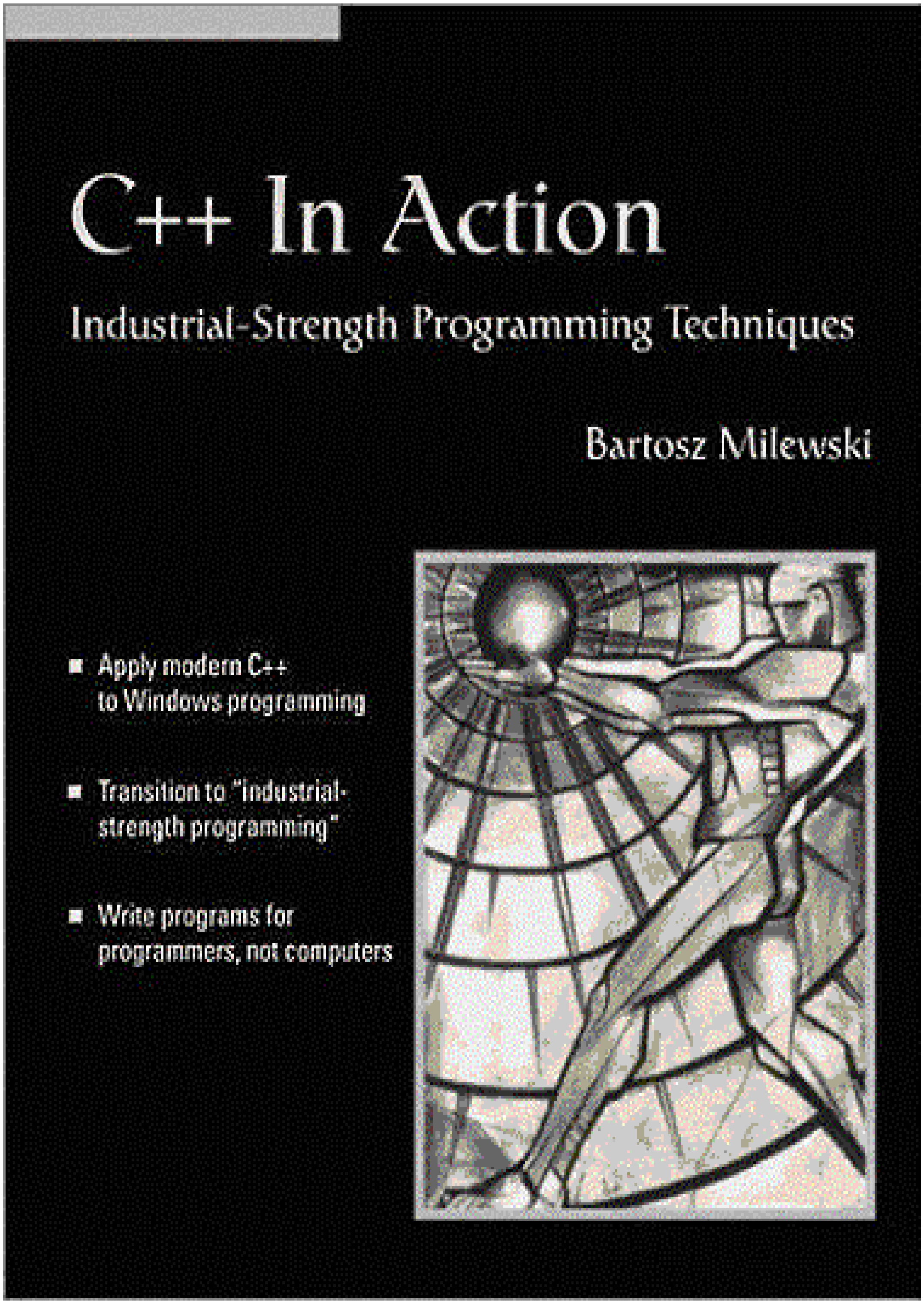


C++ In Action

Industrial-Strength Programming Techniques

Bartosz Milewski

- Apply modern C++ to Windows programming
- Transition to "industrial-strength programming"
- Write programs for programmers, not computers

Contents

Preface

Why This Book?

During the first four month of 1994 I was presented with a wonderful opportunity. My old University in Wroclaw, Poland, invited me to give two courses for the students of Computer Physics. The choice of topics was left entirely to my discretion. I knew exactly what I wanted to teach...

My work at Microsoft gave me the unique experience of working on large software projects and applying and developing state of the art design and programming methodologies. Of course, there are plenty of books on the market that talk about design, programming paradigms, languages, etc. Unfortunately most of them are either written in a dry academic style and are quite obsolete, or they are hastily put together to catch the latest vogue. There is a glut of books teaching programming in C, C++ and, more recently, in Java. They teach the language, all right, but rarely do they teach programming.

We have to realize that we are witnessing an unprecedented explosion of new hardware and software technologies. For the last twenty years the power of computers grew exponentially, almost doubling every year. Our software experience should follow this exponential curve as well. Where does this leave books that were written ten or twenty years ago? And who has time to write new books? The academics? The home programmers? The conference crowd? What about people who are active full time, designing and implementing state of the art software? They have no time!

In fact I could only dream about writing this book while working full time at Microsoft. I had problems finding time to share experiences with other teams working on the same project. We were all too busy writing software. And then I managed to get a four-month leave of absence. This is how this book started.

Teaching courses to a live, demanding audience is the best way of systematizing and testing ideas and making fast progress writing a book. The goal I put forward for the courses was to prepare the students for jobs in the industry. In particular, I asked myself the question: If I wanted to hire a new programmer, what would I like him to know to become a productive member of my team as quickly as possible?

For sure, I would like such a person to know

- C++ and object oriented programming.
- Top-down design and top-down implementation techniques.
- Effective programming with templates and C++ exceptions.
- Team work.

He (and whenever I use the pronoun *he*, I mean it as an abbreviation for *he or she*) should be able to write reliable and maintainable code, easy to understand by other members of the team. The person should know advanced programming techniques such as synchronization in a multithreaded environment, effective use of virtual memory, debugging techniques, etc.

Unfortunately, most college graduates are never taught this kind of "industrial strength" programming. Some universities are known to produce first class computer hackers (and seem to be proud of it!). What's worse, a lot of experienced programmers have large holes in that area of their education. They don't know C++, they use C-style programming in C++, they skip the design stage, they implement bottom-up, they hate C++ exceptions, and they don't work with the team. The bottom line is this: they waste a lot of their own time and they waste a lot of others' time. They produce buggy code that's difficult to maintain.

So who are you, the reader of this book? You might be a beginner who wants to learn C++. You might be a student who wants to supplement his or college education. You might be a new programmer who is trying to make a transition from the academic to the industrial environment. Or you might be a seasoned programmer in search of new ideas. This book should satisfy you no matter what category you find yourself in.

Introduction

I have divided this book into three parts, the *Language*, the *Techniques*, and the *Software Project*.

Language

The first part teaches C++, the language of choice for general-purpose programming. But it is not your usual C++ tutorial.

For the beginner who doesn't know much about C or C++, it just introduces a new object oriented language. It doesn't concentrate on syntax or grammar; it shows how to express certain ideas in C++. It is like teaching a foreign language by conversation rather than by memorizing words and grammatical rules (when I was teaching it to students, I called this part of the course "Conversational C++"). After all, this is what the programmer needs: to be able to express ideas in the form of a program written in a particular language. When I learn a foreign language, the first thing I want to know is how to say, "How much does it cost?" I don't need to learn the whole conjugation of the verb 'to cost' in the past, present and future tenses. I just want to be able to walk into a store in a foreign country and buy something.

For a C programmer who doesn't know much about C++ (other than that it's slow and cryptic--the popular myths in the C subculture) this is an exercise in unlearning C in order to effectively program in C++. Why should a C programmer unlearn C? Isn't C++ a superset of C? Unfortunately yes! The decision to make C++ compatible with C was a purely practical, marketing decision. And it worked! Instead of being a completely new product that would take decades to gain the market, it became "version 3.1" of C. This is both good and bad. It's good because backward C compatibility allowed C++, and some elements of object oriented programming, to quickly gain foothold in the programming community. It's bad because it doesn't require anybody to change his programming methodology.

Instead of having to rewrite the existing code all at once, many companies were, and still are, able to gradually phase C++ in. The usual path for such a phase-in is to introduce C++ as a 'stricter' C. In principle all C code could be recompiled as C++ . In practice, C++ has somewhat stricter type checking and the compiler is able to detect more bugs and issue more warnings. So recompiling C code using a C++ compiler is a way of cleaning up the existing code. The changes that have to be introduced into the source code at that stage are mostly bug fixes and stricter type enforcement. If the code was written in pre-ANSI C, the prototypes of all functions have to be generated. It is surprising how many bugs are detected during this ANSI-zation procedure. All this work is definitely worth the effort. A C compiler should only be used when a good C++ compiler is not available (really, a rare occurrence nowadays).

Once the C++ compiler becomes part of the programming environment, programmers sooner or later start learning new tricks and eventually they develop some kind of C++ programming methodology, either on their own or by reading various self-help books. This is where the bad news starts. There is a subset of C++ (I call it the C ghetto) where many ex-C-programmers live. A lot of C programmers start hating C++ after a glimpse of the C ghetto. They don't realize that C++ has as many good uses as misuses.

For a C-ghetto programmer this book should be a shock (I hope!). It essentially says, "whatever you did up to now was wrong" and "Kernighan and Ritchie are not gods". (Kernighan and Ritchie are the creators of C and the authors of the influential book *The C Programming Language*). I want to make this clear right here and now, in the introduction. I understand that the first, quite natural, reaction of such a programmer is to close the book immediately (or, actually, jump to another Internet site) and ask for a refund. Please don't do this! The shocking, iconoclastic value of this book is not there to hurt anybody's feelings. Seeing that there exists a drastically different philosophy is supposed to prompt one to rethink one's beliefs. Besides, the Emperor *is* naked.

For a C++ programmer, the tutorial offers a new look at the language. It shows how to avoid the pitfalls of C++ and use the language according to the way it should have been designed in the first place. I would lie if I said that C++ is a beautiful programming language. However, it is going to be, at least for some time, the most popular language for writing serious software. We may as well try to take advantage of its expressive power to write better software, rather than use it to find so many more ways to hurt ourselves. For a C++ programmer, this part of the book should be mostly easy reading. And, although the constructs and the techniques introduced there are widely known, I tried to show them from a different perspective. My overriding philosophy was to create a system that promotes maintainable, human-readable coding style. That's why I took every opportunity not only to show various programming options but also to explain why I considered some of them superior to others.

Finally, for a Java programmer, this book should be an eye-opener. It shows that, with some discipline, it is possible to write safe and robust code in C++. Everything Java can do, C++ can do, too. Plus, it can deliver unmatched performance.

But performance is not the only reason to stick with C++. The kind of elegant resource management that can be implemented in C++ is quite impossible in Java, because of Java's reliance on garbage collection. In C++ you can have objects whose lifetime is precisely defined by the scope they live in. You are guaranteed that these objects will be destroyed upon the exit from that scope. That's why you can entrust such objects with vital resources, like semaphores, file handles, database transactions, etc. Java objects, on the other hand, have undefined life spans--they are deallocated only when the runtime decides to collect them. So the way you deal with resources in Java harks back to the old C exception paradigm, where the `finally` clause had to do all the painfully explicit garbage collection.

There are no "native" speakers of C++. When "speaking" C++, we all have some accent that reveals our programming background. Some of us have a strong C accent, some use Smalltalk-like expressions, others Lisp--The goal of the tutorial is to come as close as possible to being a native speaker of C++. Language is a tool for expressing ideas. Therefore the emphasis is not on syntax and grammar but on the ways to express yourself. It is not "Here's a cute C++ construct and this is how you might use it." Instead it is more of "Here's an idea. How do I express it in C++?" Initially the 'ideas' take the form of simple sentences like "A star is a celestial body," or "A stack allows you to push and pop." Later the sentences are combined to form 'paragraphs.' describing the functionality of a software component. The various constructs of C++ are introduced as the need arises, always in the context of a problem that needs to be solved.

Techniques

Writing good software requires much more than just learning the language. Firstly, the program doesn't execute in a vacuum. It has to interact with the computer. And interacting with the computer means going through the operating system. Without having some knowledge of the operating system, it is impossible to write serious programs. Secondly, we not only want to write programs that run--we want our programs to be small, fast, reliable, robust and scaleable. Thirdly, we want to finish the development of a program in a sensible amount of time, and we want to maintain and enhance it afterwards.

The goal of the second part of the book, The *Techniques*, is to make possible the transition from 'weekend programming' to 'industrial strength programming.'

I will describe the technique that makes programming in C++ an order of magnitude more robust and maintainable. I call it "managing resources" since it is centered on the idea of a program creating, acquiring, owning and releasing various kinds of resources. For every resource, at any point in time during the execution of the program, there has to be a well-defined owner responsible for its release. This simple idea turns out to be extremely powerful in designing and maintaining complex software systems. Many a bug has been avoided or found and fixed using resource ownership analysis.

Resource management meshes very naturally with C++ exception handling. In fact, writing sensible C++ programs that use exceptions seems virtually impossible without the encapsulation of resources. So, when should you use exceptions? What do they buy you? It depends on what your response is to the following simple question: Do you always check the result of `new` (or, for C programmers, the result of `malloc`)? This is a rhetorical question. Unless you are an exceptionally careful programmer--you don't. That means you are already using exceptions, whether you want it or not. Because accessing a null pointer results in an exception called the General Protection Fault (GP-fault or Access Violation, as the programmers call it). If your program is not exception-aware, it will die a horrible death upon such an exception. What's more, the operating system will shame you by putting up a message box, leaving no doubt that it was your application that was written using sub-standard programming practices (maybe not in so many words).

My point is, in order to write robust and reliable applications--and that's what this book is about--you will sooner or later have to use exceptions. Of course, there are other programming techniques that were and still are being successfully applied to the development of reasonably robust and reliable applications. None of them, however, comes close in terms of simplicity and maintainability to the application of C++ exceptions in combination with the resource management techniques.

I will introduce the interaction with the operating system through a series of Windows programming exercises. They will lead the reader into new programming paradigms: message-based programming, Model-View-Controller approach to user interface, etc.

The advances in computer hardware paved the way to a new generation of PC operating systems. Preemptive multitasking and virtual memory are finally mainstream features on personal computers. So how does one write an application that takes advantage of multitasking? How does one synchronize multiple threads accessing the same data structure? And most importantly, how does multitasking mesh with the object-oriented paradigm and C++? I will try to answer these questions.

Virtual memory gives your application the illusion of practically infinite memory. On a 32-bit system you can address 4 gigabytes of virtual memory--in practice the amount of available memory is limited by the size of your hard disk(s). For the application you write it means that it can easily deal with multimegabyte memory based data structures. Or can it? Welcome to the world of thrashing! I will explain which algorithms and data structures are compatible with virtual memory and how to use memory-mapped files to save disk space.

Software Project

There is more to the creation of a successful application (or system) than just learning the language and mastering the techniques. Today's commercial software projects are among the most complex engineering undertakings of humankind. Programming is essentially the art of dealing with complexity. There were many attempts to apply traditional engineering methods to control software's complexity. Modularization, software reuse, software IC's, etc. Let's face it--in general they don't work. They may be very helpful in providing low level building blocks and libraries, but they can hardly be used as guiding principles in the design and implementation of complex software projects.

The simple reason is that there is very little repetition in a piece of software. Try to visually compare a printout of a program with, say, a picture of a microprocessor wafer. You'll see a lot of repetitive patterns in the layout of the microprocessor. Piece-wise it resembles some kind of a high-tech crystal. A condensed view of a program, on the other hand, would look more like a high-tech fractal. You'd see a lot of self-similarities--large-scale patterns will resemble small-scale patterns. But you'd find very few exact matches or repetitions. Each little piece appears to be individually handcrafted. Repetitions in a program are not only unnecessary but they contribute to a maintenance nightmare. If you modify, or bug-fix, one piece of code, your are supposed to find all the copies of this piece and apply identical modifications to them as well.

This abhorrence of repetition is reflected in the production process of software. The proportion of research, design and manufacturing in the software industry is different than in other industries. Manufacturing, for instance, plays only a marginal role. Strictly speaking, electronic channels of distribution could make the manufacturing phase totally irrelevant. R & D plays a vital role, more so than in many other industries. But what really sets software development apart from others is the amount of design that goes into the product. Programming *is* designing. Designing, building prototypes, testing--over and over again. Software industry is the ultimate "design industry."

In the third part of the book I will attempt to describe the large-scale aspects of software development. I will concentrate on the dynamics of a software project, both from the point of view of management and planning as well as development strategies and tactics. I will describe the dynamics of a project from its conception to shipment. I will talk about documentation, the design process and the development process. I will not, however, try to come up with ready-made recipes because they won't work--for exactly the reasons described above.

There is a popular unflattering stereotype of a programmer as a socially challenged nerd. Somebody who would work alone at night, subsist on Twinkies, avoid direct eye contact and care very little about personal hygiene. I've known programmers like that, and I'm sure there are still some around. However most of the specimens of this old culture are becoming extinct, and for a good reason. Progress in hardware and software makes it impossible to produce any reasonably useful and reliable program while working in isolation. Teamwork is the essential part of software development.

Dividing the work and coordinating the development effort of a team is always a big challenge. In traditional industries members of the team know (at least in theory) what they are doing. They learned the routine. They are performing a synchronized dance and they know the steps and hear the music. In the software industry every team member improvises the steps as he or goes and, at the same time, composes the music for the rest of the team.

I will advocate a change of emphasis in software development. Instead of the old axiom *Programs are written for computers*. I will turn the logic upside down and claim that *Programs are written for programmers*. This statement is in fact the premise of the whole book. You can't develop industrial strength software if you don't treat you code as a publication for other programmers to read, understand and modify. You don't want your 'code' to be an exercise in cryptography.

The computer is the ultimate proofing tool for your software. The compiler is your spell-checker. By running your program you attempt to test the correctness of your publication. But it's only another human being--a fellow programmer--that can understand the meaning of your program. And it is crucial that he do it with minimum effort, because without understanding, it is impossible to maintain your software.

Language

Objects and Scopes

What's the most important thing in the Universe? Is it matter? It seems like everything is built from matter-galaxies, stars, planets, houses, cars and even us, programmers. But what's matter without energy? The Universe would be dead without it. Energy is the source of change, movement, life. But what is matter and energy without space and time? We need space into which to put matter, and we need time to see matter change.

Programming is like creating universes. We need matter: data structures, objects, variables. We need energy--the executable code--the lifeforce of the program. Objects would be dead without code that operates on them. Objects need space to be put into and to relate to each other. Lines of code need time to be executed. The space-time of the program is described by scopes. An object lives and dies by its scope. Lines of executable code operate within scopes. Scopes provide the structure to program's space and time. And ultimately programming is about structure.

Arrays and References

In a program, an object is identified by its name. But if we had to call the object by its name everywhere, we would end up with one global name space. Our program would execute in a structureless "object soup." The power to give an object different names in different scopes provides an additional level of indirection, so important in programming. There is an old saying in Computer Science--every problem can be solved by adding a level of indirection. This indirection can be accomplished by using a reference, an alias, an alternative name, that can be attached to a different object every time it enters a scope.

Computers are great at menial tasks. They have a lot more patience that we humans do. It is a punishment for a human to have to write "I will not challange my teacher's authority" a hundred times. Tell the computer to do it a hundred times, and it won't even blink. That's the power of iteration (and conformity).

Pointers

Using references, we can give multiple names to the same object. Using pointers, we can have the same name refer to different objects--a pointer is a *mutable* reference.

Pointers give us power to create complex data structures. They also increase our ability to shoot ourselves in the foot. Pointer is like a plug that can be plugged into a jack. If you have too many plugs and too many jacks, you may end up with a mess of tangled cables. A programmer has to strike a balance between creating a program that looks like a breadboard or like a printed circuit.

Polymorphism

Polymorphic means *multi-shaped*. A tuner, a tape deck, a CD player--they come in different shapes but they all have the same *audio-out* jack. You can plug your earphones into it and listen to music no matter whether it came as a modulation of a carrier wave, a set of magnetic domains on a tape or a series of pits in the aluminum substrate on a plastic disk.

- ## Small Software Project

When you write a program, you don't ask yourself the question, "How can I use a particular language feature?" You ask, "What language feature will help me solve my problem?"

Objects and Scopes

Global scope

Class definition, object definition, constructor, destructor, output stream, include, main.

There is an old tradition in teaching C, dating back to Kernighan and Ritchie (The C Programming Language), to have the first program print the greeting "Hello World!". It is only appropriate that our first C++ program should respond to this greeting. The way to do it, of course, is to create the World and let it speak for itself.

The following program does just that, but it also serves as a metaphor for C++ programming. Every C++ program is a world in itself. The world is a play and we define the characters in that play and let them interact. This program in a sense is "the Mother of all C++ programs," it contains just one player, the World, and lets us witness its creation and destruction. The World interacts with us by printing messages on the computer screen. It prints "Hello!" when it is created, and "Good bye!" when it vanishes. So here we go:

```
#include <iostream>

class World
{
public:
    World ()  { std::cout << "Hello!\n"; }
    ~World () { std::cout << "Good bye!\n"; }
};

World TheWorld;

void main() {}
```

This program consists of the following parts:

- The include statement,
- The class definition,
- The object definition, and
- The main routine.

Let's start from the end, from the main function, and work our way backwards. Whenever you see something weird in C++ it is probably there for the sake of compatibility with C. That's the case with `main()`[1]. In this particular program it serves no purpose whatsoever, and in fact is quite empty. It takes no parameters-you can tell that from the empty set of parentheses; does nothing-you can tell that from the empty set of braces; and returns nothing-you can tell that from the keyword void in front of it. But it has to be there.

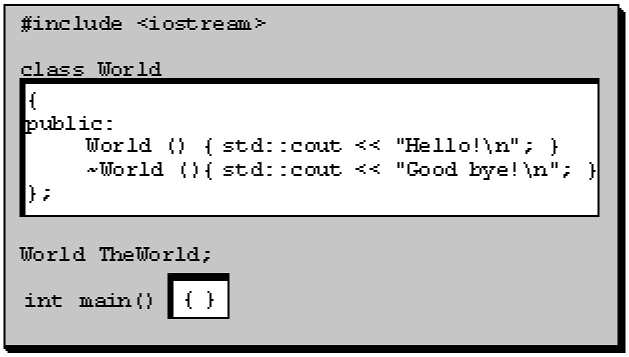
The line:

```
World TheWorld;
```

defines the object TheWorld of type World. You can also say, TheWorld is an instance of the class World. Like every statement in C++, it is delimited by a semicolon. (Quick exercise: Find all the statements in our program.)

This is the central line of the program. Show this program to a C programmer and ask him what it does. He will say "Nothing!" That's because a C programmer looks at main() and sees nothing there. The trained eye of a C++ programmer will spot the global definition of TheWorld and he will say "Ha!"

Global means "outside of any curly braces." All this space outside of the curly braces is considered the global scope, Figure 1.



```
#include <iostream>

class World
{
public:
     World () { std::cout << "Hello!\n"; }
     ~World (){ std::cout << "Good bye!\n"; }
};

World TheWorld;

int main() { }
```

Figure 1 Global scope is everything outside of curly braces.

Next, following our inverted order of analysis, is the definition of the type World. It turns out that World is a class-that is a type defined by the programmer. The definition is inside curly braces (and is delimited by a semicolon, did you miss that one?). The keyword public means that we have nothing to hide yet. Later we'll learn about data hiding and we won't be that open any more.

First inside the class definition is the constructor. Constructor is a piece of code that is to be executed every time an object of this particular class is created. It always has the same name as the class itself, but it may take arguments-that's why it has a set of parentheses following it-they're empty, so this particular one doesn't take any arguments. The constructor does something-the curly braces following it are not empty. They contain the statement

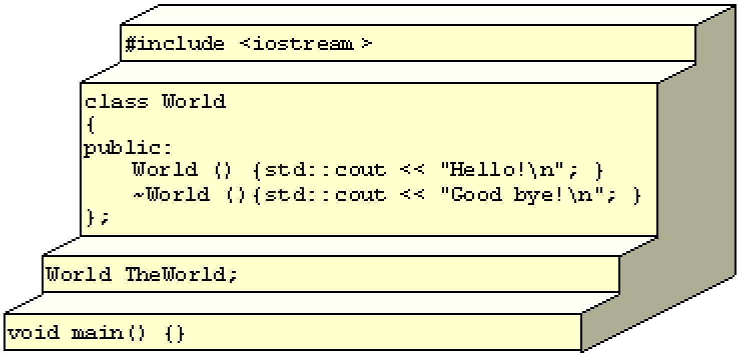
```
std::cout << "Hello!\n";
```

It means that the object std::cout is sent the string "Hello!\n". This particular predefined object std::cout represents the standard output, presumably the screen of the computer, and it prints whatever is sent to it. (Now we know where this "Hello!" came from.) By the way, '\n' (backslash n) at the end of the string means "newline"-so that the next printout will start on a new line.

Next is the destructor, which always has the same name as the class but is preceded by a tilde. It is the piece of code that is executed every time an object of this class is destroyed. It never takes any arguments, so the parentheses following its name are always empty. The destructor of World also does something. It prints "Good Bye!\n".

At the top of the file we have an include statement. It tells the compiler to find the file iostream and include it right there. If your compiler is properly installed, it will find it; if not, reinstall it or read the manual. (Sorry, I have no idea what compiler you are using.) The compiler needs this file to find out what the heck std::cout is and what can be done to it. This object (of the class iostream) is part of the standard C++ library, not part of the language.

The order in which we have analyzed this program was far from random. It is called top-down and is the right way of looking at programs (and writing them, too). If you're confused where the top is and which way is down, just imagine that the program has a third dimension and it looks sort of like a set stairs, Figure 2. The main routine sits at the top of the stairs, closer to you. Global definitions are one step lower. Class definitions are next, and so on.



```
#include <iostream >
```

```
class World
{
public:
    World () {std::cout << "Hello!\n"; }
    ~World (){std::cout << "Good bye!\n"; }
};
```

```
World TheWorld;
```

```
void main() {}
```

Figure 2. Trying to visualize the top-down view of the program. Imagine that the top is closer to you and the bottom if further from you.

The stage is now set, the main player is TheWorld. Its character is defined by the class World-it does something when it is constructed and does something when it is destroyed. The play starts. Here's what happens:

1. All global objects are constructed. In our case the constructor of TheWorld is executed and prints "Hello!" on the screen.
2. The main routine is executed. In our case it does something very important. It makes C programmers happy.
3. All global objects are destroyed. In our case the destructor of TheWorld is executed and it prints "Good bye!"

That's it!

Well, not really. If programming were that simple, everybody would be a programmer. In order to make it difficult there is a whole lot of magical incantations that have to be typed into the computer in order to make the program run. It's these incantations that provide our job security. I can tell you what incantations I had to type into my computer in order to run this program. And I am already assuming a lot. That I am in the proper directory, that I have the file world1.cpp in it, that the compiler is properly installed, and so on, so forth. I typed

```
cl world1.cpp
```

`cl` is the nickname of my C++ compiler. Yours might have a different nickname, so check with you compiler manual. Once the compilation was finished, I just typed

```
world1
```

and voila! By the way, this is the name of the resulting program on my system.

One more hint. If the compiler refuses to compile your program, look at its output. It is supposed to tell you where and what errors it has found. If you understand these error messages, you are probably an experienced programmer already. If you don't, don't despair. Only experienced programmers can understand them.

Your best bet is probably some kind of an integrated programming environment, with a built-in editor, compiler, debugger and more. You'll be able to build and run your program with a single mouse click. Well, not exactly. You usually have to tell the program that you want to create a new project. The type of the project, in our case, is a *console* application (as opposed to a *Windows* application). Then you have to create a new file, type in the code and save it under the name world1.cpp. Next, and I'm not making it up, you must tell the program to add the file you have just created (with the above mentioned program's help) to the project. Once the file is in the project you'll need to build the executable. *Building* means compiling and linking. If there are any compilation errors, you'll be able to jump to the line in your program that caused the error and try to correct it.

Troubleshooting tips:

- Your compiler can't find the header file, *iostream*. You might have an old compiler that doesn't support the new standard library. Try changing *iostream* to *iostream.h* and removing the **std::** prefixes in front of *cout*. This is a temporary fix, before you upgrade your compiler.
- Your program prints "Hello!", but doesn't print "Good bye!" You have a sub-standard library (unfortunately, even the newest VC++ v. 6.0 has this flaw). In other words, your compiler does not follow the C++ standard. Try to get a refund (good luck!). Seriously, most examples in this book won't require this feature, so don't worry too much.

At this point you might want to make a small detour into a short Visual C++ tutorial.

In most environments, after you've successfully built your program, you can single-step through it under a debugger. You'll be able to see how each statement is executed and how it changes the state of the program.

To summarize, we have introduced a class and an object. Think of a class as a blueprint for an object. It describes the object's behavior-in our case, what happens when the object is constructed and destroyed. Once you have a blueprint, you can create objects that are based on it.

[1] One could think of main as the constructor of the global Main object. This is how truly object oriented languages work.

Local scope

Scope of main, passing arguments to constructors, integer type. Private data members, initialization, embedded local scopes, the for loop.

Global scope extends outside of any braces. It is activated (that is, ready to use, all objects constructed) before main is executed, and deactivated (all objects destroyed) after main is exited. Local scopes, on the other hand, are delimited by braces (well, not always, we'll see in a moment that there are cases where braces can be omitted). Local scope is activated when the flow of the program enters it, and deactivated when it leaves it. Objects in local scope are constructed whenever their definition is encountered, and destroyed when the scope is exited. Such objects are also called *automatic* (because they are automatically created and destroyed), or stack objects (because they occupy memory that is allocated on the program's stack).

Our first example of the use of local scope creates a local `World` called `myWorld` within the scope of `main()`. The `myWorld` object is constructed right after entering `main()`, then "Hello from main" is printed, and finally the object is destroyed right before exiting the scope of `main`. The global `World` is still constructed before `main()` and destroyed after `main()`.

Another modification to our original program is the use of an integer as an argument to the constructor of `World`. An integer is of the predefined type `int`. Its size is compiler dependent. In the constructor, this argument (called `i`) is sent to the standard output. Notice how clever the `std::cout` object is. It accepts strings of characters and prints them as strings and it accepts integers and prints them as decimal numbers. And, as you can see, you can chain arguments to std::cout one after another.

Since the constructor expects an argument, we have to provide it when we create the object. Here we just specify it in parentheses after the name of the object being created

When the constructor of `TheWorld` is executed it prints "Hello from 1."

```
#include <iostream>

class World
{
public:
    World (int i)
    {
        std::cout << "Hello from " << i << ".\n";
    }

    ~World ()
    {
        std::cout << "Good bye.\n";
    }
};

World TheWorld (1);

int main()
{

    World myWorld (2);
    std::cout << "Hello from main!\n";
}
```

There's still something missing. How do we know which object printed the first "Good bye." and which object printed the second one? Our objects don't remember their identity. In fact they don't remember anything! But before we fix that, let me digress philosophically.

Here we are talking about object oriented programming and I haven't even defined what I mean by an object.

So here we go:

Definition: An object is something that has identity.

If you can think of something that doesn't have an identity, it's not an object. Beauty is not an object. But if you could say "this beauty is different from the one over there," you would give the two beauties their identities and they would become objects. We sometimes define classes that have non-object names. That just means that we give an old name a new "objectified" meaning. In general, though, it's not such a great idea. If possible one should stick to *countable nouns*.

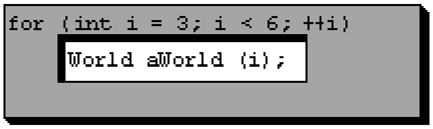
I didn't mean to say that our Worlds didn't have identity, because they did. They were just not aware of it. Let's give them memory, so that they'll be able to remember who they are.

```
#include <iostream>

class World
{
public:
    World (int id)
       : _identifier (id)
    {
        std::cout << "Hello from " << _identifier << ".\n";
    }

    ~World ()
    {
        std::cout << "Good bye from " << _identifier << ".\n";
    }
private:
    const int _identifier;
};

World TheWorld (1);

int main ()
{
    World myWorld (2);
    for (int i = 3; i < 6; ++i)
    {
        World aWorld (i);
    }
    World oneMoreWorld (6);
}
```

Several new things require explanation. Let's do it top-down. In main, we create our local World, myWorld, with an id of 2 (this time the World will remember it until it is destroyed). Next we have a for loop. The meaning of it is the following: For integer i starting from 3 as long as it's less than 6, incremented every time the body of the loop is executed (that's the meaning of

++i), do this: Open the scope, define a World called aWorld with an id equal to the current value of i, close the scope.

After the loop is done iterating, one more World is defined called oneMoreWorld. It has the id of 6. The braces delimiting the scope of the for loop may be omitted. This is because here the body of the loop consists of a single statement. So, in fact, we could have written the loop as in Figure 3 and the program would still execute exactly the same way.



```
for (int i = 3; i < 6; ++i)
    World aWorld (i);
```

Figure 3 A scope without braces.

The body of the for loop forms a separate scope within the scope of main. When the program enters this scope, the objects from the outer scope(s) are not destroyed. In fact, if there is no name conflict, they are still visible and accessible. There would be name conflict if myWorld were called aWorld. It would be perfectly okay, only that the outer aWorld would be temporarily inaccessible within the for loop: the name of the outer aWorld would be hidden by the name of the inner aWorld. We'll see later what exactly *accessible* means.

What is the scope of the variable i that is defined in the header of the for loop?

```
for (int i = 3; i < 6; ++i)
{
    World aWorld (i);
}
```

This is actually a nontrivial question. It used to be that its scope extended beyond the scope of the loop. That's no longer true. The scope of a variable defined in the head of a loop (or the if statement, when we get to it) is that of the loop itself. Notice that it's not the same as the scope of the *body* of the loop. Variables defined in the body of the loop are re-initialized on every iteration. The ones defined in the head are initialized once, at loop entry. However, once you exit the loop, neither are accessible.

We still have our global World, TheWorld, with an id of 1.

The class World has a data member _identifier. Its type is int and it's const and private. Why is it private? None of your business. Okay, okay... It's called *data hiding*. It means: today it is implemented like this, tomorrow, who knows. Maybe an int, maybe a string, and maybe not at all. If the _identifier weren't private, it would be like with these hidden functions in MS DOS that everybody knew about: Applications started using them and the developers of MS DOS had to keep them forever for compatibility. Not so in our World. The compiler will make sure that nobody, except the World itself, has access to private members.

The keyword const means that nobody can change the value of _identifier during the lifetime of the object. It is physically (or at least "compilatorily") impossible. But even a const has to be initialized some place or another. The compiler provides just one little window when we can (and in fact have to) initialize a const data member. It is in the *preamble* to the constructor. Not even in the body of the constructor--only in the preamble itself can we initialize a const data member. Preamble is the part of the constructor that starts with a colon and contains a comma-separated list of data members followed by their initializers in parentheses. In our preamble `_identifier` is initialized with the value of `id`.

```
    World (int id)
       : _identifier (id)
    {
        std::cout << "Hello from " << _identifier << ".\n";
    }
```

Once `_identifier` is initialized, its value for the particular instance of `World` never changes (later we'll learn how to modify non-constant data members). When the destructor of this object is called, it will print the value that was passed to it in the constructor. So when we see "Good bye from 1." we'll know that it's the end of `TheWorld` (as opposed to the end of `aWorld`).

Why does `_identifier` have an underscore in front of it? It's a convention. In this book the names of all private data members will start with an underscore. Once you get used to it, it actually helps readability. Of course, you can use your own convention instead. In fact everybody may come up with their own conventions, which is exactly what happens all the time.

Speaking of conventions--I will absolutely insist on the brace positioning convention, and for a good reason: In the style of programming that this book promotes, scopes play a central role. It is of paramount importance for the readability of code to make scopes stand out graphically. Therefore:

The opening brace of any scope should be vertically aligned with the closing brace of that scope.

The screen and printer paper saving convention of yesteryear

```
for (int i = 3; i < 6; ++i) { // <- BAD!!!
     World aWorld (i);
}
```

is declared harmful to program maintainability.

To summarize, objects may have their own memory, or state. The state data members are preferably kept private. They are best initialized in the preamble to the constructor, so that the object is in a consistent state immediately after its birth. The for loop creates its own scope. This scope is entered and exited during every iteration (this is why you see all these "Good bye from 3, 4, 5" messages). In fact a matched pair of braces can be put anywhere within another scope, just to form a sub-scope.

Embedded objects

Embeddings, initialization of embeddings, order of construction/destruction.

We've seen data members of type int--one of the built in types. The beauty of C++ is that it makes virtually no distinction between built in types and the ones defined by the programmer. When a data member of some class is of a user defined type, it is called an embedded object. In the following example `Matter` is embedded in `World` (I just gave matter identity. You know, this *matter* here is much more stable than the one in the neighboring universe.) Another way to put it is--`World` contains `Matter`.

```
#include <iostream>
```

```
class Matter
{
public:
    Matter (int id)
        : _identifier(id)
    {
        std::cout << "  Matter for " << _identifier << " created\n";
    }
    ~Matter ()
    {
        std::cout << "  Matter in " << _identifier << "
annihilated\n";
    }
private:
    const int _identifier;
};

class World
{
public:
    World (int id)
        : _identifier (id), _matter (_identifier) // initializing
embeddings
    {
        std::cout << "Hello from world " << _identifier << ".\n";
    }

    ~World ()
    {
        std::cout << "Good bye from world " << _identifier << ".\n";
    }
private:
    const int      _identifier;
    const Matter   _matter; // Embedded object of type Matter
};

World TheUniverse (1);

int main ()
{
    World myWorld (2);
}
```

What's interesting in this example is the preamble to the constructor of World.

```
World (int id)
        : _identifier (id), _matter (_identifier)
```

It first initializes the _identifier and than _matter. Initializing an object means calling its constructor. The constructor of Matter requires an integer, and that's what we are passing there. We made _matter const for the same reason we made _identifier const. It is never changed during the lifetime of the World. It is initialized in the preamble and never even accessed by anybody.

By the way, the double slash // is used for comments. The compiler ignores everything that follows // until the end of line. And now for the surprise: The order of initialization has *nothing to do* with the order in the preamble. In fact, if the constructor of the embedded object doesn't require any arguments, it can be omitted from the preamble whatsoever. If no explicit initialization is required, the whole preamble may be omitted.

Instead, the rule of initialization is:

> **Data members are initialized in the order in which they appear in the class definition.**

Since _identifier appears before _matter in the definition of World, it will be initialized first. That's why we could use its value to in the construction of _matter. The order of embeddings in C++ is as important as the order of statements.

The destruction of embedded objects is guaranteed to proceed in the opposite order of construction. The object that was constructed first will be destroyed last, and so on.

In the object-oriented argot, object embedding is called the "has-a" relationship. A World has a Matter (remember, we have objectified matter).

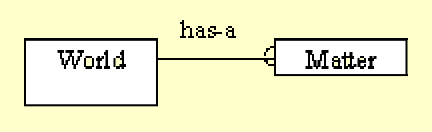


Figure 4 Graphical representation of the has-a relationship.

To summarize, objects of programmer defined types may be embedded as data members in other classes. The constructors for the embeddings are called before the body of the object's constructor is executed (conceptually you may visualize them as being called in the preamble). If these constructors need arguments, they must be passed in the preamble. The order of construction is determined by the order of embeddings. The embeddings are destroyed in the reverse order of construction.

Inheritance

Public inheritance, initialization of the base class, order of construction/destruction, type double.

Another important relationship between objects is the "is-a" relationship. When we say that a star is a celestial body, we mean that a star has all the properties of a celestial body, and maybe some others, specific to a star. In C++ this relationship is expressed using inheritance. We can say: class Star inherits from class CelestialBody, or that CelestialBody is the base class of Star.

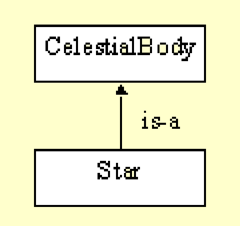


Figure 5 Graphical representation the is-a relationships between objects.

Here's an example:

```
#include <iostream>

class CelestialBody
{
public:
    CelestialBody (double mass)
        : _mass (mass)
    {
        std::cout << "Creating celestial body of mass " << _mass <<
"\n";
    }

    ~CelestialBody ()
    {
        std::cout << "Destroying celestial body of mass " <<
            _mass << "\n";
    }

private:
    const double _mass;
};

class Star: public CelestialBody  // Star is a CelestialBody
{
public:
    Star (double mass, double brightness)
        : CelestialBody (mass), _brightness (brightness)
    {
        std::cout << "Creating a star of brightness " <<
              _brightness << "\n";
    }

    ~Star ()
    {
        std::cout << "Destroying a star of brightness " <<
            _brightness << "\n";
    }

private:
    const double _brightness;
};

int main ()
{
    std::cout << "    Entering main.\n";
    Star aStar ( 1234.5, 0.1 );
    std::cout << "    Exiting main.\n";
}
```

The line

```
class Star: public CelestialBody  // Star is a CelestialBody
```

tells the compiler that `Star` inherits everything from `CelestialBody`. In particular, since `CelestialBody` has `_mass`, `Star` will have `_mass`, too.

CelestialBody has a constructor that requires an argument of the type double. Double is a built in type corresponding to double precision floating point numbers. Its single precision counterpart is called a float. Notice that our clever object std::cout has no problem printing doubles.

In the preamble to the constructor of Star we have to pass the argument to the constructor of CelestialBody. Here's how we do it-we invoke the base constructor using the name of its class:

```
    Star (double mass, double brightness)
        : CelestialBody (mass), _brightness (brightness)
```

The order of construction is very important.

First the base class is fully constructed, then the derived class.

The construction of the derived class proceeds in the usual order: first the embeddings, then the code in the constructor. Again, the order in the preamble is meaningless, and if there are no explicit initializations the whole thing may be omitted. The order of destruction is again the reverse of the order of construction. In particular, the derived class destructor is called first, the destructor of the base class follows.

Member functions and Interfaces

Input stream, member functions (methods), return values, interfaces, functions returning void.

So far we have been constructing and destroying objects. What else can we do to an object? We can't access its private data members, because the compiler wouldn't let us (not to mention that I haven't even talked about the syntax one would use to do it). It turns out that only the object itself (as specified in its class definition) can make a decision to expose some of its behavior and/or contents. Behavior is exposed via member functions, also called methods. If a member function is public, anybody can call it (we'll see the syntax in a moment).

Our first member function is called GetValue. It returns an int. From the implementation of this function we can see that the returned value is that of the private data member _num. Now we know what it means to have access to an object; it means to be able to invoke its member functions. Here, the object num of the type InputNum is defined in the scope of main and that's where we can access it. And access we do, calling num.GetValue(). To put it in different words, we invoke the method GetValue() on the object num. Or, we are getting the value of object num. In still other words (Smalltalk-speak) we send the message GetValue to the object num. What we are getting back is an int, which we print immediately using our old friend std::cout.

```
#include <iostream>

class  InputNum
{
public:
    InputNum ()
    {
        std::cout << "Enter number ";
        std::cin >> _num;
    }

    int GetValue () const {  return _num; }
private:
    int _num;
};

int main()
{
    InputNum num;
    std::cout << "The value is " << num.GetValue() << "\n";
    return 0;
}
```

A few words of syntax clarification are in order. The type of the return value is always specified in front of the member function definition (or declaration, as will be explained later). A method that is supposed to return something must have a return statement at the end (later we'll see that a return from the middle is possible, too). If a function is not supposed to return anything, its return type is void. In all our previous examples main wasn't returning anything, so it was declared void.

Not this time though. It turns out that main can return an integer. In fact it always returns an integer, even if it is defined as returning void (in that case it returns a more or less random number). By convention, main is supposed to return 0 if the program was successful. Otherwise it is supposed to return an error level. You can check for the error level after you've run your program from a batch file using the statement similar to this:

```
if not errorlevel 1 goto end
```

The keyword const following the declaration of a method, as in

```
int GetValue () const
```

means that this method does not change the state of the object. The compiler will not allow such a method to perform assignment, increment, or decrement of any data member, nor will it permit this method to call any non-constant methods of that class.

Moreover, only methods declared const may be called on a const object. So far we've seen a const object _matter inside World. If Matter had a const method, like GetValue, it would have been okay to call it in the context of the object _matter (that is: _matter.GetValue() would compile all right). We'll see examples of that later.

By the way, _num cannot be declared const any more, since it is not initialized in the preamble. The code in the body of the constructor is changing the (undefined) value of _num. Try compiling this program with a const _num and you'll see what happens.

When the object num of class InputNum is constructed in main, it prompts the user to input a number. It then waits for a number and stores it in its private data member _num. This is done through the services of the input object std::cin. Std::cin gets input from the keyboard and stores it in a variable. Depending on the type of the variable, std::cin expects different things from the keyboard. For instance, if you call it with an int, it expects an integer; if you call it with a double, it expects a floating point number, etc.

The set of public methods is called the *interface* and it defines the way clients may interact with the object. InputNum, for instance, provides read-only access to its contents. Because of that, we are guaranteed that the value of InputNum will never change once it is created. Successive calls to GetValue() will always return the same value.

If it makes sense, it is also possible for an object to grant write access to its contents--for example, in our case, by defining the method

```
void SetValue (int i) { _num = i; }
```

Then, after the call

```
num.SetValue (10);
```

subsequent calls to GetValue would return 10, no matter what number was input in the constructor of num. By the way, the equal sign in C++ signifies assignment (same as := in Pascal). The statement

```
_num = i;
```

sets the value of _num to that of i. Of course the method SetValue cannot be defined const, because it changes the state of the object (try it!). And if Matter had such a non-constant method, that method couldn't have been called in the context of the constant object _matter inside the World. The compiler would have strongly objected.

To summarize, the set of public member functions defines the interface to the object of a given class. What is even more important, a well designed and well implemented object is able to fulfill a well defined contract. Programming in C++ is about writing contracts and fulfilling them. An interface can, and should, be designed in such a way that the client's part of the contract is simple and well defined. The object may even be able to protect itself from sloppy clients who break their part of the contract. We'll talk about it when we discuss the use of assertions.

Member function scope

Member function scope, strings as arrays of chars, calling other member functions.

In the next example we will show how the body of a member function forms its own local scope. We will be able to define objects within that scope, create sub-scopes, and so on.

> But first, let's make some simplifications. You are probably tired of typing std:: in front of cin and cout. It is actually enough to tell the compiler once, before you start using them, that you are about to use them. So, once you say the magic words
>
> ```
> using std::cout;
> ```
>
> you can omit std:: in front of cout. The same goes for cin. There's an even more general way of getting rid of the std:: prefixes, by declaring
>
> ```
> using namespace std;
> ```
>
> That's because the whole standard library sits in a *namespace* called std. We'll come back to namespaces later.

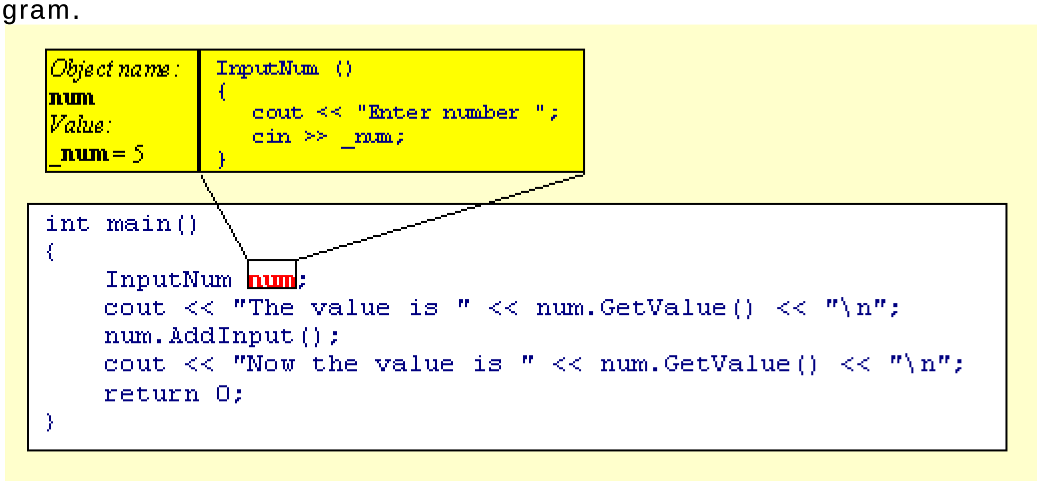
So here's a new twist in our input object.

```
#include <iostream>
using std::cout;
using std::cin;
```

```
class  InputNum
{
public:
    InputNum ()
    {
        cout << "Enter number ";
        cin >> _num;
    }

    int GetValue () const {  return _num; }

    void AddInput ()
    {
        InputNum aNum;  // get a number from user
        _num = _num + aNum.GetValue ();
    }

private:
    int _num;
};

int main()
{
    InputNum num;
    cout << "The value is " << num.GetValue() << "\n";
    num.AddInput();
    cout << "Now the value is " << num.GetValue() << "\n";
    return 0;
}
```

A new method was added to InputNum. It uses a local object of type InputNum to prompt the user for a number. This number is then retrieved using GetValue() and added to the original value.

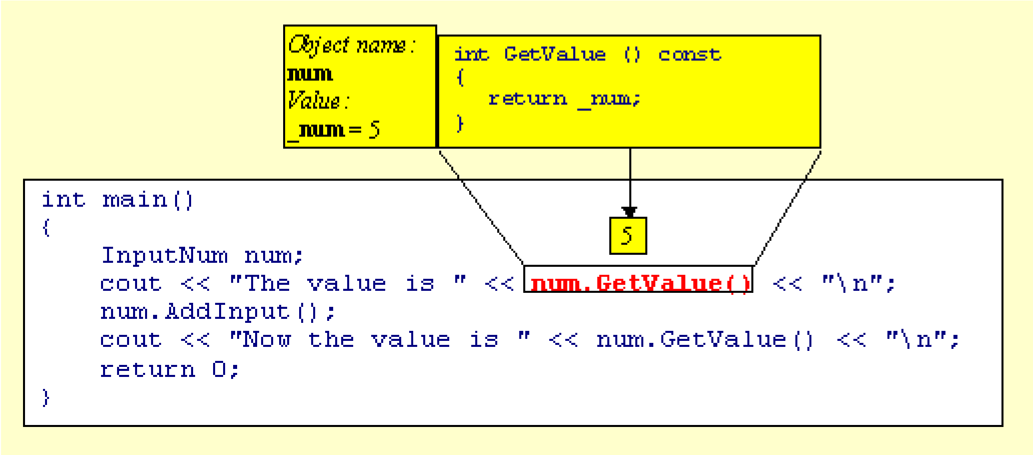
Look at the following series of pictures that visualize the execution of the program.



```
InputNum ()
{
    cout << "Enter number ";
    cin >> _num;
}
```

```
int main()
{
    InputNum num;
    cout << "The value is " << num.GetValue() << "\n";
    num.AddInput();
    cout << "Now the value is " << num.GetValue() << "\n";
    return 0;
}
```

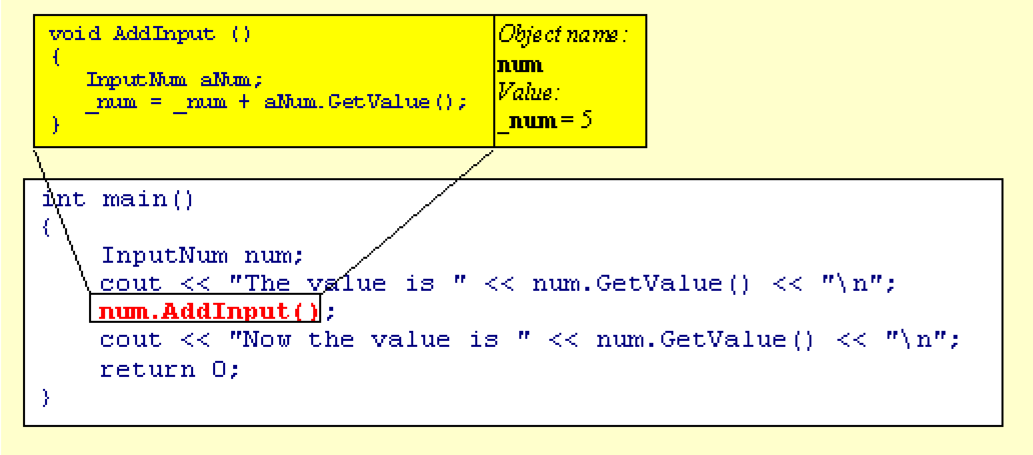
Figure 6 While executing main, the constructor of object num of class InputNum is called. It prompts the user to input a number. The user inputs 5. This value is stored in num's private data member _num.



```
int GetValue () const
{
   return _num;
}
```

```
int main()
{
    InputNum num;
    cout << "The value is " << num.GetValue() << "\n";
    num.AddInput();
    cout << "Now the value is " << num.GetValue() << "\n";
    return 0;
}
```

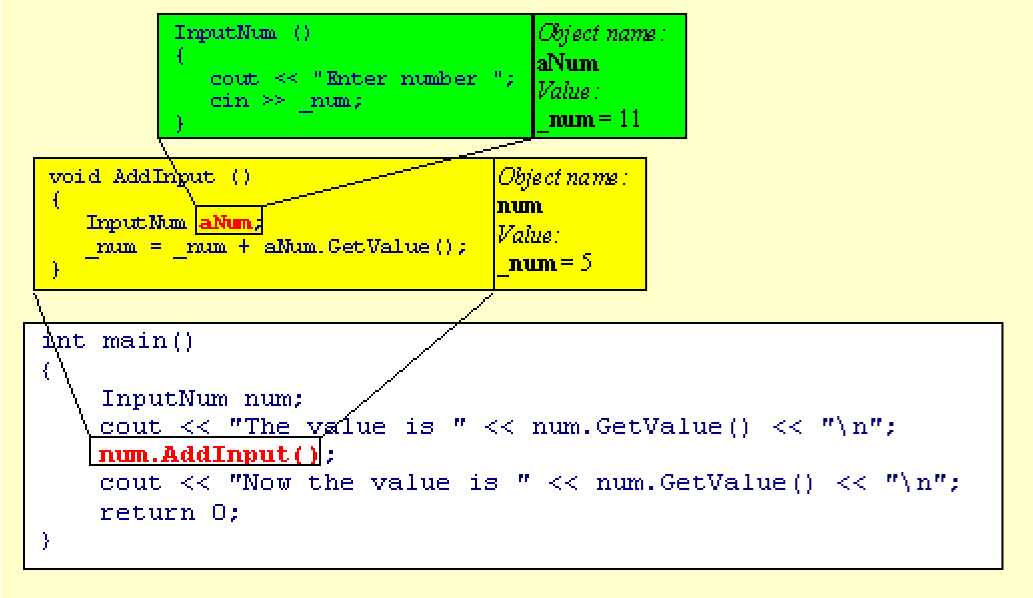
Figure 7 The GetValue method of the object num is called. It retrieves and returns the value stored in num's private data member _num. This value is 5.



```
void AddInput ()
{
   InputNum aNum;
   _num = _num + aNum.GetValue();
}
```

```
int main()
{
    InputNum num;
    cout << "The value is " << num.GetValue() << "\n";
    num.AddInput();
    cout << "Now the value is " << num.GetValue() << "\n";
    return 0;
}
```

Figure 8 The AddInput method of the object num is called next.

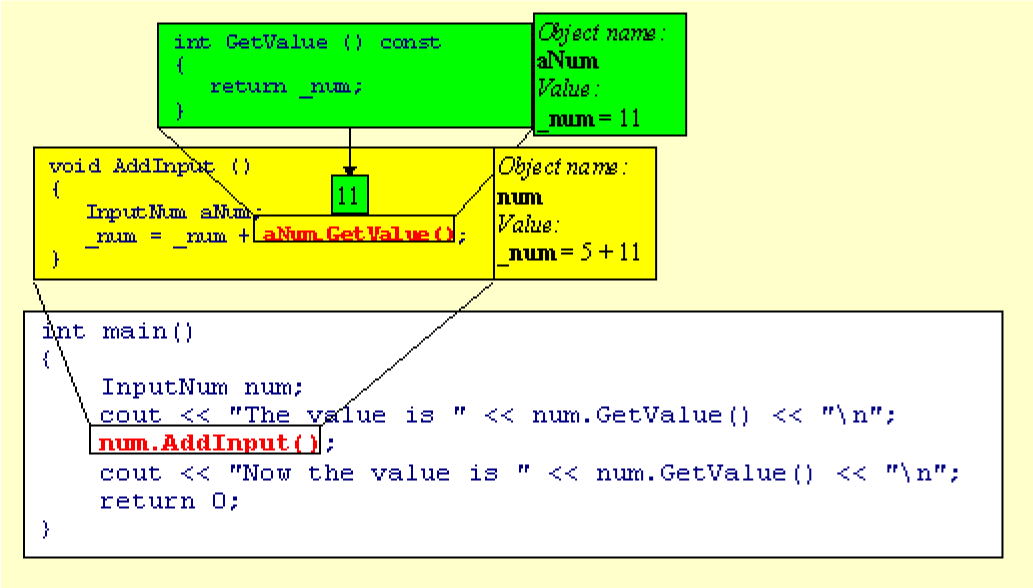


```
InputNum ()
{
   cout << "Enter number ";
   cin >> _num;
}
```

```
void AddInput ()
{
   InputNum aNum;
   _num = _num + aNum.GetValue();
}
```

```
int main()
{
    InputNum num;
    cout << "The value is " << num.GetValue() << "\n";
    num.AddInput();
    cout << "Now the value is " << num.GetValue() << "\n";
    return 0;
}
```

Figure 9 The AddInput method of num creates the object aNum of the class InputNumber. The constructor of aNum prompts the user to input a number. User enters 11 and this value is stored in aNum's private data member _num.

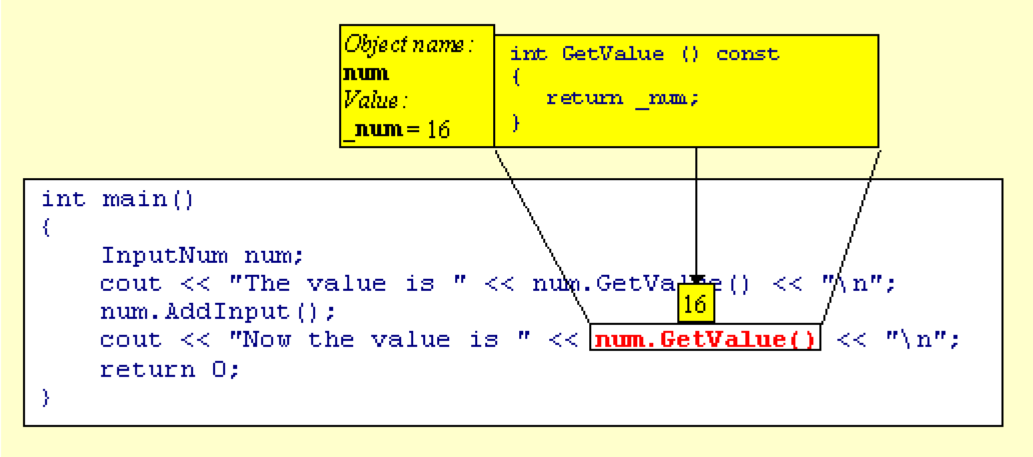


```
int GetValue () const
{
   return _num;
}
```

```
void AddInput ()
{
   InputNum aNum;
   _num = _num + aNum.GetValue();
}
```

```
int main()
{
    InputNum num;
    cout << "The value is " << num.GetValue() << "\n";
    num.AddInput();
    cout << "Now the value is " << num.GetValue() << "\n";
    return 0;
}
```

Figure 10 While still executing the AddInput method of num, the method GetValue of the object aNum is called. It retrieves and returns the value stored in aNum's private data member _num. This value, equal to 11, is then added to the num's private data member _num. It's value was 5, but after the addition of 11 it changes to 16.



```
int GetValue () const
{
   return _num;
}
```

```
int main()
{
    InputNum num;
    cout << "The value is " << num.GetValue() << "\n";
    num.AddInput();
    cout << "Now the value is " << num.GetValue() << "\n";
    return 0;
}
```

Figure 11 The method GetValue of the object num is called again. It retrieves and returns the value of num's private data member _num, which is now equal to 16.

You might wonder, how it is done that the program executes a method, then returns to the caller and continues exactly where it left off. Where does it keep the information about what it was doing before it made the call? The answer is, it's using the program (or the execution) stack. Calling a procedure means pushing some data on this (invisible) stack. Returning from the procedure means popping this data from the stack and returning to the previous state.

In particular, a calling sequence might look something like this. First, all the arguments to be passed to the procedure are pushed on the stack. Then the return address is pushed--that's the address of the next instruction to be executed after the return from the call. Then the execution jumps to the actual code of the procedure. The procedure starts by making room on the stack for its local variables. This way, each procedure invocation gets a fresh copy of all local variables. The procedure is executed, then it pops its local variables, the return address and the arguments passed to it, and jumps back to the return address. If there is a return value, it is usually passed in one of the processor's registers.

Take into account that this is only a conceptual model. The actual details depend on the particular *calling convention* and are both processor- and compiler-dependent.

At this point you might want to make a small detour into a short debugging tutorial.

In C++ you can do standard arithmetic operations on numbers and variables. Below is a list of basic arithmetic operators.

Arithmetic Binary Operators		
+	addition	a + b
-	subtraction	a - b
*	multiplication	a * b
/	division	a / b
%	division modulo	a % b
Arithmetic Unary Operators		
+	plus	+a
-	minus	-a
Assignment Operator		
=	assignment	a = b

Some of the operators require a word of explanation. For instance, why are there two division operators?

First of all, if you are operating on floating point numbers (for instance, of the type double) you should use the regular division operator /. Even if only one of the two numbers or variables is a decimal fraction, like 3.14, or a double, the result of the division will be a number of the type double. However, if both operands are integers--literal numbers without a decimal point or variables defined as int (or other integral types that we'll talk about soon)--applying operator / results in an integer equal to the *integral part* of the quotient. For instance, 3 / 2 will produce 1, which is the integral part of 1.5. Similarly, 10 / 3 will produce 3, etc.

The % operator, which can only be applied to integral types, yields the *reminder* from the division. For instance, 3 % 2 (pronounced "3 modulo 2") is equal to 1. Similarly, 11 % 3 yields 2, etc.

It is always true that

(a / b) * b + a % b is equal to a

However, the results of applying operators / and % when one of the numbers is negative are not well defined (although they still fulfill the equality above). So don't rely on them if you are dealing with numbers of either sign. If you think this sucks, you are right. The arguments for implementing such behavior in C++ are really lame: compatibility with C and efficient implementation in terms of processor instructions. I rest my case.

The unary plus does nothing. It is there for symmetry with the unary minus, which inverts the sign of a number. So if a = 2 then -a yields -2 and +a yields 2.

One thing is kind of awkward in the last program. The prompt "Enter number " is always the same. The user has no idea what's going on. Wouldn't it be nice to vary the prompt depending on the context? Ideally we would like to have code like this, in main:

```
InputNum num( "Enter number " );
num.AddInput( "Another one  " );
num.AddInput( "One more     " );
```

Some of the InputNum's methods would have to take strings as arguments. No problem. Here's how it's done:

```
#include <iostream>
using std::cout;
using std::cin;

class  InputNum
{
public:
    InputNum (char msg [])
    {
        cout << msg;
        cin >> _num;
    }

    int GetValue () const {  return _num; }

    void AddInput (char msg [])
    {
        InputNum aNum (msg);
        _num = GetValue () + aNum.GetValue ();
    }

private:
    int _num;
};

const char SumString [] = "The sum is   ";

int main()
{
    InputNum num ("Enter number ");
    num.AddInput ("Another one  ");
    num.AddInput ("One more     ");
    cout << SumString << num.GetValue () << "\n";
    return 0;
}
```

Several things are going on here. A new built-in type, char, has been introduced. As the name suggests, this type is used to store characters. Unless you work on some really weird machine, you can safely assume that `char`s are 8-bit quantities.

We also learn that a string is an *array* of characters--that's the meaning of brackets after the name of the string variable. What we don't see here is the fact that literal strings are zero terminated; that is, after the last explicit character, a *null* char is added by the compiler. For instance `"Hi"` is really an array of three characters, `'H'`, `'i'`, and 0 (or `'\0'` for purists). The null character is distinctly different from the character representing the digit zero. The latter is written as `'0'`. See the difference? Null character: `0` or `'\0'`, zero: `'0'`. Since we are at it, `"\n"` is an array of two characters `'\n'` and `'\0'`. This terminating null tells various functions operating on strings, including our friend cout, where the end of the string is. Otherwise it would have no clue.

By the way, instead of sending the string "\n" to the output, you can get the same result sending a special thing called `endl`. The syntax is, `cout << endl;` and it works like magic (meaning, I don't want to explain it right now). Since `endl` is defined in the standard library, you either have to prefix it with `std::` or declare, up front, your intent of `using std::endl`.

> There is a special `string` type defined in the standard library. It's extremely useful if you're planning on doing any string manipulations. We'll get to it later.

Another curious thing about this program is the way we call `GetValue()` inside `AddInput()`. We make two calls, one with no object specified, the other targeted at `aNum`. We know that the second one invokes `GetValue` on the object `aNum`. The first one, it turns out, invokes the `GetValue()` method on the object we are sitting in. The "this" object, as we say in the C++ argot. It simply retrieves its own value of `_num`. It does it, however, in a way that isolates us from the details of its implementation. Here it may look like an overkill of generality (why don't we just use `_num` directly?), but many situations it may mean a big win in future maintenance.

Some of you are probably asking yourselves the question: how expensive it is. Calling a member function instead of retrieving the value of `_num` directly. Or even, why not make `_num` public and shortcut our way to it everywhere. The syntax for accessing a public data member is:

```
myNum._num
```

You won't believe it, but the call is actually free. I'm not kidding, it costs zero additional cycles. Nada, nil, niente, zilch! (Later, I'll explain the meaning of *inline* functions).

To summarize, the body of a member function forms a separate local scope. This is also true about the body of a constructor and of a destructor. One can define objects within this local scope as well as create sub-scopes inside of it.

A string is a null terminated array of chars. Arrays are declared with rectangular brackets following their names. Arrays of chars can be initialized with explicit literal strings. Arrays may be defined within global or local scopes and passed as arguments to other functions.

Types

Built in types, typedefs.

Here is the list of built-in types with a short description of each of them.

- `bool`-Boolean type. Can take predefined values `true` and `false`.

- int -generic signed integral type. Its size is machine dependent, it is considered the most efficient implementation of an integral type for a given architecture. At least 2 bytes long.
- double - generic signed floating point type.
- char -generic 8-bit character type. Used in ASCII strings. Depending on the compiler, the default *signedness* of char may be signed or unsigned.
- short and long are a little better defined signed integral types. Usually short is 2-byte long and long is 4-byte long, but on a 64-bit machine it may be 8-bytes long (same as int). If you are writing your program to be run on only one type of processor architectures, you may make assumptions about the sizes of shorts and longs (and chars, of course). Sizes become crucial when you start storing data on removable disks or move them across the network. Short is often used as a cheap int, space-wise; for example, in large arrays. Long is often used when short or int may overflow. Remember, in two bytes you can store numbers between -32767 and 32767. Your arithmetic can easily overflow such storage, and the values will wrap around.
- float is a cheaper version of double (again, space-wise), but it offers less precision. It makes sense to use it in large arrays, where size really matters.
- Additional floating point type of even larger size and precision is called long double.
- All integral types may be prefixed by the keyword unsigned or signed (default for all types--except, as mentioned above, char--is signed). Of unsigned types, of special interest is unsigned char corresponding to a machine byte, and unsigned long often used for low-level bit-wise operations.

When you are tired of typing unsigned long over and over again, C++ lets you define a shortcut called a typedef. It is a way of giving a different name to any type--built-in or user defined. For instance, an unsigned long is often typedef'd to (given the name of) ULONG:

```
typedef unsigned long ULONG;
```

After you define such a typedef, you can use ULONG anywhere you would use any other type, for example,

```
ULONG ValueHolder::SwapValue (ULONG newValue)
{
    ULONG oldValue = _ulNumber;
    _ulNumber = newValue;
    return oldValue;
}
```

Similarly one often uses typedefs for BYTE

```
typedef unsigned char BYTE;
```

Like every feature of C++, typedefs can be abused. Just because you can redefine all types using your own names, doesn't mean you should. What do you make of types like INT or LONG when you see them in somebody else's code? Are these typedef'd names for int or long? If so, why not use directly int or long? And if not, then what the heck is going on? I can't think of any rational reason to typedef such basic data types, other than following some kind of coding standard from hell.

Besides these basic types there is an infinite variety of derived types and user defined types. We've had a glimpse of derived types in an array of chars. And we've seen user-defined types called classes.

Summary

Objects of any type (built-in, derived, or user defined) can be defined within the global scope or local scopes. Bodies of functions (such as `main()`) and member functions form local scopes. Within a given local scope one can create sub-scopes which can have sub-sub-scopes, and so on. A sub-scope is usually delimited by braces, but in certain constructs, such as the for loop, or the conditional if/else (we'll see examples later) a single-statement body may be braceless and yet it will form a sub-scope.

A class is a user-defined type. Class behavior is specified by the interface--the set of member functions (including constructors and a destructor) together with the description of their behavior. A member function takes zero or more arguments and returns a value (which can be empty--the void return type). The implementation of the class is specified by the bodies of member functions and the types of data members. The implementation fulfills the contract declared by the interface.

Objects can be combined by embedding and inheritance--semantically corresponding to the has-a and is-a relationships. Embedding and inheritance may be combined in various way leading to objects containing objects that inherit from other objects, that inherit from yet other objects, that contain..., etc.

Word of Caution

If you've come that far, you may congratulate yourself. You've learned the essence of C++. The rest is filling in the details. (Well, there is still polymorphism-there's no object oriented programming without it. And references. And operator overloading.) Can you imagine that there are people who claim to know C++, and even write programs in C++, without knowing all that? Here I'm telling you all about the importance of scopes, constructors and destructors, data hiding, interfaces, contracts, etc., and they will say: "What, constructors? I don't use them in C++. I declare all my variables at the top of the function (Member function? No, I mean global function) and then initialize them whenever they are needed." Next thing they'll tell you is that they can compile their C++ programs using a C compiler ("For backward compatibility, you know.").

It's just a warning. When you rush to your office first thing tomorrow morning, panting, eager to share your new fascination with C++ with your fellow programmers, that's what you may encounter. Not to mention these programmers who will say: "C++ is too slow for me. C gives me more control."

Exercises

Notice: If you have background in C and I haven't scared you off yet, these exercises will seem trivial to you. Please do them anyway. Get used to the new way of thinking and structuring your problems. All the exercises should be solved using only the constructs introduced so far. You may compare your solutions with my solutions.

1. Write a program using class World, whose output is "Hello from 1! Good bye from 1. Hello from 2! Good bye from 2." Find at least two different solutions.
2. Find a programmer's error in this class definition. Give two different ways of correcting it.

```
class Glove
{
public:
    Glove (int cFingers)
    : _n (cFingers), _hand (_n)
    {
        cout << "Glove with " << _n << " fingers\n";
    }
private:
    Hand  _hand;
    int   _n;
};
```

3. Define class HorBar whose constructor prints the following pattern

```
+----------+
```

4. where the number of minus signs is passed as an int argument to the constructor. Similarly class VerBar should print a vertical bar of height n, like this:

```
|
|
|
|
```

5. Define class Frame that contains (embedded) the upper horizontal bar, the vertical bar and the lower horizontal bar. Its constructor should take two arguments, the horizontal and the vertical extent of the frame

```
Frame (int hor, int ver)
```

6. The Frame object can be defined and initialized in the program using the following syntax:

```
Frame myFrame (10, 2);
```

7. Define class Ladder that contains two frames and a VerBar between them. The constructor should take two arguments, one for the horizontal extent and one for the vertical distance between horizontal bars. It should print the following pattern when created with (5, 2):

```
+-----+
|
|
+-----+
|
|
+-----+
|
|
+-----+
```

8. Write a program that asks for a number and calculates its factorial. Factorial of *n* is the product of all positive integers up (and including *n*. *n*! = 1 * 2 * 3 * ... * *n* . To multiply two numbers put an asterisk, *, between them.
9. Define class Planet that inherits from CelestialBody and is characterized by an albedo (how reflective its surface is--use a double to store it).
10. In the following class definition, replace dummy strings with actual words in such a way that during the construction of the object, the string "program makes objects with class" is printed

```
class One: public Two
{
public:
    One ()
    {
        Three three;
        cout << "dummy ";
    }
private:
    Four    _four;
};
```

Each of the classes Two, Three, and Four prints some dummy string in its constructor.

Design a similar class which, during construction, would print the above string, and during destruction print the same words in reverse order. The constraint is that each class has to have exactly the same code (print the same string) in its constructor and in its destructor.

11. Design, implement and test class Average. An object of type Average should accept values that are put into it. At any point in time it should be possible to retrieve their cumulative arithmetic average.
12. Design a class hierarchy (chain of inheritance) that derives a Human from a LivingBeing. Make sure you use the is-a relationship. Now add branches for a Tomato and an Elephant. Write a little test program that defines one of each inside main. Each class should have a constructor that prints a short but pointed message.
13. Design a Human exploring the has-a relationship. A human has a head, the head has a nose, the nose has a nose hair, etc. Derive a Man and a Woman. Implement simple classes whose constructors print messages and create a couple of Humans.

Abstract Data Types

Abstract data types, include files, non-inline member functions, assertions.

It is now time to design some serious classes that actually do something useful. Let's start with something simple and easy to understand--an abstract data type--stack of integers. A stack is defined by its interface. Major things that you can do to a stack is to push a number into it and pop a number from it.

What makes a stack a stack, is its LIFO (Last In-First Out) behavior. You can push values into it and, when you pop them, they come up in reverse order.

Our implementation choice is to use an array of integers. It would be nice to be able to separate the interface from the implementation. In some languages it is actually possible. Not so in C++ (or, at least, not without some gymnastics). The reasons are mostly technological. The compiler would have to have knowledge about all the files involved in the project in order to allow for such separation.

In C++, the best we can do is to separate the details of the implementation of *some member functions* into the implementation file. The interface file, however, must contain the definition of the class, and that involves specifying all the data members. Traditionally, interfaces are defined in header files with the .h extension. Here is the stack.h interface file.

```
const int maxStack = 16;

class IStack
{
public:
    IStack () :_top (0) {}
    void Push (int i);
    int  Pop ();
private:
    int _arr [maxStack];
    int _top;
};
```

The constructor of IStack initializes the top-of-the-stack index to zero--the start of the array. Yes, in C++ array elements are numbered from zero to *n*-1, where *n* is the size of the array. In our case the size is 16. It is defined to be so in the statement

```
const int maxStack = 16;
```

The modifier const tells the compiler that the value of maxStack will never change. The compiler is therefore free to substitute every use of the symbol maxStack with the actual value of 16. And that's exactly what it does.

The line

```
int _arr [maxStack];
```

declares _arr to be an array of maxStack integers. The size of an array has to be a constant. You don't have to specify the size when the array is explicitly initialized; as we have seen in the case of strings of characters.

Notice that member functions Push and Pop are *declared* but not *defined* within the definition of the class IStack. The difference between function declaration and function definition is that the latter includes the code--the implementation--(and is not followed by a semicolon). The former does not: It only specifies the types of parameters and the type of the return value. So where is the implementation of Push and Pop? In the separate implementation file stack.cpp.

First thing in that file is the inclusion of the header stack.h, which we have just seen. Next we include the new header file cassert. This file contains the definition of the very important function assert. I will not go into the details of its implementation, suffice it to say that this magical function can be turned off completely by defining the symbol NDEBUG. However, as long as we *don't* define NDEBUG, the assertion checks its argument for logical truth, that is, for a non-zero value. In other words, it asserts the truth of its argument. We define

NDEBUG only after the final program is thoroughly tested, and in one big swoop we get rid of all the assertions, thus improving the program's speed[2]. What happens when the argument of the assertion is not true (or is equal to zero)? In that unfortunate case the assertion will print a message specifying the name of the source file, the line number and the condition that failed. Assertions are a debugging tool. When an assertion fails it signifies a programmer's error--a bug in the program.

We usually don't anticipate bugs, they appear in unexpected places all by themselves. However there are some focal points in our code where they can be caught. These are the places where we make assumptions. It is okay to make certain assumptions, they lead to simpler and faster code. We should however make sure, at least during development and testing, that nobody violates these assumptions. Let's have a look at the implementations of Push and Pop:

```
#include "stack.h"
#include <cassert>
#include <iostream>
using std::cout;
using std::endl;

//compile with NDEBUG=1 to get rid of assertions

void IStack::Push (int i)
{
    assert (_top < maxStack);
    _arr [_top] = i;
    ++_top;
}

int IStack::Pop ()
{
    assert (_top > 0);
    --_top;
    return _arr [_top];
}
```

The first thing worth noticing is that, when the definition of a member function is taken out of the context of the class definition, its name has to be qualified with the class name. There's more to it than meets the eye, though. The methods we've been defining so far were all inline. Member functions whose definition is embedded inside the definition of the class are automatically made inline by the compiler. What does it mean? It means that the compiler, instead of generating a function call, will try to generate the actual code of the function right on the spot where it was invoked. For instance, since the method GetValue of the object Input was inline, it's invocation in

```
cout << input.GetValue ();
```

is, from the point of view of generated code, completely equivalent to

```
cout << input._num;
```

On the other hand, if the definition of the member function is taken out of the class definition, like in the case of Pop, it automatically becomes non-inline.

Should one use inline or non-inline methods? It depends on the complexity of the method. If it contains a single statement, it is usually cheaper execution-wise and code-size-wise to make it inline. If it's more complex, and is invoked from many different places, it makes more sense to make it non-inline.

In any case, inline functions are absolutely vital to programming in C++. Without inlining, it would be virtually impossible to convince anybody to use such methods as GetValue or SetValue instead of simply making _num public (even *with* inlining it is still difficult to convince some programmers). Imposing no penalty for data hiding is a tremendous strength of C++. Typing a few additional lines of code here and there, without even having to think much about it, is a price every experienced programmer is happy to pay for the convenience and power of data hiding.

The main function does a few pushes and pops to demonstrate the workings of the stack.

```
void main ()
{
    IStack stack;
    stack.Push (1);
    stack.Push (2);
    cout << "Popped " << stack.Pop() << endl;
    cout << "Popped " << stack.Pop() << endl;
}
```

Exercises

You may compare your solutions with my solutions.

1. Add method Top () that returns the value from the top of the stack without changing its state. Add method Count that returns the count of elements on the stack.
2. Modify the main function so that one of the stack contracts is violated. See what happens when you run the program with the assertions turned on.
3. Modify the stack so that it becomes CharStack--the stack of characters. Use it to invert strings by pushing all the characters of a string, and then popping and printing them one by one.
4. Design the abstract data type Queue of doubles with methods Put and Get and the FIFO (First-In-First-Out) behavior. Implement it using an array and two indices: the put index, and the get index that trails the put index. The contract of this particular queue reads: The Put method shall never be called more than maxPuts times during the lifetime of the Queue. Thou shalt not Get when the Queue is empty.
5. Design and implement the DblArray data type with the methods:

   ```
   void Set (int i, double val)
   ```

6. and

   ```
   double Get (int i) const
   ```

7. to set and get the values of the particular cells in the array. Add one more method

   ```
   bool IsSet (int i) const
   ```

8. that returns false if the cell has never been set before and true otherwise. The contract is that no Sets or Gets shall be done with the index greater than maxCells; that one shall never try to get a value that hasn't been set before,

[2] Believe it or not: Failure to turn off assertions and to turn on optimizations is often the reason for false claims of C++ slowness. (Others usually involve misuse of language features.)

and that one shouldn't try set a value that has already been set before (write once-read many times array). The array may store any values of the type `double`.

9. Hint: if you want to negate a Boolean resut, put an exclamation mark in front of it. For instance

```
!IsSet (i)
```

10. is `true` when the cell is *not* set and `false` when it is.

Arrays and References

References

References to objects, friends, passing by reference, initializing a reference.

So far we've been exploring the *has-a* and *is-a* types of relationships between objects. The third kind of relationship is ***the has-access-to*** relationship. The *has-access-to* relationship is not "destructive" like the others. Objects to which A has access are not destroyed when A is destroyed.

Of course, every object has access to all the global objects and to all objects within the encompassing scopes. However, in order to access these objects, the object in question (more precisely its class) has to have built-in knowledge of the actual *names* of these objects. For instance, class `World` knows about the existence of the globally accessible object `cout`. There's nothing wrong with having a few global objects whose names are known to lower level objects. They form the scenery of our program. They can be used to abstract some globally available services. However, the need for one object to access another object is so prevalent and dynamic that the above mechanism is not sufficient. The *has-access-to* relationship finds its expression in C++ in the form of references.

A reference refers to an object. It is like a new temporary name for an existing object--an alias. It can be used just like the original object; that is, the syntax for accessing a reference is the same as that for accessing an object. One can pass references, return references, even store them inside other objects. Whenever the contents of the object changes, all the references to this object show this change. Conversely, any reference can be used to modify the object it refers to (unless it's a `const` reference).

We declare a reference variable by following the type name with an ampersand (actually, the ampersand *binds* with the name of the variable):

```
Istack & stack;
```

One important thing about a reference is that, once it's created, and initialized to refer to a particular object, it will always refer to the same object. Assignment to a reference does not make it point to another object. It overwrites the object it refers to with the new value. For instance, consider this code

```
int i = 5;
int j = 10;
int & iref = i;  // iref points to i
iref = j;
```

The execution of the last statement will result in changing the value of `i` from 5 to 10. It will not make `iref` point to `j`. Nothing can make `iref` point to anything other than `i`.

A reference cannot be created without being initialized. If a reference is stored within an object, it must be initialized in the preamble to the constructor of that object.

We'll use references in the definition and implementation of the stack sequencer. A sequencer is an object that is used to present some other object as a sequence of items; it returns the values stored in that object one by one. In order to retrieve values from the stack, the sequencer needs access to private data members of the stack. The stack may grant such access by making the sequencer its friend. All the stack's private data members are still invisible to the rest of the world, except for its new friend, the stack sequencer. Notice that friendship is a one way relationship: StackSeq can access private data of IStack, but not the other way around.

> The combo IStack and StackSeq is an examples of a larger granularity structure, a pattern that unifies several data structures into one functional unit.

```
class IStack
{
    friend class StackSeq; // give it access to private members
public:
    IStack (): _top (0) {}
    void Push (int i);
    int  Pop ();
private:
    int _arr [maxStack];
    int _top;
};
```

```
class StackSeq
{
public:
    StackSeq (IStack const & stack);
    bool AtEnd () const;    // are we done yet?
    void Advance (); // move to next item
    int GetNum ()const;   // retrieve current item
private:
    IStack const & _stack;  // reference to stack
    int     _iCur;   // current index into stack
};
```

The constructor of StackSeq is called with a reference to a const IStack. One of the data members is also a reference to a const IStack. A const reference cannot be used to change the object it refers to. For instance, since Push is not a const method, the compiler will not allow StackSeq to call _stack.Push(). It will also disallow any writes into the array, or any change to _stack._top. A reference to const can be initialized by any reference, but the converse is not true--a reference that is not const cannot be initialized by a reference to const. Hence we can pass any stack to the constructor of StackSeq. The compiler, in accordance with the declaration of the constructor, converts it to a reference to a const IStack. Then _stack, which is also a reference to a const IStack, may be initialized in the preamble to the constructor of the StackSeq (remember, references *must* be initialized in the preamble).

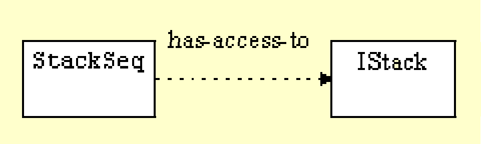


Figure 1 The graphical representation of the *has-access-to* relationship between objects.

Data member _iCur stores the state of the sequencer--an index into the array _arr inside the referenced stack. The sequencer being a friend of the class IStack has full knowledge of the stack's implementation.

```
StackSeq::StackSeq (IStack const & stack )
    : _iCur (0), _stack (stack) // init reference
{}
```

The implementation of the remaining methods is rather straightforward

```
bool StackSeq::AtEnd () const
{
    return _iCur == _stack._top;  // friend: can access _top
}

void StackSeq::Advance ()
{
    assert (!AtEnd()); // not at end
    ++_iCur;
}

int StackSeq::GetNum () const
{
    return _stack._arr [_iCur]; // friend: can access _arr
}
```

Notice that the dot syntax for accessing object's data members through a reference is the same as that of accessing the object directly. The variable _stack is a (read-only) alias for a stack object. Due to the sequencer's *friend* status, StackSeq is accessing IStack's private data members with impunity.

New logical operators have been introduced in this program as well. The exclamation point logically negates the expression that follows it. !AtEnd() is true when the sequencer is not done and false when it's done. You read it as "not at end."

The double equal sign '==' is used for equality testing. The result is false if two quantities are different and true if they are equal.

Warning: The similarity between the assignment operator = and the equality operator == is a common source of serious programming mistakes.

It is very unfortunate that, because of the heritage of C, a single typo like this may cause an almost undetectable programming error. Don't count on the compiler to flag the statement

```
x == 1;
```

or the conditional

```
if (x = 0)
```

as errors (some compilers will warn, but only at a high warning level). Some people try to prevent such errors by getting into the habit of reversing the order of equality testing against a constant. For instance

```
if (0 == x)
```

is safer than

```
if (x == 0)
```

because the omission of one of the equal signs will immediately cause a compiler error. I used to be one of the advocates of this style, but I changed my mind. I don't like a style that decreases the readability of code.

The *not-equal-to* relation is expressed using the operator !=, as in x != 0.

The main procedure tests the functionality of the sequencer. It pushes a few numbers onto the stack and then iterates through it. Notice how the object TheStack is passed to the constructor of the iterator. The compiler knows that what we really mean is a reference, because it has already seen the declaration of this constructor in the class definition of IStack. Since the constructor was declared to take a reference, the reference will automatically be passed.

When a formal argument of a method is specified as a reference, we say that this argument is passed by reference. When you call a method like that, you pass it a reference to a variable, and the method may change the value of that variable. This is in contrast to passing an argument by value. When you call a method that expects to get a variable by value, the method will not change the value of this variable. It will operate on a copy of its value.

Compare these two methods.

```
class Dual
{
    void ByValue (int j)
    {
        ++j;
        cout << j << endl;
    }

    void ByRef (int & j)
    {
        ++j;
        cout << j << endl;
    }
};

void main ()
{
    Dual dual;
    int i = 1;
    dual.ByValue (i);
    cout << "After calling ByValue, i = " << i << endl;
    dual.ByRef (i);
    cout << "After calling ByRef, i = " << i << endl;
}
```

The first method does not change the value of i, the second does.

If you are worried about the fact that from the caller's point of view it is not obvious whether a reference was meant--don't! As a general rule, assume that objects are passed by reference, built-in types by value (the example above notwithstanding).

Only in some special cases will we pass objects by value, and it will be pretty obvious why (see: value classes). So, when reading most programs, you can safely assume that, when an object is passed to a function, it is passed by reference.

```
void main ()
{
    IStack TheStack;
    TheStack.Push (1);
    TheStack.Push (2);
    TheStack.Push (3);

    for ( StackSeq seq (TheStack);
        !seq.AtEnd();
        seq.Advance() )
    {
        cout << "    " << seq.GetNum() << endl;
    }
}
```

Let us follow the path of the reference. A reference to TheStack is passed to the constructor of StackSeq under the (read-only) alias of stack. This is a new temporary name for the object TheStack. This reference cannot be used to modify TheStack in any way, since it is declared const. Next, we initialize the reference data member _stack of the object seq. Again, this reference cannot be used to modify TheStack. From the time the object seq is constructed until the time it is destroyed, _stack will be a read-only alias for TheStack. The sequencer will know TheStack under the read-only alias _stack.

Notice the creative use of the for loop. We are taking advantage of the fact that the for-loop header contains a *statement*, an *expression*, and a *statement*. The first statement--loop ***initialization***--is executed only once before entering the loop. In our case it contains the definition of the sequencer. The following expression--the loop ***condition***--is tested every time the loop is (re-) entered. If the expression is true (non-zero) the iteration starts (or continues). When it is false, the loop is exited. In our case the condition is that the sequencer is not at the end. The third statement--the loop ***increment***--is executed after every iteration. In our example it advances the sequencer. Notice that after a normal exit from the loop, the loop condition always ends up false.

Any of the three parts of the for-loop header may be empty. An infinite loop, for instance, is usually implemented like this:

```
for (;;) // do forever
{
    // There must be some way out of here!
```

```
 }
```

(Of course you can exit such loop using a `break` or a `return` statement.)

Here's a list of some of the operators that create Boolean values.

Equality Testing Operators	
==	equal to
!=	not equal to
Relational Operators	
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
Logical Negation Operator	
!	not

Stack-based calculator

Top down design, make, private member functions, if statement, do/while loop.

Our next step will be to design and implement a small but useful program using the techniques that we learned so far. It will be a simple stack based calculator. Even though it is a very small project, we will follow all the steps of the design and implementation process. We'll start with the functional specification, then proceed with the architectural design, and finally implement the program.

Functional Specification

The stack-based calculator has a simple interface. It accepts user input in the form of numbers and operators. Numbers are stored in the LIFO type memory--they are pushed on the stack. Binary operators are applied to the two top level numbers popped from the stack, and the result of the operation is pushed on top of the stack. If there's only one number in the stack, it is substituted for both operands. After every action the contents of the whole stack is displayed.

Since effectively the calculator implements reverse Polish notation, there is no need for parentheses. For simplicity we will implement only four basic arithmetic operations. A sample session may look like this (user input follows the prompt '>'):

```
 > 3
       3
 > 2
       3
       2
 > +
       5
 > +
    10
```

Design

The obvious top-level object is the *calculator* itself. It stores numbers in its memory and performs operations on them. Since we would like to be able to display the contents of the calculator's stack after every action, we need access to a stack *sequencer*. Finally, in order to completely decouple input/output operations from the calculator, we should have an *input* object that obtains input from the user and does some pre-processing (tokenization) to distinguish between numbers and operators.

We'll use the standard cout object for output.

The minimum lifetimes of these three objects are the following:

- The calculator has to be alive during the whole session since it has long term memory.
- The scope of the input object is associated with a single input operation.
- The scope of the sequencer is associated with every stack display operation that follows a successful calculator action.

The input object obtains a string from the standard input. It can distinguish between numbers and operators, returning different tokens depending on the input. If the token is 'number' the string is converted to an int and the input object remembers its value.

The calculator object accepts the input object, from which it can get pre-digested data, and executes the command. If the data is a number, it stores it in a stack; if it is an operator, it performs the operation. The results can be viewed by creating a stack sequencer--the calculator may simply give access to its stack for the purpose of iteration.

Notice that we have only three types of objects interacting at the top level, plus one object, the stack, that we can treat as a black box (we don't call any of its methods, we just pass access to it from one component to another). This is not just a side effect of the simplicity of our project--we should always strive to have only a small number of top level objects.

Once the top level is established, we can start our top-down descent. In our design we may go one level deeper into the *calculator* object. We already know that it has a stack. The stack object will thus be embedded in it. We will re-use the stack that we used in the previous paragraph.

Stubbed Implementation

The top-down design will be followed by the top-down implementation. Based on our architectural specification we start to write the main procedure.

```
void main ()
{
    Calculator TheCalculator;
    bool status;
    do
    {
        // Prompt for input
        cout << "> ";
        Input input;
        status = TheCalculator.Execute (input);
        if (status)
        {
            for (StackSeq seq (TheCalculator.GetStack ());
                !seq.AtEnd ();
                seq.Advance () )
            {
                cout << "    " << seq.GetNum () << endl;
            }
        }
    } while (status);
}
```

We have introduced some new constructs here, the `do`/`while` loop and the `if` statement. The execution of the body of the *do*/*while* loop is repeated as long as the condition in the *while* clause remains true. Notice that, unlike in the case of the *for* loop, the body of the *do*/*while* loop is always executed at least once. As should be obvious by now, the body of the loop forms a separate local scope (even if it is a single statement and the braces are omitted).

The body of the *if* statement is entered only if the condition in the *if* clause is true (different from zero). Otherwise it is skipped altogether. And again, the body of the *if* statement forms a local scope even if it is only one statement, in which case the braces may be omitted.

Notice also that the variable `status` is defined without being initialized. We try to avoid such situations in C++. Here I took the liberty of not initializing it, since it is always initialized inside the body of the *do*/*while* loop (and we know that it is executed at least once). I couldn't define the variable `status` inside the scope of the loop, because it is tested in the *while* clause which belongs to the outer scope. The *while* clause is evaluated during each iteration, after the body of the loop is executed.

Next, following the top-down approach, we'll write stub implementations for all classes. The stack is passed from the calculator to the sequencer and we don't need to know anything about it (other than that it's an object of class `IStack`). Hence the trivial implementation:

```
class IStack {};
```

The sequencer has all the methods stubbed out. I have added to it a dummy variable, `_done`, to simulate the finiteness of the stack. `GetNum` returns the arbitrarily chosen number 13.

```
class StackSeq
{
public:
    StackSeq (IStack const & stack) : _stack (stack), _done (false)
    {
        cout << "Stack sequencer created\n";
    }
    bool AtEnd () const { return _done; }
    void Advance () { _done = true; }
    int GetNum () const { return 13; }
private:
    IStack const &  _stack;
    bool            _done;
};
```

At this level of detail, the class Input exposes only its constructor:

```
class Input
{
public:
    Input ()
    {
        cout << "Input created\n";
    }
};
```

The calculator, again, has a dummy variable whose purpose is to break out of the loop in main after just one iteration.

```
class Calculator
{
public:
    Calculator () : _done (false)
    {
        cout << "Calculator created\n";
    }
    bool Execute (Input& input)
    {
        cout << "Calculator::Execute\n";
        return !_done;
    }
    IStack const & GetStack () const
    {
        _done = true;
        return _stack;
    }
private:
    IStack  _stack;
    bool    _done;
};
```

The method GetStack returns a const reference to IStack. In other words it makes a read-only alias for the calculator's private object _stack and makes it available to the caller. The user may use this alias to access _stack, but only through its const methods, or, if it is an IStack's friend, by reading the values of _top and those stored in the array _arr. This is exactly what the sequencer needs. Notice also that the statement return _stack is interpreted by the compiler to return a *reference* to _stack. This is because GetStack was declared as returning a reference. If GetStack were declared as

```
IStack const GetStack () const;
```

the compiler would return a read-only *copy* of the stack. Copying the stack is somehow more expensive than providing a reference to it. We'll come back to this problem later, when we talk about value classes.

With all the dummies in place, we can compile and execute the test program. Its output shows that everything works as expected.

```
Calculator created
> Input created
```

```
Calculator::Execute
Stack sequencer created
    13
> Input created
Calculator::Execute
```

Implementation

Calculator: Implementation

Now is the time to start the top to bottom descent. The first candidate for implementation is the Calculator itself. When implementing the calculator we'll find out what we need from the Input object. Let's start with the Execute method. First, it should retrieve the token from the input. We may expect the following tokens: the number token, any of the arithmetic operator tokens, or the *error* token. For each type we do a different thing. For the *number* token we retrieve the value of the number from the input and push it on the stack. For the *operator*, we pop two numbers (or one if there aren't two), pass them to the Calculate method and push the result.

{
source

```
bool Calculator::Execute (Input const & input)
{
    int token = input.Token ();
    bool status = false; // assume failure

    if (token == tokError)
    {
        cout << "Unknown token\n";
    }
    else if (token == tokNumber)
    {
        if (_stack.IsFull ())
        {
            cout << "Stack is full\n";
        }
        else
        {
            _stack.Push (input.Number ());
            status = true; // success
        }
    }
    else
    {
        assert (token == '+' || token == '-'
              || token == '*' || token == '/');

        if (_stack.IsEmpty ())
        {
            cout << "Stack is empty\n";
        }
        else
        {
            int num2 = _stack.Pop ();
            int num1;

            // Special case, when only one number on the stack:
            // use this number for both operands.

            if (_stack.IsEmpty ())
                num1 = num2;
            else
                num1 = _stack.Pop ();

            _stack.Push (Calculate (num1, num2, token));
            status = true;
        }
    }
    return status;
}
```

We have introduced the extension of the conditional--the else clause. The body of the *else* statement is executed only if the preceding *if* statement hasn't been executed (because the condition was false).

If/*else* statement can be staggered like this:

```
if (A)
    ...
else if (B)
    ...
else if (C)
    ...
else
    ...
```

Calculate is the new method of Calculator. Since it is used only inside the implementation of Calculator, it is made private. It takes two numbers and a token and it returns the result of the operation. I have separated this code into a method mostly because the Execute method was getting too large. Calculate has well defined functionality, quite independent from the rest of the code. It is implemented as one large staggered *if*/*else* construct.

```
int Calculator::Calculate (int num1, int num2, int token) const
{
    int result;

    if (token == '+')
        result = num1 + num2;
    else if (token == '-')
        result = num1 - num2;
    else if (token == '*')
        result = num1 * num2;
    else if (token == '/')
    {
        if (num2 == 0)
        {
            cout << "Division by zero\n";
            result = 0;
        }
        else
            result = num1 / num2;
    }
    return result;
}
```

Notice the use of character literals such as

```
'+', '-', '*', '/'.
```

Unlike string literals, character literals are surrounded by *single* quotes. Instead of assigning special values to *operator* tokens, we just used their character literal (ASCII) values.

Let's have a look at the modified class definition of the calculator.

```
class Calculator
{
public:
    bool Execute ( Input& input );
    // give access to the stack
    IStack const & GetStack () const { return _stack; }
private:
    int Calculate (int n1, int n2, int token) const;

    IStack  _stack;
};
```

We may now add our old implementation of IStack and StackSeq, extend the dummy Input by adding dummy implementations of the methods Token and Number, recompile and test.

Input: Implementation

Now it's the time to implement the Input class. It's also time to split our project into separate files. We should have three header files--calc.h, stack.h and input.h--as well as three implementation files--calc.cpp, stack.cpp and input.cpp. To build an executable program out of multiple source files one needs the help of a *linker*. Fortunately, in an integrated environment you can create a project to which you add all the implementation files and let the environment *build* the program for you. We don't want to get too involved in such details here.

The more header files we have, the more likely it is that we will include the same file twice. How can this happen? Suppose that we include input.h inside calc.h. Then we include both calc.h and input.h inside calc.cpp, and we're in trouble. The compiler will see two definitions of class Input (they're identical, but so what). The way to protect ourselves from such situations is to guard the header files with conditional compilation directives. Let's enclose the whole body of input.h with the pair

```
#if !defined input_h
...
#endif
```

If the symbol `input_h` is not defined, the compiler will process the code between these two *preprocessor directives* (as they are called). In the beginning, this symbol is not defined by anybody. So the first time the compiler encounters the `#include "input.h"` directive, it will go ahead processing `input.h`. However, the first directive inside the *if*/*endif* pair defines the symbol `input_h`

```
#define input_h
```

Next time the `#include "input.h"` directive is encountered, the compiler will skip everything between `#if !defined input_h` and `#endif` since `input_h` has already been defined.

```
#if !defined input_h   // prevent multiple inclusions
#define input_h

const int maxBuf = 100;

// Tokens are tokNumber, tokError, +, -, *, /.

const int tokNumber = 1;
const int tokError  = 2;

// Gets input from stdin, converts to token.

class Input
{
public:
        Input ();
        int Token () const { return _token; }
        int Number () const;
private:
        int  _token;
        char _buf [maxBuf];
};

#endif // input_h
```

Notice that the methods `Token` and `Number` are declared `const`. The `Calculator` who has read-only access to input (through a *const* reference) will still be able to call these methods. The buffer `_buf` is where the string obtained from the user will be stored.

The implementation file `input.cpp` includes two new standard headers.

```
#include <cctype>
#include <cstdlib>
```

The file `cctype` contains the definitions of (very efficient) macros to recognize character type. The `isdigit` macro, for instance, returns true if the character is one of the digits between '0' and '9' and false otherwise. The other file, `cstdlib`, is needed for the declaration of the function `atoi` that converts an ASCII string representing a decimal number to an integer. You can find out more about functions like that by studying the standard library. You can either use online help or read the compiler manuals (or some other books).

```
Input::Input ()
{
    cin >> _buf;

    // first char of input is usually enough to decide
    // what token it is

    int c = _buf [0];

    if (isdigit (c))
        _token = tokNumber;
    else if (c == '+' || c == '*' || c == '/')
        _token = c;
    else if (c == '-') // allow entering negative numbers
    {
        if (isdigit (_buf [1])) // peek at next char
            _token = tokNumber;
        else
            _token = c;
    }
    else
        _token = tokError;
}
```

The constructor of Input reads a line of text from the standard input into a character buffer. This is yet another amazing trick performed by our friend cin. (By the way, for the time being we are not even thinking about what could happen if the buffer overflows. We are still at the level of *weekend programming*.)

Depending on the first character in the buffer, we decide what token to make out of it. Notice the special treatment of the minus sign. It could be a binary minus, or it could be a unary minus in front of a number. To find out which one it is, we peek at the next character in the buffer. Notice the use of one of the character classification macros I mentioned before. isdigit returns true when the character is a digit (that is one of '0', '1', '2'... '9'). The character classification macros are implemented using a very efficient *table lookup* method. It is more efficient to call isdigit than execute code like this

```
if ( c >= '0' && c <= '9' )
```

The header cctype contains other useful macros, besides isdigit, like isspace, islower and isupper.

If the token is tokNumber, the Calculator needs to know its value. The method Number converts the string in the buffer into an int value using the library function atoi declared in cstdlib.

```
int Input::Number () const
{
    assert (_token == tokNumber);
    return atoi (_buf);   // convert string to integer
}
```

The Makefile

One final word for those of us who, for one reason or another, don't use an integrated environment to develop their code. When the project grows beyond a single file, it is time to start using the *make* utility. Below is a very simplified *makefile* that establishes the dependencies between various files in the project and provides the recipes to generate them:

```
calc.exe : calc.obj stack.obj input.obj
    cl calc.obj stack.obj input.obj

calc.obj : calc.cpp calc.h input.h stack.h
    cl -c calc.cpp

stack.obj : stack.cpp stack.h
    cl -c stack.cpp

input.obj : input.cpp input.h
    cl -c input.cpp
```

If the *makefile* is called `calc.mak`, you just call `make calc.mak` (or `nmake calc.mak`) and all the necessary compilation and linking steps will be executed for you to create the executable `calc.exe`. Again, use online help or read manuals to find out more about *make* (or *nmake*).

Exercises

Compare with solutions.

In C++ one can perform bitwise operations on integral numbers. Binary bitwise operators AND and OR are denoted by ampersand & and vertical bar | respectively. The usage is

```
int i = 3;  // binary 0011
int j = 5;  // binary 0101
int k = i & j; // binary 0001
k = i | j;  // binary 0111
k = k | 8;  // binary 1111
```

Bitwise complement is denoted by a tilde ~. For instance (on a four-bit computer), if *k* = 13 (binary 1101) than its complement *j* = ~*k* would be equal to 2 (binary 0010) (ones become zeros and zeros become ones).

<table>
<tr><th colspan="2">Binary Bitwise Operators</th></tr>
<tr><td>&</td><td>bitwise and</td></tr>
<tr><td>|</td><td>bitwise (inclusive) or</td></tr>
<tr><td>^</td><td>bitwise exclusive or (xor)</td></tr>
<tr><td>>></td><td>right shift by n bits</td></tr>
<tr><td><<</td><td>left shift by n bits</td></tr>
<tr><th colspan="2">Unary Bitwise Operator</th></tr>
<tr><td>~</td><td>one's complement</td></tr>
</table>

1. Add bitwise operations to our calculator.
2. Add a new command, 'x', that toggles between decimal and hexadecimal outputs.

To output a number in the hexadecimal (base sixteen) one sends a modifier "hex" to the output.

```
using std::hex;
int i = 11;
cout << hex << i;   // i will be printed in hex as 'b'.
```

3. Extend the calculator by adding *symbolic variables* denoted by lowercase letters. Add the assignment operation that assigns a value to a symbolic variable. For instance, one should be able to do the following

```
> a
> 5
> =     // a = 5
> 2
> a
> *    // 2 * 5
```

4. Consider what should the `Calculator` do with a symbol. For instance, what should it push on the stack?

Pointers

Pointers

A pointer variable stores the memory address of an object (for instance, of a variable, an array, a function, an instance of a class, etc.). Using a pointer one can access the contents of the object it points to. In this sense a pointer acts just like a reference, it is an alias of some other object. There is, however, one big difference between a pointer and a reference. A pointer is not permanently attached to an object-- it can be moved. The programmer can change the pointer to point to another object. This is a very powerful feature and, at the same time, a very dangerous one. The majority of bugs involve, in one way or another, the use of pointers. In C++ (and C) we have the following types of pointers:

- Uninitialized pointers,
- Pointers to deallocated memory,
- Pointers beyond the end of an array,
- Pointers that think they point to something else,
- Cross your fingers, knock on wood kind of pointers.

Obviously we don't want to use such pointers. We want to use pointers that at all times point to what they think (and we think, and the next person thinks) they point to. And that requires a lot of discipline on the programmer's part. Therefore the first rule of pointer use is:

Don't use pointers unless there is no other way.

Use references instead, whenever possible. You'll definitely avoid the problem of uninitialized pointers. The use of references gives you also the opportunity to fine tune the scope of objects and variables. References, unlike pointers, have to be initialized at the point of definition. If they are inside an object, you'll have to define the object in the right spot. It must be created within the scope where the objects to be accessed by these references are available. That imposes a certain natural order of creation.

Another misuse of pointers, a heritage of C, is the use of pointers in place of array indexes. The old argument was that scanning an array using a pointer results in faster code. Not so with modern optimizing compilers. It turns out that a compiler is more likely to do some aggressive (but correct!) optimizations when the algorithm is expressed using indexes rather than pointers. So here goes another myth from the dark ages of computing. We'll discuss this issue in more detail soon.

Finally, many C-programmers are used to combining pointers with error codes, that is using a null pointer as a sign of error. Although there are ways of faking "null references," they won't be discussed in this book for obvious reasons (they are ugly hacks!). The solution is either to separate error codes from references, or to use exceptions (see the chapter on resource management) to signal abnormal conditions. Of course, there is nothing wrong with functions that return pointers *by design* and use the null-pointer trick to signal special conditions.

There is an opposing school of thought about dealing with pointers—fighting fire with fire. Some object oriented languages, notably Java, use pointers as the only way of accessing objects. The trick is—they don't allow C-style manipulation of pointers. In Java, pointers are treated like immutable handles. Moreover, such languages like Java, Smalltalk or Eiffel implement garbage collection. The programmer is freed from the chore of keeping track of pointers and objects they point to, because the runtime system takes care of it. It reference-counts every object and automatically destroys it when it's not referenced by any pointer. Such service doesn't come for free, though. There is a runtime cost associated with it. Moreover, not having direct control over when the destructors are executed makes the methods of resource management impossible to implement in such systems. But I am getting ahead of myself. We'll discuss these topics later in more detail.

Pointers vs. References

A pointer variable is declared with an asterisk between the type name and the variable name (the asterisk binds with the variable name). For instance,

```
int *pValue;
```

declares pValue to be a pointer to an integer. The pointer can be initialized with (or assigned to, using the assignment operator =) an address of some other variable. The *address of* operator is denoted by the ampersand (there is no conflict between this ampersand and the reference ampersand--they appear in different contexts). For instance,

```
int TheValue = 10;
pValue = &TheValue;
```

assigns the address of the variable TheValue to the pointer pValue.

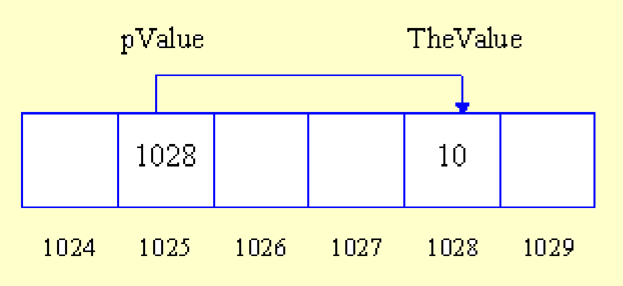


Figure 1. The pointer pValue stores the address of—points to—the variable TheValue.

If we want to access the value to which the pointer points, we use the *dereference* operator, the asterisk (again, its double meaning doesn't lead to confusion). For instance,

```
int i = *pValue;
```

assigns the value that pValue points to, to the integer i. In our example i will change its value to 10. Conversely,

```
*pValue = 20;
```

will change the value that pValue points to, that is, the value of TheValue. (The value of TheValue will now be 20). As you can see, this is a little bit more complicated than the use of references. By the way, the same example with references would look like this:

```
int TheValue = 10;
int& aliasValue = TheValue; // aliasValue refers to TheValue
int i = aliasValue; // access TheValue through an alias
aliasValue = 20;  // change TheValue through an alias
```

References definitely win the simplicity competition.

Here's another silly example that shows how one could, but shouldn't, use pointers. The only new piece of syntax here is the ***member access*** operator denoted by an arrow (actually a combination of a minus sign with a greater than sign) ->. It combines dereferencing a pointer and accessing a member (calling a member function).

```
IStack TheStack;
IStack* pStack = &TheStack;
pStack->Push (7);  // push 7 on TheStack using a pointer
// equivalent to (*pStack).Push (7)
```

The same can be done using a reference:

```
IStack TheStack;
IStack& aliasStack = TheStack;
aliasStack.Push (7);  // push 7 on TheStack using an alias
```

Once more, references rule.

Pointers and Arrays

Arrays are passed around as pointers. In fact, a C++ array is implemented as a pointer to the first element of the array. In practical terms it means that we can access a pointer as if it were an array--that is using an index. Conversely, we can use a pointer to access elements of an array. We can increment a pointer to move from one element of the array to the next. Of course, in both cases it is our responsibility to make sure that the pointer actually points to an element of an array.

A string is a good example of an array. There is a function in the standard library called strlen that calculates the length of a null terminated string. Let's write our own implementation of this function, which we will call StrLen

```
int StrLen (char const str [] )
```

```
{
    for (int i = 0; str [i] != '\0'; ++i)
        continue;
    return i;
}
```

The ***continue*** keyword is used here instead of an empty body of the loop. It's less error prone this way.

Here's the main procedure that passes an array to StrLen:

```
int main ()
{
    char aString [] = "the long string";
    int len = StrLen (aString);
    cout << "The length of " << aString << " is " << len;
}
```

We are scanning the string for a terminating null and returning the index of this null. Pretty obvious, isn't it?

Here's a more traditional "optimized" version:

```
int StrLen (char const * pStr)
{
    char const * p = pStr;
    while (*p++);
    return p - pStr - 1;
}
```

We initialize p to point to the beginning of the string. The while loop is a little cryptic. We dereference the pointer, test it for Boolean truth and ***post-increment*** it, all in one statement. If the character obtained by dereferencing the pointer is different from zero (zero being equivalent to Boolean false) we will continue looping. The post-increment operator moves the pointer to the next position in the array, but only after it has been used in the expression (yielding true of false).

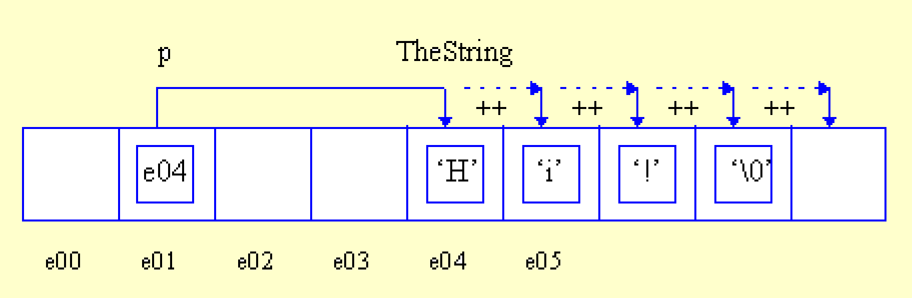


Figure 2. The pointer p initially points at the first character of the string "Hi!" at address 0xe04 (I use hexadecimal notation for addresses). Subsequent increments move it through the characters of the string until the null character is reached and processed.

By the way, there is also a ***pre-increment*** operator that is written in front of a variable, ++p. It increments the variable *before* its value is used in the expression.

Increment Operators (acting on i)	
i++	post-increment

i++	post-increment
++i	pre-increment
Decrement Operators (acting on i)	
i--	post-decrement
--i	pre-decrement

In the spirit of terseness, I haven't bothered using the continue statement in the empty body of the loop—the semicolon (denoting the empty statement) is considered sufficient for the old-school C programmers.

Finally, we subtract the two pointers to get the number of array elements that we have gone through; and add one, since we have overshot the null character (the last test accesses the null character, and then it increments the pointer anyway). By the way, I never get this part right the first time around. I had to pay the penalty of an additional edit-compile-run cycle. If you have problems understanding this second implementation, you're lucky. I won't have to convince you any further not to write code like this.

The question is: Will you prefer to write the simpler and more readable index implementation of procedures like StrLen after seeing both versions? If you have answered *yes*, you may go directly to the following paragraph. If you want more gory details, click here and see some assembly code.

Soapbox

The art of programming is in a very peculiar situation. It is developing so fast, that people who started programming when C was in its infancy are still very active in the field. In other sciences a lot of progress was made through natural attrition. The computer revolution happened well within one generation. Granted, a lot of programmers made enough money to be able to afford doing volunteer work for the rest of their lives. Still, many others are carrying around their old (although only a few years old) bag of tricks that they learned when they were programming XT's with 64k memory.

New programmers learn programming from the classics like Kernighan and Ritchie's "The C programming language." It's a great book, don't get me wrong, but it teaches the programming style of the times long gone.

The highest authority in algorithms and data structures is Donald Knuth's great classic "The Art of Programming." It's a beautiful and very thorough series of scientific books. However, I've seen C implementations of quicksort that were based on the algorithms from these books. They were pre-structured-programming monstrosities.

If your compiler is unable to optimize the human readable, maintainable version of the algorithm, and you have to double as a human compiler--**buy a new compiler!** Nobody can afford human compilers any more. So, have mercy on yourself and your fellow programmers who will have to look at your code.

Don't use pointers where an index will do.

Exercises

Compare with solutions.

1. Implement the function:

```
void StrCpy (char* strDest, char* strSrc);
```

2. that copies the source string up to and with the terminating null into the destination string. The destination string is assumed to have enough space (unfortunately, we can’t simply assert it). Use the "indexing the array" method rather than pointers to char.
3. Implement the function

```
int StrCmp (char* str1, char* str2);
```

4. that lexicographically compares two strings. It returns zero when the strings are equal. Otherwise it returns a number greater than or less than zero if the first differing character in string 1 corresponds to, respectively, greater or smaller ASCII code than the corresponding character in string 2. The comparison stops when a null character is encountered in either string. If string 1 is longer, a positive number is returned; if string 2 is longer, a negative number is returned. Use the index implementation.
5. Another of these old "optimizing" tricks is to run a loop backwards. Instead of incrementing the counter we decrement it until it becomes negative or reaches zero. (At some point every programmer gets bitten by this trick when the loop variable is of the unsigned type.)

   Find a bug and rewrite the function

```
void StrNCpy (char* strDest, char* strSrc, size_t len)
{
    while (--len >= 0)
        *strDest++ = *strSrc++;
}
```

where size_t is some unsigned integral type defined, for instance, in cstring. Get rid of the "optimizing" tricks introduced here by a human compiler.

Pointers and Dynamic Memory Allocation

Despite all the negative advertising in this book, pointers are invaluable whenever dynamic data structures are involved. Dynamic data structures have to be able to grow and shrink. Growing involves acquiring new areas of memory and cultivating them; shrinking means recycling them. How do we acquire new areas of memory? We have to ask the system to give us memory for a particular type of variable or object. We do it using ***operator new***. The memory returned by new is allocated from the so called *heap*—the area of memory managed by the C++ runtime. How do we access this new memory? Using a **pointer** to that particular type of variable or object. For instance, to get memory enough to store a single integer, we'd use the following syntax:

```
int * pNumber = new int;
```

Even better, we could (and presumably should) immediately initialize this memory with a particular value we want to store there:

```
int * pNumber = new int (2001);
```

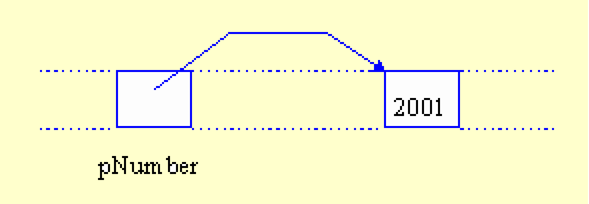


Figure 3. New memory for an integer was allocated and initialized with the value of 2001. The pointer pNumber contains the address of this new memory.

The initialization is mandatory when allocating an object that has a constructor. There are two cases: When there is a constructor that takes no arguments you don't have to do anything--such constructor is called by default. An argument-less constructor may, for instance, initialize member variables to some pre-defined values (e.g., zeros).

```
IStack* pStack = new IStack; // default constructor is called
```

When the object has a constructor that requires arguments, we specify them in the call to new:

```
Star* pStar = new Star (1234.5, 10.2);
```

If we want to store n integers, we need to allocate an array of n integers

```
int* pNumbers = new int [n];
```

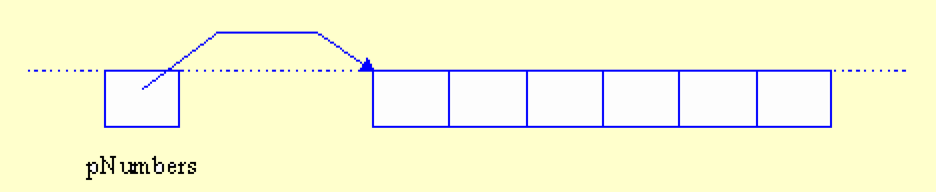


Figure 4. The result of allocating an array.

There is no direct way to initialize the contents of a dynamically allocated array. We just have to iterate through the newly allocated array and set the values by hand.

```
for (int i = 0; i < n; ++i)
    pNumbers [i] = i * i;
```

Consequently, if we are allocating an array of *objects*, there is no way to pass arguments to objects' constructors. Therefore it is required that the objects that are stored in such an array have a no-argument constructor. In particular, it's okay if no constructor is defined for a given class-—we are allowed to allocate an array of such objects. No constructors will be called during initialization.

If the class *has* constructors, one of them must take no arguments (a class may have more than one constructor if they differ by the number or the type of parameters). This no-argument constructor will be then called automatically for every object in the array. For instance, when allocating an array of stacks

```
IStack* aStack = new IStack [4];
```

a constructor will be called for each of the four stacks.

To recycle, or free, the memory allocated with new, we use ***operator delete***:

```
delete pNumber;
```

If the object being deleted has a destructor, it will be called automatically. For instance,

```
delete pStar;
```

will print "Destroying a star of brightness ..."

In the case of arrays, we mark the pointer to be deleted with square brackets. Curiously enough, we put the brackets in front of the pointer name, rather than after it.

```
delete [] pNumbers;
```

Again, if we are dealing with an array of objects, the destructor for each of them is automatically called.

Not so in the case of an *array of pointers*! When the array of pointers is deleted, the objects pointed to by the elements of the array are *not* automatically deleted. You have to actually go through a loop and delete each of them. The rule of thumb is: Try to allocate memory in the constructor of an object and deallocate it in its destructor. The deallocation should than reflect the allocation. If you had to call operator new in a loop, you'll have to call operator delete in a similar loop.

Deleting a null pointer is safe. Hint: don't check for a null pointer before calling delete—delete will do it for you anyway.

What happens if we forget to call delete for a dynamically allocated object or array? The memory it occupies will never be recycled (at least not until the program terminates). We call it a ***memory leak***. Is it a bug? During program termination all memory is freed anyway (although no destructors are called for the leaked objects), so who cares?

It depends. If the program repeatedly leaks more and more memory during its execution, it will eventually run out of memory. So that's a bug. (And a very difficult one to catch, I might add.) Good programming practice, therefore, is to ***always deallocate*** everything that has ever been allocated. Incidentally, there are tools and techniques to monitor memory leaks. I'll talk about it later.

Dynamic Data Structures

Linked lists, recursive type definitions, hash tables, arrays and pointers.

It is time to introduce a few fundamental data structures that will serve as building blocks for the creation of more complex objects. We will start with a dynamic array (actually a stack), a linked list and a string buffer. Then we will introduce hash tables, and finally we will put all these components together to form a string table--a data structure for efficient storage and retrieval of strings.

It's good to know that almost all of these data structures are available through the C++ standard library. A dynamic array is called a `vector`; there is also a `list`, a `string` and much more. There are, however, several reasons why I'd like you to first learn to implement these data structures without the use of the standard library. First, it will be a good exercise and a good way to learn the language step-by-step. Second, I want you to get an idea how these data structures might be implemented in the standard library. Third, the standard library uses very advanced features of the language (e.g., templates, operator overloading, etc.) which we haven't talked about yet. We'll get there in time, though.

Dynamic Stack

Dynamic arrays

We will now modify the contract of our stack. There will no longer be a strict limit on how many integers can be pushed. We’ll be able to keep pushing until we run out of memory.

However, we would like our implementation not to be a memory hog, at least not when it’s not necessary. So every time we run out of space in the stack, we’ll just double its size. Obviously, we won’t need the method IsFull. Otherwise the interface will be the same as that of our ordinary stack.

Here’s how our new stack works: There's a private helper function Grow that's used internally to double the stack size when necessary. We store the current capacity of the stack in a private variable _capacity. For the purpose of testing we'll set the initial capacity of the stack to one, so that we'll be able to see the stack grow almost immediately.

```
const int initStack = 1;

class IStack
{
public:
      IStack ();
      ~IStack ();
      void Push ( int i );
      int  Pop ();
      int  Top () const;
      bool IsEmpty () const;
private:
      void Grow ();

      int * _arr;
      int   _capacity; // size of the array
      int   _top;
};
```

Notice that _arr is declared as a pointer rather than an array. A dynamically allocated array *has to* be declared as a pointer.

We allocate the initial array in the constructor of IStack and we delete the memory in the destructor.

```
IStack::IStack ()
      : _top (0), _capacity (initStack)
{
      _arr = new int [initStack]; // allocate memory
}

IStack::~IStack ()
{
       delete []_arr; // free memory
}
```

This is a new kind of relationships between objects. Not only can we say that IStack has access to the array, we can say that IStack owns the array. The ***owns-a*** relationship is expressed through a pointer. Object A owns object B if object A is responsible for object B's deallocation. We'll talk more about it when we discuss resource management.

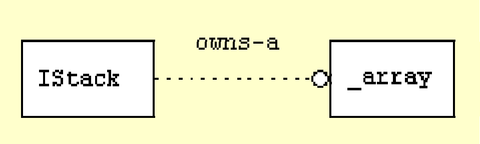


Fig. The owns-a relationship between objects.

IStack acquires the memory for the array in its constructor and deallocates it in its destructor. In a moment we'll see that the method Grow does some reallocation of its own.

So what happens when we Push an integer and there is no space left in the array? We call the Grow method and then complete the push.

```
void IStack::Push (int i)
{
      assert (_top <= _capacity);
      if (_top == _capacity)
            Grow ();
      _arr [_top] = i;
      ++_top;
}
```

Let's see what Grow does. It has to go through the following steps:

- Allocate new array of twice the size of the current array
- Copy all entries from the old array into the first half of the new array
- Double the _capacity variable
- Delete the old array
- Set the pointer _arr to point to the newly allocated array.

```
void IStack::Grow ()
{
      cout << "Doubling stack from " << _capacity << ".\n";
      // allocate new array
      int * arrNew = new int [2 * _capacity];
      // copy all entries
      for (int i = 0; i < _capacity; ++i)
            arrNew [i] = _arr [i];
      _capacity = 2 * _capacity;
      // free old memory
      delete []_arr;
      // substitute new array for old array
      _arr = arrNew;
}
```

The statement

```
_arr = arrNew;
```

is a ***pointer assignment***. We are changing the pointer _arr to point into the new location--the same location arrNew is pointing at. This is the one operation that we cannot do with references. Notice: we are not changing the *contents* of the location. That would be done in the statement

```
*_arr = *arrNew;
```

or its equivalent

```
_arr [0] = arrNew [0];
```

We are changing the pointer _arr itself.

```
_arr = arrNew;
```

For completeness, here are the implementations of the rest of the methods. Notice that although _arr is a pointer, it is accessed like an array.

```
int IStack::Pop ()
{
      // Do not Pop an empty stack
      assert (_top > 0);
      --_top;
      return _arr [_top];
}

int IStack::Top () const
{
      // Don't call Top on an empty stack
      assert (_top > 0);
      return _arr [_top - 1];
}

bool IStack::IsEmpty () const
{
      assert (_top >= 0);
      return _top == 0;
}
```

And here's a simple test program that forces the stack to grow a few times. Notice that we never shrink the stack. This is called a ***high water mark*** type of the algorithm. It would be easy though to add the Shrink method called by Pop every time _top gets much below _size/2.

```
int main ()
{
      IStack stack;
      for (int i = 0; i < 5; ++i)
      {
            cout << "Push " << i << endl;
            stack.Push (i);
      }
      for (int j = 0; j < 5; ++j)
      {
            cout << "Pop " << stack.Pop () << endl;
      }
}
```

Linked List

A linked list is a dynamic data structure with fast prepending and appending and linear searching. A list consists of links. Suppose that we want to store integer id's in a linked list. A Link will contain a field for an id and a pointer to the next link. To see how it works in practice, let's look at the picture.

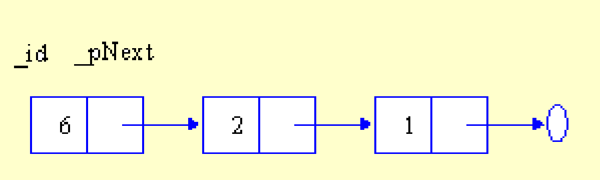


Figure 5. A linked list storing numbers 6, 2 and 1.

The last link in the list has a null (zero) pointer _pNext. That's the list terminator. A null pointer is never a valid pointer, that is, it cannot be dereferenced. It can however be checked against zero.

```
class Link
{
public:
    Link (Link* pNext, int id)
        : _pNext (pNext), _id (id) {}
    Link *  Next () const { return _pNext; }
    int     Id () const { return _id; }
private:
    int     _id;
    Link *  _pNext;
};
```

Notice how the definition of Link is self-referencing--a Link contains a pointer to a Link. It's okay to use *pointers* and *references* to types whose definitions haven't been seen by the compiler, yet. The only thing needed by the compiler in such a case is the *declaration* of the type. In our case the line

```
class Link
```

serves as such a declaration.

A linked ***List*** stores a pointer to the first link. Conceptually, a list is a different object than the link. To begin with, it has a different interface. One can add new items to the list, one can remove them. One can iterate the list, etc. These operations make no sense as methods of Link. Yet, in some sense, every link is a beginning of a list. This recursive way of looking at things is very popular in languages like Lisp. We usually try to avoid such recursive concepts in object oriented programming. So for us a List is a different creature than a Link.

Starting from the first link pointed to by _pHead, we can traverse the whole list following the pointers _pNext.

In the beginning, the list is empty and the pointer _pHead is initialized to zero.

```
class List
{
public:
    List (): _pHead (0) {}
    ~List ();
    void Add ( int id );
    Link const * GetHead () const { return _pHead; }
private:
    Link* _pHead;
};
```

The list has a destructor in which it deallocates all the links. The list ***owns*** the links.

```
List::~List ()
{
    // free the list
    while ( _pHead != 0 )
    {
        Link* pLink = _pHead;
        _pHead = _pHead->Next(); // unlink pLink
        delete pLink;
    }
}
```

The algorithm for deallocating a list works as follows: As long as there is something in the list, we point a temporary pointer, pLink, at the first link; point the _pHead at the next link (could be null), which is equivalent to unlinking the first link; and delete this link. The statement

```
_pHead = _pHead->Next ();
```

is a pointer assignment. _pHead will now point at the same link that is pointed at by _pHead->_pNext

Again, this is something we wouldn't be able to do with references.

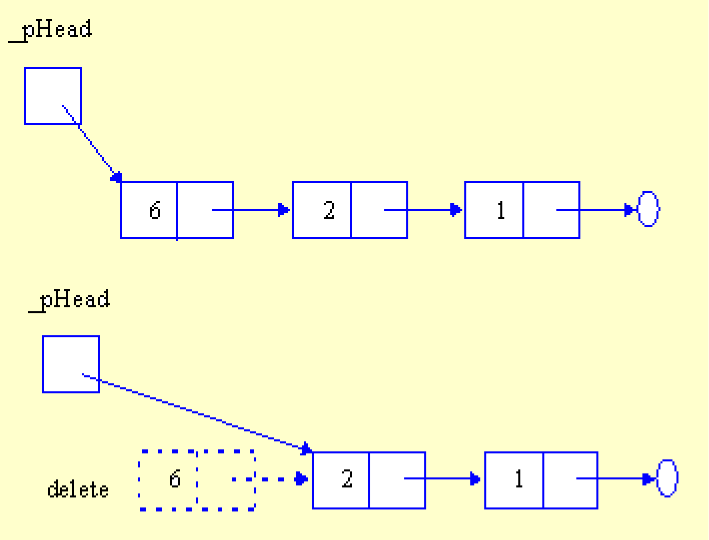


Figure 6. The unlinking and deleting of the first element of the list.

There is a simpler, recursive, implementation of the linked list destructor:

```
List::~List ()
{
    delete _pHead;
}

Link::~Link ()
{
    delete _pNext;
}
```

The recursion stops automatically as soon as it reaches the terminating null pointer.

Simplicity is the obvious advantage of the recursive solution. The price you pay for this simplicity is speed and memory usage. Recursive function calls are more expensive than looping. They also consume the program's stack space. If stack is at premium (e.g., when you're writing the kernel of an operating system), recursion is out of the question. But even if you have plenty of stack, you might still be better off using the iterative solution for really large linked lists. On the other hand, because of memory thrashing, really large linked lists are a bad idea in itself (we'll talk about it later). In any case, whichever way you go, it's good to know your tradeoffs.

Adding a new id to the list is done by allocating a new link and initializing it with the id and the pointer to the next item. The next item is the link that was previously at the beginning of the list. (It was pointed to by _pHead.) The new link becomes the head of the list and _pHead is pointed to it (pointer assignment). This process is called prepending.

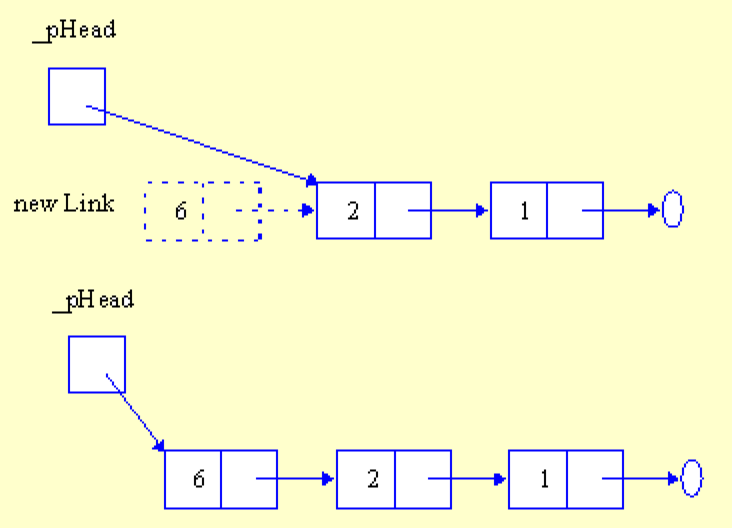


Figure 7. Prepending a new element in front of the list.

```
void List::Add ( int id )
{
    // add in front of the list
    Link * pLink = new Link (_pHead, id);
    _pHead = pLink;
}
```

Since the list grows dynamically, every new Link has to be allocated using operator new. When a ***new object*** is allocated, its constructor is automatically called. This is why we are passing the constructor arguments in new Link (_pHead, id).

List iteration is simple enough. For instance, here's how we may do a search for a link containing a given id

```
for (Link const * pLink = list.GetHead();
     pLink != 0;
     pLink = pLink->Next ())
{
    if (pLink->Id() == id)
        break;
}
```

Notice the use of pointers to const. The method GetHead is declared to return a pointer to a const Link.

```
Link const * GetHead () const;
```

The variable we assign it to must therefore be a pointer to a const Link, too.

```
Link const * pLink = list.GetHead ();
```

The meaning of a pointer to const is the same as the meaning of a reference to const--the object pointed to, or referred to, cannot be changed through such a pointer or reference. In the case of a pointer, however, we have to distinguish between a pointer to const and a const pointer. The latter is a pointer that, once initialized, cannot be pointed to anything else (just like a reference). The syntax for these two cases is

```
Link const * pLink; // pointer to const
Link * const pLink = pInitPtr;  // const pointer
```

Finally, the two can be combined to form a const pointer to const

```
Link const * const pLink = pInitPtr; // const pointer to const
```

This last pointer can only be used for read access to a single location to which it was initialized.

There is some confusion in the naming of pointers and references when combined with const. Since there is only one possibility of const-ness for a reference, one often uses the terms reference to const and const reference interchangeably. Unfortunately, the two cases are often confused when applied to pointers. In this book const pointer will always mean a pointer that cannot be moved, and pointer to const, a pointer through which one cannot write.

But wait, there's even more to confuse you! There is an equivalent syntax for a pointer to const

```
const Link * pLink; // pointer to const
```

and, of course,

```
const Link * const pLink = pInitPtr; // const pointer to const
```

The source of all this confusion is our insistence on reading text from left to right (people in the Middle East will laugh at us). Since C was written for computers and not humans, the direction of reading didn't really matter. So, to this day, declaration are best read right to left.

Let's play with reversing some of the declarations. Just remember that asterisk means "pointer to a."

```
Link const * pLink;
```

becomes Figure 8.

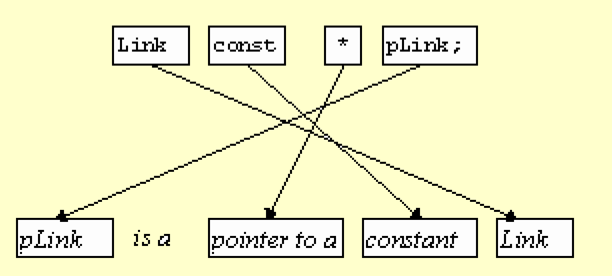


Figure 8. Reading type declarations: pLink is a pointer to a constant Link.

Similarly

```
Link * const pLink;
```

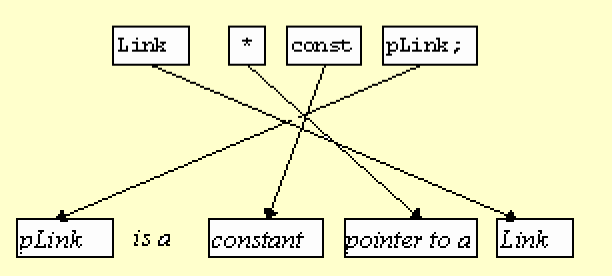


Figure 9. Reading type declarations: pLink is a constant pointer to a Link.

```
Link const * const pLink = pInitPtr;
```

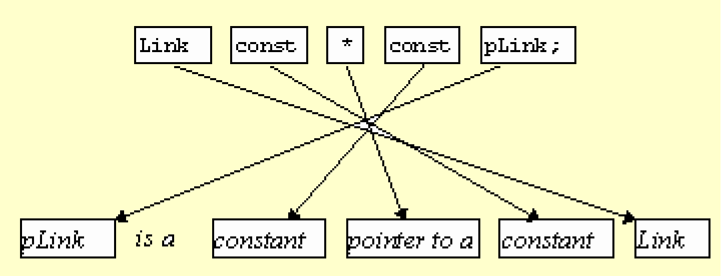


Figure 10. Reading type declarations: pLink is a constant pointer to a constant Link.

String Table

A string table maps integers into strings and strings into integers. The first mapping is easily implemented using an array of strings (pointers, or even better, offsets into a buffer). The second one is more difficult to implement efficiently. Our goal is to have constant time access both ways. It means that, independently of how many strings there are in the table (within some reasonable limits), the mappings of an int into a string and a string into an int take, on average, constant time (in the programmer speak O(1)). Notice that in the simplest implementation—going through the list of strings and comparing each with the string to be mapped—the execution takes longer and longer as the list grows (we say, it grows like O(N)). However, there is a data structure called the *hash table* that can accomplish constant time mapping. We will use it to implement an efficient string table.

With this performance goal in mind, we are ready to design the string table. Since we want to be able to add strings to it, we will need a large buffer to store them. The StringBuffer object stores a string and returns an offset at which the string can be found. This offset, in turn, is stored in another data structure, an array of offsets, that maps integer id's to string offsets. Each string is thus assigned an id, which is simply the index into the array of offsets.

When adding a new string, the appropriate entry is also added to the hash table. (For the moment let's treat the hash table as a black box that maps strings into short lists of id's.)

```
const int idNotFound = -1;

const int maxStrings = 100;
// String table maps strings to ints
// and ints to strings

class StringTable
{
public:
      StringTable ();
      int ForceAdd (char const * str);
      int Find (char const * str) const;
      char const * GetString (int id) const;
private:
      HTable         _htab;  // string -> short list of id's
      int            _offStr [maxStrings]; // id -> offset
      int            _curId;
      StringBuffer   _strBuf; // offset -> string
};
```

The translation from an id into a string is done in the GetString method, the translation from the string into an id is done in Find, and the addition of new strings without looking for duplicates is done in ForceAdd.

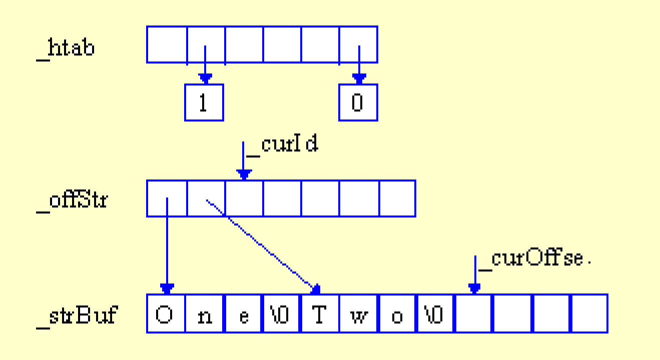


Figure 11. The hash table stores indexes into the offset array. The offset array stores indexes into the string buffer. The string buffer stores strings.

```
StringTable::StringTable ()
```

```
    : _curId (0)
{}

int StringTable::ForceAdd (char const * str)
{
      int len = strlen (str);
      // is there enough space?
      if (_curId == maxStrings || !_strBuf.WillFit (len))
      {
            return idNotFound;
      }
      // point to the place where the string will be stored
      _offStr [_curId] = _strBuf.GetOffset ();
      _strBuf.Add (str);
      // add mapping to hash table
      _htab.Add (str, _curId);
      ++_curId;
      return _curId - 1;
}
```

The hash table (still a black box for us) finds a short list of entries, one of them containing the id of the given string. We search this list in order to find the correct id.

```
int StringTable::Find (char const * str) const
{
      // Get a short list from hash table
      List const & list = _htab.Find (str);
      // Iterate over this list
      for (Link const * pLink = list.GetHead ();
           pLink != 0;
           pLink = pLink->Next ())
      {
            int id = pLink->Id ();
            int offStr = _offStr [id];
            if (_strBuf.IsEqual (offStr, str))
                  return id;
      }
      return idNotFound;
}

// map integer into string. Must be valid id

char const * StringTable::GetString (int id) const
{
      assert (id >= 0);
      assert (id < _curId);
      int offStr = _offStr [id];
      return _strBuf.GetString (offStr);
}
```

String Buffer

Our string buffer is implemented as an array of characters. We keep copying null terminated strings into this array until we run out of space. The variable `_curOffset` is an index into the array and it always points at the next free area where a string can be copied. It is initialized to zero, thus pointing at the beginning of the buffer. Before adding a string we make sure that it will fit in the remaining space. Adding a string means copying it to the place where _curOffset points to, and moving _curOffset past the string's terminating null. GetOffset returns the current offset, which can be later used to access the string about to be copied. IsEqual compares the string at a given offset with a given string, and GetString returns a const string given the offset.

```
const int maxBufSize=500;

class StringBuffer
{
public:
      StringBuffer () : _curOffset (0) {}
      bool WillFit (int len) const
      {
            return _curOffset + len + 1 < maxBufSize;
      }
      void Add (char const * str)
      {
            // strcpy (_buf + _curOffset, str);
            strcpy (&_buf [_curOffset], str);
            _curOffset += strlen (str) + 1;
      }
      int GetOffset () const
      {
            return _curOffset;
      }
      bool IsEqual (int offStr, char const * str) const
      {
            // char const * strStored = _buf + offStr;
            char const * strStored = &_strBuf [offStr];
            // strcmp returns 0 when strings are equal
            return strcmp (str, strStored) == 0;
      }
      char const * GetString (int offStr) const
      {
            //return _buf + offStr;
            return &_buf [offStr];
      }
private:
      char    _buf [maxBufSize];
      int     _curOffset;
};
```

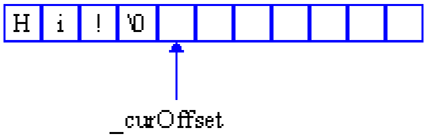


Figure 12. Current offset in the string buffer.

There are two ways of creating a pointer that points to a particular entry in an array. One can take the address of the nth element

```
p = &arr [n];  // address of the n’th element
```

or one can use the fact that one can add an integer to a pointer in order to move it n entries ahead

```
p = arr + n;  // pointer plus integer
```

Both are correct and produce the same code-- in the example above, pointer equivalents are shown as comments. The array notation seems less confusing.

I also used a generalized assignment operator += that adds the right hand side expression to the left hand side variable. For instance,

```
n += i;
```

is equivalent to

```
n = n + i;
```

The code produced is of course the same (although originally the += operator was introduced in C as an optimization), but the shorthand remains useful, once you get used to it

Finally, I used some of the string functions from the standard library. The header file that contains their declaration is called cstring. You can find a detailed description of these functions in your compiler’s help.

Table Lookup

Before explaining how the hash table works, let me make a little digression about algorithms that use table lookup. Accessing a table is a very fast operation (unless the table doesn’t fit in physical memory; but that’s a different story). So, if we have a function whose values can be pre-computed and stored in a table, we can trade memory for speed. The isdigit function (macro) is a prime example of such a tradeoff. The naive implementation would be

```
inline bool IsDigitSlow (char c)
{
    return c >= '0' && c <= '9';
}
```

However, if we notice that there can only be 256 different arguments to isdigit, we can pre-compute them all and store in a table. Let’s define the class CharTable that stores the pre-computed values

```
class CharTable
{
public:
      CharTable ();
      bool IsDigit (unsigned char c) { return _tab [c]; }
private:
      bool _tab [UCHAR_MAX + 1];  // limits.h
};

CharTable::CharTable ()
{
      for (int i = 0; i <= UCHAR_MAX; ++i)
      {
            // use the slow method
            if (i >= '0' && i <= '9')
                  _tab [i] = true;
            else
                  _tab [i] = false;
      }
}

CharTable TheCharTable;
```

Now we could quickly find out whether a given character is a digit by calling

```
TheCharTable.IsDigit (c)
```

In reality the isdigit macro is implemented using a lookup of a statically initialized table of bit fields, where every bit corresponds to one property, such as being a digit, being a white space, being an alphanumeric character, etc.

Hash Table

The hash table data structure is based on the idea of using table lookup to speed up an arbitrary mapping. For our purposes, we are interested in mapping strings into integers. We cannot use strings directly as indices into an array. However, we can define an auxiliary function that converts strings into such indices. Such a function is called a hash function. Thus we could imagine a two-step process to map a string into an integer: for a given string calculate the hash function and then use the result to access an array that contains the pre-computed value of the mapping at that offset.

Such hashing, called perfect hashing, is usually difficult to implement. In the imperfect world we are usually satisfied with a flawed hash function that may occasionally map two or more different strings into the same index. Such situation is called a collision. Because of collisions, the hash table maps a string not into a single value but rather into a "short list" of candidates. By further searching this list we can find the string we are interested in, together with the value into which it is mapped.

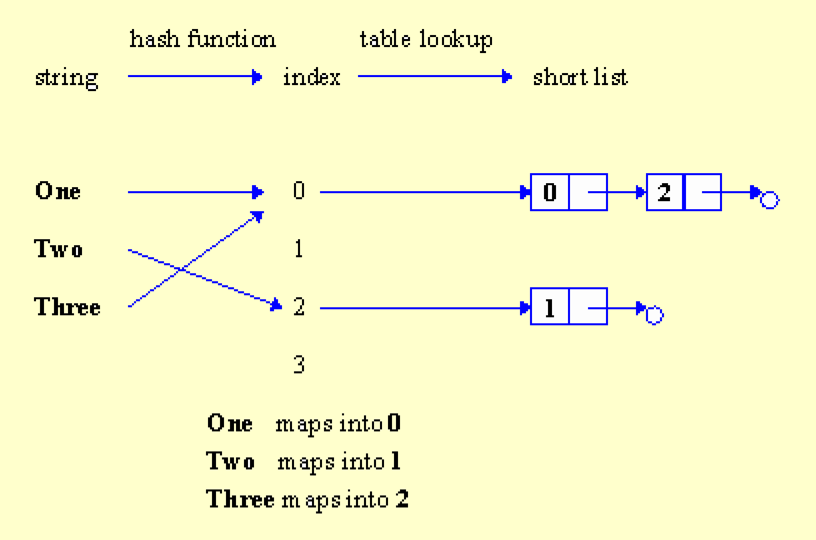


Figure 13. The hash function for the string One is the same as for the string Three. The collision is dealt with by creating a short list that contains the id's for both strings.

This algorithm becomes efficient when the number of strings to be mapped is large enough. Direct linear search among N strings would require, on average, N/2 comparisons. On the other hand, if the size of the hash table is larger than N, the search requires, on average, one comparison (plus the calculation of the hash function). For instance, in our string table we can store at most 100 strings. Finding a given string directly in such a table would require, on average, 50 string comparisons. If we spread these strings in a 127-entry array using a hashing function that randomizes the strings reasonably well, we can expect slightly more than one comparison on the average. That's a significant improvement.

Here is the definition of the class HashTable . The table itself is an array of lists (these are the "short lists" we were talking about). Most of them will contain zero or one element. In the rare case of a ***conflict***, that is, two or more strings hashed into the same index, some lists may be longer than that.

```
const int sizeHTable = 127;

// Hash table of strings

class HTable
{
public:
      // return a short list of candidates
      List const & Find (char const * str) const;
      // add another string->id mapping
      void Add (char const * str, int id);
private:
      // the hashing function
      int hash (char const * str) const;

      List _aList [sizeHTable]; // an array of (short) lists
};

// Find the list in the hash table that may contain
// the id of the string we are looking for

List const & HTable::Find (char const * str) const
{
      int i = hash (str);
      return _aList [i];
}

void HTable::Add (char const * str, int id)
{
      int i = hash (str);
      _aList [i].Add (id);
}
```

The choice of a hash function is important. We don't want to have too many conflicts. The shift-and-add algorithm is one of the best string randomizers.

```
int HTable::hash (char const * str) const
{
      // no empty strings, please
      assert (str != 0 && str [0] != 0);
      unsigned h = str [0];
      for (int i = 1; str [i] != 0; ++i)
            h = (h << 4) + str [i];
      return h % sizeHTable; // remainder
}
```

The expression `h << 4` is equal to *h* shifted left by 4 bits (that is multiplied by 16).

In the last step in the hashing algorithm we calculate the remainder of the division of `h` by the size of the hash table. This value can be used directly as an index into the array of `sizeHTable` entries. The size of the table is also important. Powers of 2 are worst--they create a lot of conficts; prime numbers are best. Usually a power of 2 plus or minus one will do. In our case 127 = 2^7 - 1, which happens to be a prime number.

The hash function of the string "One" is 114. It is calculated as follows

char	ASCII	h
'O'	0x4F	0x4F
'n'	0x6E	0x55E
'e'	0x65	0x5645

The remainder of division of h by 127 is 114, so the id of string "One" will be stored at offset 114 in the hash table array.

Test

This is how we may test the string table:

```
int main ()
{
      StringTable strTable;
      strTable.ForceAdd ("One");
      strTable.ForceAdd ("Two");
      strTable.ForceAdd ("Three");

      int id = strTable.Find ("One");
      cout << "One at " << id << endl;
      id = strTable.Find ("Two");
      cout << "Two at " << id << endl;
      id = strTable.Find ("Three");
      cout << "Three at " << id << endl;
      id = strTable.Find ("Minus one");
      cout << "Minus one at " << id << endl;

      cout << "String 0 is " << strTable.GetString (0) << endl;
      cout << "String 1 is " << strTable.GetString (1) << endl;
      cout << "String 2 is " << strTable.GetString (2) << endl;
}
```

Exercises

See the solutions

1. Add a private Shrink method to the dynamic stack. Let Pop call it when _top reaches some low water mark. Make sure the stack doesn't have a size at which alternating Pushes and Pops lead to alternating allocations and deallocations.
2. Create a 2-deep stack of strings (char *). It's a stack that "remembers" only the last and the one before last strings. When a new string is pushed, the "before last" string is deleted, the "last" string takes its place and the new one becomes "last." Pop returns the last string; it also moves up the "before last" to the "last" position one and stores zero in its place. In other words, the third Pop in a row will always return a zero pointer.
3. Create a queue of integers with the FIFO (First In First Out) protocol. Use a linked list to implement it.
4. Create a sequencer for a linked list. Create another sequencer that keeps track of both, the current and the previous item. Implement method Unlink on such a sequencer. It should remove the current item from the list that is being iterated. Test for special cases (remove first, remove last, one element list, etc.).
5. Create a doubly linked list item, DLink, with the _pNext as well as _pPrev pointers. Implement the Unlink method of the DLink object. Create a FIFO queue that uses a circular doubly linked list; that is, the last element points at the first element and the first element points back at the last element. Always create a new DLink with both links closed on itself. The pointer to itself can be accessed through the keyword this. For instance _pNext = this; will point _pNext at the current DLink.

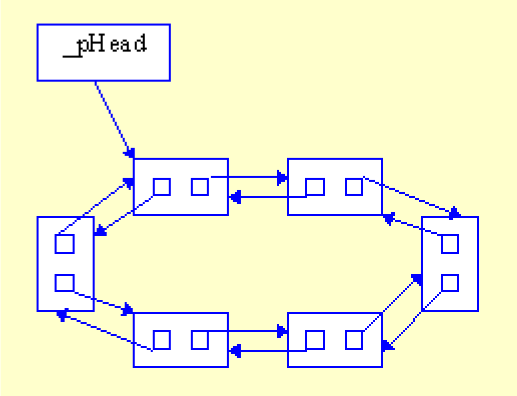


Figure 14. A circular doubly linked list.

6. Design and implement a sequencer for the hash table. It should go through all the entries in the array and follow all the non-empty linked lists.
7. Using the hash table sequencer from the previous problem, design and implement new StringTable that uses a dynamic hash table that grows by allocating a larger array, iterates through the old hash table and re-hashes each element into the new array. Notice that the new array's hash function will differ from the old one (the division modulo size will use a different size).

Polymorphism

Polymorphism

The Meaning of is-a

The fact that a Star is a CelestialBody means not only that a Star has all the member variables and methods of a CelestialBody (and more). After all that would be just syntactic sugar over straight embedding. One could embed a CelestialBody as a public member of Star and, apart from awkward semantics of the has-a relationship, one could imitate inheritance. (As a matter of fact, this is probably how your compiler implements inheritance.)

There is however one big difference. Due to inheritance, a Star can pass as a CelestialBody. What does it mean? It means that a function that expects a reference or a pointer to a CelestialBody will happily accept a reference or a pointer to a Star. Here’s an example: The class BlackHole has a method Gobble that accepts any CelestialBody

```
void BlackHole::Gobble (CelestialBody * pBody);
```

Since a Star is a CelestialBody, it can be Gobbled by a BlackHole:

```
Star* pStar = new Star (1, 2);
TheBlackHole.Gobble (pStar);  // Yumm
```

How is the CelestialBody treated inside a BlackHole?

```
void BlackHole::Gobble (CelestialBody* pBody)
{
    delete pBody;
}
```

It is destroyed. Now the big question is: whose destructor is called? On the one hand, we know that we sent a Star to its fiery death, so we should hear "Destroying a star ..." On the other hand, the BlackHole has no idea that it is swallowing a Star. It will expect to hear "Destroying celestial body..." What’s more, at compile time, the compiler has no idea what will be sent into the BlackHole. Imagine a situation like this:

```
Star * pStar = new Star (1, 2);
Planet * pPlanet = new Planet (3, 4);

TheBlackHole.Gobble (pStar);  // Yumm
TheBlackHole.Gobble (pPlanet);  // Yumm, yumm
```

In both cases the same method Gobble is called, the same code is executed, therefore it is obvious that inside Gobble the compiler may only put the call to (or expand inline) CelestialBody’s destructor. The compiler dispatches the call based on the ***type of the pointer***. Notice that the same would apply to any other method of CelestialBody overridden by Star or Planet. If Gobble called any of these, the compiler would call the CelestialBody's implementation and not the one provided by Star or Planet.

This solution is cheap and in many cases adequate. However, for a small additional fee, C++ can offer a very powerful feature called ***polymorphism***. If you want to hear different final cries from the insides of a BlackHole depending on what has fallen into it, you must declare the CelestialBody's destructor ***virtual***. When a virtual function is overridden in a derived class, the dispatching of calls to that function is done by ***actual type*** of the object pointed to, rather than by the ***type of the pointer***.

> If you are curious about how virtual functions are implemented and what kind of runtime overhead they incur, go on a little implementation sidetrip.

Parse Tree

Virtual member functions, virtual destructors, pure virtual functions, protected data members.

I will demonstrate the use of polymorphism in an example of a data structure—the arithmetic tree. An arithmetic expression can be converted into a tree structure whose nodes are arithmetic operators and leaf nodes are numbers. Figure 2-3 shows the example of a tree that corresponds to the expression 2 * (3 + 4) + 5. Analyzing it from the root towards the leaves we first encounter the plus node, whose children are the two terms that are to be added. The left child is a product of two factors. The left factor is number 2 and the right factor is the sum of 3 and 4. The right child of the top level plus node is number 5. Notice that the tree representation doesn't require any parentheses or the knowledge of operator precedence. It uniquely describes the calculation to be performed.

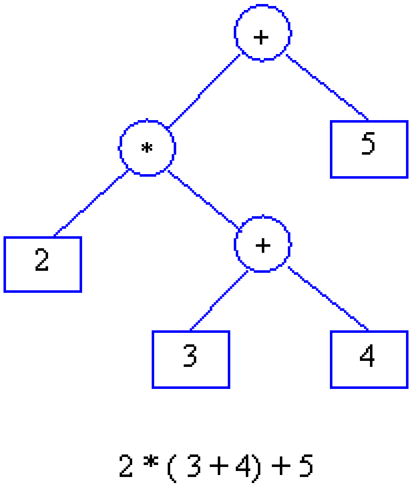


Figure 2-3 The arithmetic tree corresponding to the expression 2 * (3 + 4) + 5.

We will represent the nodes of the arithmetic tree as objects inheriting from a single class Node. The direct descendants of the Node are NumNode representing a number and BinNode representing a binary operator. For simplicity, we will restrict ourselves to only two classes derived from BinNode, the AddNode and the MultNode. Figure 2-4 shows the class hierarchy I have just described. Abstract classes are the classes that cannot be instantiated, they only serve as parents for other classes. I'll explain this term in a moment

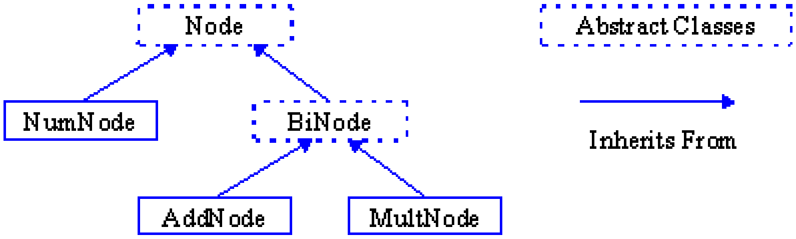


Figure 2-4 The class hierarchy of nodes.

What are the operations we would like to perform on a node? We would like to be able to calculate its value and, at some point, destroy it. The Calc method returns a double as the result of the calculation of the node's value. Of course, for some nodes the calculation may involve the recursive calculations of its children. The method is const since it doesn't change the node itself. Since each type of node has to provide its own implementation of the Calc method, we make this function virtual. However, there is no "default" implementation of Calc for an arbitrary Node. The function that has no implementation (inherited or otheriwise) is called ***pure virtual***. That's the meaning of = 0 in the declaration of Calc.

A class that has one or more pure virtual functions is called an ***abstract class*** and it cannot be instantiated (no object of this class can be created). Only classes that are derived from it, and which provide their own implementations of all the pure virtual functions, can be instantiated. Notice that our sample arithmetic tree has instances of AddNodes, MultNodes and NumNodes, but no instances of Nodes or BinNodes.

A rule of thumb is that, if a class has a virtual function, it probably needs a virtual destructor as well--and once we decide to pay the overhead of a vtable pointer, all subsequent virtual functions don't increase the size of the object. So, in such a case, adding a virtual destructor doesn't add any significant overhead.

In our case we can anticipate that some of the descendant nodes will have to destroy their children in their destructors, so we really need a virtual destructor. A destructor cannot be made pure virtual, because it is actually called by the destructors of the derived classes. That's why I gave it an empty body. (Even though I made it inline, the compiler will create a function body for it, because it needs to stick a pointer to it into the virtual table).

```
class Node
{
public:
    virtual ~Node () {}
    virtual double Calc () const = 0;
};
```

NumNode stores a double value that is initialized in its constructor. It also overrides the Calc virtual function. In this case, Calc simply returns the value stored in the node.

```
class NumNode: public Node
{
public:
    NumNode (double num) : _num (num ) {}
    double Calc () const;
private:
    const double _num;
};

double NumNode::Calc () const
{
    cout << "Numeric node " << _num << endl;
    return _num;
}
```

BinNode has two children that are pointers to nodes. They are initialized in the constructor and deleted in the destructor—this is why I could make them const pointers (but not *pointers to* const, since I have to call the non-const method on them—the destructor). The Calc method is still pure virtual, inherited from Node, only the descendants of BinNode will know how to implement it.

```
class BinNode: public Node
{
public:
    BinNode (Node * pLeft, Node * pRight)
      : _pLeft (pLeft), _pRight (pRight) {}
    ~BinNode ();
protected:
    Node * const _pLeft;
    Node * const _pRight;
};

BinNode::~BinNode ()
{
    delete _pLeft;
    delete _pRight;
}
```

This is where you first see the advantage of polymorphism. A binary node can have children which are arbitrary nodes. Each of them can be a number node, an addition node, or a multiplication node. There are nine possible combinations of children—it would be silly to make separate classes for each of them (consider, for instance, AddNodeWithLeftMultNodeAndRightNumberNode). We had no choice but to accept and store pointers to children as more general pointers to Nodes. Yet, when we call destructors through them, we need to call different functions to destroy different nodes. For instance, AddNode has a different destructor than a NumNode (which has an empty one), and so on. This is why we had to make the destructors of Nodes virtual.

Notice that the two data members of BinNode are not private—they are protected. This qualification is slightly weaker than private. A private data member or method cannot be accessed from any code outside of the implementation of the given class (or its friends). Not even from the code of the *derived* class. Had we made _pLeft and _pRight private, we'd have to provide public methods to set and get them. That would be tantamount to exposing them to everybody. By making them `protected` we are letting classes *derived* from `BinNode` manipulate them, but, at the same time, bar anybody else from doing so.

Table 1

Access specifier	**Who can access such member?**
`public`	anybody
`protected`	the class itself, its friends and derived classes
`private`	only the class itself and its friends

The class `AddNode` is derived from `BinNode`.

```
class AddNode: public BinNode
{
public:
    AddNode (Node * pLeft, Node * pRight)
        : BinNode (pLeft, pRight) {}
    double Calc () const;
};
```

It provides its own implementation of `Calc`. This is where you see the advantages of polymorphism again. We let the child nodes calculate themselves. Since the `Calc` method is virtual, they will do the right thing based on their actual class, and not on the class of the pointer (`Node *`). The two results of calling `Calc` are added and the sum returned.

```
double AddNode::Calc () const
{
    cout << "Adding\n";
    return _pLeft->Calc () + _pRight->Calc ();
}
```

Notice how the method of `AddNode` directly accesses its parent's data members `_pLeft` and `_pRight`. Were they declared private, such access would be flagged as an error by the compiler.

For completeness, here's the implementation of the `MultNode` and a simple test program.

```
class MultNode: public BinNode
{
public:
    MultNode (Node * pLeft, Node * pRight)
        : BinNode (pLeft, pRight) {}
    double Calc () const;
};

double MultNode::Calc () const
{
    cout << "Multiplying\n";
    return _pLeft->Calc () * _pRight->Calc ();
}

int main ()
{
    // ( 20.0 + (-10.0) ) * 0.1
    Node * pNode1 = new NumNode (20.0);
    Node * pNode2 = new NumNode (-10.0);
    Node * pNode3 = new AddNode (pNode1, pNode2);
    Node * pNode4 = new NumNode (0.1);
    Node * pNode5 = new MultNode (pNode3, pNode4);
    cout << "Calculating the tree\n";
    // tell the root to calculate itself
    double x = pNode5->Calc ();
    cout << x << endl;
    delete pNode5; // and all children
}
```

Do you think you can write more efficient code by not using polymorphism? Think twice! If you're still not convinced, go on a little sidetrip into the alternative universe of C.

Exercises

See solutions.

1. Design a program that calculates all the prime numbers up to 100 by eliminating all the numbers divisible by 2, 3, 5 and 7. Create an abstract class Sieve with one pure virtual method NextNumber. Implement a SourceSieve that simply iterates over all numbers from 1 to 100 in order. Implement Sieve2 that takes a reference to a Sieve in its constructor, and when asked to give the next number, keeps retrieving numbers from its child sieve until it finds one that is not divisible by 2 and returns it. Do the same for 3, 5 and 7. Create all these Sieves as local objects in main, chain them, and print all the numbers returned by the top Sieve. What order of Sieves is best?
2. Create an abstract class Command with two pure virtual methods Execute and Undo. Create command classes for our stack-based calculator, corresponding to all possible calculator inputs. For every user input construct a corresponding Command object, Execute it and push it on a two-deep stack of Command s. Add the ‘u’ (undo) command to the calculator. The u-command pops the last Command from the command stack and calls its virtual method Undo.

Small Software Project

Software Project

Top down design and implementation. Scanner, top-down recursive parser, symbol table. The grammar, problems with associativity. Allocation and ownership of objects returned from a function. Function pointers, switch statements.

Programmers are rarely totally satisfied with their code. So after I've written the program for this chapter—a simple parser-calculator—I immediately noticed that it could have been done better. Especially the top-level view of the program was not very clear. The nodes of the parsing tree needed access to such components as the `Store` (the calculator's "memory"), the symbol table (to print meaningful error messages), maybe even to the function table (to evaluate built-in functions). I could have either made all these objects global and thus known to everybody, or pass references to them to all the nodes of the tree.

The order of construction of these objects was important, so I felt that maybe they should be combined into one high-level object, the `Calculator`. The temptation to redesign and rewrite the whole program was strong. Maybe after the nth iteration I could come up with something close to the ideal?

Then I decided I shouldn't do it. Part of making a program is coming up with not-so-good solutions and then improving upon them. I would have cheated if I had come up with the best, and then reverse engineered it to fit the top-down design and implementation process. So here it is—the actual, real-life process of creating a program.

Specification

The purpose of the program is to accept arithmetic expressions from the user, evaluate them, and display the results. The expressions are to be parsed by a simple top-down recursive descent parser (if you have no idea what I'm talking about, don't panic-it's just a name for what I'm going to describe in detail later). The parser's goal is to convert the input string into an arithmetic tree. The simple grammar accepted by the parser is defined as follows: The parser is looking for an expression.

1. An ***expression*** is
    a. a term followed by a plus or a minus sign followed by another expression.
    b. Sometimes an expression doesn't contain any plus or minus signs, and then it is just equal to a term.
2. A ***term*** is
    a. a ***factor*** multiplied or divided by another ***term***.
    b. Sometimes a term contains no multiplication or division and then it is just equal to a ***factor***.
3. A ***factor*** can be
    a. a number,
    b. an identifier corresponding to a variable,
    c. a minus sign followed by a factor (unary minus), or
    d. the whole expression enclosed within parentheses.

For instance, the expression 1 + x * (2 - y) is parsed as in Figure 2-5

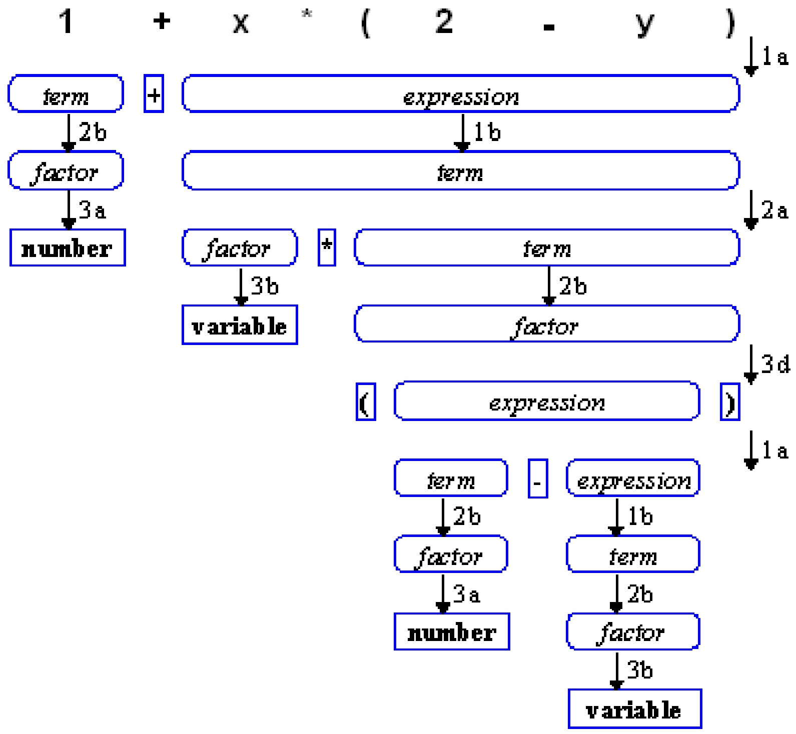


Figure 2-5 The parsing of an arithmetic expression. The arrows with numbers correspond to rules of grammar that were used to make each parsing step.

This grammar is simple and natural, but it has one flaw—the arithmetic operators are right associative instead of being left associative. That means, for instance, that a - b + c will be parsed as a - (b + c) which is not exactly what we'd expect. There are standard ways of fixing this grammar or modifying the parser, but that's beyond the scope of this section. (We will revisit this issue later.)

Since we want our calculator to be able to use variables, we will need a symbol table. A symbol table remembers names of variables. We will reuse the string table described in the previous section.

We'll also need to be able to assign values to variables. We can do it by expanding the definition of expression to include the following clause

1. An ***expression*** can also be
   a term followed by the equal sign followed by an expression.

Not every term though is a correct left-hand side of an assignment. It has to be an ***lvalue***—something that can be the left hand side of an assignment. In our case the only allowed type of lvalue will be an identifier corresponding to a variable.

We will also introduce a scanner, which will convert the input string into a series of tokens. It will recognize arithmetic operators, numbers, and identifiers. This way the parser will have an easier job--matching sequences of tokens to grammatical productons.

Stubbed Implementation

I will follow the top-down implementation strategy. It means that I will run the program first and implement it later. (Actually, I will run it as soon as I create stubs for all the top level classes.)

Let's start with the Scanner. It is constructed out of a buffer of text (a line of text, to be precise). It keeps a pointer to that buffer and, later, it will be able to scan it left to right and convert it to tokens. For now the constructor of the Scanner stub announces its existence to the world and prints the contents of the buffer.

```
class Scanner
{
public:
    Scanner (char const * buf);
private:
    char const * const _buf;
};

Scanner::Scanner (char const * buf)
    : _buf (buf)
{
    cout << "Scanner with \"" << buf << "\"" << endl;
}
```

The SymbolTable stub will be really trivial for now. We only assume that it has a constructor.

```
class SymbolTable
{
public:
    SymbolTable () {}
};
```

The Parser will need access to the scanner and to the symbol table. It will parse the tokens retrieved from the Scanner and evaluate the resulting tree. The method Eval is supposed to do that. It should return a status code that would depend on the result of parsing. We combine the three possible statuses into an *enumeration*. An enum is an integral type that can only take a few predefined values. These values are given symbolic names and are either initialized to concrete values by the programmer or by the compiler. In our case we don't really care what values correspond to the various statuses, so we leave it to the compiler. Using an enum rather than an int for the return type of Eval has the advantage of stricter type checking. It also prevents us from returning anything other than one of the three values defined by the enum.

```
enum Status
{
    stOk,
    stQuit,
    stError
};

class Parser
{
public:
    Parser (Scanner & scanner, SymbolTable & symTab);
    ~Parser ();
    Status Eval ();
private:
    Scanner &        _scanner;
    SymbolTable &    _symTab;
};

Parser::Parser (Scanner & scanner, SymbolTable & symTab)
    : _scanner (scanner), _symTab (symTab)
{
    cout << "Parser created\n";
}

Parser::~Parser ()
{
    cout << "Destroying parser\n";
}

Status Parser::Eval ()
{
    cout << "Parser eval\n";
    return stQuit;
}
```

Finally, the main procedure. Here you can see the top level design of the program in action. The lifetime of the symbol table has to be equal to that of the whole program, since it has to remember the names of all the variables introduced by the user during one session. The scanner and the parser, though, can be created every time a line of text is entered. The parser doesn't have any state that has to be preserved from one line of text to another. If it encounters a new variable name, it will store it in the symbol table that has a longer lifespan.

In the main loop of our program a line of text is retrieved from the standard input using the getline method of cin, a scanner is constructed from this line, and a parser is created from this scanner. The Eval method of the parser is then called to parse and evaluate the expression. As long as the status returned by Eval is different from stQuit, the whole process is repeated.

```
const int maxBuf = 100;

int main ()
{
    char buf [maxBuf];
    Status status;
    SymbolTable symTab;
    do
    {
        cout << "> ";  // prompt
        cin.getline (buf, maxBuf);
        Scanner scanner (buf);
        Parser  parser (scanner, symTab);
        status = parser.Eval ();
    } while (status != stQuit);
}
```

This program compiles and runs thus proving the validity of the concept.

Expanding Stubs

The first stub to be expanded into a full implementation will be that of the Scanner. The scanner converts the input string into a series of tokens. It works like an iterator. Whenever the Parser needs a token it asks the Scanner for the current token. When this token is parsed, the Parser accepts it by calling Scanner::Accept(). The Accept method scans the string further trying to recognize the next token.

There is a finite, well-defined set of tokens. It is convenient to put them into the enumeration EToken.

```
enum EToken
{
    tEnd,
    tError,
    tNumber,
    tPlus,
    tMult
};
```

We would also like the Scanner to be able to convert the part of the input string recognized as a number to a floating-point value. This is done by the Number() method that may be called only when the current token is tNumber (see the assertion there). Notice the use of the type char const * const—a const pointer to a const string. The pointer is initialized in the constructor and never changes again. Its contents is read-only, too.

```
class Scanner
{
public:
    Scanner (char const * buf);
    EToken  Token () const { return _token; }
    EToken  Accept ();
    double Number ()
    {
        assert (_token == tNumber);
        return _number;
    }
private:
    void EatWhite ();

    char const * const   _buf;
    int                  _iLook;
    EToken               _token;
    double               _number;
};
```

The constructor of the Scanner, besides initializing all member variables, calls the method Accept. Accept recognizes the first available token and positions the index _iLook past the recognized part of the buffer.

```
Scanner::Scanner (char const * buf)
    : _buf (buf), _iLook(0)
{
    cout << "Scanner with \"" << buf << "\"" << endl;
    Accept ();
}
```

EatWhite is the helper function that skips whitespace characters in the input.

```
void Scanner::EatWhite ()
{
    while (isspace (_buf [_iLook]))
        ++_iLook;
}
```

Accept is just one big ***switch*** statement. Depending on the value of the current character in the buffer, _buf[_iLook], the control passes to the appropriate ***case*** label within the switch. For now I have implemented the addition and the multiplication operators. When they are recognized, the _token variable is initialized to the appropriate enumerated value and the variable _iLook incremented by one.

The recognition of digits is done using the fall-through property of the case statements. In a switch statement, unless there is an explicit ***break*** statement, the control passes to the next case. All digit cases fall through, to reach the code after case '9'. The string starting with the digit is converted to an integer (later we will convert this part of the program to recognize floating-point numbers) and _iLook is positioned after the last digit.

The ***default*** case is executed when no other case matches the character in the switch.

```
EToken Scanner::Accept ()
{
    EatWhite ();
    switch (_buf[_iLook])
    {
    case '+':
        _token = tPlus;
        ++_iLook;
        break;
    case '*':
        _token = tMult;
        ++_iLook;
        break;
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
        _token = tNumber;
        _number = atoi (&_buf [_iLook]);
        while (isdigit (_buf [_iLook]))
            ++_iLook;
        break;
    case '\0': // end of input
        _token = tEnd;
        break;
    default:
        _token = tError;
        break;
    }
    return Token ();
}
```

You might ask: Can this switch statement be replaced by some clever use of polymorphism? I really don't know how one could do it. The trouble is that the incoming data—characters from the buffer—is ***amorphic***. At the stage where we are trying to determine the meaning of the input, to give it *form*, the best we can do is to go through a multi-way conditional, in this case a switch statement. Dealing with amorphic input is virtually the only case when the use of a switch statement is fully legitimate in C++. Parsing user input, data stored in a file or external events, all require such treatment.

The dummy parser is implemented as a for loop retrieving one token after another and printing the appropriate message.

```
Status Parser::Parse ()
{
    for (EToken token = _scanner.Token ();
        token != tEnd;
        token = _scanner.Accept ())
    {
        switch (token)
        {
        case tMult:
            cout << "Times\n";
            break;
        case tPlus:
            cout << "Plus\n";
            break;
        case tNumber:
            cout << "Number: " << _scanner.Number () << "\n";
            break;
        case tError:
            cout << "Error\n";
            return stQuit;
        default:
            cout << "Error: bad token\n";
            return stQuit;
        }
    }
    return stOk;
}
```

Once more, this partially implemented program is compiled and tested. We can see the flow of control and the correct recognition of a few tokens.

Final Implementation. Not!

There is no "final" implementation. Let me reiterate:

There is no "final" implementation.

So how do we know when to stop? The truth is, we don't. Sometimes we just get bored with the program, sometimes we start a new interesting project and we abandon the current one (promising ourselves that we'll come back to it later). Having a delivery date planned ahead (even if we slip a little), and a manager who tells us to stop adding new features and ship—helps.

With this caveat in mind, let me go over the pseudo-final version of the calculator to show you how the stubs get expanded to create a working program.

Scanner

Pointer to pointer type, passing simple types by reference, C++ identifiers, passing buffers.

The list of tokens was enlarged to include four arithmetic operators, the assignment operator, parentheses and a token representing an identifier. An identifier is a symbolic name, like *pi*, *sin*, *x*, etc.

```
enum EToken
{
    tEnd,
    tError,
    tNumber,    // literal number
    tPlus,      // +
    tMult,      // *
    tMinus,     // -
    tDivide,    // /
    tLParen,    // (
    tRParen,    // )
    tAssign,    // =
    tIdent      // identifier (symbolic name)
};
```

The Accept method was expanded to recognize the additional arithmetic symbols as well as floating point numbers and identifiers. Decimal point was added to the list of digits in the scanner's switch statement. This is to recognize numbers like .5 that start with the decimal point. The library function strtod (string to double) not only converts a string to a floating point number, but it also updates the pointer to the first character that cannot possibly be part of the number. This is very useful, since it lets us easily calculate the new value of _iLook after scanning the number.

```
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
case '.':
{
    _token = tNumber;
    char * p;
    _number = strtod (&_buf [_iLook], &p);
    _iLook = p - _buf; // pointer subtraction
    break;
}
```

The function strtod has two outputs: the value of the number that it has recognized and the pointer to the first unrecognized character.

```
double strtod (char const * str, char ** ppEnd);
```

How can a function have more than one output? The trick is to pass an argument that is a reference or a pointer to the value to be modified by the function. In our case the additional output is a pointer to char. We have to pass a reference or a pointer to this pointer. (Since strtod is a function from the standard C library it uses pointers rather than references.)

Let's see what happens, step-by-step. We first define the variable which is to be modified by strtod. This variable is a pointer to a char

```
char * p;
```

Notice that we don't have to initialize it to anything. It will be overwritten in the subsequent call anyway. Next, we pass the address of this variable to strtod

```
_number = strtod (&_buf [_iLook], &p);
```

The function expects a ***pointer to a pointer*** to a char

```
double strtod (char const * str, char ** ppEnd);
```

By dereferencing this pointer to pointer, strtod can overwrite the value of the pointer. For instance, it could do this:

```
*ppEnd = pCurrent;
```

This would make the original p point to whatever pCurrent was pointing to.

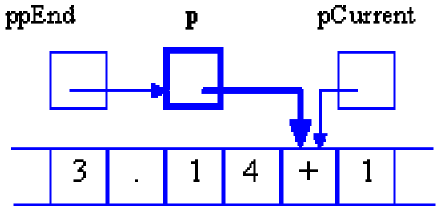


Figure 2-6

In C++ we could have passed a reference to a pointer instead (not that it's much more readable).

```
char *& pEnd
```

It is not clear that passing simple types like char* or int by reference leads to more readable code. Consider this

```
char * p;
_number = StrToDouble (&_buf [_iLook], p);
```

It looks like passing an uninitialized variable to a function. Only by looking up the declaration of StrToDouble would you know that p is passed by reference

```
double StrToDouble (char const * str, char *& rpEnd)
{
    ...
    rpEnd = pCurrent;
    ...
}
```

Although it definitely is a good programming practice to look up at least the declaration of the function you are about to call, one might argue that it shouldn't be necessary to look it up when you are reading somebody else's code. Then again, how can you understand the code if you don't know what StrToDouble is doing? And how about a comment that will immediately explain what is going on?

```
char * p;  // p will be initialized by StrToDouble
_number = StrToDouble (&_buf [_iLook], p);
```

You should definitely put a comment whenever you define a variable without immediately initializing it. Otherwise the reader of your code will suspect a bug.

Taking all that into account my recommendation would be to go ahead and use C++ references for passing simple, as well as user defined types by reference.

Of course, if strtod were not written by a human optimizing compiler, the code would probably look more like this

```
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
case '.':
```

```
{
    _token = tNumber;
    _number = StrToDouble (_buf, _iLook); // updates _iLook
    break;
}
```

with StrToDouble declared as follows

```
double StrToDouble (char const * pBuf, int& iCurrent);
```

It would start converting the string to a double starting from pBuf [iCurrent] advancing iCurrent past the end of the number.

Back to Scanner::Accept(). Identifiers are recognized in the default statement of the big switch. The idea is that if the character is not a digit, not a decimal point, not an operator, then it must either be an identifier or an error. We require an identifier to start with an uppercase or lowercase letter, or with an underscore. By the way, this is exactly the same requirement that C++ identifiers must fulfill. We use the isalpha() function (really a macro) to check for the letters of the alphabet. Inside the identifier we (and C++) allow digits as well. The macro isalnum() checks if the character is alphanumeric. Examples of identifiers are thus i, pEnd, _token, __iscsymf, Istop4digits, SERIOUS_ERROR_1, etc.

```
default:
    if (isalpha (_buf [_iLook]) || _buf [_iLook] == '_')
    {
        _token = tIdent;
        _iSymbol = _iLook;
        int cLook; // initialized in the do loop
        do {
            ++_iLook;
            cLook = _buf [_iLook];
        } while (isalnum (cLook) || cLook == '_');

        _lenSymbol = _iLook - _iSymbol;
        if (_lenSymbol > maxSymLen)
            _lenSymbol = maxSymLen;
    }
    else
        _token = tError;
    break;
```

To simplify our lives as programmers, we chose to limit the size of symbols to maxSymLen. Remember, we are still weekend programmers!

Once the Scanner recognizes an identifier, it should be able to provide its name for use by other parts of the program. To retrieve a symbol name, we call the following method

```
void Scanner::SymbolName (char * strOut, int & len)
{
    assert (len >= maxSymLen);
    assert (_lenSymbol <= maxSymLen);
    strncpy (strOut, &_buf [_iSymbol], _lenSymbol);
    strOut [_lenSymbol] = 0;
    len = _lenSymbol;
}
```

Notice that we *have to* make a copy of the string, since the original in the buffer is not null terminated. We copy the string to the caller's buffer strOut of length len. We do it by calling the function strncpy (string-n-copy, where n means that there is a maximum count of characters to be copied). The length is an in/out parameter. It should be initialized by the caller to the size of the buffer strOut. After SymbolName returns, its value reflects the actual length of the string copied.

How do we know that the buffer is big enough? We make it part of the contract—see the assertions.

The method SymbolName is an example of a more general pattern of passing buffers of data between objects. There are three main schemes: caller's fixed buffer, caller-allocated buffer and callee-allocated buffer. In our case the buffer is passed by the caller and its size is fixed. This allows the caller to use a local fixed buffer—there is no need to allocate or re-allocate it every time the function is called. Here's the example of the Parser code that makes this call—the buffer strSymbol is a local array

```
char strSymbol [maxSymLen + 1];
int lenSym = maxSymLen;
_scanner.SymbolName (strSymbol, lenSym);
```

Notice that this method can only be used when there is a well-defined and reasonable maximum size for the buffer, or when the data can be retrieved incrementally in multiple calls. Here, we were clever enough to always truncate the size of our identifiers to maxSymLen.

If the size of the data to be passed in the buffer is not limited, we have to be able to allocate the buffer on demand. In the case of caller-allocated buffer we have two options. Optimally, the caller should be able to first ask for the size of data, allocate the appropriate buffer and call the method to fill the buffer. There is a variation of the scheme—the caller re-allocated buffer—where the caller allocates the buffer of some arbitrary size that covers, say, 99% of the cases. When the data does not fit into the buffer, the callee returns the appropriate failure code and lets the caller allocate a bigger buffer.

```
char * pBuf = new char [goodSize];
int len = goodSize;
if (FillBuffer (pBuf, len) == errOverflow)
{
    // rarely necessary
    delete [] pBuf;
    pBuf = new char [len]; // len updated by FillBuffer
    FillBuffer (pBuf, len);
}
```

This may seem like a strange optimization until you encounter situations where the call to ask for the size of data is really expensive. For instance, you might be calling across the network, or require disk access to find the size, etc.

The callee-allocated buffer seems a simple enough scheme. The most likely complication is a memory leak when the caller forgets to deallocate the buffer (which, we should remember, hasn't been explicitly allocated by the caller). We'll see how to protect ourselves from such problems using smart pointers (see the chapter on managing resources). Other complications arise when the callee uses a different memory allocator than the caller, or when the call is remoted using, for instance, remote procedure call (RPC). Usually we let the callee allocate memory when dealing with functions that have to return dynamic data structures (lists, trees, etc.). Here's a simple code example of callee-allocated buffer

```
char * pBuf = AcquireData ();
// use pBuf
delete pBuf;
```

The following decision tree summarizes various methods of passing data to the caller

```
if (max data size well defined)
{
    use caller’s fixed buffer
}
else if (it's cheap to ask for size)
{
    use caller-allocated buffer
}
else if ((caller trusted to free memory
            && caller uses the same allocator
            && no problems with remoting)
        || returning dynamic data structures)
{
    use callee-allocated buffer
}
else
{
    use caller-re-allocated buffer
}
```

In the second part of the book we'll talk about some interesting ways of making the callee-allocated buffer a much more attractive and convenient mechanism.

Symbol Table

Explicit, function overloading.

With a few minor modifications we are going to reuse the hash table and the string buffer code to implement the symbol table. But first we'll make the size of the hash table settable during construction (not that it matters much).

```
explicit HTable (int size): _size (size)
{
    _aList = new List [size];
}

~HTable ()
{
    delete []_aList;
}
```

Notice the keyword explicit in front of the constructor. I put it there to tell the compiler not to use this constructor to do implicit conversions. Normally, whenever you define a constructor that takes a single argument of type T, the compiler can use it to quietly convert an object of type T to an object of the class whose constructor it is. For instance, since the constructor of the hash table takes a single argument of type int, the compiler would accept a statement like this:

```
HTable table (13);
table = 5;
```

It would just "convert" an integer, 5, to a hash table as easily as it converts an integer to a double. Such implicit conversions are sometimes extremely useful--at other times they are a source of unexpected problems. The code above makes little sense and the error would be easy to spot. Imagine however that somebody, by mistake, passed an int to a function expecting a const reference to a hash table. The compiler would actually create a temporary hash table of the size equal to the value of the integer and pass it to the function. To protect yourself from this kind of errors you should make it a habit to declare single-argument constructors explicit--that is, unless you really want to define an implicit conversion.

Another modification we'd like to make to our hash table is to allow the method Find (and also ForceAdd) to take the length of the string as a parameter. It so happens that when the parser calls these methods, it has the length handy, so it's a waste of time to calculate it again inside these methods.

This is easily done, but what if we have to find a string in a hash table and we don't have its length handy? Of course, we could always call strlen and pass the result to the appropriate method. There's a simpler solution, though. We can define an alternative version of Find which does not take the length of the string, but calculates it internally and calls the other version. Like this:

```
List const & Find (char const * str) const
{
    return Find (str, strlen (str));
}
```

Notice that we didn't have to invent different names for these two version of Find. It's enough if they differ by the number or type of arguments--the compiler will figure out which one to call. This mechanism of defining more than one method with the same name is called overloading.

In C++ you can overload free functions, methods, as well as constructors. For instance, for clients who don't have a preference for the size of the hash table, or who want to create arrays of hash tables, we could add a second constructor HTable () with no arguments. It would be implemented exactly like the first constructor, only it would arbitrarily set the size to, say, 127 (remember our first implementation of the hash table?). There is, however, a simpler way to accomplish the same thing. Instead of defining two constructors, you could specify the default value for its argument. Such declaration would look like this:

```
HTable (int size = 127);
```

You can specify default values for arguments not only in constructors, but also in free functions and methods. The rule is that, in the declaration, the arguments with default values have to follow the ones without defaults. When you call such a function with fewer than the full set of arguments, the missing trailing arguments will be set to their defaults. Here's a simple example.

```
void fun (int x, double y = 0.0, char * str = "");
// different ways of calling fun
fun (10); // y = 0.0, str = ""
fun (5, 1.2) // str = "";
fun (1, 0.2, "hello!");
```

The symbol table uses a hash table to map strings into short lists of string id's. New strings are stored in the string buffer at the first available offset, _curStrOff, and given the first available id, _curId. This offset is then stored in the array _offStr under the index equal to the string's id.

```
const int idNotFound = -1;

class SymbolTable
{
public:
    explicit SymbolTable (int size);
    ~SymbolTable ();
    int ForceAdd (char const * str, int len);
    int Find (char const * str, int len) const;
    char const * GetString (int id) const;
private:
    HTable    _htab;
    int     * _offStr; // offsets of strings in buffer
    int       _size;
    int       _curId;
    char    * _strBuf;
    int       _bufSize;
    int       _curStrOff;
};
```

Here are the constructor and the destructor of the symbol table. Other methods are just copied from our earlier implementation of the string buffer.

```
SymbolTable::SymbolTable (int size)
    : _size (size), _curId (0), _curStrOff (0), _htab (size + 1)
{
    _offStr = new int [size];
    _bufSize = size * 10;
    _strBuf = new char [_bufSize];
}

SymbolTable::~SymbolTable ()
{
    delete []_offStr;
    delete []_strBuf;
}
```

Store

Forward declarations.

Our calculator can deal with symbolic variables. The user creates a variable by inventing a name for it and then using it in arithmetic operations. Every variable has to be initialized—assigned a value in an assignment expression—before it can be used in evaluating other expressions. To store the values of user defined variables our calculator will need some kind of "memory." We will create a class Store that contains a fixed number, size, of memory cells. Each cell can store a value of the type double. The cells are numbered from zero to size-1. Each cell can be in either of two states—uninitialized or initialized.

```
enum { stNotInit, stInit };
```

The association between a symbolic name—a string—and the cell number is handled by the symbol table. For instance, when the user first introduces a given variable, say *x*, the string "x" is added to the symbol table and assigned an integer, say 3. From that point on, the value of the variable *x* will be stored in cell number 3 in the Store object.

We would also like to pre-initialize the symbol table and the store with some useful constants like *e* (the base of natural logarithms) and *pi* (the ratio of the circumference of a circle to its diameter). We would like to do it in the constructor of Store, therefore we need to pass it a reference to the symbol table. Now here’s a little snag: We want to put the definition of the class Store in a separate header file, ***store.h***. The definition of the class SymbolTable is in a different file, ***symtab.h***. When the compiler is looking at the declaration of the constructor of Store

```
    Store (int size, SymbolTable & symTab);
```

it has no idea what SymbolTable is. The simple-minded solution is to include the file ***symtab.h*** in ***store.h***. There is nothing wrong with doing that, except for burdening the compiler with the processing of one more file whenever it is processing ***symtab.h*** or any file that includes it. In a really big project, with a lot of header files including one another, it might become a real headache. If you are using any type of dependency checker, it will assume that a change in ***symtab.h*** requires the recompilation of all the files that include it directly or indirectly. In particular, any file that includes ***store.h*** will have to be recompiled too. And all this unnecessary processing just because we ed to let the compiler know that SymbolTable is a name of a class? Why don’t we just say that? Indeed, the syntax of such a ***forward declaration*** is:

```
class SymbolTable;
```

As long as we are *only* using pointers or references to SymbolTable, this will do. We don’t need to include ***symtab.h***. On the other hand, a forward declaration would not be sufficient if we wanted to call any of the methods of SymbolTable (including the constructor or the destructor) or if we tried to embed or inherit from SymbolTable.

```
class SymbolTable; // forward declaration

class Store
{
public:
    Store (int size, SymbolTable & symTab);
    ~Store ()
    {
        delete []_cell;
        delete []_status;
    }
    bool IsInit (int id) const
    {
        return (id < _size && _status [id] != stNotInit);
    }
    double Value (int id) const
    {
        assert (IsInit (id));
        return _cell [id];
    }
    void SetValue (int id, double val)
    {
        if (id < _size)
        {
            _cell [id] = val;
            _status [id] = stInit;
        }
    }
private:
    int              _size;
    double         * _cell;
    unsigned char  * _status;
};
```

Store contains two arrays. The array of cells and the array of statuses (initialized/uninitialized). They are initialized in the constructor and deleted in the destructor. We also store the size of these arrays (it's used for error checking). The client of Store can check whether a given cell has been initialized, get the value stored there, as well as set (and initialize) this value.

Why am I using two arrays instead of a single array of two-field objects? I could have defined a class Cell

```
class Cell
{
    // …
private:
    double            _value;
    unsigned char     _status;
};
```

I could have had a single array of Cell 's in Store instead of two separate arrays for values and statuses.

Both solutions have their merits. Using Cell improves code understandability and maintenance. On the other hand, using two separate arrays may be more space efficient. How come? Because of the *alignment* of data.

On most computer architectures it is cheaper to retrieve data from memory if it's aligned on 32-bit or 64-bit boundary. So, unless you specify otherwise, the compiler will force the alignment of user-defined data structures on such boundaries. In particular, it might add padding to your data structures. It's very likely that the compiler will add three (or even seven) bytes of padding to every Cell. This way, the size of a Cell in memory will be guaranteed to be a multiple of 32 (or 64) bits, and accessing an array of Cells will be very fast.

As long as you're dealing with reasonably small arrays, you shouldn't care. But if your program uses lots of memory, you might want to be more careful. In our implementation we tried to reduce the memory overhead by declaring the array _status as an array of bytes (unsigned chars). In fact, we could get away with an array of bits. After all, status is a two-valued variable and it can be encoded using a single bit. Implementing a bit array is nontrivial and involves explicit bit-wise operations on every access, but in some cases it might save you oodles of memory.

So here's a rule of thumb: If you anticipate that you might have to shrink your data structures because of memory constraints, implementing the shrinkable part separately will give you more flexibility. In our case, we could, in the future, re-implement the _status array as a bit array without changing much code around it. On the other hand, if you anticipate that you might have to add more *under-sized* variables to each element of the array, you'd be better off combining them into a single class. In our case, if we expected that we might, say, need some kind of type enumeration for every variable, we could easily add it to the Cell class.

Making choices like this is what makes programming interesting.

The constructor of Store is defined in the source file ***store.cpp***. Since the constructor calls actual methods of the SymbolTable, the forward declaration of this class is no longer sufficient and we need to explicitly include the header ***symtab.h*** in ***store.cpp***.

```
Store::Store (int size, SymbolTable & symTab): _size (size)
{
    _cell = new double [size];
    _status = new unsigned char [size];
    for (int i = 0; i < size; ++i)
        _status [i] = stNotInit;
    // add predefined constants
    // Note: if more needed, do a more general job
    cout << "e = " << exp(1) << endl;
    int id = symTab.ForceAdd ("e", 1);
    SetValue (id, exp (1));
    cout << "pi = " << 2 * acos (0.0) << endl;
    id = symTab.ForceAdd ("pi", 2);
    SetValue (id, 2.0 * acos (0.0));
}
```

We add the mapping of the string "e" of size 1 to the symbol table and then use the returned integer as a cell number in the call to SetValue. The same procedure is used to initialize the value of "pi."

Function Table

Meta-programs. Pointers to functions. Explicit array initialization. Explicit class initialization.

For a good scientific calculator, built-in functions are a must. We have to be able to calculate square roots, logarithms, trigonometric functions, etc. We are quite lucky because the standard C library implements most of the basic math functions. All we need is for the parser to recognize the function call and then call the appropriate library function. The only tricky part is to make the connection between the function name—the string recognized by the parser—and the call to the appropriate function.

One way would be to create a multi-way conditional that compares the string to a list of predefined function names, and, when successful, calls the corresponding function.

```
if (strcmp (string, "sin") == 0)
{
    result = sin (arg);
}
else if ...
...
```

As long as the number of built-in functions is reasonably small, this solution is good enough. Let's pretend though that, instead of a toy calculator, we are writing an industrial strength program that will have to handle hundreds, if not thousands, of built in functions. The problem than becomes: Given a string, match it against hundreds of pre-defined strings. Clearly, doing hundreds of string comparisons every time is not acceptable.

But wait! We already have a string-matching object--the symbol table. Since it's implemented as a hash table, it can perform a match in constant time, independent of the size of the table. The symbol table converts a string into an integer. We can pre-fill the table with built in function names (just like we did with built-in constants) and dispatch the function calls based on integers rather than strings. We could, for instance, insert the string "sin" in the zeroth slot of the symbol table, "cos" in the first slot, etc., and then dispatch calls using a switch statement:

```
case 0:
    result = sin (arg);
    break;
case 1:
    result = cos (arg);
    break;
...
```

A switch statement that uses a set of consecutive labels, 0, 1, 2, etc., is implemented by the compiler as a jump table with a constant switching time independent of the number of labels. Seems like a perfect, constant-time solution.

But how do we make sure that sin always corresponds to 0, cos to 1, etc.? Well, we can always initialize the symbol table with these strings in this particular order. After all, we know that the symbol table assigns consecutive indexes starting with zero. Is it okay though? These are implementation secrets of the symbol table. What if the next guy to maintain our program rewrites the symbol table to use an even better algorithm? One that does not assign consecutive indexes starting with zero?

And how about expandability of such code? Suppose that at some point in the future we will want to add one more built-in function. What changes will we have to make to this implementation? We'll have to:

- Add one more case to the switch statement,
- Add one more string to the symbol table, and
- Make sure that the string is inserted at the offset that is the case label of the corresponding function.

Notice what we have just done. We have written a meta-program: a set of instructions to be followed in order to add a new built-in function to our calculator. These are not machine instructions, these are instructions for the programmer. In fact, they don't appear anywhere in the program, even as comments—they are implicit. This kind of invisible meta-code adds to the hidden complexity of a program. What does meta-code describe? It describes the steps to be taken to implement the most likely extension to the program, as well as the assertions that have to be preserved when making such extensions.

When comparing various implementations, take into account not only the complexity of code but also the complexity of meta-code.

Let's see if we can find a better implementation for built-in functions. Optimally, we would like to have some kind of a table listing all the functions. (It is almost always a good idea to shift the complexity from code to data structures.) Adding a new function would be equivalent to adding a new entry to this table. The meta-program for such an implementation would consist of a single statement:

- Add a new entry to the function array

In order to make connection between data and executable code we need a new device: ***pointers to functions***. A pointer to function is just like a regular pointer, only that instead of pointing to data it points to executable code. By using a pointer to data, you can read or modify data; by using a pointer to function you can call functions. Just like with any other type of pointer, we'll have to be able to

- declare a pointer to function,

- initialize a pointer to point to a particular function,
- make a function call using the pointer.

For instance, we may declare a pointer to function taking a `double` (as an argument) and returning a `double`. There are many such functions and we can initialize the pointer to point to any one of these. In particular we may initialize the pointer to point to the function `double sin (double x)`. After that, if we make a function call through our pointer, we will be calling `sin`. Had we initialized the pointer to point to `double cos (double x)`, we would have called `cos` instead. The beauty of a pointer to function is that the same pointer may point to different functions during the execution of the program. The part of the program that makes the function call doesn't have to know what function it is calling.

Let's see some actual code to see what the syntax looks like. First let's declare a pointer `pFun` to a function taking a `double` and returning a `double`:

```
double (* pFun) (double x);
```

Compare it with the declaration of a function `sin` taking a `double` and returning a `double`:

```
double sin (double x);
```

Why did we have to enclose the asterisk and the pointer name within parentheses? We had to do it in order to distinguish the declaration of a pointer to function from a declaration of a function returning a pointer. Without the parentheses

```
double * pFun (double x);
```

declares a function taking a `double` and returning a pointer to `double`. Quite a difference!

You can declare a pointer to any type of a function by taking the declaration of any function of this type and replacing the function name with `(* pFun)`. Of course, you can use an arbitrary name instead of `pFun`.

Remember the function `strtod`? Its declaration was:

```
double strtod (char const * str, char ** ppEnd);
```

A pointer to a function of this type would be declared as:

```
double (* pStrtod) (char const * str, char ** ppEnd);
```

It's that simple!

Because of pointers to functions, we have to augment the rules of reading declarations. We start reading backwards, as usual: `pStrtod` *is a pointer...* But when we hit the parenthesis, we immediately go to the matching parenthesis and change the direction of parsing. The first thing we see now is a left parenthesis that starts the argument list. A parenthesized argument list means that we are dealing with a function, so we continue reading *...to a function taking* `str` *which is a pointer to a* `const char` *and* `ppEnd` *which is a pointer to a pointer to a* `char`. Here we hit the closing parenthesis, so we zigzag back to where we left our leftward scanning and continue: *...returning double*. Similarly, when we encounter a left square bracket, we know we are dealing with an array, so, for instance

```
    int (* pArr) [3];
```

reads: <- pArr *is a pointer -> to an array of three <- integers*. (The arrows mark the change of direction.) Let's try something more complicated:

```
double (* (* pFunArr) [6]) (double x);
```

<- pFunArr *is a pointer -> to an array of six <- pointers -> to functions taking double* x *<- and returning a double*. Finally, as an exercise to the reader, a nice little declaration:

```
    double (* (* pFunFunFun) (double (* pFun)(double x)) (double
x);
```

Please, don't try it at home!

Now for the second step: the initialization. In order to make the pointer point to a function, we have to take the address of the function and assign it to the pointer. It so happens that the name of the function *is* the address of the function. We can therefore initialize our pointer like this:

```
double (* pFun) (double x) = sin;
```

or do the assignment at a later point:

```
pFun = sin;
```

Finally, in order to invoke a function through a pointer, we dereference the pointer and pass the appropriate argument(s):

```
double x = 3.1415;
double y = (* pFun) (x);
```

Pointers to functions can be used as building blocks in more complex data structures such as: arrays, classes, pointers to pointers, etc. To simplify the declarations of such data structures, type definitions are often used.

We start building our implementation of the built-in function table with a type definition:

```
typedef double (*PtrFun) (double);
```

All our built-in functions take one argument of the type double and return a result which is also double. The connection between a function and its string name is established in a class FunctionEntry.

```
class FunctionEntry
{
public:
    PtrFun pFun;
    char* strFun;
};
```

You might be wondering why we have made all the data members of this class public and provided no constructor to initialize them. That's because we want to be able to initialize them explicitly, like this:

```
FunctionEntry funEntry = { sin, "sin" };
```

This kind of initialization, where you assign values directly to the object's data members, is possible only if and only if the object has no private data and no constructor. Still, I haven't explained why anyone would want to use this kind of direct initialization instead of simply providing an appropriate constructor? Here's why:

```
const int maxIdFun = 16;

FunctionEntry funArr[maxIdFun] =
{
    log,    "log",
    log10,  "log10",
    exp,    "exp",
    sqrt,   "sqrt",
    sin,    "sin",
    cos,    "cos",
    tan,    "tan",
    CoTan,  "cotan",
    sinh,   "sinh",
    cosh,   "cosh",
    tanh,   "tanh",
    asin,   "asin",
    acos,   "acos",
    atan,   "atan",
    0,      ""
};
```

In one big swoop we've been able to initialize the whole array of FunctionEntries. That's exactly what we wanted. Notice how easy it will be now to add a new built-in function: just add one more line with function pointer and string name between any two lines in this list and it'll just work (you'll need to increment maxIdFun as well, but that's something the compiler will remind you of, if you forget). If you want to know more about explicit initialization of aggregates (such as class objects and arrays), I will talk more about it at the end of this chapter.

Now that we are through with the preliminaries, let's get back to our original problem: the initialization of the symbol table with built-in function names. This should be done in the construction stages of the program—the calculator is not ready to be used unless the symbol table is pre-filled. We will therefore define an object called the FunctionTable with the dual purpose of initializing the symbol table with the names and translating symbol id's to function pointers for all the built-in functions.

```
class SymbolTable;

class FunctionTable
{
public:
    FunctionTable (SymbolTable & symTab, FunctionEntry funArr []);
    int Size () const { return _size; }
    PtrFun GetFun (int id) { return _pFun [id]; }
private:
    PtrFun  _pFun [maxIdFun];
    int   _size;
};
```

The constructor of the FunctionTable takes a reference to the symbol table and fills it with strings from our array of FunctionEntries. At the same time it copies corresponding function pointers into its own array (strictly speaking the copying could be avoided if we let the function table use the function array directly—that's an optimization that is best left as an exercise to the reader).

```
FunctionTable::FunctionTable(SymbolTable &symTab,FunctionEntry funArr
[])
    : _size (0)
{
    for (int i = 0; i < maxIdFun; ++i)
    {
        int len =  strlen (funArr [i].strFun);
        if (len == 0)
            break;
        _pFun [i] = funArr [i].pFun;
        cout << funArr[i].strFun << endl;
        int j = symTab.ForceAdd (funArr [i].strFun, len);
        assert (i == j);
        ++_size;
    }
}
```

Notice the very important assertion in the loop. We are making sure that the assumption that the symbol table assigns consecutive indexes to our functions is indeed true. We want the string "log" to correspond to id 0, because we store the pointer to the function log at offset 0, etc.

All of the built-in functions so far have been library functions (defined in the standard C library) except for one, the cotangent. The cotangent is just the inverse of the tangent, so it is easy to write our own implementation of it. The only tricky part is dealing with the inverse of zero—division by zero causes an exception, which would terminate the program. That's why we test for zero and return HUGE_VAL as its inverse. That's not entirely correct (the result of division by zero is undefined), but it'll do for now.

```
double CoTan (double x)
{
    double y = tan (x);
    if (y == 0)
    {
        cout << "cotan of " << x << " undefined\n";
        return HUGE_VAL;
    }
    return 1.0 / y;
}
```

Note that the appropriate line in the initialization of funArr

```
CoTan, "cotan",
```

contains the pointer to our own implementation of cotangent, namely CoTan.

To summarize: We went through quite a bit of trouble in order to produce an implementation that avoids some of the hidden complexities of something we called *meta-code*. In this particular case, it was well worth the effort because we knew that the table of built-in functions would be the first thing to be modified in the future. We made sure such modifications would be easy and virtually fool-proof. We have localized them in one place (the initialization of funArr, made trivial mistakes compiler detectable (forgetting to update maxIdFun) and positioned an assertion in a strategic place (inside the FunctionTableconstructor).

Nodes

We have to add a few more node types. Corresponding to built-in functions are the objects of type FunNode. A FunNode contains a pointer to function and a child node—the argument to the function. I have a confession to make: When I was writing this program I forgot to add the destructor to FunNode. The program worked fine, it was just leaking memory. Every time a FunNode was constructed and then destroyed, it would leave its child node orphaned. The memory for the whole subnode would never be reclaimed. If you kept this program running for hours and hours (which I never did), it would eventually chew up all its memory and die. I caught this bug by instrumenting the memory allocator when I was writing the second part of the book—*The Techniques*.

```
class FunNode: public Node
{
public:
    FunNode (PtrFun pFun, Node * pNode)
        : _pNode (pNode), _pFun (pFun)
    {}
    ~FunNode () { delete _pNode; }
    double Calc () const;
private:
    Node * const _pNode;
    PtrFun       _pFun;
};
```

The Calc method illustrates the syntax for the invocation of a function through a pointer.

```
double FunNode::Calc () const
{
    assert (_pFun != 0);
    return (*_pFun)(_pNode->Calc ());
}
```

The assignment node is a little trickier. It is a binary node with the children corresponding to the left hand side and the right hand side of the assignment. For instance, the parsing of the line

```
x = 1
```

will produce an assignment node with the left node corresponding to the symbolic variable x and the right node corresponding to the value 1. First of all, not every node can be a left hand side of an assignment. For instance

```
1 + 2 = 1
```

is clearly not acceptable. We have to have a way of deciding whether a given node can appear on the left hand side of an assignment—whether it can be an ***lvalue***. In our calculator, the only possible lvalue is a symbolic variable.

In ordert to be able to verify if a given node is an lvalue, we will add the virtual method IsLvalue to the base class Node and provide the default implementation

```
bool Node::IsLvalue () const
{
    return 0;
}
```

The only type of node that will override this default will be the symbolic-variable node. We'll make the parser perform the IsLvalue test on the left hand side of the assignment before creating the AssignNode. Here, inside the assignment node, we'll only assert that the parser did its job.

```
class AssignNode : public BinNode
{
public:
    AssignNode (Node * pLeft, Node * pRight)
        : BinNode (pLeft, pRight)
    {
        assert (pLeft->IsLvalue ());
    }
    double Calc () const;
};
```

The Calc method calls Assign method of the left child with the value calculated by the right child. Again, we will add the virtual method Assign to the base class Node, together with the default implementation that does nothing.

```
virtual void Assign (double value) {}
```

Only the variable node will override this implementation.

Having said that, the implementation of AssignNode::Calc is straightforward

```
double AssignNode::Calc () const
{
    double x = _pRight->Calc ();
    _pLeft->Assign (x);
    return x;
}
```

Next, we have to define the node corresponding to a symbolic variable. The value of the variable is stored in the Store object. VarNode has to have access to this object in order to calculate itself. But, as we know, VarNode can also be used on the left hand side of an assignment, so it has to override the virtual methods IsLvalue and Assign.

```
class Store;

class VarNode: public Node
{
```

```
public:
    VarNode (int id, Store & store)
        : _id (id), _store (store) {}
    double Calc () const;
    bool IsLvalue () const;
    void Assign (double val);
private:
    const int _id;
    Store & _store;
};

double VarNode::Calc () const
{
    double x = 0.0;
    if (_store.IsInit (_id))
        x = _store.Value (_id);
    else
        cout << "Use of uninitialized variable\n";
    return x;
}

void VarNode::Assign (double val)
{
    _store.SetValue (_id, val);
}

bool VarNode::IsLvalue () const
{
    return true;
}
```

Parser

Private methods.

The Parser requires a major makeover. Let’s start with the header file, ***parse.h***. It contains a lot of forward declarations

```
class Node;
class Scanner;
class Store;
class FunctionTable;
class SymbolTable;
```

All these classes are mentioned in ***parse.h*** either through pointers or references—there is no need to include the header files that define them.

The class Parser has a constructor that takes references to all the major objects it will need in order to parse the stream of tokens from the Scanner. It needs the Store to create VarNodes, FunctionTable to create FunNodes and the SymbolTable to recognize variables and function names. Once the Parser is created, we can only ask it to evaluate the expression passed to it in the Scanner. The Eval method has the side effect of printing the result on the screen (if you're cringing now, I promise you'll be relieved when you read the next chapter of this book).

Parser has a number of private methods that it uses to parse expressions, terms and factors; and a method to execute the evaluation of the expression tree. Private methods are a useful device for code structuring. They cannot be called from outside of the class—they are just helper functions used in the implementations of other, public or private, methods.

```
class Parser
{
public:
    Parser (Scanner & scanner,
        Store & store,
        FunctionTable & funTab,
        SymbolTable & symTab);
    ~Parser ();
    Status Eval ();
private:
    void   Parse();
    Node * Expr();
    Node * Term();
    Node * Factor();
    void   Execute ();

    Scanner        & _scanner;
    Node           * _pTree;
    Status           _status;
    Store          & _store;
    FunctionTable  & _funTab;
    SymbolTable    & _symTab;
};
```

The constructor (we are already looking at the implementation file, ***parse.cpp***) initializes all the private variables.

```
Parser::Parser (Scanner & scanner,
        Store & store,
        FunctionTable & funTab,
        SymbolTable & symTab )
: _scanner (scanner),
  _pTree (0),
  _status (stOk),
  _funTab (funTab),
  _store (store),
  _symTab (symTab)
{
}
```

The destructor deletes the parse tree:

```
Parser::~Parser ()
{
    delete _pTree;
}
```

The Eval method calls private methods to parse the input and execute the calculation:

```
Status Parser::Eval ()
{
    Parse ();
    if (_status == stOk)
        Execute ();
    else
        _status = stQuit;
    return _status;
}
```

Execute calls the Calc method of the top node of the parse tree and prints the result of the (recursive) calculation

```
void Parser::Execute ()
{
    if (_pTree)
    {
        double result = _pTree->Calc ();
        cout << "  " << result << endl;
    }
}
```

The parsing starts with the assumption that whatever it is, it’s an expression.

```
void Parser::Parse ()
{
    _pTree = Expr ();
}
```

The expression starts with a term. The term can be followed by one of the operators that bind terms: the plus sign, the minus sign, the assignment sign; or nothing at all. These correspond to the four productions:

```
Expression is Term + Expression
or Term - Expression
or Term = Expression  // the Term must be an lvalue
or just Term
```

That’s why, after parsing a term we ask the Scanner for the next token and test it against tPlus, tMinus and tAssign. Immediately after a token is recognized we tell the Scanner to accept it.

```
Node * Parser::Expr ()
{
    Node * pNode = Term ();
    EToken token = _scanner.Token ();
    if (token == tPlus)
    {
        _scanner.Accept ();
        Node * pRight = Expr ();
        pNode = new AddNode (pNode, pRight);
    }
    else if (token == tMinus)
    {
        _scanner.Accept ();
        Node * pRight = Expr ();
        pNode = new SubNode (pNode, pRight);
    }
    else if (token == tAssign)
    {
        _scanner.Accept ();
        Node * pRight = Expr ();
        if (pNode->IsLvalue ())
        {
            pNode = new AssignNode (pNode, pRight);
        }
        else
        {
            _status = stError;
            delete pNode;
            pNode = Expr ();
        }
    }
    return pNode;
}
```

We proceed in a similar fashion with the `Term`, following the productions

```
Term is Factor * Term
or Factor / Term
or just Factor
```

```
Node * Parser::Term ()
{
    Node * pNode = Factor ();
    if (_scanner.Token () == tMult)
    {
        _scanner.Accept ();
        Node * pRight = Term ();
        pNode = new MultNode (pNode, pRight);
    }
    else if (_scanner.Token () == tDivide)
    {
        _scanner.Accept ();
        Node * pRight = Term ();
        pNode = new DivideNode (pNode, pRight);
    }
    return pNode;
}
```

A `Factor`, in turn, can be one of these

```
Factor is ( Expression ) // parenthesized expression
or Number // literal floating point number
or Identifier ( Expression ) // function call
or Identifier // symbolic variable
or - Factor  // unary minus
```

```
Node * Parser::Factor ()
```

```
{
    Node * pNode;
    EToken token = _scanner.Token ();

    if (token == tLParen)
    {
        _scanner.Accept (); // accept '('
        pNode = Expr ();
        if (_scanner.Token() != tRParen)
            _status = stError;
        _scanner.Accept (); // accept ')'
    }
    else if (token == tNumber)
    {
        pNode = new NumNode (_scanner.Number ());
        _scanner.Accept ();
    }
    else if (token == tIdent)
    {
        char strSymbol [maxSymLen + 1];
        int lenSym = maxSymLen;
        // copy the symbol into strSymbol
        _scanner.SymbolName (strSymbol, lenSym);
        int id = _symTab.Find (strSymbol, lenSym);
        _scanner.Accept ();
        if (_scanner.Token() == tLParen) // function call
        {
            _scanner.Accept (); // accept '('
            pNode = Expr ();
            if (_scanner.Token () == tRParen)
                _scanner.Accept (); // accept ')'
            else
                _status = stError;
            if (id != idNotFound && id < _funTab.Size ())
            {
                pNode = new FunNode (
                    _funTab.GetFun (id), pNode);
            }
            else
            {
                cout << "Unknown function \"";
                cout << strSymbol << "\"\n";
            }
        }
        else
        {
            if (id == idNotFound)
                id = _symTab.ForceAdd (strSymbol, lenSym);
            pNode = new VarNode (id, _store);
        }
    }
    else if (token == tMinus) // unary minus
    {
        _scanner.Accept (); // accept minus
        pNode = new UMinusNode (Factor ());
    }
    else
    {
        _scanner.Accept ();
        _status = stError;
        pNode = 0;
    }
    return pNode;
}
```

To see how the parser works, let’s single step through a simple example. Suppose that the user typed

```
x = 1
```

First, a scanner containing the input string is created. It scans the first token and decides that it's an identifier, tIdent. Next, the parser is called to Eval the expression. It starts parsing it by assuming that it is, indeed, an expression

```
_pTree = Expr ();
```

Expression must start with a term, so Expr calls Term

```
Node * pNode = Term ();
```

Term, on the other hand, expects to see a Factor

```
Node * pNode = Factor ();
```

Finally Factor looks at the first token

```
EToken token = _scanner.Token ();
```

and compares it to tLParen, tNumber, tIdent and tMinus--in our case the first token is an identifier. It then ask the scanner for the name of the symbol (it is "x") and searches for it in the symbol table. It's done with the first token so it tells the scanner to accept it.

```
// copy the symbol into strSymbol
_scanner.SymbolName (strSymbol, lenSym);
int id = _symTab.Find (strSymbol, lenSym);
_scanner.Accept ();
```

The scanner scans for the next token, which is the assignment operator tAssign. The parser looks at this token to see if it a left parenthesis--that would signify a function call. Since it isn’t, it creates a VarNode using the id of the symbol found in the symbol table. If this is a new symbol for which there wasn’t any id, it just adds it to the symbol table and creates a new id.

```
if (id == idNotFound)
    id = _symTab.ForceAdd (strSymbol, lenSym);
pNode = new VarNode (id, _store);
```

Factor is now done and it returns a pointer to the node it has just created. Back to Term. It has a node, so now it looks at the next token to see if it is one of tMult or tDivide. Since it’s not, it returns the same node. Back to Expr. Again, a look at the token: is it tPlus, tMinus or tAssign? It is tAssign! It accepts the token (the scanner positions itself at tNumber whose value is 1). So far, the parser has seen a node followed by an equal sign. An equal sign can be followed by an arbitrary expression, so Expr now calls itself to parse the rest of the input which is "1".

```
_scanner.Accept ();
Node* pRight = Expr ();
```

Expr again goes through Term and Factor, which creates a NumNode

```
pNode = new NumNode (_scanner.Number ());
_scanner.Accept ();
```

Back to Term and Expr with the new node pRight. The parser has now seen a node, pNode, followed by the equal sign and another node, pRight. It looks very much like an assignment statement, except that we don't know if pNode represents an lvalue. In our case, the node being a VarNode, it does. An assignment node is created:

```
if (pNode->IsLvalue ())
{
    pNode = new AssignNode (pNode, pRight);
}
```

Expr returns this AssignNode to Parse. The parse tree now looks like this (Figure 2-7)

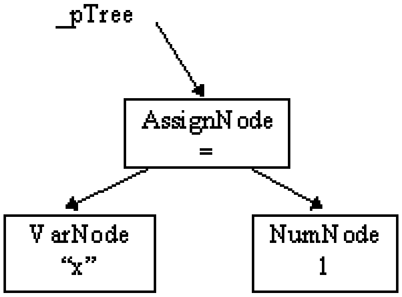


Figure 2-7

Since the parsing was successful, we call Execute, which calls _pTree->Calc () . The virtual Calc method of AssignNode calculates the value of the right node—NumNode returns 1—and call the Assign method of VarNode with this value. VarNode has access to the Store and stores the value of 1 in the slot corresponding to "x".

Main

The main function drives the whole thing. Its overall structure hasn't changed much since our last iteration--except for the addition of the function table, the symbol table and the store.

```
const int maxBuf = 100;
const int maxSymbols = 40;

int main ()
```

```
{
    // Notice all these local objects.
    // A clear sign that there should be
    // a top level object, say, the Calculator.
    // Back to the drawing board!

    char buf [maxBuf];
    Status status;
    SymbolTable symTab (maxSymbols);
    FunctionTable funTab (symTab, funArr);
    Store store (maxSymbols, symTab);

    do
    {
        cout << "> ";  // prompt
        cin.getline (buf, maxBuf);
        Scanner scanner (buf);
        Parser  parser (scanner, store, funTab, symTab);
        status = parser.Eval ();
    } while (status != stQuit);
}
```

Notice that for every line of input we create a new scanner and a new parser. We keep however the same symbol table, function table and the store. This is important because we want the values assigned to variables to be remembered as long as the program is active. The parser’s destructor is called after the evaluation of every line. This call plays an important role of freeing the parse tree.

There is a comment at the top of `main`, which hints at ways of improving the structure of the program. There are five local variables/objects defined at the top of `main`. Moreover, they depend on each other: the symbol table has to be initialized before the function table and before the store. The parser’s constructor takes references to three of them. As a rule of thumb, whenever you see too many local variables, you should think hard how to combine at least some of them into a separate object. In our case, it's pretty obvious that this object sould be called `Calculator`. It should combine `SymbolTable`, `FunctionTable` and `Store` as its embeddings.

We’ll come back to this program in the next part of the book to see how it can be made into a professional, "industrial strength" program.

Initialization of Aggregates

Explicit initialization of classes and arrays.

Just as you can explicitly initialize an array of characters using a literal string

```
char string [] = "Literal String";
```

you can initialize other aggregate data structures--classes and arrays. An object of a given class can be explicitly initialized if and only if all its non-static data members are public and there is no base class, no virtual functions and no user-defined constructor. All public data members must be explicitly initializable as well (a little recursion here). For instance, if you have

```
class Initializable
{
public:
    // no constructor
    int          _val;
    char const * _string;
    Foo        * _pFoo;
};
```

you can define an instance of such class and initialize all its members at once

```
Foo foo;
Initializable init = { 1, "Literal String", &foo };
```

Since Initializable is initializable, you can use it as a data member of another initializable class.

```
class BigInitializable
{
public:
    Initializable   _init;
    double          _pi;
};

BigInitializable big = { { 1, "Literal String", &foo }, 3.14 };
```

As you see, you can nest initializations.

You can also explicitly initialize an array of objects. They may be of a simple or aggregate type. They may even be arrays of arrays of objects. Here are a few examples.

```
char string [] = { 'A', 'B', 'C', '\0' };
```

is equivalent to its shorthand

```
char string [] = "ABC";
```

Here's another example

```
Initializable init [2] = {
        { 1, "Literal String", &foo1 },
        { 2, "Another String", &foo2 } };
```

We used this method in the initialization of our array of FuncitionEntry objects.

If objects in the array have single-argument constructors, you can specify these arguments in the initializer list. For instance,

```
CelestialBody solarSystem = { 0.33, 4.87, 5.98, 0.64, 1900, 569, 87,
103, 0.66 };
```

where masses of planets are given in units of 10^{24}kg.

Exercises

1. Create a top level Calculator object and make appropriate changes to lower level components.
2. Add two new built in functions to the calculator, sqr and cube. Sqr squares its argument and cube cubes it (raises to the third power).
3. Add the recognition of unary plus to the calculator. Make necessary modifications to the scanner and the parser. Add a new node, UPlusNode. The calculator should be able to deal correctly with such expressions as

   ```
   x = +2
   2 * + 7
   1 / (+1 - 2)
   ```

4. Add powers to the calculator according to the following productions

   ```
   Factor is SimpleFactor ^ Factor // a ^ b (a to the power of b)
         or SimpleFactor

   SimpleFactor is ( Expression ) // parenthesized expression
         or Number    // literal floating point number
         or Identifier ( Expression )// function call
         or Identifier// symbolic variable
         or - Factor  // unary minus
   ```

5. To all nodes in the parse tree add virtual method Print(). When Print() is called on the root of the tree, the whole tree should be displayed in some readable form. Use varying indentation (by printing a number of spaces at the beginning of every line) to distinguish between different levels of the tree. For instance

   ```
   void AddNode::Print (int indent) const
   {
       _pLeft->Print (indent + 2);
       Indent (indent);
       cout << "+" << endl;
       _pRight->Print (indent + 2);
       cout << endl;
   }
   ```

   where Indent prints as many spaces as is the value of its argument.
6. Derivatives of some of the built-in functions are, respectively

   ```
   sin(x) -> cos(x)
   cos(x) -> -sin(x)
   exp(x) -> exp(x)
   log(x) -> 1/x
   sqrt(x) -> 1/(2 * sqrt(x))
   ```

   The derivative of a sum is a sum of derivatives, the derivative a product is given by the formula

   ```
   (f(x) * g(x))’ = f’(x) * g(x) + f(x) * g’(x)
   ```

   where prime denotes a derivative. The derivative of a quotient is given by the formula

```
(f(x) / g(x))’ = (f(x) * g’(x) - f’(x) * g(x)) / (g(x) * g(x))
```

and the derivative of the superposition of functions is given by

```
(f(g(x))’ = g’(x) * f’(g(x)).
```

Rewrite the calculator to derive the symbolic derivative of the input by transforming the parse tree according to the formulas above. Make sure no memory is leaked in the process (that is, you must delete everything you allocate).

Operator overloading

You can pretty much do any kind of arithmetic in C++ using the built-in integral and floating-point types. However, that's not always enough. Old-time engineers swear by Fortran, the language which has built-in type *complex*. In a lot of engineering applications, especially in electronics, you can't really do effective calculations without the use of complex numbers.

C++ does not support complex arithmetics. Neither does it support matrix or vector calculus. Does that mean that engineers and scientists should stick to Fortran? Not at all! Obviously in C++ you can define new classes of objects, so defining a complex number is a piece of cake. What about adding, subtracting, multiplying, etc.? You can define appropriate methods of class `complex`. What about notational convenience? In Fortran you can add two complex numbers simply by putting the plus sign between them. No problem! Enter operator overloading.

In an expression like

```
double delta = 5 * 5 - 4 * 3.2 * 0.1;
```

you see several arithmetic operators: the equal sign, the multiplication symbol and the minus sign. Their meaning is well understood by the compiler. It knows how to multiply or subtract integers or floating-point numbers. But if you want to teach the compiler to multiply or subtract objects of some user-defined class, you have to *overload* the appropriate operators. The syntax for operator overloading requires some getting used to, but the use of overloaded operators doesn't. You simply put a multiplication sign between two complex variables and the compiler finds your definition of complex multiplication and applies it. By the way, a `complex` type is conveniently defined for you in the standard library.

An equal sign is an operator too. Like most operators in C++, it can be overloaded. Its meaning, however, goes well beyond arithmetics. In fact, if you don't do anything special about it, you can assign an arbitrary object to another object of the same class by simply putting an equal sign between them. Yes, that's right, you can, for instance, do that:

```
SymbolTable symTab1 (100);
SymbolTable symTab2 (200);
symTab1 = symTab2;
```

Will the assignment in this case do the sensible thing? No, the assignment will most definitely be wrong, and it will result in a very nasty problem with memory management. So, even if you're not planning on overloading the standard arithmetic operators, you should still learn something about the assignment operator; including when and why you would want to overload it. And that brings us to a very important topic--value semantics.

Passing by Value

Copy constructor, overloading the assignment operator, default copy constructor and operator =, return by value, passing by value, implicit type conversions.

So far we've been careful to pass objects from and to methods using references or pointers. For instance, in the following line of code

```
Parser  parser (scanner, store, funTab, symTab);
```

all the arguments to the `Parser`'s constructor--`scanner`, `store`, `funTab` and `symTab`--are passed by reference. We know that, because we've seen the following declaration (and so did the compiler):

```
Parser (Scanner & scanner,
        Store & store,
        FunctionTable & funTab,
        SymbolTable & symTab);
```

When we construct the parser, we don't give it a *copy* of a symbol table. We give it *access* to an existing symbol table. If we gave it a private copy, we wouldn't have been able to see the changes the parser made to it. The parser may, for instance, add a new variable to the symbol table. We want our symbol table to remember this variable even after the current parser is destroyed. The same goes for `store`--it must remember the values assigned to symbolic variables across the invocations of the parser.

But what about the scanner? We don't really care whether the parser makes a scratch copy of it for its private use. Neither do we care what the parser does to the function table. What we *do* care about in this case is performance. Creating a scratch copy of a large object is quite time consuming.

But suppose we didn't care about performance. Would the following work?

```
Parser (Scanner scanner,
        Store & store,
        FunctionTable funTab,
        SymbolTable & symTab);
```

Notice the absence of ampersands after `Scanner` and `FunctionTable`. What we are telling the compiler is this: When the caller creates a Parser, passing it a scanner, make a temporary copy of this scanner and let the Parser's constructor operate on that copy.

> This is, after all, the way built-in types are passed around. When you call a method that expects an integer, it's the copy of that integer that's used inside the method. You can modify that copy to your heart's content and you'll never change the original. Only if you explicitly request that the method accept a *reference* to an integer, can you change the original.

There are many reasons why such approach will not work as expected, unless we make several further modifications to our code. First of all, the temporary copy of the scanner (and the function table) will disappear as soon as the execution of the Parser's constructor is finished. The parser will store a reference to it in its member variable, but that's useless. After the end of construction the reference will point to a non-exitsent scratch copy of a scanner. That's not good.

If we decide to pass a copy of the scanner to the parser, we should also *store* a copy of the scanner inside the parser. Here's how you do it--just omit the ampersand.

```
class Parser
{
    ...
private:
    Scanner         _scanner;
    Node          * _pTree;
    Status          _status;
    Store         & _store;
    FunctionTable   _funTab;
    SymbolTable   & _symTab;
};
```

But what is really happening inside the constructor? Now that neither the argument, scanner, nor the member variable, _scanner, are references, how is _scanner initialized with scanner? The syntax is misleadingly simple.

```
Parser::Parser (Scanner scanner,
            Store & store,
            FunctionTable funTab,
            SymbolTable & symTab)
: _scanner (scanner),
  _pTree (0),
  _status (stOk),
  _funTab (funTab),
  _store (store),
  _symTab (symTab)
{
}
```

What happens behind the scenes is that Scanner's *copy constructor* is called. A copy constructor is the one that takes a (possibly const) reference to the object of the same class and clones it. In our case, the appropriate constructor would be declared as follows,

```
Scanner::Scanner (Scanner const & scanner);
```

But wait a minute! Scanner does *not* have a constructor of this signature. Why doesn't the compiler protest, like it always does when we try to call an undefined member function? The unexpected answer is that, if you don't explicitly declare a copy constructor for a given class, the compiler will create one for you. If this doesn't sound scary, I don't know what does.

Beware of default copy constructors!

The copy constructor generated by the compiler is probably wrong! After all, what can a dumb compiler know about copying user defined classes? Sure, it tries to do its best--it

- does a bitwise copy of all the data members that are of built-in types and

- calls respective copy constructors for user-defined embedded objects.

But that's it. Any time it encounters a pointer it simply duplicates it. It *does not* create a copy of the object pointed to by the pointer. That might be okay, or not--only the creator of the class knows for sure.

This kind of operation is called a *shallow copy*, as opposed to a *deep copy* which follows all the pointers. Shallow copy is fine when the pointed-to data structures can be easily shared between multiple instances of the object.

But consider, as an example, what happens when we make a shallow copy of the top node of a parse tree. If the top node has children, a shallow copy will not clone the child nodes. We will end up with two top nodes, both pointing to the same child nodes. That's not a problem until the destructor of one of the top nodes is called. It promptly deletes its children. And what is the second top node pointing to now? A piece of garbage! The moment it tries to access the children, it will stomp over reclaimed memory with disastrous results. But even if it does nothing, eventually its own destructor is called. And that destructor will attempt to delete the same children that have already been deleted by the first top node. The result? Memory corruption.

But wait, there's more! C++ not only sneaks a default copy constructor on you. It also provides you with a convenient default assignment operator.

Beware of default assignments!

The following code is perfectly legal.

```
SymbolTable symTab1 (100);
SymbolTable symTab2 (200);
// ...
symTab1 = symTab2;
```

Not only does it perform a shallow copy of symTab2 into symTab1, but it also clobbers whatever already was there in symTab1. All memory that was allocated in symTab1 is lost, never to be reclaimed. Instead, the memory allocated in symTab2 will be double deleted. Now that's a bargain!

> Why does C++ quietly let a skunk into our house and waits for us to run into it in the dark? If you've followed this book closely, you know the answer--compatibility with C! You see, in C you can't define a copy constructor, because there aren't any constructors. So every time you wanted to copy something more complex than an int, you'd have to write a special function or a macro. So, in the traditional spirit of letting programmers shoot themselves in the foot, C provided this additional facility of quietly copying all the user-defined struct's. Now, in C this wasn't such a big deal--a C struct is just raw data. There is no data hiding, no methods, no inheritance. Besides, there are no references in C, so you are much less likely to inadvertently copy a struct, just because you forgot one ampersand. But in C++ it's a completely different story. Beware--many a bug in C++ is a result of a missing ampersand.

But wait, there's even more! C++ not only offers a free copy constructor and a free assignment, it will also quietly use these two to return objects from functions. Here's a fragment of code from our old implementation of a stack-based calculator, except for one small modification. Can you spot it?

```
class Calculator
```

```
{
public:
    int Execute (Input & input);
    IStack const GetStack () const { return _stack; }
private:
    int Calculate (int n1, int n2, int token) const;

    IStack  _stack;
};
```

What happened here was that I omitted the ampersand in the return type of Calculator::GetStack and IStack is now returned by value. Let's have a very close look at what happens during such transfer. Also, let's assume for a moment that the compiler doesn't do any clever optimizations here. In particular, let's define GetStack out of line, so a regular function call has to be executed.

```
IStack const Calculator::GetStack () const
{
    return _stack;
}
//...
IStack const stk;
stk = calc.GetStack ();  // <- by value!
```

The process of ***returning an object by value*** consists of two steps: copy construction and assignment.

- First of all, before executing the call to GetStack, the compiler pre-allocates some scratch space on the stack (I'm talking here about the internal call stack, where function arguments and local variables live at runtime). This scratch space is then passed to the function being called. To copy the value of _stack into scratch space, GetStack calls IStack's copy constructor.
- After returning from GetStack, the assignment operator is called to copy the value from scratch space to the local variable stk.

It might look pretty complicated at first, but if you think about it, this is really the only sensible fool-proof way of returning an object by value. Especially if the programmer took care of both defining a copy constructor and overloading the assignment operator for the class in question.

For some classes the compiler will not be able to generate a default assignment. These are the classes that contain data members that cannot be assigned outside of the constructor's preamble. This is true for any const members (including const pointers) and for references. For instance, if you tried to assign one parser to another, like this:

```
parser1 = parser;
```

```
Parser parser1 (parser);
```

will compile. The references in parser1 will be correctly initialized to refer to the same objects as the ones in parser. However, since value semantics requires both--the copy constructor and the assignment--class Parser in its present form cannot be passed by value.

Value Semantics

Let's try to make some sense of this mess. We've just learned about several features of C++ that, each taken separately, may turn our code against us. It's time to find out what they can do for us when used in proper context.

The question whether a given object should be passed by reference or by value should be decided on the level of its class. For some objects it makes little sense to be passed by value, for others it makes a lot of sense. By designing the class properly, we can accommodate either case.

First of all, we can easily protect objects of a given class from being passed by value by declaring a private copy constructor and a private assignment operator. No implementation of these methods is necessary. They are there just to stop the compiler from doing the magic behind our backs.

Suppose we want to prevent IStack objects from being passed by value. All we have to do is add these two lines to its class definition (strictly speaking, one of them would suffice).

```
class IStack
{
public:
    // ...
private:
    IStack (IStack const & i);
    IStack & operator = (IStack const & i);
    // ...
};
```

Now try compiling any of the code that attempted to pass an IStack to a function, return an IStack from a function or assign one IStack to another. It simply won't compile! If you make a mistake of omitting an ampersand following IStack's class name in any such context, the compiler will immediately catch it for you.

Should you, therefore, be adding copy constructors and assignment overloads to every class you create? Frankly, I don't think it's practical. There are definitely some classes that are being passed around a lot--these should have this type of protection in their declarations. With others--use your better judgment.

However, there are some classes of objects for which passing by value makes a lot of sense. Such classes are said to have *value semantics*. Usually, but not always, objects of such classes have relatively small footprint. Size is important if you care about performance. On the other hand, quite often the only alternative to passing some objects by value is to keep allocating new copies from the free store using new--a much more expensive choice. So you shouldn't dismiss value semantics out of hand even for larger object. Such would be the case, for instance, with some algebraic classes that we'll discuss in the following chapter.

In many cases giving a class value semantics does not require any work. As we've seen earlier, the compiler will gladly provide a default copy constructor and the default assignment operator. If the default shallow copy does adequate job of duplicating all the relevant data stored in the object, you shouldn't look any further.

In case shallow copy is not enough, you should provide the appropriate copy constructor and overload the assignment operator. Let's consider a simple example.

Object of class Value represents an integer value. However, the value is not stored as an int--it's stored as a string. What's more, the strings records the "arithmetic history" of the number.

```
class Value
{
public:
    Value ()
    {
        cout << "  Default constructor\n";
        _numString = new char [1];
        _numString [0] = '\0';
    }

    Value (int i)
    {
        cout << "  Construction/conversion from int " << i << endl;

        stringstream buffer;
        buffer << i << ends; // terminate string
        Init (buffer.str ().c_str ());
        Display ();
    }

    Value (Value const & v)
    {
        cout << "  Copy constructor ( " << v._numString << " )\n";
        Init (v._numString);
        Display ();
    }

    Value & operator= (Value const & v)
    {
        cout << "  operator = ( " << v._numString << " )\n";
        if (_numString != v._numString)
        {
            delete _numString;
            Init (v._numString);
        }
        Display ();
        return *this;
    }

    friend Value operator+ (Value const & v1, Value const & v2 );
private:
    void Init (char const * buf)
    {
        int len = strlen (buf);
        _numString = new char [len + 1];
        strcpy (_numString, buf);
    }

    void Display ()
    {
        cout << "\t" << _numString << endl;
    }

    char * _numString;
};
```

A lot of things are happening here. First of all, we have a bunch of constructors. Let's use this opportunity to learn more about various types of constructors.

- The *default constructor* with no arguments. You must have it, if you want to create arrays of objects of this class. A default constructor could also be a constructor with the defaults provided for all its arguments. For instance, `Value::Value (int i = 0);` would work just fine.
- The constructor with a single argument. Unless declared as `explicit`, such constructor provides *implicit conversion* from the type of its argument to the type represented by the class. In this case, we are giving the compiler a go-ahead to convert any `int` to a `Value`, if such conversion is required to make sense of our code. We'll see examples of such conversions in a moment.
- The *copy constructor*. It takes a (usually `const`) reference to an object of the same class and clones it.

Next, we have the overloading of the assignment operator. By convention it takes a (usually `const`) reference to an object and copies it into an existing object of the same class. Unlike the copy constructor, the assignment operator must be able to deal with an already initialized object as its target. Special precautions, therefore, are required. We usually have to deallocate whatever resources the target object stores and allocate new resources to hold the copy of the source. However, beware of the following, perfectly legitimate use of assignment:

```
Value val (1);
val = val;
```

Here, testing for "source equal to target" in the assignment operator will prevent us from trying to deallocate the source before making the copy. In fact, no copying is necessary when target is identical to the source.

There's also a bit of magic from the standard C++ library that I used to convert an `int` to its string representation. It's just like printing an `int` to `cout`, only that instead of `cout` I used a `stringstream`. It's a stream that has a string for its output. You can get hold of this string by calling the method `stringstream::str`. (Before you do that, make sure to null-terminate the string by outputting the special `ends` (end string) object). To get the stringstream functionality you have to #include <sstream> in your code and add the appropriate using statements.

Finally, to make things really interesting, I added an overloading of the *plus operator* for objects of type Value.

```
inline Value operator+ (Value const & v1, Value const & v2 )
{
    cout << "  operator + (" << v1._numString << ", "
         << v2._numString << ")\n";
    stringstream buffer;
    buffer << v1._numString << " + " << v2._numString << ends;
    Value result;
    result.Init (buffer.str ().c_str ());
    cout << "  Returning by value\n";
    return result;
}
```

Our operator+ takes two arguments of the type const reference to Value and returns another Value--their "sum"--by value. Next time the compiler sees an expression like val1 + val2, it will turn it into the call our method Value::operator+. In fact, the following two statements are equivalent:

```
val3 = val1 + val2;
val3.operator= (operator+ (val1, val2));
```

For us, humans, the first one is usually much easier to grasp.

Notice that I haven't declared operator+ to be a method of Value. Instead it is a *free function*. You can execute it outside of the context of any particular object. There is a good reason for doing that and I am about to explain it. But first, let me show you the alternative approach.

```
Value Value::operator+ (Value const & val) const
{
    cout << "  Value::operator + (" << val._numString << ")\n";
    stringstream buffer;
    buffer << _numString << " + " << val._numString << ends;
    Value result;
    result.Init (buffer.str ());
    cout << "  Returning by value\n";
    return result;
}
```

Here, operator+ is defined as a member function of Value, so it requires only one argument. The first addend is implicit as the this object. The syntax for using this version of operator+ is identical to the free-function one and the equivalence works as follows,

```
val3 = val1 + val2;
val3.operator= (val1.operator+ (val2));
```

There is just one subtle difference between these two definitions--the way the two addends are treated in the "method" implementation is not symmetric. Normally that wouldn't be a problem, except when you try to add a regular integer to a Value object. Consider this,

```
Value v (5);
Value vSum = v + 10;
```

In general you'd need to define a separate overload of operator+ to deal with such mixed additions. For instance, using the free-function approach,

```
Value operator+ (Value const & val, int i);
```

or, using the "method" approach,

```
Value Value::operator+ (int i);
```

You see what the problem is? You can deal with adding an int to a Value using either method. But if you want to do the opposite, add a Value to an int, only the free-function approach will work. Like this:

```
Value operator+ (int i, Value const & val);
Value v (5);
Value vSum = 10 + v;
```

By the way, notice how smart the compiler is when parsing arithmetic expressions. It looks at the types of operands and, based on that, finds the correct overloading of the operator. Actually, it is even smarter than that!

I told you that you needed a special overloading of operator+ to deal with mixed additions. It's not entirely true. In fact both, adding an int to a Value and adding a Value to an int would work without any additional overloads of operator+. How? Because there is an implicit conversion from an int to a Value that the compiler may, and will, use in its attempts to parse a mixed arithmetic expression. This is only a slight generalization of what happens when you are adding an int to a double. For instance,

```
double x = 1.2;
int n = 3;
double result = x + n;
```

In order to perform this addition, the compiler converts n to a double and uses its internal implementation of "double" addition. Similarly, in

```
Value v (12);
int n = 3
Value result = v + n;
```

the compiler is free to convert n to a Value and use the overloaded operator+ that takes two (const references to) Values. In case you're wondering about the *implicit conversion* from int to Value, look again at the set of Value's constructors.

A constructor that takes a single argument, defines an implicit conversion from the type of its argument to its own type.

But converting an int into a Value is one thing. Here, the compiler has to do even more. It has to convert an int into a *const reference to* Value. It does it by creating a temporary Value, initializing it with an integer and passing a const reference to it to the called function. The tricky part is that the temporary

Value has to be kept alive for the duration of the call. How the compiler does it is another story. Suffice it to say that *it works*.

What does *not* work is if you tried to do the same with a non-const reference. A function that takes a non-const reference expects to be able to have a side effect of changing the value of that argument. So using a temporary as a go-between is out of the question. In fact, the compiler will refuse to do implicit conversions to non-const references even it the case of built-in primitive types. Consider the following example,

```
void increment (double & x) { x += 1.0; }
int n = 5;
increment (n);   // <- error!
```

There is no way the compiler could figure out that you were expecting n to change its value from 5 to 6. And it can't pass the address of an int where the address of a double is expected, because the two have a completely different memory layout (and possibly size). Therefore it will tell you something along the lines, "*Error: There's no conversion from int to (non-const) reference to double*."

In contrast, the following code compiles just fine,

```
void dontIncrement (double const & x);
int n = 5;
dontIncrement (n);  // <- ok!
```

because the compiler is free to generate a temporary double, initialize it with the value on n and pass a ***const*** reference to it.

One last thing. Why did I declare operator= to return a reference to Value and operator+ to return Value by value? Let's start with operator=. It has to return something (by reference or by value) if you like this style of programming,

```
Value v1, v2, v3 (123);
v1 = v2 = v3;
```

or

```
if (v1 = v2) // notice, it's not ==
    ...
```

The chaining of assignments works, because it is interpreted as

```
v1.operator= (v2.operator= (v3));
```

and the assignment as condition works because it is equivalent to

```
if (v1.operator= (v2))
    ...
```

and the return value from operator= is interpreted as true if it's different from zero.

On the other hand, if you don't care for this style of programming, you may as well declare operator= to return void.

```
void Value::operator= (Value const & val)
```

I wouldn't recommend the same when overloading operator+, because we are usually interested in the result of addition. So instead, let me try to explain why returning the result of addition *by value* is preferable to returning it by reference. It's simple--just ask yourself the question, "Reference to what?" It can't be the reference to any of the addends, because they are supposed to keep their original values. So you have to create a new object to store the result of addition. If you create it on the stack, as a local variable, it will quickly disappear after you return from operator+. You'll end up with a reference to some stack junk. If you create it using new, it won't disappear from under your reference, but then you'll never be able to delete it. You'll end up with a memory leak.

Think of operator+ (or any other binary operator for that matter) as a two-argument constructor. It constructs a new entity, the sum of the two addends, and it has to put it somewhere. There is very little choice as to where to store it, especially when you're inside a complicated arithmetic expression. Return it by value and let the compiler worry about the rest.

To summarize, primitive built-in types are usually passed around by value. The same kind of value semantics for a user-defined type is accomplished by the use of a copy constructor and an overloaded operator=. Unless specifically overridden, the compiler will create a default copy constructor and default assignment for any user-defined type.

Techniques

Code Review 1: The Cleanup

Code Review 2: Hiding Implementation Details

Code Review 3: Sharing

Code Review 4: Removing Limitations

Code Review 5: Resource Management

Code Review 6: Making Use of Standard Template Library

Code Review 7: Serialization and Deserialization

- The Calculator Object
- Command Parser
- Serialization and Deserialization
- In-memory (De-) Serialization
- Multiple Inheritance
- Transactions

Overloading operator new

- Class-specific new
- Global new

Techniques

I had the second part of this book all figured out: I would introduce advanced features of C++ one by one, with examples. I would do templates and exceptions and write a chapter about my favorite resource management. Then I would show the perils of concurrent programming and how to avoid them, the clever tricks with virtual memory and I'd finish up with a series of Windows programs.

Then I started preparing a course for students and suddenly realized that it wouldn't work. Well, it would in some way, but not the way I wanted it. I didn't want to come up with a random bag of tricks, like so many "experience" books do. (Nor did I want to produce a new theory of program development.) What I really wanted is to show how programming works in real-life situations. In real life you don't come up with a new trick and start looking around for the opportunity to use it. A much more likely scenario is that you are faced with a problem that has to be solved and you need to know what your options are. Or you are trying to spot potential problems in your or somebody else's code and prevent them.

The problem is that, in order to demonstrate the usefulness of advanced techniques, one needs to work with examples of more substantial programs. Of course, I could keep inventing stories like: "Imagine you wrote a large system that was supposed to distribute luggage in a big airport. The system was just allocating an entry for a suitcase when it ran out of memory. A null suitcase was created and caused a plane to GP-fault."

And then I realized that there is no better opportunity for a programmer to learn about programming than by taking part in code reviews. This is how my coworkers and I honed our skills, both when defending our own code and when critiquing others' code.

The rules of engagement were very simple: The reviewer was always right. If the reviewer couldn't understand somebody's code, the code had to be redesigned. If the reviewer thought that the names of variables were too cryptic, they had to be changed. Of course, arguing was allowed; but if the reviewed was not able to convince the reviewer, the reviewed was the loser.

Instead of writing yet another program for the course and code reviewing it, I looked at what I already had--the calculator from the first part of the book. The code looked okay, it was object oriented, nicely structured, seemingly bug free. Yet I couldn't say it was a solid piece of programming that I would like to sign with my name. It worked, but:

The fact that the program works has no relevance.

Any fool can write a program that runs. I decided to give the calculator a thorough code review. I imagined that it was written by a novice programmer (someone who had just learned C++ by reading the first part of this book). My role was to help the person turn this piece of amateur code into an industrial strength program.

What follows is a series of code reviews that result in quite substantial changes in the program. Besides demonstrating various advanced programming techniques they prove one more important point: Substantial changes are possible and beneficial. The saying "If it ain't broke, don't fix it." should be banned from serious discussion about software development. Instead another saying should be adopted:

If you can’t write programs that can be modified, don’t bother.

In fact, it is often for the best to toss out the unmaintainable code and start all over from scratch (although it might be impossible to convince the management about it). Fortunately, we don’t have to toss out the calculator. As you’ll see it is highly maintainable. Not bad for an amateurish program.

To get the most out of this part of the book, you should try working with the project's source code. Try making all the changes described in code reviews and, after each chapter, compare your code with the one I provided. Being able to modify a program is probably the most important skill of any programmer.

Code Review 1: The Cleanup

As much as we would like a program to follow the design, there is always an element of ad libbing when filling out the details. We create auxiliary classes and functions, add methods and data members to existing classes. This process reminds me of the growth of a yeast culture. As each cell keeps growing, it starts to bloat and bud. Eventually new buds separate and start a life of their own.

In software, the growing and (especially) the bloating happens quite naturally. It's the budding and the separation that requires conscious effort on the part of the programmer. Code reviews help this process greatly. We'll see examples of overgrown files, methods and classes throughout our code reviews. Our response will be to create new files, new methods and, especially, new classes.

We'll be always on the lookout for potential code reuse.

When reviewing code, we will also figure out which parts require more thorough explanation and add appropriate comments.

Improving Code Grouping

The obvious place to start the code review is the file **parse.cpp**. It contains the `main` procedure. And that's the first jarring feature. `Main` does not belong in the implementation file of the `Parser` class. I put it there because its purpose was mainly to test the `Parser`. Now it's time to separate it away from the `Parser` and create a new file **main.cpp**.

Look also at the file **symtab.h**. I definitely see some potential for code reuse there. Such classes as `Link` and `List` are very general and can be used in many applications. This particular list stores direct *values* rather than pointers to objects. So let's create a separate header file for it and call it **vlist.h**. The implementation will go to another file **vlist.cpp**.

Notice that we haven't tried to make any changes to the classes `Link` and `List` to make them more general. This is a useful rule of thumb:

> **Don't try to generalize until there's a need for it.**

Over-generalization is the bane of code reuse.

Finally, I split the files **symtab.h** and **symtab.cpp** into three groups of files corresponding to the symbol table, the function table and the hash table.

Table: The correspondence between files and classes, functions and global objects.

main.cpp	`main ()`
parser.h parser.cpp	`Parser`
scan.h scan.cpp	`EToken, Scanner`
funtab.h funtab.cpp	`FunctionEntry, funArr, FunctionTable`
symtab.h symtab.cpp	`SymbolTable`
vlist.h vlist.cpp	`Link, List`

htab.h htab.cpp	HTable
store.h store.cpp	Store
tree.h tree.cpp	Node *and its children*

Decoupling the Output

Let’s look again at the main loop of our calculator.

```
  cout << "> ";  // prompt
  cin.getline (buf, maxBuf);
  Scanner scanner (buf);
  Parser  parser (scanner, store, funTab, symTab);
  status = parser.Eval ();
```

The first two lines take care of user input. The next three lines do the parsing and evaluation. Aren’t we missing something? Shouldn’t the calculator display the result? Well, it does, inside Eval. Sending the result of the calculation to the output is a side effect of evaluation.

What it means, however, is that somebody will have to look inside the implementation of Eval in order to understand what the calculator is doing. Okay, that sounds bad--but isn't it true in general that you'd have to look inside the implementation of Eval in order to know what it's doing? Otherwise, how would you know that it first parses and then evaluates the user input, which has been passed to the parser in the form of a scanner.

Well, that's a good point! Indeed, why don’t we make it more explicit and rewrite the last line of the main loop.

```
 cout << "> ";  // prompt
 cin.getline (buf, maxBuf);
 Scanner scanner (buf);
 Parser  parser (scanner, store, funTab, symTab);
 status = parser.Parse ();
 double result = parser.Calculate ();
 cout << result << endl;
```

This way, every statement does just one well defined thing. I can read this code without having to know any of the implementation details of any of the objects. Here’s how I read it:

1. Prompt the user,
2. Get a line of input,
3. Create a scanner to encapsulate the line of input,
4. Create a parser to process this input,
5. Let the parser parse it,
6. Let the parser calculate the result,
7. Display the result.

I find this solution esthetically more pleasing, but esthetics alone isn't usually enough to convince somebody to rewrite their code. I had to learn to translate my esthetics into practical arguments. Therefore I will argue that this version is *better* than the original one because:

- It separates three distinct and well defined actions: parsing, calculation and display. This decoupling will translate into easier maintenance.

- Input and output are now performed at the same level, rather than input being done at the top level and output interspersed with the lower level code.
- Input and output are part of user interface which usually evolves independently of the rest of the program. So it's always a good idea to have it concentrated in one place. I'm already anticipating the possible switch from command line interface to a simple Windows interface.

The changes to `Parser` to support this new split are pretty obvious. I got rid of the old `Parse` method and pasted its code directly in the only place from which it was called (all it did was: `_pTree = Expr()`. Then I renamed the `Eval` to `Parse` and moved the calculation to the new method `Calculate`. By the way, I also fixed a loophole in error checking: the input line should not contain anything following the expression. Before that, a line like 1 + 1 = 1 would have been accepted without complaint.

```
Status Parser::Parse ()
{
    // Everything is an expression
    _pTree = Expr ();
    if (!_scanner.IsDone ())
        _status = stError;
    return _status;
}

double Parser::Calculate () const
{
    assert (_status == stOk);
    assert (_pTree != 0);
    return _pTree->Calc ();
}

bool Scanner::IsDone () const
{
    return _buf [_iLook] == '\0';
}
```

My students convinced me that `Calculate` should not be called, if the parsing resulted in an error. As you can see, that certainly simplifies the code of `Parse` and `Calculate`. Of course, now we have to check for errors in main, but that's okay; since we have the status codes available there, why not do something useful with them.

By the way, as long as we're at it, we can also check the scanner for an empty line and not bother creating a parser in such a case. Incidentally, that will allow us to use an empty line as a sign from the user to quit the calculator. Things are just nicely falling in place.

```
cout << "\nEnter empty line to quit\n";
// Process a line of input at a time
do
{
    cout << "> ";  // prompt
    cin.getline (buf, maxBuf); // read a line
    Scanner scanner (buf); // create a scanner
    if (!scanner.IsEmpty ())
    {
        // Create a parser
        Parser  parser (scanner, store, funTab, symTab);
        status = parser.Parse ();
        if (status == stOk)
        {
            double result = parser.Calculate ();
            cout << result << endl;
        }
        else
        {
            cout << "Syntax error.\n";
        }
    }
    else
    {
        break;
    }
} while (status != stQuit);
```

Notice what we have just done: We have reorganized the top level of our program with very little effort. In traditional programming this is a big no-no. Changing the top level is synonymous with rewriting the whole program. We must have done something right, if we were able to pull such a trick. Of course, this is a small program and, besides, *I* wrote it, so it was easy for *me* to change it. I have some good news: I have tried these methods in a project a hundred times bigger than this and it worked. Better than that--I would let a newcomer to the project go ahead and make a top level change and propagate it all the way to the lowest level. And frankly, if I weren't able to do just that, I would have been stuck forever with some early design mistakes. Just like the one in the calculator which we have just fixed.

There are several other output operations throughout the program. They are mostly dealing with error reporting. Since we don't have yet a coherent plan for error reporting and error propagation, we'll just do one cosmetic change. Instead of directing error messages to the standard output cout, we'll direct them to the standard error cerr, like this:

```
cerr << "Error: division by zero\n";
```

The only practical difference is that, if you redirect standard output of your program to a file, the error messages will still go to your console. Not that we care much, but when we port this program to Windows, we would like to be able to quickly search the source for stderr and, for instance, convert these printouts to message boxes.

Fighting Defensive Programming

"Defensive programming" sounds good, doesn't it? A program written by a defensive programmer should be safer and more robust, right?

Wrong! Defensive programming is a dangerous form of engineering malpractice (in a show of self-abasement I'll shortly expose examples of such practices in my own code). Defensive programming promotes writing code that is supposed to work even in the face of programmers' errors, a.k.a. bugs. In practice, it gives the bugs the opportunity to cover their tracks. Anybody who spent a sleepless night chasing a bug that could have been caught early on, if it weren't for defensive programming, will understand my indignation. In most cases defensive programming is a result of not understanding the exact contract between various parts of the program. A confused or lazy programmer, instead of trying to understand the logic of the program, might write something that should work *no matter what*. That code will obscure the program's logic even more, making further development even sloppier. The programmer will eventually lose control of the code.

One source of confusion that is often covered up by defensive programming is the discrepancy between the length of a string and the size of the array into which it will fit. The array must have room for the terminating null which is not counted in the length of the string. Does the MAX_PATH defined in a system header take into account the terminating null? Should one use MAX_PATH or MAX_PATH + 1 when allocating a buffer?

Now look at the confusion in our calculator. Scanner::GetSymbolName assumes that its argument len is the maximum size of a string that can fit in the buffer. So the buffer size is one larger than len. Hence the assertions:

```
assert (len >= maxSymLen);
assert (_lenSymbol <= maxSymLen);
```

But when we are calling GetSymbolName from the parser, we are allocating a buffer of maxSymLen characters and pass maxSymLen as its length. Wait a moment, that looks like a bug, doesn’t it? We have just said that GetSymbolName expects the buffer to be one char longer than len. So how does it work? Because, in the scanner, we have defensively truncated the length of a symbolic name to *one less* than maxSymLen.

```
if (_lenSymbol >= maxSymLen)
    _lenSymbol = maxSymLen - 1;
```

So now it turns out that maxSymLen includes the terminating null and Scanner::GetSymbolName expects len to be the size of the buffer, not the string; although when it returns, it sets len to the actual size of the string--without the null. Hey, it works! Look how clever I am! Not!

Let’s just try to rationalize our assumptions. First of all, maxSymLen should be the maximum length of a symbolic name *without* the terminating null. Null termination is an implementation detail--we should be able to switch to counted strings without having to change the value of maxSymLen. Scanner::Accept should therefore truncate symbolic names like this:

```
if (_lenSymbol > maxSymLen)
    _lenSymbol = maxSymLen;
```

This is how GetSymbolName should be called

```
char strSymbol [maxSymLen + 1];
int lenSym = _scanner.GetSymbolName (strSymbol, maxSymLen + 1);
```

and this is how it should be implemented

```
int Scanner::GetSymbolName (char * strOut, int lenBuf)
{
    assert (lenBuf > maxSymLen);
    assert (_lenSymbol < lenBuf);
    strncpy (strOut, &_buf[_iSymbol], _lenSymbol);
    strOut [_lenSymbol] = 0;
    return _lenSymbol;
}
```

The in-argument, lenBuf, is the length of the buffer. The return value is the length of the string copied into the buffer. That’s *much* less confusing.

Another excuse for all kinds of lame coding practices is any situation that is *not really supposed to happen*. Our calculator has some built in limitations. For instance, the size of the Store is limited to maxSymbols, which is set to 40. That’s plenty, and we don’t expect to hit this limitation under normal circumstances. Even if we did, we have a line of defense in Store:

```
void Store::SetValue (int id, double val)
{
    if (id < _size)
    {
        _cell [id] = val;
        _status [id] = stInit;
    }
}
```

That’s very fortunate, because it let us get away with some sloppy error checking in the parser (we’ll look at it shortly).

You see what happens when you lower your defenses against defensive programming? You get sloppy and sooner or later it will bite you. So let’s turn this method into an example of *offensive programming*: we are so sure of the correctness of our (or others’) code that we’re not afraid to assert it.

```
void Store::SetValue (int id, double val)
{
    assert (id < _size);
    _cell [id] = val;
    _status [id] = stInit;
}
```

We’ll fix the caller’s code shortly.

A Case of Paranoid Programming

The initialization of the function table is an example of paranoid defensive programming. First we define maxIdFun to be the size of the array of function entries. Then we define the array of size maxIdFun, but we initialize it with a smaller number of entries. We also add a sentinel with a null pointer and an empty string. Then, in the constructor of FunctionTable, we set up the loop to go from zero to maxIdFun - 1, but we break from it when we encounter the sentinel. All this mess is there to prevent the mismatch between the size of the function array and the number of entries in the function table.

Instead, one simple assertion would suffice:

```
    assert (maxIdFun == sizeof FunctionArray / sizeof FunctionArray
[0]);
```

The sizeof operator returns the size (in bytes) of its argument. When applied to a static array, it returns the size of the whole array. When applied to a pointer, even if this pointer points to an array, it returns the size of the pointer--4 bytes on a 32-bit computer. (The argument to sizeof may also be a type name, it will return the size of an object of that type.)

Notice that I had to correct an awkward naming conflict for this assertion to work. I had named the argument to FunctionTable constructor the same name as the global array--they were both called funArr. Guess what, when I first wrote the assertion, it didn't work as planned: sizeof funArr kept returning 4. I checked the language reference several times, tried using brackets--nothing worked. I was ready to blame the compiler when suddenly I realized that my sizeof was applied to the *argument passed to the constructor* and not to the global array, as I thought. The local name funArr obscured the global name funArr and, consequently, I was sizing a pointer rather than a static array.

I learned (or rather re-learned for the nth time) two lessons: don't use the same name for different variables and, if operator sizeof returns 4 when applied to an array, don't blame the compiler. The irony of this situation was that I usually, but not this time, follow the convention of starting a global name with a capital letter (and also making it less abbreviated).

After changing the name of the global array to FunctionArray, I set maxIdFun correctly to 14 and let the compiler size the array by counting the initializers (that's why I left the brackets empty).

```
const int maxIdFun = 14;
FunctionEntry FunctionArray [] =
{
    log, "log",
    …
    atan, "atan",
};
```

Now I could finally add the assertion that would catch the mismatch between the size of FunctionArray and the array FunctionTable::_pFun was pointing to.

```
FunctionTable::FunctionTable (SymbolTable & symTab, FunctionEntry *
funArr)
: _size(0)
{
    assert (maxIdFun == sizeof FunctionArray / sizeof FunctionArray
[0]);
    …
}
```

When I showed this solution to students, they immediately noticed the problem: What if the constructor of FunctionTable is called with a pointer to an array *different* than FunctionArray? There's no way to prevent it. Obviously the design was evolving away from the generality of FunctionTable as an object that can be initialized by any array of function entries and towards making FunctionArray a very close partner of FunctionTable. In fact, that's what I was planning to do in the next code review. Here's the code I wrote for that purpose

```
static FunctionEntry FunctionArray [] = { … }

FunctionTable::FunctionTable (SymbolTable & symTab)
    : _size(0)
```

```
{
    assert (maxIdFun == sizeof FunctionArray / sizeof FunctionArray
[0]);
    …
}
```

I made FunctionArray *static*, which means that it was invisible outside of the file **funtab.cpp** where it was defined. In this sense *static* and *extern* are on the opposite ends of the spectrum of visibility. Data hiding is always a good idea.

Is it possible to make FunctionArray and FunctionTable even more intimately related? How about making FunctionArray a member of the class FunctionTable? Now, remember, the whole idea of FunctionArray was to be able to *statically* initialize it--compile-time instead of run-time. It turns out that there is a construct in C++ called--notice the overloading of this keyword--a *static* member.

> A static member can be viewed as a member of the *class object*, rather than a member of the particular *instance object*. In some object-oriented languages, for example in Smalltalk, there is a clear distinction between the instance object and the class object. There can be many instances of objects of the same class, but they all share the same class object. A class object may have its own data members and methods. The values of *class object* data members are shared between all instances. One can access class data or call class methods without having access to any instance object. Class methods can only operate on class data, not on the data of a particular instance.
>
> Although in C++ there is no notion of class object, static data members and static methods effectively provide the equivalent functionality. There is only one instance of each static data member *per class*, no matter how many objects of a given class are created. A static method is essentially a regular function (in that it doesn't have the this pointer); with the visibility restricted by its membership in the class.

This is the syntax for declaring a private static data member

```
class FunctionTable
{
    …
private:
    static FunctionEntry  _functionArray [];
    …
};
```

And this is how it's initialized in the implementation file:

```
FunctionEntry FunctionTable::_functionArray [] =
{
    log, "log",
    …
};
```

Notice that, even though _functionArray is private, its initialization is done within global scope.

Unfortunately, when _functionArray is made a static member with undefined size (look at the empty brackets in the declaration), the sizeof operator does not return the initialized size of the array, as we would like it. The compiler looks only at the declaration in FunctionTable and it doesn't find the information about the size there. When you think about it, it actually makes sense.

There was another interesting suggestion. Why not use directly sizeof FunctionArray / sizeof FunctionArray [0] as the size of the array _pFun inside the declaration of FunctionTable?

```
    Pfun _pFun [sizeof FunctionArray / sizeof FunctionArray [0]];
```

In principle it's possible, but in practice we would have to initialize FunctionArray before the declaration of FunctionTable, so that the compiler could calculate the sizeof. That would mean putting the initialization in the header file and, since the header had to be included in several implementation files, we would get a linker error (multiple definition).

The proposed solution was then modified to make FunctionTable a const array (or, strictly speaking, an array of constants) so that the linker wouldn't complain that it is initialized in multiple files. After all, constructs like this

```
    const int maxLen = 10;
```

are perfectly acceptable in header files. So should

```
    const FunctionEntry FunctionArray [] =
    {
        log, "log",
        …
    };
```

And indeed, the linker didn't complain. This solution is good, because it makes the table of function pointers automatically adjust its size when the function array grows. That's better than hitting the assertion and having to adjust the size by hand. On the other hand, the array FunctionArray that is only needed internally to initialize the function table, suddenly becomes visible to the whole world. This is exactly the opposite of the other proposal, which made the array a private static member of the function table. It violates the principle of data hiding.

So what are we to do? Here's my solution: Let's hide the function array in the implementation file **funtab.cpp** by making it static (but not a static data member). But instead of trying at all costs to size _pFun statically, let's use dynamic allocation. At the point of allocation, the static size of FunctionTable is well known to the compiler.

```
FunctionTable::FunctionTable (SymbolTable& symTab)
: _size (sizeof FunctionArray / sizeof FunctionArray [0])
{
    _pFun = new PFun [_size];
    for (int i = 0; i < _size; ++i)
    {
        int len =  strlen (FunctionArray[i].strFun);
        _pFun [i] = FunctionArray [i].pFun;
        cout << FunctionArray[i].strFun << endl;
        int j = symTab.ForceAdd (FunctionArray[i].strFun, len);
        assert (i == j);
    }
}
```

Let’s not forget to deallocate the array

```
    FunctionTable::~FunctionTable ()
    {
        delete []_pFun;
    }
```

We have already seen examples of classes forming patterns (stack and its sequencer). Here we see an example of a static array teaming up with a class. In practice such patterns will share the same header and implementation file.

What have I sacrificed by choosing this solution? I have sacrificed static allocation of an array. The drawbacks of dynamic allocation are twofold: having to remember to deallocate the object--a maintainability drawback--and an infinitesimal increase in the cost of access, which I mention here only for the sake of completeness, because it is *really* irrelevant. In exchange, I can keep the function array and its initializer list hidden, which is an important maintainability gain.

The code of the FunctionTable is not immediately reusable, but I could easily make it so by changing the constructor to accept an array of function entries and the count of entries. Since I don’t see a lot of potential for reuse, I won’t do it. This way I reduce the number of objects visible at the highest abstraction level--at the level of main.

Fringes

Dealing with boundary cases and error paths is the area where beginning programmers make most mistakes. Not checking for array bounds, not testing error returns, ignoring null pointers, etc., are the type of errors that usually don’t manifest themselves during superficial testing. The program works, except for weird cases. Such an approach is not acceptable in writing professional software. Weird cases are as important as the main execution path--in fact they often contribute to the bulk of code (although we will learn to avoid such imbalances).

I’ve found an example of sloppy error checking in the parser. It so happens that the method SymbolTable::ForceAdd may fail if there is no more space for symbolic variables. Yet we are not testing for it in the parser. The code works (sort of) due to the case of defensive programming in the symbol table. Now that we have just replaced the defensive code with the offensive one, we have to fix this fragment too:

```
// Factor := Ident
if (id == idNotFound)
{
    // add new identifier to the symbol table
    id = _symTab.ForceAdd (strSymbol, lenSym);
    if (id == idNotFound)
    {
        cerr << "Error: Too many variables\n";
        _status = stError;
        pNode = 0;
    }
}
if (id != idNotFound)
    pNode = new VarNode (id, _store);
```

The rule of thumb is to check for the error as soon as possible. In this case we shouldn't have waited until VarNode was created and then silently break the contract of that class (VarNode::Calc was not doing the correct thing!).

Improving Communication Between Classes

So far, we've been passing strings between classes always accompanied by the count of characters. At some point it looked like a good idea, but lately it became more and more obvious that the count is superfluous. For instance, it is very easy to eliminate the count from the hash functions. Since we are scanning the string anyway, we can just as easily look for the terminating null,.

```
int HTable::hash (char const * str) const
{
    // must be unsigned, hash should return positive number
    unsigned h = str [0];
    assert (h != 0); // no empty strings, please
    for (int i = 1; str[i] != 0; ++i)
        h = (h << 4) + str [i];
    return h % _size;  // small positive integer
}
```

In ForceAdd, where we use the length in several places, we can simply apply strlen to it.

```
int SymbolTable::ForceAdd (char const * str)
{
    int len = strlen (str);
    assert (len > 0);
    …
}
```

Next, let's have a closer look at the interactions between various classes inside the symbol table. The hash table is an array of lists. The search for a string in the hash table returns a short list of candidates. Symbol table code then iterates over this short list and tries to find the string using direct comparison. Here's the code that does it:

```
int SymbolTable::Find (char const * str) const
{
    // Get a short list from hash table
    List const & list = _htab.Find (str);
    // Iterate over this list
    for (Link const * pLink = list.GetHead();
        pLink != 0;
        pLink = pLink->Next ())
    {
        int id = pLink->Id ();
        int offStr = _offStr [id];
```

```
         // char * strStored = &_strBuf [offStr];
         char const * strStored = _strBuf + offStr;
         if (strcmp (str, strStored) == 0) // they're equal
         {
             return id;  // success!
         }
     }
     return idNotFound;
}
```

Objects of class List play the role of communication packets between the hash table and the code of the symbol table. As a code reviewer I feel uneasy about it. Something is bothering me in this arrangement--does the HTable have to expose the details of its implementation, namely linked lists, to its client? After all there are implementations of hash tables that don’t use linked lists. As a code reviewer I can express my uneasiness even if I don’t know how to fix the situation. In this case, however, I *do* know what to do. The one common feature of all implementations of hash tables is that they have to be able to deal with collisions (unless they are perfect hash tables--but we won’t worry about those here). So they will *always*, at least conceptually, return an overflow list. All the client has to do is sequence through this list in order to finish the search.

The keyword here is ***sequence***. A sequencer object is easyly implemented on top of a linked list. It’s just a repackaging of the code we’ve seen above. But a sequencer has a much bigger potential. We've already seen a stack sequencer; one can write a sequencer over an array, a sequencer that would skip entries, and so on. We could make the hash table return a sequencer rather than a list. That way we would isolate ourselves from one more implementation detail of the hash table. (If you're wondering why I'm calling it a sequencer and not an iterator--I reserve the name *iterator* for the pointer-like objects of the standard library--we'll get there in due time.)

Figure 1 shows the interaction of the client’s code (SymbolTable code) with the hash table using the sequencer object which encapsulates the results of the hash table search.

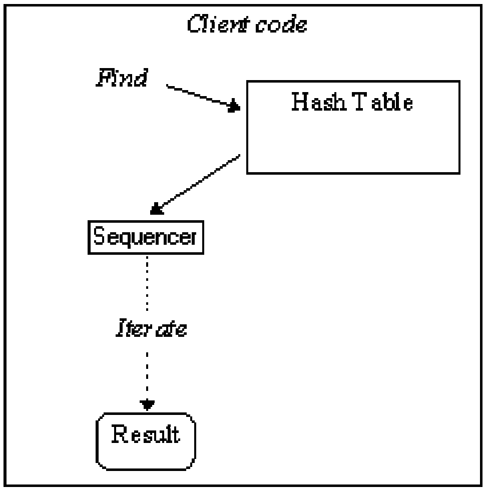


Figure: Using a sequencer for communications between the hash table and the client

The straightforward and simple-minded translation of Figure 3-1 into code would produce a method like this:

```
ListSeq * HashTable::Find (char const * str);  // BAD BAD BAD !!!
```

The sequencer would be allocated inside Find using new and would have to be deleted once the sequencing is finished.

We have created a monster! We substituted simple, fool-proof code with a dangerous kludge of a code. Don't laugh! You'll find this kind of coding practices turned into methodology in some commercially available libraries or API sets.

Our solution to this is ***object inversion***. Instead of creating an auxiliary object inside the callee and making the caller responsible for deleting it, we'll let the caller create an auxiliary *local* object (on the stack) which will be freed automatically on exit from the scope. The constructor of this object will take a reference to the source object (in our case, the hash table) and negotiate with it the transfer of data. In fact, we've already seen this mechanism in action when we first introduced a sequencer. (It was a stack sequencer whose constructor took reference to the stack object.)

Here's the new implementation of SymbolTable::Find that makes use of such a sequencer, IdSeq. It takes a reference to the hash table and obtains, from it, a list corresponding to string str.

```
int SymbolTable::Find (char const * str) const
{
    // Iterate over a short list from hash table
    for (IdSeq seq (_htab, str);
        !seq.AtEnd ();
         seq.Advance ())
    {
        int id = seq.GetId ();
        int offStr = _offStr [id];
        // char const * strStored = &_strBuf [offStr];
        char const * strStored = _strBuf + offStr;
        if (strcmp (str, strStored) == 0) // they're equal
        {
            return id;  // success!
        }
    }
    return idNotFound;
}
```

That's how it looks from the client's side. Searching through the hash table is reduced to the act of creating a sequencer corresponding to the pair: hash table, string. In other words, we have transformed a situation where a string sent to a hash table resulted in a sequencer into an equivalent one, where a sequencer is created from the interaction between the hash table and a string.

I still haven't explained how the whole thing is implemented. Obviously IdSeq is a special class that knows something about the workings of the hash table. Behind our backs it still obtains a linked list from the hash table and uses it for sequencing. The difference though is that the hash table no longer gives linked lists away to just anybody. The HTable::Find method is private and it can only be accessed by our friend--and now HTable's friend--IdSeq.

```
class HTable
```

```
{
    friend class IdSeq;
     …
private:

    List & Find (char const * str) const;
    …
};
```

One more cosmetic issue: The linked list sequencer is really a more general class, i. e., it belongs in the file **vlist.h** and should have nothing to do with the hash table. Something like that would be perfect:

```
// List sequencer
// Usage:>
// for (ListSeq seq (list);
//           !seq.AtEnd ();
//          seq.Advance ())
// {
//           int id = seq.GetId ();
//          ...
// }

class ListSeq
{
public:
    ListSeq (List const & list)
        : _pLink (list.GetHead ()) {}
    bool AtEnd () const { return _pLink == 0; }
    void Advance () { _pLink = _pLink->Next (); }
    int GetId () const { return _pLink->Id (); }
private:
    Link const * _pLink; // current link
};
```

On the other hand, we need a specialized sequencer that interacts with the hash table in its constructor.

Nothing is simpler than that. Here's a sequencer that inherits everything from the generic sequencer. It only provides its own constructor, which obtains the linked list from the hash table and passes it to the parent's constructor. That's exactly what we need—-no less no more.

```
// The short list sequencer
// The client creates this sequencer
// to search for a given string
class IdSeq: public ListSeq
{
public:
    IdSeq (HTable const & htab, char const * str)
        : ListSeq (htab.Find (str)) {}
};
```

Let's summarize the situation. There are five actors in this game, so it is still possible to keep them all in mind at once. Table 1 lists them and explains their roles.

Table:

List	**vlist.h**	Generic singly linked list storing data by value
ListSeq	**vlist.h**	Generic seqencer of a singly linked list
IdSeq	**htab.h**	Specialized sequencer: friend of HTable, inherits behavior from ListSeq.
HTable	**htab.h**	Hash table. For clients, the access is only through IdSeq. For friends, private method Find returns a List
SymbolTable	**symtab.h**	The client of HTable. Accesses HTable only through specialized IdSeq.

Notice how the classes are grouped: List and ListSeq form one team. They are generic enough to be ready for reuse. IdSeq and HTable form another team that depends for its implementation on the List team. They are also ready for reuse. The SymbolTable depends on the HTable and a rather disorganized mix of objects to store strings. (Let’s make a note to revisit this part of the code at a later time and do some cleanup.)

How do we measure our accomplishment? Suppose that we decide to change the implementation of the hash table to something that is called *closed hashing*. In such scheme the overflow entries are stored in the same array as the direct entries. The overflow list is no longer a linked list, but rather a prescription how to jump around that array. It can be encapsulated in a sequencer (which, of course, has to be a friend of the hash table).

So what’s the meta-program that transforms the symbol table from one implementation of the hash table to another? Provided we named our new classes the same names as before (HTable and IdSeq), the meta-program is *empty*. We don’t have to touch a single line of code in the symbol table. That’s quite an accomplishment!

Correcting Design Flaws

As I mentioned before, our parser of arithmetic expressions has a flaw--its grammar is right associative. That means that an expression like this

```
8 - 2 + 1
```

will be parsed as

```
8 - (2 + 1) = 5
```

rather than, as we would expect,

```
(8 - 2) + 1 = 7.
```

We could try to correct the grammar to contain left associative productions of the type

```
Expression := Expression '+' Term
```

but they would immediately send our recursive parser into an infinite recursive spin (Expr would start by calling Expr that would start by calling Expr, that would…). There are more powerful parsing algorithms that can deal with left recursion but they are too complicated for our simple calculator. There is however a middle-of-the-road solution that shifts associativity from grammar to execution. In that approach, the expression parser keeps parsing term after term and stores them all in a new type of multi-node. All terms that are the children of the multi-node have the same status, but when it comes to calculation, they are evaluated from left to right. For instance, the parsing of the expression

```
    8 - 2 + 1
```

would result in a tree in Figure 3-2.

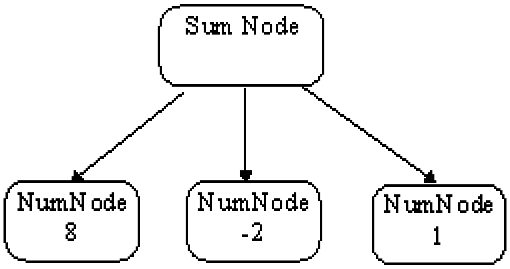


Figure. The result of parsing 8 - 2 + 1 using a multi-node.

The Calc method of the multi-node SumNode sums the terms left to right, giving in the correct answer

```
    8 - 2 + 1 = 7
```

We can do the same trick with the parsing of terms--to correct the associativity of factors.

The grammar is modified to contain productions of the type (this is called the *extended* BNF notation)

```
    Expr := Term { ('+' | '-') Term }
    Term := Factor { ('*' | '/') Factor }
```

or, in more descriptive language, Expression is a Term followed by zero or more occurrences of the combination of a sign (*plus* or *minus*) with a Term. Term, in turn, is a Factor followed by zero or more occurrences of the combination of a *multiplication* or *division* sign with a Factor. For those with spatial imagination, Figure 3-3 shows these relationships pictorially.

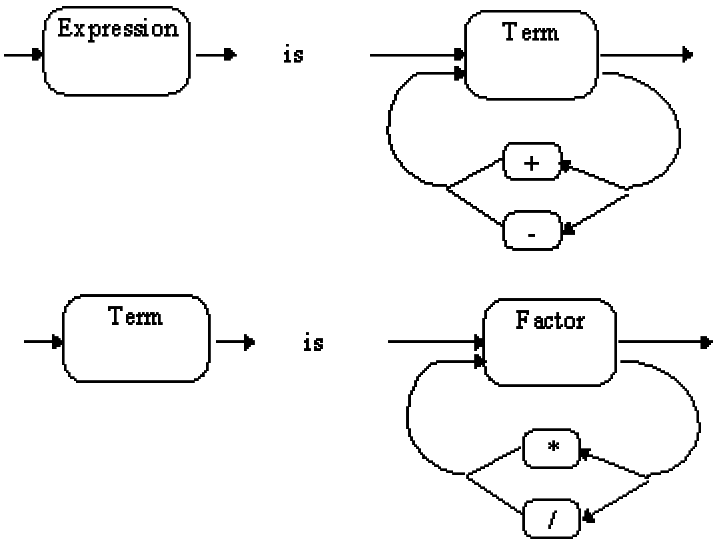


Figure. Extended productions for expressions and terms.

The phrase "zero or more occurrences" translates directly into the do/while loop in a recursive parser. Below is the corrected implementation of `Parser::Expr`.

```
Node * Parser::Expr ()
{
    // Parse a term
    Node * pNode = Term ();
    EToken token = _scanner.Token ();
    if (token == tPlus || token == tMinus)
    {
        // Expr := Term { ('+' | '-') Term }
        MultiNode * pMultiNode = new SumNode (pNode);
        do
        {
            _scanner.Accept ();
            Node * pRight = Term ();
            pMultiNode->AddChild (pRight, (token == tPlus));
            token = _scanner.Token ();
        } while (token == tPlus || token == tMinus);
        pNode = pMultiNode;
    }
    else if (token == tAssign)
    {
        _scanner.Accept ();
        // Expr := Term = Expr
        Node * pRight = Expr ();
        // provided the Term is an lvalue
        if (pNode->IsLvalue ())
        {
            // Assignment node
            pNode = new AssignNode (pNode, pRight);
        }
        else
        {
            _status = stError;
            delete pNode;
            pNode = Expr ();
        }
    }
    // otherwise Expr := Term
    return pNode;
}
```

The coding of Parser::Term undergoes a similar transformation.

```
Node * Parser::Term ()
{
    Node * pNode = Factor ();
    EToken token = _scanner.Token ();
    if (token == tMult || token == tDivide)
    {
        // Term := Factor { ('*' | '/') Factor }
        MultiNode * pMultiNode = new ProductNode (pNode);
        do
        {
            _scanner.Accept ();
            Node * pRight = Factor ();
            pMultiNode->AddChild (pRight, (token == tMult));
            token = _scanner.Token ();
        } while (token == tMult || token == tDivide);
        pNode = pMultiNode;
    }
    // otherwise Term := Factor
    return pNode;
}
```

We have introduced two new types of Nodes: SumNode and ProductNode. They both share the property that they can have multiple children, and that these children have a 'sign' associated with each of them. In the SumNode the sign is plus or minus, in the ProductNode it could be the multiplication or the division sign. We will store pointers-to-children in one array and the corresponding signs in another array. The convention will be that addition and multiplication signs are stored as *true* while subtraction and division signs are stored as *false* in the _isPositive Boolean array. The distinction will be obvious in the classes derived from the (abstract) base class MultiNode. MultiNode will represent what’s common between SumNode and ProductNode. For our first sloppy implementation we will arbitrarily restrict the number of children to 8--to be fixed later! Sloppy or not, we will test for the overflow and return an error when the user writes an expression (or term) with more than 8 children nodes.

```
const int MAX_CHILDREN = 8;
// Generic multiple node: an abstract class
class MultiNode: public Node
{
public:
    MultiNode (Node * pNode)
    : _isError (false)
    {
        _aChild [0] = pNode;
        _aIsPositive[0] = true;
        _iCur = 1;
    }
    ~MultiNode ();
    void AddChild (Node * pNode, bool isPositive)
    {
        if (_iCur == MAX_CHILDREN)
        {
            _isError = true;
            return;
        }
        _aChild [_iCur] = pNode;
        _aIsPositive [_iCur] = isPositive;
        ++_iCur;
    }
protected:
    bool     _isError;
    int      _iCur;
    Node   * _aChild [MAX_CHILDREN];
    bool     _aIsPositive [MAX_CHILDREN];
};
```

An array of Boolean values can be most simply implemented as an array of bool. It is however not the most space efficient implementation. To store two possible values, true and false, all one needs is a bit. The standard C++ library uses this optimization in its implementation of the Boolean array.

Here are the definitions of the two derived classes. Notice that since MultiNode *is a* Node, so is SumNode and ProductNode (the is-a relation is transitive).

```
// Summing
class SumNode: public MultiNode
{
public:
    SumNode (Node * pNode)
        : MultiNode (pNode) {}
    double Calc () const;
};

// Multiplying and dividing.
// Sign in this case refers to
// the exponent: positive means multiply,
// negative means divide
class ProductNode: public MultiNode
{
public:
    ProductNode (Node * pNode)
        : MultiNode (pNode) {}
    double Calc () const;
};
```

Don’t forget to delete the children after we’re done.

```
MultiNode::~MultiNode ()
{
    for (int i = 0; i < _iCur; ++i)
        delete _aChild [i];
}
```

And here comes our *pièce de résistance*. The Calc methods that makes our algebra left associative.

```
double SumNode::Calc () const
{
    if (_isError)
    {
        cerr << "Error: too many terms\n";
        return 0.0;
    }

    double sum = 0.0;
    for (int i = 0; i < _iCur; ++i)
    {
        double val = _aChild [i]->Calc ();
        if (_aIsPositive [i])
            sum += val;
        else
            sum -= val;
    }
    return sum;
}

// Notice: the calculation is left associative
double ProductNode::Calc () const
{
    if (_isError)
    {
        cerr << "Error: too many terms\n";
        return 0.0;
    }

    double prod = 1.0;
    for (int i = 0; i < _iCur; ++i)
    {
        double val = _aChild [i]->Calc ();
        if (_aIsPositive [i])
            prod *= val;
        else if (val != 0.0)
        {
            prod /= val;
        }
        else
        {
            cerr << "Error: division by zero\n";
            return HUGE_VAL;
        }
    }
    return prod;
```

```
 }
```

Code Review 2: Hiding Implementation Details

Embedded classes, protected constructors, hiding constants, anonymous enums, namespaces.

A good software engineer is like a spy. He exposes information to his collaborators on the *need-to-know* basis, because knowing too much may get them in trouble. I'm not warning here about the dangers of industrial espionage. I'm talking about the constant struggle with complexity. The more details are hidden, the simpler it is to understand what's going on.

Using Embedded Classes

The class Link is only used internally by the linked list and its friend the sequencer. Frankly, nobody else should even know about its existence. The potential for code reuse of the class Link outside of the class List is minimal. So why don't we hide the definition of Link inside the private section of the class definition of List.

```
class List
{
    friend class ListSeq;
public:
    List ();
    ~List ();
    void Add (int id);
private:
    // nested class definition
    class Link
    {
    public:
        Link (Link * pNext, int id)
            : _pNext (pNext), _id (id) {}

        Link *   Next () const { return _pNext; }
        int        Id () const { return _id; }
    private:
        Link     * _pNext;
        int        _id;
    };

private:
    Link const * GetHead () { return _pHead; }

    Link* _pHead;
};
```

The syntax of class embedding is self-explanatory.

Class ListSeq has a data member that is a pointer to Link. Being a friend of List, it has no problem accessing the private definition of class Link. However, it has to qualify the name Link with the name of the enclosing class List--the new name becomes List::Link.

```
class ListSeq
{
public:
    bool AtEnd () const { return _pLink == 0; }
    void Advance () { _pLink = _pLink->Next(); }
    int GetId () const { return _pLink->Id (); }
protected:
    ListSeq (List const & list)
        : _pLink (list.GetHead ()) {}
private:
    // usage of nested class
    List::Link const *_pLink;
};
```

One way to look at a class declaration is to see it as a collection of methods, data members and types. So far we've been dealing only with methods and data members; now we can see how a type--in this case, another class--is defined within a class. A class may also create aliases for other types using typedefs. We'll see more examples of these techniques later.

The classes List and ListSeq went through some additional privatization (in the case of ListSeq, it should probably be called "protectization"). I made the GetHead method private, but I made ListSeq a friend, so it can still call it. I also made the constructor of ListSeq protected, because we never create it in our program--we only use objects of the derived class, IdSeq.

I might have gone too far with privatization here, making these classes more difficult to reuse. It's important, however, to know how far you can go and make an informed decision when to stop.

Combining Classes

Conceptually, the sequencer object is very closely tied to the list object. This relationship is somehow reflected in our code by having ListSeq be a friend of List. But we can do much better than that--we can embed the sequencer class inside the list class. This time, however, we don't want to make it private--we want the clients of List to be able to use it. As you know, ouside of the embedding class, the client may only access the embedded class by prefixing it with the name of the outer class. In this case, the (scope-resolution) prefix would be List::. It makes sense then to shorten the name of the embedded class to Seq. On the outside it will be seen as List::Seq, and on the inside (of List) there is no danger of name conflict.

Here's the modified declaration of List:

```
class List
{
public:
    List ();
    ~List ();
    void Add (int id);
private:
    class Link
    {
    public:
        Link (Link * pNext, int id)
            : _pNext (pNext), _id (id) {}

        Link *  Next () const { return _pNext; }
        int     Id () const { return _id; }
    private:
        Link *  _pNext;
        int     _id;
    };

public:
    class Seq
    {
    public:
        Seq (List const & list)
            : _pLink (list.GetHead ()) {}
        bool AtEnd () const { return _pLink == 0; }
        void Advance () { _pLink = _pLink->Next (); }
        int GetId () const { return _pLink->Id (); }
    private:

        Link const * _pLink; // current link
    };

    friend Seq;
private:
    Link const * GetHead () const { return _pHead; }

    Link * _pHead;
};
```

Notice, by the way, how I declared Seq to be a friend of List following its class declaration. At that point the compiler knows that Seq is a class.

The only client of our sequencer is the hash table sequencer, IdSeq. We have to modify its definition accordingly.

```
class IdSeq: public List::Seq
{
public:
    IdSeq (HTable const & htab, char const * str)
        : List::Seq (htab.Find (str)) {}
};
```

And, while we're at it, how about moving this class definition where it belongs, inside the class HTable? As before, we can shorten its name to Seq and export it as HTable::Seq. And here's how we will use it inside SymbolTable::Find

```
for (HTable::Seq seq (_htab, str);
     !seq.AtEnd ();
     seq.Advance ())
```

Combining Things using Namespaces

There is one more example of a set of related entities that we would like to combine more tightly in our program. But this time it's a mixture of classes and data. I'm talking about the whole complex of FunctionTable, FunctionEntry and FunctionArray (add to it also the definition of CoTan which is never used outside of the context of the function table). Of course, I could embed FunctionEntry inside FunctionTable, make CoTan a static method and declare FunctionArray a static member (we discussed this option earlier). There is however a better solution. In C++ we can create a higher-level grouping called a namespace. Just look at the names of objects we're trying to combine. Except for CoTan, they all share the same prefix, Function. So let's call our namespace Function and start by embedding the class definition of Table (formerly known as FunctionTable) in it.

```
namespace Function
{
    class Table
    {
    public:
        Table (SymbolTable & symTab);
        ~Table () { delete []_pFun; }
        int Size () const { return _size; }
        PFun GetFun (int id) { return _pFun [id]; }
    private:
        PFun * _pFun;
        int    _size;
    };
}
```

The beauty of a namespace it that you can continue it in the implementation file. Here's the condensed version of the file **funtab.cpp**:

```
namespace Function
{
    double CoTan (double x) {...}
    class Entry {...};
    Entry Array [] =
    {...};
    Table::Table (SymbolTable & symTab)
        : _size(sizeof Array / sizeof Array [0])
    {...}
}
```

As you might have guessed, the next step is to replace all occurrences of FunctionTable in the rest of the program by Function::Table. The only tricky part is the forward declaration in the header **parse.h**. You can't just say class Function::Table;, because the compiler hasn't seen the declaration of the Function namespace (remember, the point of using a forward declaration was to avoid including **funtab.h**). We have to tell the compiler not only that Table is a class, but also that it's declared inside the Function namespace. Here's how we do it:

```
namespace Function
{
    class Table;
}

class Parser
{
public:
    Parser (Scanner & scanner,
            Store & store,
            Function::Table & funTab,
            SymbolTable & symTab );
    ...
};
```

> As a general rule, a lot of traditional naming conventions for classes, functions and objects are being replaced in modern C++ by by the use of embedding classes or namespaces and scope resolution prefixes.

By the way, the whole C++ Standard Library is enclosed in a namespace. Its name is std. Now you understand these prefixes std:: in front of cin, cout, endl, etc. (You've also learned how to avoid these prefixes using the using keyword.)

Hiding Constants in Enumerations

There are several constants in our program that are specific to the implementation of certain classes. It would be natural to hide the definitions of these constants inside the definitions of classes that use them. It turns out that we can do it using enums. We don't even have to give names to enums—they can be anonymous.

Look how many ways of defining constants there are in C++. There is the old-style C #define preprocessor macro, there is a type-safe global const and, finally, there is the minimum-scope enum. Which one is the best? It all depends on type. If you need a typed constant, say, a double or a (user defined) Vector, use a global const. If you just need a generic integral type constant—as in the case of an array bound—look at its scope. If it's needed by one class only, or a closely related group of classes, use an enum. If its scope is larger, use a global const int. A #define is a hack that can be used to bypass type checking or avoid type conversions--avoid it at all costs. By the way, debuggers don't see the names of constants introduced through #defines. They appear as literal numerical values. It might be a problem sometimes when you don't remember what that 74 stood for.

Here's the first example of hiding constants using enumerations. The constant idNotFound is specific to the SymbolTable.

```
class SymbolTable
{
public:
    // Embedded anonymous enum
    enum { idNotFound = -1 };
    …
}
```

No change is required in the implementation of Find. Being a method of SymbolTable, Find can access idNotFound with no qualifications.

Not so with the parser. It can still use the constant idNotFound, since it is defined in the public section of SymbolTable, but it has to qualify its name with the name of the class where it's embedded.

```
if (_scanner.Token () == tLParen) // function call
{
    _scanner.Accept (); // accept '('
    pNode = Expr ();
    if (_scanner.Token() == tRParen)
        _scanner.Accept (); // accept ')'
    else
        _status = stError;
    // The use of embedded enum
    if (id != SymbolTable::idNotFound
        && id < _funTab.Size ())
    {
        pNode = new FunNode (
            _funTab.GetFun (id), pNode );
    }
    else
    {
        cerr << "Unknown function \"";
        cerr << strSymbol << "\"\n";
    }
}
else
{
    // Factor := Ident
    if (id == SymbolTable::idNotFound)
    {
        // add new identifier to the symbol table
        id = _symTab.ForceAdd (strSymbol);
        if (id == SymbolTable::idNotFound)
        {
            cerr << "Error: Too many variables\n";
            _status = stError;
            pNode = 0;
        }
    }
    if (id != SymbolTable::idNotFound)
        pNode = new VarNode (id, _store);
}
```

Maximum symbol length might be considered an internal limitation of the Scanner (although one might argue that it is a limitation of the "language" that it recognizes--we'll actually remove this limitation later).

```
class Scanner
{
public:
    // Embedded anonymous enum
    enum { maxSymLen = 80 };
    …
};
```

Hiding Constants in Local Variables

A constant that is only used within a single code fragment should not, in general, be exposed in the global scope. It can as well be defined within the scope of its usefulness. The compiler will still do inlining of such constants (that is, it will substitute the occurrences of the constant name with its literal value, rather than introducing a separate variable in memory). We have a few such constants that are only used in main

```
int main ()
{
    const int maxBuf = 100;
    const int maxSymbols = 40;

    char buf [maxBuf];
    Status status;
    SymbolTable symTab (maxSymbols);
    …
}
```

Code Review 3: Sharing

Have you noticed how we oscillate between global and local perspective in our code reviews? Whenever we talk about sharing, we take the more global stance. Whenever we talk about data or implementation hiding, we take a local stance. It's time to swing the pendulum towards sharing.

Isolating Global Program Parameters

From the global perspective some of the constants that we've been so busily burying inside local scopes are actually tunable parameters of our program. What if somebody (say, our client) wants to increase the maximum length of symbolic names? Or what if he wants to be able to add more entries to the symbol table? Or desperately needs to increase the size of the input buffer? We'll have to dig deep into our classes and functions to find the appropriate constants.

Fortunately, there are several ways to make such fine tuning less cumbersome. I'll only show the most basic one-collecting all tunable constants in a single file **params.h**. Here's the contents of this file

```
const int maxBuf = 100;        // size of input buffer
const int maxSymbols = 40;     // size of symbol table
const int maxSymLen = 80;      // max length of symbol name
```

There will always be some tunable parameters in your program whose change will require recompilation. Their place is in **params.h**.

There usually is another set of parameters that should be tunable by the user (or the administrator). These are called user preferences and are stored persistently in some convenient location. Even these parameters have their default values whose place is, you guessed it, in **params.h**.

Testing Boundary Conditions

We know that our program has built-in limitations: see **params.h**. Obviously we want to make these limitations as unobtrusive as possible. We generously allocate space for symbols, characters in buffers, etc., so that in day-to-day operation the user doesn't hit these limits. Unfortunately, that also means that we, the programmers, are unlikely to hit these limitations in our testing. And rest assured--whatever boundaries we don't hit, our users will!

So let's make a little experiment. Let's edit the file **params.h** and set all the limits to some relatively low values. We might even keep two sets of parameters--one for regular operation and one for testing. I used conditional compilation directives to switch between the two sets. Changing 0 to 1 in the #if directive will switch me back to the regular set.

```
#if 0
const int maxBuf = 100;     // size of input buffer
const int maxSymbols = 40;  // size of symbol table
const int maxSymLen = 80;   // max length of symbol name
#else
const int maxBuf = 8;     // size of input buffer
const int maxSymbols = 5;  // size of symbol table
const int maxSymLen = 4;   // max length of symbol name
#endif
```

After recompiling the whole program, I ran it and immediately hit an assertion in the constructor of the Function::Table. Quick inspection showed that the program was trying to cram 14 symbols for built-in functions into our symbol table, whose maximum capacity was set to 5. That's good! Obviously the value of 5 for the size of the symbol table was too small, but we were able to catch it immediately.

Is an assertion good enough protection from this kind of a problem? In general, yes. This problem might only arise when somebody changes the value of maxSymLen or ads more built-in functions. We are safe in that case, because:

After every code change the program will be re-built in the debugging version (with the assertions turned on) and run at least once before it's released to others.

By "others" I don't mean customers--I mean other programmers on the same team. Before releasing a program to customers, much stricter testing is required.

So let's now set the size of the symbol table to 14--just enough for built-in functions--and try to enter an expression that introduces a new symbolic variable. Kaboom! We get a general protection fault!

What happened? In Parser::Factor we correctly discovered a problem, even printed out "Error: Too many variables," and then returned a null pointer. Well, what else were we supposed to do?! Unfortunately, the caller of Parser::Factor never expected a null pointer. Back to the drawing board!

Let's add checks for a null pointer to all the callers of Parser::Factor (as well as the callers of Parser::Term, who can propagate the null obtained from Parser::Factor).

One more test--this time everything works fine; that is until we exit the program. Remember, all this time we are running a debug build of our program under the debugger. A good debug runtime will do a heap check when your program frees memory. A heap check should discover such problems as buffer overflows, double deletions, etc. And indeed it does! The destructor of Store reports an overwritten memory block. A little more sleuthing and the culprit becomes obvious. Look at this code in the constructor of Store:

```
int id = symTab.ForceAdd ("e");
SetValue (id, exp (1));
cerr << "pi = " << 2 * acos (0.0) << endl;
id = symTab.ForceAdd ("pi");
SetValue (id, 2 * acos (0.0));
```

When our symbol table overflows, it returns -1. SymbolTable::SetValue is called with -1 and happily obliges.

```
void SetValue (int id, double val)
{
    assert (id < _size);
    _cell [id] = val;
    _status [id] = stInit;
}
```

We were smart enough to assert that id be less than the size of the table, but not smart enough to guard against a negative value of id.

Okay, let's change that and try again. This time we hit the assertion immediately. Great! One more test with maxSymbols = 16 and we're in the open.

Now let's test the restriction on the size of the input buffer. Let's input a large expression, something like one with ten zeros. The program works, but not exactly as the user would expect. It first prints the incorrect result, then exits.

First of all, why does it exit? It turns out that, since it didn't retrieve all the characters from the input on the first try, the next time it calls cin.getline it returns immediately with an empty buffer (I'm not sure why it's empty). But that's not our biggest problem. A program that quietly returns incorrect results is much worse than the one that crashes. Incorrect information is worse than no information at all. We have to fix it immediately.

```
char buf [maxBuf+1];
// ...
cin.getline (buf, maxBuf+1);
if (strlen (buf) == maxBuf)
{
    cerr << "Error: Input buffer overflow\n";
    status = stError;
    break;
}
```

Notice what I have done. I increased the size of the buffer by one, so that I could grab one character more than the self-imposed limit. Now I can detect if my greedy request was satisfied and, if so, I know that the input was larger than maxBuf and I bail out.

Now for the last test--maximum symbol length. Before we try that, we have to increase maxSymbols, so that we can enter a few test symbol into our program. What we discover is that, although the program works, it quietly lies to the user. Try inputing and expression toolong = 6 and then displaying the value of another variable tool. Instead of complaining about the use of unitialized variable, the program quietly treats both names as representing the same variable. The least we can do is to display a warning in Scanner::Accept.

```
if (_lenSymbol > maxSymLen)
{
    cerr << "Warning: Variable name truncated\n";
    _lenSymbol = maxSymLen;
}
```

Scary, isn't it? Why was this program so buggy? Was this the norm or an exception? The truth is that, no matter how good you are, your programs will contain errors. The compiler will catch most of the simple errors--typos, type mismatches, wrong number of arguments, etc. Many errors will become obvious after some elementary testing. The rest are the "hard bugs".

There are many techniques to eradicate hard bugs. One is testing--in particular you can make testing easier by instrumenting your program--writing code whose only purpose is to catch bugs. We've done something like that by instrumenting the file **params.h**. I'll show you more testing and instrumentation techniques later.

There is also the matter of attitude. I wrote this program without much concern for error checking and exceptional conditions. Normally I would be much more careful.

Finally, there is the matter of policy. Dealing with exceptional conditions is not something you do on a whim. You have to develop some rules. We'll talk about it some more.

Templates

We have put the class List together with its sequencer in a standalone file with code reuse in mind. Any time in the future we need a linked list of integers we can just include this well tested and solid code. Did I just say "a linked list of integers"? What if we need a list of unsigned longs or doubles or even Stars for that matter? We need a class that is parametrized by a type. Enter templates.

To turn List into a template, you need to preface its class definition with

```
template <class T>
```

and replace ints with Ts. You are parametrizing the definition of List with the parameter T that stands for any type. Once you have the template defined, you can instantiate it by substituting any type for T--it doesn't even have to be a class. For instance,

```
List<int> myList;
```

will define a List of integers called myList. But you are also free to declare other lists, such as a list of unsigned longs, List<unsigned long>, a list of doubles, List<double> or a list of Stars, List<Star>.

Here's the definition of the List template.

```
template<class T>
class List
{
public:
    List ();
    ~List ();
    void Add (T value);
private:
    class Link
    {
    public:
        Link (Link * pNext, T value)
            : _pNext (pNext), _value (value) {}

        Link *  Next () const { return _pNext; }
        T       GetValue () const { return _value; }
    private:
        Link *  _pNext;
        T       _value;
    };

public:
    class Seq
    {
    public:
        Seq (List const & list)
            : _pLink (list.GetHead ()) {}
        bool AtEnd () const { return _pLink == 0; }
        void Advance () { _pLink = _pLink->Next (); }
        T GetValue () const { return _pLink-gt;GetValue (); }
    private:

        Link const * _pLink; // current link
    };

    friend Seq;
private:
    Link const * GetHead () const { return _pHead; }

    Link * _pHead;
};
```

To be more general, I've changed the name of the method that retrieves the stored value to GetValue.

For technical reasons, today's compilers require all template code to be visible in the place of template instantiation (this requirement might change in future compilers). What that means is that we have to put all the method implementations in the same header file as the template class definition. It doesn't make these methods inline (unless they are declared as such) nor does it imply that their code will be instantiated multiple times (unless you have a very cheap compiler).

Here's how you define the constructor--notice that only the first List, the one before the double colon, is followed by <T>.

```
template <class T>
List<T>::List ()
    : _pHead (0)
{}
```

Same with the destructor:

```
template <class T>
List<T>::~List ()
{
    // free linked list
    while (_pHead != 0)
    {
        Link * pLink = _pHead;
        _pHead = _pHead->Next ();
        delete pLink;
    }
}
```

Other methods follow the same pattern.

```
template <class T>
void List<T>::Add (T val)
{
    Link * pLink = new Link (_pHead, val);
    _pHead = pLink;
}
```

The second part of "templatization" is to substitute all the uses of List of integers with List<int>

```
class HTable
{
public:
    explicit HTable (int size): _size(size)
    {
        _aList = new List<int> [size];
    }

    ~HTable ()
    {
        delete []_aList;
    }

    void Add (char const * str, int id);
public:
    class Seq: public List<int>::Seq
    {
    public:
        Seq (HTable const & htab, char const * str)
            : List<int>::Seq (htab.Find (str)) {}
    };

    friend Seq;
private:
    List<int> const & Find (char const * str) const;
    int hash (char const * str) const;

    List<int> * _aList;
    int            _size;
};
```

To summarize: A class template is a definitions of a class that is parametrized by one or more types. The type parameters are listed within angle brackets after the keyword template.

template <class type1, class type2 ...>

To instantiate the template it is enough to use it, with the type parameters substituted by actual types (either built-in or user-defined).

Code Review 4: Removing Limitations

Dynamic arrays, overloading of the array access operator, separating string table

{
source

Don't you just hate it when you're trying to save a few hours' worth of editing and suddenly you get a message box saying "Cannot save: Too many windows open." or some such nonsense? What is it? Unfortunately, it's a common occurrence--somebody who considers himself or herself a programmer hardcoded a fixed size array to store some important program resources. Then somebody found a problem during testing. The limitation probably caused a fault when the bounds were first overwritten. A bug appeared in the database, "The program GP-faulted when user tried to save a big document." Then another genius fixed it by adding a boundary test and putting up the message box. At least that's how I imagine it happening. As you can see, I don't think highly of code that has unreasonable limitations built-in. And with templates it's so easy to remove the unnecessary limits!

Let's first look around to see where in our program we managed to impose unreasonable limitations due to our ignorance of templates. There's one we have just created in `MultiNode` by arbitrarily limiting the number of children to 8. The whole symbol table is infested with limitations: the buffer for string is limited, the array of offsets is finite. Finally, the storage of variable values has an arbitrary limit imposed on it.

You might ask, "What's the likelihood of user hitting one of the limitations? Is it worth going into all this trouble in order to deal with such unlikely cases?" The answer is:

It is less trouble to use dynamic arrays than it is to deal with limitations.

Just think of all the error messages you'll have to display, the propagation of these errors (by the way, we'll deal with this topic separately later), the localization of message strings. Believe me, you don't want to do it.

Dynamic Array

Almost all cases of artificial limits can be solved by just one template--the dynamic array. Let's go together through its design and implementation. In the process you'll not only learn more about templates and operator overloading, but you'll also prepare yourself for the general-purpose dynamic `vector` from the Standard Library.

Let's first try to come up with a set of requirements for this data structure. As much as possible we would like the dynamic array to look just like an array. That means we would like to access its elements by indexing, both to read and to write. Adding new elements is a bit more tricky. For performance reasons, we don't want the array to check for bounds on every access (except in the debugging version) to see if it's necessary to extend the array. Second, we never actually need to add new elements at an arbitrary offset. In all our applications we keep adding elements to the end of the array. It would be tempting to just keep track of the last added element and have method `Push` to add and extend the array. Instead I decided to be a little more general and have a method `Add` that can add a new element at an arbitrary offset. That's because

I want to be able to use pairs of synchronized arrays that share the same insertion index--we'll see an example in a moment.

One more thing: in many cases it makes sense to pre-fill the array with some default values. For instance, an array of pointers could be pre-filled with null pointers, integers could be preset to zero or minus one, enumerated types to some default value, etc. We'll store this default value in the dynamic array, so that we could use it to pre-fill the new pieces of the array when we grow it.

```
template <class T>
class DynArray
{
    enum { initDynArray = 8 };
public:
    explicit DynArray (T const valDefault);
    ~DynArray ();
    void Add (int i, T const val);
    T operator [] (int i) const;
    T & operator [] (int i);
    int InRange (int i) const { return i < _capacity; }
private:
    void Grow (int i);

    T     * _arr;
    int     _capacity; // size of the array
    T const _valDefault;
};
```

The line

```
T operator [] (int i) const;
```

is another example of operator overloading. Here we are overloading the array access operator [] (square brackets) in order to be able to access dynamic arrays just like regular arrays.

Here's how we can define a dynamic array of integers, add a value to it using Add, and retrieve it using array access.

```
DynArray<int> a (0);    // dynamic array with 0 default value
a.Add (3, 10);  // add entry with value 10 at offset 3
int i = a [3];  // access it like a regular array
```

The compiler will translate the last line into the call to operator [], just like this:

```
int i = a.operator [] (3);
```

which, by the way, is also correct alternative syntax.

Notice that we have another overloading of the same operator in the same class:

```
T & operator [] (int i);
```

How can this be done? It's the same operator, takes the same arguments, but it returns T by reference, not by value. Is it enough for the compiler to decide between these two when is sees, for instance, a[3]? The answer is, no--two overloads that only differ by return type are not allowed. What really makes a difference here is the word const. One method is constant and the other is not. When acting on a const object, the compiler will pick the const method, otherwise it will use the non-const one.

The non-const, reference-returning version of the operator allows us to use the result of array indexing as an lvalue. The following statement "does the right thing"

```
a [3] = 11;
```

Compare it with the more verbose version:

```
int & r = a.operator [] (3);  // reference to the third element of 'a'
r = 11; // change value through a reference
```

In the above example, if a were defined to be const, the compiler would flag it as an error. And rightly so! You are not supposed to change the value of a const object. That, by the way, shows you why I had to define two versions of this method.

If I only had the non-const version, I wouldn't be able to use it on const objects or in const methods of DynArray. In fact our program wouldn't compile for exactly that reason.

If I only had the const version, I wouldn't be able to modify the elements of the array using indexing--I couldn't use them as lvalues.

There is a third possibility: I could say that by returning a reference I am not really changing the state of the object. Let's try this

```
T & operator [] (int i) const // <-- const method?!
{
    return _arr[i];  // <-- ERROR!
}
```

Oops! It didn't work! It's really getting harder and harder to shoot yourself in the foot, doesn't it?

Let me try to explain why the compiler wouldn't let us compile the above code. It's because of the this pointer. The this pointer is the pointer to the object upon which a given method was invoked. You can use this explicitly; or implicitly, by accessing the members of the object. The above code, for instance, is equivalent to

```
T & operator [] (int i) const
{
    return this->_arr[i]; // <- ERROR!
}
```

The type of the this pointer depends on the class of the object. It also depends on whether the method is const or not. Inside a non-const method of, say, class Foo , this acts as if it were declared as

```
Foo * const this;
```

whereas, in a const method, this becomes

```
Foo const * const this;
```

Therefore, inside a const method, the expression this-->_arr[i] produces a const integer, because it dereferences a pointer to const. The compiler catches us red-handed trying to create a "reference to non-const" (the return type) from a const integer. It's as bad as

```
const double speedOfLight = 300000.0; // km/s
double & mySpeed = speedOfLight;   // <-- ERROR! Won't compile!
// now let me slow down
mySpeed = 0.0;  // Oops! Changed the speed of light!
```

The bottom line is: If you want to have a method that can be used as an lvalue, because it returns a reference to the object's data, such method cannot be const. If you also want to overload the same method (that is have another method with the same name and argument types) to provide read-only access, you can do it by making the second method const. Its return type must be different, though. It could be a reference-to-const; or just the type of the value itself, if we want to return the data by value.

Although this technique is mostly used in the context of operator overloading, it can be used with any other kind of overloading. In principle, one could have

```
class Foo
{
public:
    Foo (double x) :_x(x) {}
    double & Get () { return _x; }
    double Get () const { return _x; }
    // Another possibility:
    // double const & Get () const { return _x; }
private:
    double _x;
};
```

and use `Get` both as an lvalue and an rvalue (right-hand-side value), as in

```
Foo varFoo (1.1);
const Foo constFoo (3.14);

varFoo.Get () = varFoo.Get () + 2.0 * constFoo.Get ();
```

I'm not saying that I recommend this kind of programming. A much better solution would be to provide a separate method `Set` and avoid overloading whatsoever.

In any case, it should be obvious that it is very important to be consistent--the two overloads should give access to the same piece of data.

The implementation of the dynamic array is pretty straightforward. Here's the constructor and the destructor.

```
template <class T>
DynArray <T>::DynArray (T const valDefault)
    : _capacity (initDynArray), _valDefault (valDefault)
{
    _arr = new T [initDynArray]; // allocate memory
    for (int i = 0; i < initDynArray; ++i)
        _arr[i] = _valDefault;
}

template <class T>
```

```
DynArray <T>::~DynArray ()
{
    delete []_arr;
}
```

Method Add has the potential of extending the array.

```
template <class T>
void DynArray <T>::Add (int i, T const val)
{
    if ( i >= _capacity )
        Grow(i);
    _arr [i] = val;
}
```

Here are the two versions of the array access operator. Notice that none of them can be used to extend the array--the assertions are there to make sure that nobody even tries.

```
template <class T>
inline T DynArray<T>::operator [] (int i) const
{
    assert (i < _capacity);
    return _arr[i];
}

template <class T>
inline T & DynArray<T>::operator [] (int i)
{
    assert (i < _capacity);
    return _arr[i];
}
```

The Grow method works by doubling the size of the array. However, when asked to extend the array beyond doubling, it will oblige, too.

```
// private method
template <class T>
void DynArray <T>::Grow (int idxMax)
{
    int newSize = 2 * _capacity;
    if (idxMax >= newSize)
        newSize = idxMax + 1;
    // allocate new array
    T * arrNew = new T [newSize];
    // copy all entries
    int i;
    for (i = 0; i < _capacity; ++i)
        arrNew [i] = _arr [i];
    for (; i < newSize; ++i)
        arrNew [i] = _valDefault;
    _capacity = newSize;
    // free old memory
    delete []_arr;
    // substitute new array for old array
    _arr = arrNew;
}
```

Now it's time to make use of our dynamic array template to see how easy it really is. Let's start with the class `MultiNode`. In the old, limited, implementation it had two arrays: an array of pointers to `Node` and an array of Boolean flags. Our first step is to change the types of these arrays to, respectively, `DynArray<Node*>` and `DynArray<bool>`. We have to pass default values to the constructors of these arrays in the preamble to `MultiNode` constructor. These methods that just access the arrays will work with no changes (due to our overloading of operator []), except for the places where we used to check for array bounds. Those are the places where we might have to extend the arrays, so we should use the new `Add` method. It so happens that the only place we do it is inside the `AddChild` method and the conversion is straightforward.

```
class MultiNode: public Node
{
public:
    MultiNode (Node * pNode)
        : _aChild (0),
          _aIsPositive (false),
          _iCur (0)
    {
        AddChild (pNode, true);
    }
    ~MultiNode ();
    void AddChild (Node * pNode, bool isPositive)
    {
        _aChild.Add (_iCur, pNode);
        _aIsPositive.Add (_iCur, isPositive);
        ++_iCur;
    }
protected:
    int                   _iCur;
    DynArray<Node*>       _aChild;
    DynArray<bool>        _aIsPositive;
};

MultiNode::~MultiNode ()
{
    for (int i = 0; i < _iCur; ++i)
        delete _aChild [i];
}
```

Let's have one more look at the `Calc` method of `SumNode`. Other than for the removal of error checking (we have gotten rid of the unnecessary flag, `_isError`), it works as if nothing have changed.

```
double SumNode::Calc () const
{
    double sum = 0.0;
    for (int i = 0; i < _iCur; ++i)
    {
        double val = _aChild [i]->Calc ();
        if (_aIsPositive[i])
            sum += val;
        else
            sum --= val;
    }
    return sum;
}
```

The only difference is that when we access our arrays _aChild [i] and _aIsPositive [i], we are really calling the overloaded operator [] of the respective dynamic arrays. And, by the way, since the method Calc is const, it is the const version of the overload we're calling. Isn't that beautiful?

Separating Functionality into New Classes

I'm not happy with the structuring of the symbol table. Just one look at the seven data members tells me that a new class is budding. (Seven is the magic number.) Here they are again:

```
HTable   _htab;
int    * _offStr;
int      _capacity;
int      _curId;
char   * _strBuf;
int      _bufSize;
int      _curStrOff;
```

The rule of thumb is that if you have too many data members you should group some of them into new classes. This is actually one of the three rules of class formation. You need a new class when there are

- too many local variables in a function,
- too many data members in a class or
- too many arguments to a function.

It will become clearer why these rules make perfect sense (and why seven is the magic number) when we talk about ways of dealing with complexity in the third part of this book.

The last three data members of the symbol table are perfect candidates for a new string-buffer object. The string buffer is able to store strings and assign them numbers, called offsets, that uniquely identify them. As a bonus, we'll make the string buffer dynamic, so we won't have to worry about overflowing it with too many strings.

```
class StringBuffer
{
public:
    StringBuffer ()
        : _strBuf (0), _bufSize (0), _curStrOff (0)
    {}
    ~StringBuffer ()
    {
        delete _strBuf;
    }
    int AddString (char const * str);
    char const * GetString (int off) const
    {
        assert (off < _curStrOff);
        return &_strBuf [off];
    }
private:
    void Reallocate (int addLen);

    char  * _strBuf;
    int     _bufSize;
    int     _curStrOff;
};
```

When the buffer runs out of space, the AddString method reallocates the whole buffer.

```
int StringBuffer::AddString (char const * str)
{
    int len = strlen (str);
    int offset = _curStrOff;
    // is there enough space?
    if (_curStrOff + len + 1 >= _bufSize)
    {
        Reallocate (len + 1);
    }
    // copy the string there
    strncpy (&_strBuf [_curStrOff], str, len);
    // calculate new offset
    _curStrOff += len;
    _strBuf [_curStrOff] = 0;  // null terminate
    ++_curStrOff;
    return offset;
}
```

The reallocation follows the standard doubling pattern--but making sure that the new string will fit no matter what.

```
void StringBuffer::Reallocate (int addLen)
{
    int newSize = _bufSize * 2;
    if (newSize <= _curStrOff + addLen)
        newSize = _curStrOff + addLen;
    char * newBuf = new char [newSize];
    for (int i = 0; i < _curStrOff; ++i)
        newBuf [i] = _strBuf [i];
    delete []_strBuf;
    _strBuf = newBuf;
    _bufSize = newSize;
}
```

Now the symbol table is much simpler. It only has four data members. I also used this opportunity to turn the array _aOffStr into a dynamic array.

```
class SymbolTable
{
    // Private embedded anonymous enum
    enum { cSymInit = 64 };
public:
    // Public embedded anonymous enum
    enum { idNotFound = -1 };
    SymbolTable ();
    int ForceAdd (char const * str);
    int Find (char const * str) const;
    char const * GetString (int id) const;
private:
    int               _curId;
    HTable            _htab;
    DynArray<int>     _aOffStr; // offsets of strings in buffer
    StringBuffer      _bufStr;
};
```

Here's the new improved implementation of ForceAdd. We don't have to worry any more about overflowing the offset table or the string buffer.

```
int SymbolTable::ForceAdd (char const * str)
{
    int offset = _bufStr.AddString (str);
    _aOffStr.Add (_curId, offset);
    _htab.Add (str, _curId);
    ++_curId;
    return _curId - 1;
}
```

It's always a pleasure to be able to simplify some code. This is the part of programming that I like best. ForceAdd now reads almost like a haiku.

This is how the constructor of the symbol table shrinks

```
SymbolTable::SymbolTable ()
    : _curId (0),
      _htab (cSymInit + 1),
      _aOffStr (0)
{}
```

And this is how simple the parsing of a symbolic variable becomes

```
// Factor := Ident
if (id == SymbolTable::idNotFound)
    id = _symTab.ForceAdd (strSymbol);
assert (id != SymbolTable::idNotFound);
pNode = new VarNode (id, _store);
```

Class Store needs two dynamic arrays and a few little changes where the bounds used to be checked.

```
class Store
{
private:
    // private anonymous enum
    enum { stNotInit, stInit };
public:
    explicit Store (SymbolTable & symTab);
    bool IsInit (int id) const
    {
        assert (id >= 0);
        return _aStatus.InRange (id)) &&
               _aStatus [id] != stNotInit;
    }
    double Value (int id) const
    {
        assert (id >= 0);
        assert (IsInit (id));
        return _aCell [id];
    }
    void SetValue (int id, double val)
    {
        assert (id >= 0);
        if (_aCell.InRange (id))
        {
            _aCell [id] = val;
            _aStatus [id] = stInit;
        }
        else
        {
            AddValue (id, val);
        }
    }

    void AddValue (int id, double val)
    {
        assert (id >= 0);
        _aCell.Add (id, val);
        _aStatus.Add (id, stInit);
    }
private:
    DynArray<double>           _aCell;
    DynArray<unsigned char>    _aStatus;
};
```

In the preamble to the constructor of Store we construct the dynamic arrays. The memory cells are initialized to zero and the statuses to stNotInit.

```
Store::Store (SymbolTable & symTab)
    : _aCell (0.0), _aStatus (stNotInit)
{
    // add predefined constants
    // Note: if more needed, do a more general job
    cerr << "e = " << exp (1) << endl;
    int id = symTab.ForceAdd ("e");
    AddValue (id, exp (1));
    cerr << "pi = " << 2 * acos (0.0) << endl;
    id = symTab.ForceAdd ("pi");
    AddValue (id, 2 * acos (0.0));
}
```

Finally, our getting rid of built in limitations is reflected in some nice simplifications in main. We no longer need to pass size arguments to the constructors.

```
int main ()
```

```
{
    ...
    SymbolTable symTab;
    Function::Table funTab (symTab);
    Store store (symTab);
    ...
}
```

Standard Vector

Now that you know how a dynamic array works, it's time to introduce the most useful member of the Standard Library, the vector. The vector is all that you might expect from a dynamic array and more. Without further ado, let's quickly substitute all the occurrences of DynArray with std::vector (yes, it's a member of the std namespace), making sure we include the appropriate header:

```
#include <vector>
```

Let's start with MultiNode and its two dynamic arrays.

```
std::vector<Node*> _aChild;
std::vector<bool>  _aIsPositive;
```

Before our code can compile, we have to make a few adjustments. For instance, std::vector doesn't have the Add method. We can, however, append new elements by calling push_back, which has the side effect of growing the array if necessary. (Notice, by the way, the particular naming convention of the stadard library. All names are lower case with underscores for word separators. You'll have no problems distinguishing them from our own names.)

Another thing, there seems to be no simple way of providing the default values for the elements of a vector. Fortunately, the standard library uses default constructors to fill empty spaces in vectors and other containers. It so happens that C++ provides default constructors for all built-in types. For instance, the default value for an int or a pointer is zero, for a bool, false, etc. Try it!

```
int i = int ();
bool b = bool ();
typedef void * voidptr;
void * p = voidptr ();
cout << "int " << i << ", bool " << b ", pointer " << p << endl;
```

It turns out that the introduction of std::vector further simplified our implementation of class MultiNode:

```
class MultiNode: public Node
{
public:
    MultiNode (Node * pNode)
    {
        AddChild (pNode, true);
    }
    ~MultiNode ();
    void AddChild (Node * pNode, bool isPositive)
    {
        _aChild.push_back (pNode);
        _aIsPositive.push_back (isPositive);
    }
protected:
    std::vector<Node*> _aChild;
    std::vector<bool>  _aIsPositive;
};
```

I have eliminated the member variable _iCur as it's no longer needed. We can quickly find out how many children there are by calling the vector's size () method. However, if all we need is to iterate over the elements of a vector (like we do in the destructor of MultiNode) there is a better, more "standard," way: We can use an iterator.

If you think of a vector as an array, an iterator is like a pointer. Remember when I told you that it's better to use an index rather than a pointer to access elements of an array? With vectors it's sort of the opposite, at least as far as iteration goes.

You can get an iterator pointing to the beginning of a vector by calling te vector's begin () method. Similarly, you can get the iterator pointing *one beyond the last element* of a vector by calling its end () method. You can move the iterator to point to the next slot of the vector by applying the increment, ++, operator. You can test whether two iterators are equal (point to the same slot in the vector) using the (un-) equality operators. Finally, you can get the value of an element of the vector by dereferencing the iterator using the star (asterisk) operator.

This is what the new destructor of MultiNode looks like:

```
MultiNode::~MultiNode ()
{
    typedef std::vector<Node *>::iterator NodeIter;
    for (NodeIter it = _aChild.begin (); it != _aChild.end (); ++it)
        delete *it;
}
```

The type "iterator" is internal to the template class vector. That makes for a very unwieldy type name, std::vector<Node *>::iterator. It is customary to declare local typedefs for such complex type names.

In case you're wondering, vector::iterator is in all likelihood implemented as a pointer in most versions of the Standard Library. And if, for some reason or another, it's not, you can still do with it virtually anything you can do with a pointer. What's even more important is that you get from it the same kind of performance as from a pointer. You can pass it around just like a poiner. You can return it from a function like a pointer. In fact, performance-wise, there is no difference between the code above and the following:

```
Node * aChild [size];
typedef Node ** NodeIter;
for (NodeIter it = aChild; it != aChild + size; ++it)
    delete *it;
```

I want to stress this point, because some programmers might hesitate to use some of the features of the Standard Library for fear of performance degradation. No need to worry--the creators of the Standard Library made it their highest priority not to jeopardize the performance. That's why, for instance, they selected the pointer-like approach to iteration, rather than the, somehow safer, sequencer approach. Pointer-like iteration requires two separate pieces of data--the current position and the end position. A sequencer has access to both. With a sequencer, there is no danger that you might, by mistake, compare the current position in one vector against the end position that belongs to some other vector. In fact, if you are worried about this kind of errors, you might create a sequencer class out of two iterators.

```
class NodeSeq
{
public:
    NodeSeq (std::vector<Node *> & vec)
        : _cur (vec.begin ()), _end (vec.end ())
    {}
    bool AtEnd () const { return _cur == _end; }
    void Advance () { ++_cur; }
    Node * Get () { return *_cur; }
private:
    std::vector<Node *>::iterator _cur;
    std::vector<Node *>::iterator _end;
};
```

The only drawback of the sequencer approach is that it's slightly more expensive to pass a sequencer by value than it is to pass an iterator--as you can see a sequencer is usually twice as large than the corresponding iterator. And, as you'll see later, iterators are passed around a lot, once you start using standard algorithms.

Next on our list of dynamic array substitutions is the symbol table with its _aOffStr array. We'll have to re-code the ForceAdd method; incidentally getting rid of the _curId member.

```
int SymbolTable::ForceAdd (char const * str)
{
    int offset = _bufStr.AddString (str);
    int id = _aOffStr.size ();
    _aOffStr.push_back (offset);
    _htab.Add (str, id);
    return id;
}
```

As you might remember, we use the current position in the array of offsets as our string id. Again, the size () method of the vector helps us locate the current end of the vector.

By the way, did you notice that the compiler didn't complain about our initializing the vector in the constructor of SymbolTable? Unlike DynArray, std::vector doesn't take a default element value. However, it has a one-argument constructor that accepts the initial size for the vector.

Finally, let's get rid of the two DynArrays in Store. Here the situation is a bit more tricky. One of the arrays stores enumerated values, for which the compiler doesn't know the default. Instead of coming up with a clever trick, why don't we simply replace this array with a vector of bool. All we ever ask this array is whether a given element has been initialized or not. That's a yes/no question. So here are the new data members of Store:

```
std::vector<double>  _aCell;
std::vector<bool>    _aIsInit;
```

Next, we have to substitute all calls to DynArray::InRange with a test against vector::size (). For instance,

```
bool IsInit (int id) const
{
    assert (id >= 0);
    return id < _aIsInit.size () && _aIsInit [id];
}
```

The AddValue method has to be able to insert an item beyond the current end of the vector. The way to do it is to first resize them. The resize method increases the size of the vector and fills the new slots with default values.

```
void AddValue (int id, double val)
{
    assert (id >= 0);
    _aCell.resize (id + 1);
    _aIsInit.resize (id + 1);
    _aCell [id] = val;
    _aIsInit [id] = true;
}
```

To summarize: There is no reason for any program to have unreasonable limitations due to the use of fixed-size arrays. Standard Library vector should be used whenever a resizable array is needed.

Code Review 5: Resource Management

There's one important flaw in our program. It assumes that memory allocation never fails. This assumption is blatantly false. Granted, in a virtual-memory system allocation limits are rather high. In fact, the computer is not even limited by the size of its physical memory. The only way memory allocation may fail is if you run out of space in the swap file on your disk. Now, that's not something unheard of in the world of huge applications and multimedia files. An industrial-strength program should protect itself from it. The question is, how? What should a program do when the call to new fails? First of all, if we don't do anything about it (as indeed we didn't), the system will terminate us with extreme prejudice. That's totally unacceptable!

The next, slightly more acceptable thing would be to print a good-bye message and exit. For a program such as our calculator that might actually be the correct thing. It's not like the user runs the risk of losing hours of work when the calculator unexpectedly exits.

There are a few different ways to globally intercept a memory failure. One is to catch the bad_alloc exception--more about it in a moment--another, to create your own "new handler" and register it with the runtime at the beginning of your program. The function set_new_handler is defined in the header **<new>**. It accepts a pointer to a function that's to be called in case new fails. Here's an example of a user-defined new handler:

```
void NewHandler ()
{
    cerr << "Out of memory\n";
    exit (1);
    // not reached
    return 0;
}
```

It displays a message end exits by calling a special function, exit from **<cstdlib>**.

This is how you register your handler--preferably at the beginning of main.

```
set_new_handler (&NewHandler);
```

> If you have a non-standard-compliant compiler you might have to modify this code. For instance, in Microsoft VC++ 6.0 there is a function called _set_new_handler that serves a similar purpose. Its declaration is in **<new.h>**. You'll also have to change the signature of the function you pass to it--to int NewHandler (size_t size).

Although in many cases such a solution is perfectly acceptable, most commercial programs require more sophisticated techniques (and maybe, at some point in the future, will actually apply them!). Just try to imagine a word processor (or a compiler) suddenly exiting without giving you a chance to save your work. Sounds familiar?!

What happens when memory allocation fails? In standard C++, operator new generates an exception. So let's talk about exceptions.

Exceptions

in C++ we deal with exceptional failures using exceptions. Essentially it is up to the programmer to decide what is exceptional and what isn't. A memory allocation failure is considered exceptional--so is a failed disk read or write. On the other hand "file not found" or "end of file" are considered normal events. When an exceptional condition occurs, the program can throw an exception. A throw bypasses the normal flow of control and lands us in the catch part of the nearest enclosing try/catch block. So throw really behaves like a non-local goto (non-local meaning that the control can jump across function calls).

We can best understand the use of exceptions using the calculator as an example. The only exceptional condition that we can anticipate there is the memory allocation failure. As I mentioned before, when new is not able to allocate the requested memory, it throws an exception of the type bad_alloc.

Again, this is true in the ideal world. At the time of this writing most compilers still haven't caught up with the standard. But don't despair, you can work around this problem by providing your own new-failure handler that throws an exception.

```
class bad_alloc {};

int NewHandler (size_t size)
{
    throw bad_alloc ();
    return 0;
}
```

Then call _set_new_handler (&NewHandler); first thing in main. (Notice: the handler creates an object of the type bad_alloc, by calling its default constructor, which does nothing. The type bad_alloc itself is a dummy--we define it only to be able to distinguish between different types of exceptions.)

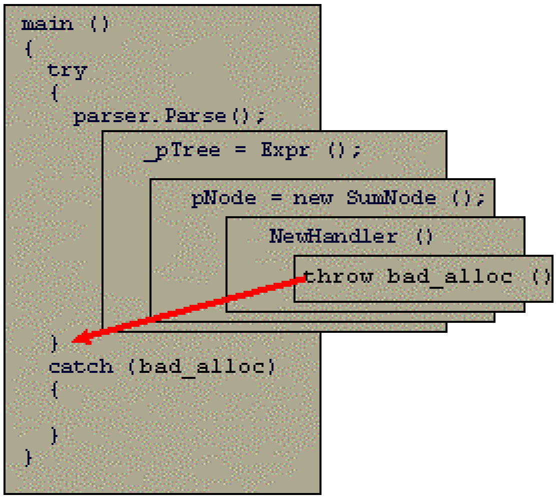
If we don't catch this exception, the program will terminate in a rather unpleasant way. So we better catch it. The program is prepared to catch an exception as soon as the flow of control enters a try/catch block. So let's open our try block in main before we construct the symbol table (whose constructor does the first memory allocations). At the end of the try block one must start a catch block that does the actual catching of the exception.

```
int main ()
{
    // only if your new doesn't throw by default
    _set_new_handler (&NewHandler);

    try
    {
        char buf [maxBuf+1];
        Status status;
        SymbolTable symTab;
        Function::Table funTab (symTab);
        Store store (symTab);
        ...
    }
    catch (bad_alloc)
    {
        cerr << "Out of memory!\n";
    }
    catch (...)
    {
        cerr << "Internal error\n";
    }
}
```

Here we have two catch blocks, one after another. The first will catch only one specific type of exception--the bad_alloc one--the second , with the ellipsis, will catch all the rest. The "internal error" may be a general protection fault or division by zero in our program. Not that we plan on allowing such bugs! But it's always better to display a message end exit than to let the system kill us and display its own message.

Always enclose the main body of your program within a try/catch block.

Figure 1 illustrates the flow of control after an exception is thrown during the allocation of the SumNode inside Parser::Expr that was called by Parser::Parse called from within the try block inside main.



```
main ()
{
  try
  {
    parser.Parse();
          _pTree = Expr ();
              pNode = new SumNode ();
                    NewHandler ()
                         throw bad_alloc ()
  }
  catch (bad_alloc)
  {
  }
}
```

Figure 1. Throw and catch across multiple function calls.

The use of exceptions solves the problem of error propagation (see sidebar) by bypassing the normal flow of control.

Error propagation.

In the old times, memory allocation failure used to be reported by new returning a null pointer. A programmer who wanted to protect his program from catastrophic failures had to test the result of every allocation. Not only was it tedious and error-prone, but it wasn't immediately clear what to do when such a failure occured. All a low level routine could do is to stop executing and return an error code to the calling routine. That, of course, put the burden of checking for error returns on the caller. In most cases the caller had little choice but to pass the same error code to its caller, and so on. Eventually, at some higher level--maybe even in main--somebody had the right context to do something about the problem, e.g., report it to the user, try to salvage his work, and maybe exit. Not only was such methodology error prone (forgetting to check for error returns), but it contributed dangerously to the obfuscation of the main flow of control. By the way, this problem applies to all kinds of errors, not only memory allocation failures.

Exceptions and Debugging

Since exceptions should not happen during normal program execution, it would make sense for your debugger to stop whenever an exception is thrown. This is particularly important when you have a catch-all clause in your program. It will not only catch those exceptions that are explicitly thrown--it will catch hardware exceptions caused by accessing a null pointer or division by zero. Those definitely require debugging.

Unfortunately, not all debuggers stop automatically on exceptions. For instance, Microsoft Visual C++ defaults to "stop only if not caught". You should change it. The appropriate dialog is available from the *Debug* menu (which is displayed only during debugging). Select the *Exceptions* item and change the action on all exceptions to "stop always."

Stack Unwinding

Figuring out how to propagate an error is the easy part. The real problem is the cleanup. You keep allocating memory in your program and, hopefully, you remember to deallocate it when it's no longer needed. If this weren't hard enough within the normal flow of control, now you have to remember to deallocate this memory when propagating an error.

Many a scheme was invented to deal with the cleanup during traditional error propagation. Since in most cases memory resources were freed at the end of procedures, one method was to use gotos to jump there whenever an error was detected. A typical procedure might look something like this:

```
ERRCODE foo ()
{
    ERRCODE status = errNoError;
    Type1 * p1 = 0;
    Type2 * p2 = 0;
    p1 = new Type1;
    if (p1 == 0)
    if (status != errNoError)
        goto Cleanup;
    // more of the same
Cleanup:
    delete p2;
    delete p1;
    return status;
}
```

All this code just to allocate and destroy two objects and make two function calls (the code not dealing with error detection and propagation has been highlighted). If you think I'm exaggerating, don't! I've seen files after files of commercial code written in this style. Incidentally, adding a return statement in the middle of such a procedure will totally invalidate the whole cleanup logic.

Wait, it gets worse before it gets better! When you use exceptions to propagate errors, you don't get a chance to clean-up your allocations unless you put try/catch pairs in every procedure. Or so it seems. The straightforward translation of the sidebar example would look something like this:

```
void foo ()
{
    Type1 * p1 = 0;
    Type2 * p2 = 0;
    try
    {
        p1 = new Type1;
        call1 (p1);
        p2 = new Type2;
        status = call2 (p2);
        // more of the same
    }
    catch (...)
    {
        delete p2;
        delete p1;
        throw;
    }
    delete p2;
    delete p1;
}
```

This code looks cleaner but it's still very error-prone. Pointers are zeroed in the beginning of the procedure, assigned to in the middle and deallocated at the end (notice the ugly, but necessary, repetition of the two delete statements). Plenty of room for mistakes. By the way, the throw with no argument re-throws the same exceptions that was caught in the enclosing catch block.

At this point you might feel desperate enough to start thinking of switching to Java or some other language that offers automatic garbage collection. Before you do, read on.

There is a mechanism in C++ that just begs to be used for cleanup--the pairing of constructors and destructors of automatic objects. Once an object is constructed in some local scope, it is always automatically destroyed when the flow of control exits that scope. The important thing is that it really doesn't matter how the scope is exited: the flow of control may naturally reach the closing bracket, or it may jump out of the scope through a goto, a break or a return statement. Automatic objects are **always** destroyed. And yes--this is very important--when the scope is exited through an exception, automatic objects are cleaned up too! Look again at Figure 1--all the stack based objects in all the nested scopes between the outer try block and the inner throw block are automatically destroyed--their destructors are executed in the inverse order of construction.

I'll show you in a moment how to harness this powerful mechanism, called stack unwinding, to do all the cleanup for us. But first let's consider a few tricky situations.

For instance, what happens when an exception is thrown during construction? Imagine an object with two embeddings. In its constructor the two embedded objects are constructed, either implicitly or explicitly (through the preamble). Now suppose that the first embedding has been fully constructed, but the constructor of the second embedding throws an exception. What do you think should happen? Shouldn't the first embedding be destroyed in the process of unwinding? Right on! That's exactly what happens--the destructors of all fully constructed sub-objects are executed during unwinding.

What if the object is dynamically allocated using new and its constructor throws an exception? Shouldn't the memory allocated for this object be freed? Right again! That's exactly what happens--the memory allocated by new is automatically freed if the constructor throws an exception.

What if the heap object has sub-objects and the constructor of the nth embedding throws an exception? Shouldn't the n-1 sub-objects be destroyed and the memory freed? You guessed it! That's exactly what happens.

Remember the rules of object construction? First the base class is constructed, then all the embedded objects are constructed, finally the body of the constructor is executed. If at any stage of the construction an exception is thrown, the whole process is reversed and all the completely constructed parts of the object are destroyed (in the reverse order of construction).

What if an exception is thrown during the construction of the nth element of an array. Shouldn't all the n-1 fully constructed elements be destroyed? Certainly! That's exactly what happens. It all just works!

These rules apply no matter whether the object (array) is allocated on the stack or on the heap. The only difference is that, once a heap object is *fully constructed*, it (and its parts) will not take part in the process of stack unwinding.

There is only one situation when stack unwinding might not do the right thing. It's when an exception is thrown from within a destructor. So don't even think of allocating some scratch memory, saving a file to disk, or doing any of the many error-prone actions in a destructor. A distructor is supposed to quietly clean up.

Never perform any action that could throw an exception inside a destructor.

Now take our symbol table.

```
class SymbolTable
{
...
private:
    HTable            _htab;
    std::vector<int>  _aOffStr;
    StringBuffer      _bufStr;
};
```

It has three embedded objects, all with nontrivial constructors. The hash table allocates an array of lists, the vector of offsets might allocate some initial storage and the string buffer allocates a buffer for strings. Any one of these allocations may fail and throw an exception.

If the constructor of the hash table fails, there is no cleanup (unless the whole SymbolTable object was allocated using new--in that case the memory for it is automatically freed). If the hash table has been successfully constructed but the vector fails, the destructor of the hash table is executed during unwinding. Finally, if the exception occurs in the constructor of the string buffer, the destructors of the vector and the hash table are called (in that order).

The constructor of the hash table itself could have some potentially interesting modes of failure.

```
HTable::HTable (int size)
    : _size (size)
{
    _aList = new List<int> [size];
}
```

Assume for a moment that the constructor of List<int> is nontrivial (for example, suppose that it allocates memory). The array of 65 (cSymInit + 1) lists is constructed in our calculator. Imagine that the constructor of the 8th list throws an exception. The destructors of the first seven entries of the array will automatically be called, after which the memory for the whole array will be freed. If, on the other hand, all 65 constructors succeed, the construction of the hash table will be successful and the only way of getting rid of the array will be through the destructor of the hash table.

But since the hash table is an embedded object inside the symbol table, and the symbol table is an automatic object inside the try block in main, its destruction is guaranteed. Lucky accident? Let's have another look at main.

```
int main ()
{
    _set_new_handler (&NewHandler);
    try
    {
        ...
        SymbolTable symTab;
        Function::Table funTab (symTab);
        Store store (symTab);
        ...
        {
            Scanner scanner (buf);
            Parser  parser (scanner, store, funTab, symTab);
            ...
        }
    }
    catch (...)
    {
        ...
    }
}
```

It looks like all the resources associated with the symbol table, the function table, the store, the scanner and the parser are by design guaranteed to be released. We'll see in a moment that this is indeed the case.

Resources

So far we've been dealing with only one kind of resource--memory. Memory is acquired from the runtime system by calling new, and released by calling delete. There are many other kinds of resources: file handles, semaphores, reference counts, etc. They all have one thing in common--they have to be acquired and released. Hence the definition:

A resource is something that can be acquired and released.

The acquiring part is easy--it's the releasing that's the problem. An unreleased resource, or resource leak, can be anywhere from annoying (the program uses more memory than is necessary) to fatal (the program deadlocks because of an unreleased semaphore).

We can guarantee the matching of the acquisition and the release of resources by following the simple ***First Rule of Acquisition***:

Acquire resources in constructors, release resources in matching destructors.

In particular, the acquisition of memory through new should be only done in a constructor, with the matching delete in the destructor. You'll find such matching pairs in the constructors and destructors of FunctionTable, HTable (and std::vector<T>, if you look it up in **<vector>**). (The other occurrences of new in our program will be explained later.)

However, if you look back at the symbol table right before we separated out the string buffer, you'll see something rather disturbing:

```
SymbolTable::SymbolTable (int size)
    : _size (size), _curId (0), _curStrOff (0), _htab (size + 1)
{
    _offStr = new int [size];
    _bufSize = size * 10;
    _strBuf = new char [_bufSize];
}

SymbolTable::~SymbolTable ()
{
    delete []_offStr;
    delete []_strBuf;
}
```

The ***First Rule of Acquisition*** is being followed all right, but the code is not exception safe. Consider what would happen if the second allocation (of _strBuf) failed. The first allocation of _offStr would have never been freed.

To prevent such situations, when programming with exceptions, one should keep in mind the following corollary to the First Rule of Acquisition:

Allocation of resources should either be the last thing in the constructor or be followed by exception safe code.

If more than one resource needs to be allocated in a constructor, one should create appropriate sub-objects to hold these resources. They will be automatically released even if the constructor fails before completion.

As you can see, there are many different arguments that lead to the same conclusion--it makes sense to separate sub-objects! We can even enhance our rule of thumb that suggested the separation of sub-object when a class had too many data members: Create sub-objects when more than one data member points to a resource. In our symbol table it was the introduction of the following two sub-objects that did the job.

```
    std::vector<int>  _aOffStr;
    StringBuffer      _bufStr;
```

Ownership of Resources

An object or a block of code owns a resource if it is responsible for its release. The concept of *ownership of resources* is central to the resource management paradigm. The owns-a relationship between objects parallels the more traditional is-a (inheritance) and has-a (embedding) relationships. There are several forms of ownership.

A block of code owns all the automatic objects defined in its scope. The job of releasing these resources (calling the destructors) is fully automated.

Another form of ownership is by embedding. An object owns all the objects embedded in it. The job of releasing these resources is automated as well (the destructor of the big object calls the destructors for the embedded objects).

In both cases the lifetime of the resource is limited to the lifetime of the embedding entity: the activation of a given scope or the lifetime of an embedding object. Because of that, the compiler can fully automate destructor calls.

The same cannot be said for dynamically allocated objects that are accessed through pointers. During their lifetimes, pointers can point to a sequence of resources. Also, several pointers can point to the same resource. The release of such resources is not automated. If the object is being passed from pointer to pointer and the ownership relations cannot easily be traced, the programmer is usually in for a trouble. The symptoms may be various: uninitialized pointers, memory leaks, double deletions, etc.

Imposing the ***First Rule of Acquisition*** clarifies the ownership relationships and guarantees the absence of most errors of the types we have just listed. A block of code can only own its automatic objects--no naked pointers should be allowed there. An object can own other objects by embedding as well as through pointers. In the latter case though, the acquisition of the resources (initialization of pointers) has to take place in the object's constructor, and their release in the destructor. Constructors are the only places where a naked pointer may appear and then only for a very short time.

Notice that if all resources have owners, they are guaranteed to be released. The objects that own resources are either defined in the global scope, or in a local scope or are owned by other objects--through embeddings or through dynamic allocation in constructors. Global objects are destroyed after exit from main, local objects are destroyed upon exit from their scope, embeddings are either destroyed when the enclosing objects are destroyed, or when the enclosing constructor is aborted through an exception. Since all the owners are eventually destroyed, or the resources are eventually freed.

Access to Resources

A block of code or an object may operate on a resource without owning it, that is, without being responsible for its release. When granting or passing access, as opposed to ownership, one should try to use references whenever possible. We have already learned that the has-access-to relationship is best expressed using references. A reference cannot express ownership because an object cannot be deleted through a reference (at least not without some trickery).

There are rare cases when using a reference is awkward or impossible. In such cases one can use a **weak pointer**--a pointer that doesn't express ownership--only access. In C++ there is no syntactic difference between strong and week pointers, so this distinction should be made by the appropriate commenting of the code.

We will graphically represent the has-access-to relationship (either through a reference or through a weak pointer) by a dotted arrow, as in Figure 2.

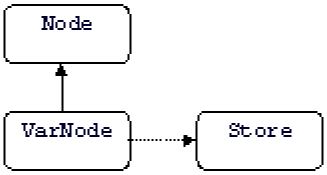


Figure 2. Graphical representation of the has-access-to relationship between VarNode and Store.

If access to a resource is reference-counted, the ownership rules apply to the reference count itself. Reference count should be acquired (incremented) in a constructor and released (decremented) in a destructor. The object itself may be passed around using a reference or a pointer, depending on whether there is one distinguished owner, or the ownership is distributed (the last client to release the reference count, destroys the object).

Smart Pointers

Our calculator was implemented with no regard for resource management. There are naked pointers all over the place, especially in the parser. Consider this fragment.

```
do
{
    _scanner.Accept();
    Node * pRight = Term ();
    pMultiNode->AddChild (pRight, (token == tPlus));
    token = _scanner.Token();
} while (token == tPlus || token == tMinus);
```

The call to Term returns a node pointer that is temporarily stored in pRight. Then the MultiNode's method AddChild is called, and we know very well that it might try to resize its array of children. If the reallocation fails and an exception is thrown, the tree pointed to by pRight will never be deallocated. We have a memory leak.

Before I show you the systematic solution to this problem, let's try the obvious thing. Since our problem stems from the presence of a naked pointer, let's create a special purpose class to encapsulate it. This class should acquire the node in its constructor and release it in the destructor. In addition to that, we would like objects of this class to behave like regular pointers. Here's how we can do it.

```
class NodePtr
{
public:
    NodePtr (Node * pNode) : _p (pNode) {}
    ~NodePtr () { delete _p; }
    Node * operator->() const { return _p; }
    Node & operator * () const { return _p; }
private:
    Node * _p;
};
```

Such objects are called safe or smart pointers. The pointer-like behavior is implemented by overloading the pointer-access and pointer-dereference operators. This clever device makes an object behave like a pointer. In particular, one can call all the public methods (and access all public data members, if there were any) of Node by "dereferencing" an object of the type NodePtr.

```
{
    Node * pNode = Expr ();
    NodePtr pSmartNode (pNode);
    double x = pSmartNode->Calc (); // pointer-like behavior
    ...
    // Automatic destruction of pSmartNode.
    // pNode is deleted by its destructor.
}
```

Of course, a smart pointer by itself will not solve our problems in the parser. After all we don't want the nodes created by calling Term or Factor to be automatically destroyed upon normal exit from the scope. We want to be able to build them into the parse tree whose lifetime extends well beyond the local scope of these methods. To do that we will have to relax our First Rule of Acquisition .

Ownership Transfer: First Attempt

When the lifetime of a given resource can be mapped into the lifetime of some scope, we encapsulate this resource in a smart pointer and we're done. When this can't be done, we have to pass the resource between scopes. There are two possible directions for such transfer: up and down. A resource may be passed up from a procedure to the caller (returned), or it can be passed down from a caller to the procedure (as an argument). We assume that before being passed, the resource is owned by some type of owner object (e.g., a smart pointer).

Passing a resource down to a procedure is relatively easy. We can simply pass a reference to the owner object (a smart pointer, in our case) and let the procedure acquire the ownership from it. We'll add a special method, Release, to our smart pointer to release the ownership of the resource.

```
Node * NodePtr::Release ()
{
    Node * tmp = _p;
    _p = 0;
    return tmp;
}
```

The important thing about Release is that it zeroes the internal pointer, so that the destructor of NodePtr will not delete the object (delete always checks for a null pointer and returns immediately). After the call to Release the smart pointer no longer owns the resource. So who owns it? Whoever called it better provide a new owner!

This is how we can apply this method in our program. Here, the node resource is passed from the Parser's method Expr down to the MultiNode's method AddChild.

```
do
{
    _scanner.Accept();
    NodePtr pRight (Term ());
    pMultiNode->AddChild (pRight, (token == tPlus));
    token = _scanner.Token();
} while (token == tPlus || token == tMinus);
```

AddChild acquires the ownership of the node by calling the Release method and passes it immediately to the vector _aChild (if you see a problem here, read on, we'll tackle it later).

```
void MultiNode::AddChild (NodePtr & pNode, bool isPositive)
{
    _aChild.push_back (pNode.Release ());
    _aIsPositive.push_back (isPositive);
}
```

Passing a resource up is a little trickier. Technically, there's no problem. We just have to call Release to acquire the resource from the owner and then return it back. For instance, here's how we can return a node from Parser::Expr.

```
Node * Parser::Expr ()
{
    // Parse a term
    NodePtr pNode (Term ());
    ...
    return pNode.Release ();
}
```

What makes it tricky is that now the caller of Expr has a naked pointer. Of course, if the caller is smart, he or she will immediately find a new owner for this pointer--presumably a smart pointer--just like we did a moment ago with the result of Term. But it's one thing to expect everybody to take special care of the naked pointers returned by new and Release, and quite a different story to expect the same level of vigilance with every procedure that happens to return a pointer. Especially that it's not immediately obvious which ones are returning strong pointers that are supposed to be deleted, and which return weak pointers that must not be deleted.

Of course, you may chose to study the code of every procedure you call and find out what's expected from you. You might hope that a procedure that transfer ownership will be appropriately commented in its header file. Or you might rely on some special naming convention--for instance start the names of all resource-returning procedures with the prefix "Query" (been there!).

Fortunately, you don't have to do any of these horrible things. There is a better way. Read on!

To summarize, even though there are some big holes in our methodology, we have accomplished no mean feat. We have encapsulated all the resources following the First Rule of Acquisition. This will guarantee automatic cleanup in the face of exceptions. We have a crude method of transfering resources up and down between owners.

Ownership Transfer: Second Attempt

So far our attempt at resource transfer through procedure boundaries have been to release the resource from its owner, pass it in its "naked" form and then immediately encapsulate it again. The obvious danger is that, although the passing happens within a few nanosecond in a running program, the code that accepts the resource may be written months or even years after the code that releases it. The two sides of the procedure barrier don't necessarily talk to each other.

But who says that we have to "undress" the resource for the duration of the transfer? Can't we pass it together with its encapsulator? The short answer is a resounding yes! The longer answer is necessary in order to explain why it wouldn't work without some unorthodox thinking.

First of all, if we were to pass a smart pointer "as is" from a procedure to the caller, we'd end up with a dangling pointer.

```
NodePtr Parser::Expr ()
{
    // Parse a term
    NodePtr pNode = Term (); // <- assignment
    ...
    return pNode; // <- by value
}

NodePtr Parser::Term ()
{
    NodePtr pNode = Factor (); // <- assignment
    ...
    return pNode; // <- by value
}
```

Here's why: Remember how objects are returned from procedures? First, a copy constructor is called to construct a temporary, then the assignment operator is called to copy this temporary into the caller's variable. If an object doesn't define its own copy constructor (or assignment operator), one will be provided for it by the compiler. The default copy constructor/assignment makes a shallow copy of the object. It means that default copying of a smart pointer doesn't copy the object it points to. That's fine, we don't want a copy of our resource. We end up, however, with two smart pointers pointing to the same resource. That's bad in itself. To make things even worse, one of them is going out of scope--the smart pointer defined inside the procedure. It will delete the object it points to, not realizing that its clone brother has a pointer to it, too. We've returned a smart pointer, all right, but it points to a deleted object.

What we need is to enforce the rule that there may only be one owner of a given resource at a time. If somebody tries to clone a smart pointer, we have to steal the resource from the original owner. To that end we need to define our own copy constructor and assignment operator.

```
NodePtr::NodePtr (NodePtr & pSource)
    : _p (pSource.Release ())
{}

NodePtr & NodePtr::operator= (NodePtr & pSource)
{
    if (_p != pSource._p)
    {
        delete _p;
        _p = pSource.Release ();
    }
}
```

Notice that these are not your usual copy constructor and assignment operator. For one, they don't take const references to their source objects. They can't, because they modify them. And modify they do, drastically, by zeroing out their contents. We can't really call it "copy semantics," we call it "transfer semantics." We provide a class with transfer semantics when we give it a "transfer constructor" and overload its assigment operator such that each of these operations takes away the ownership of a resource from its argument.

It's time to generalize our accomplishments. We need a template class that is a smart pointer with transfer semantics. We can actually find it in the standard library, in the header **<memory>**, under the name auto_ptr. Instead of NodePtr, we'll start using std::auto_ptr<Node> whenever node ownership is concerned.

> The standards committee strugled with the definition of auto_ptr almost till the last moment. Don't be surprised then if your compiler's library contains an older version of this template. For instance, you might discover that your auto_ptr contains an ownership flag besides the pointer. Such implementation is now obsolete.

Let's begin with UMinusNode, which has to accept the ownership of its child node in the constructor. The child node is passed, of course, in an auto_ptr. We could retrieve it from the auto_ptr by calling its release method. We could then store the pointer and delete it in the destructor of UminusNode--all in the spirit of Resource Management. But there's a better way! We can embed an auto_ptr inside UminusNode and let *it* deal with ownership chores.

```
class UMinusNode: public Node
{
public:
    UMinusNode (auto_ptr<Node> & pNode)
```

```
        : _pNode (pNode) // <- "transfer" constructor
    {}
    double Calc () const;
private:
    auto_ptr<Node> _pNode;
};
```

Look what happened--there no longer is a need for UMinusNode to have a destructor, because embeddings are destroyed automatically. We don't have to call release on the argument to the constructor, because we can pass it directly to the "transfer" constructor of our embedded auto_ptr. Internally, the implementation of UMinusNode::Calc remains unchanged, because the syntax for accessing a pointer and an auto_ptr is identical.

We can immediately do the same conversion with FunNode, BinNode and AssignNode. BinNode contains two auto_ptrs.

```
class BinNode: public Node
{
public:
    BinNode (auto_ptr<Node> & pLeft, auto_ptr<Node> & pRight)
        : _pLeft (pLeft), _pRight (pRight)
    {}
protected:
    auto_ptr<Node> _pLeft;
    auto_ptr<Node> _pRight;
};

class AssignNode : public BinNode
{
public:
    AssignNode (auto_ptr<Node> & pLeft, auto_ptr<Node> & pRight)
        : BinNode (pLeft, pRight)
    {}

    double Calc () const;
};
```

As before, we were able to get rid of explicit destructors.

We can continue our destructor derby with the Parser itself. It can own its parsing tree through an auto_ptr and gone is its explicit destructor.

```
    auto_ptr<Node>    _pTree;
```

We can't get rid of the destructor of MultiNode yet. For the time being, let's just rewrite it so that it accepts auto_ptrs as arguments to its methods. We'll return to the issue "owning containers" later.

```
class MultiNode: public Node
{
public:
    MultiNode (auto_ptr<Node> & pNode)
    {
        AddChild (pNode, true);
    }
    ~MultiNode ();
    void AddChild (auto_ptr<Node> & pNode, bool isPositive)
    {
        _aChild.push_back (pNode.release ());
        _aIsPositive.push_back (isPositive);
    }
protected:

    std::vector<bool>  _aIsPositive;
};
```

And now let's look at the Parser::Expr method after it's been generously sprinkled with auto_ptrs.

```
auto_ptr<Node> Parser::Expr ()
{
    // Parse a term
    auto_ptr<Node> pNode = Term ();
    if (pNode.get () == 0)
        return pNode;

    EToken token = _scanner.Token ();
    if (token == tPlus || token == tMinus)
    {
        // Expr := Term { ('+' | '-') Term }
        auto_ptr<MultiNode> pMultiNode (new SumNode (pNode));
        do
        {
            _scanner.Accept ();
            auto_ptr<Node> pRight = Term ();
            pMultiNode->AddChild (pRight, (token == tPlus));
            token = _scanner.Token ();
        } while (token == tPlus || token == tMinus);
        // with member template support
        pNode = pMultiNode; // <- Up-casting!
        // pNode = up_cast<Node> (pMultiNode);
    }
    else if (token == tAssign)
    {
        _scanner.Accept ();

        auto_ptr<Node> pRight = Expr ();
        // provided the Term is an lvalue
        if (pNode->IsLvalue ())
        {
            // Assignment node
            pNode = auto_ptr<Node> (new AssignNode (pNode, pRight));
        }
        else
        {
            _status = stError;
            pNode = Expr ();
        }
    }
    // otherwise Expr := Term
    return pNode;
}
```

There is one tricky point in this code that requires some explaining. We are dealing with two types of auto_ptrs--one encapsulating a Node and another encapsulating a MultiNode (only MultiNode has the AddChild method). At some point, when we are done adding children, we must convert a pointer to MultiNode to a pointer to Node. With pointers it was no trouble, because a MultiNode "is-a" Node (they are related through inheritance). But auto_ptr<Node> is a totally different class from auto_ptr<MultiNode>. So this simple line of code:

```
pNode = pMultiNode; // <- Up-casting!
```

hides some pretty interesting stuff. In fact, there is a special mechanism called ***member template*** that's used inside the auto_ptr to make possible this types of conversions (called up-casting, because they cast the pointer up the class hierarchy).

Look a the following code--a transfer constructor and an assignment operator override are added to the auto_ptr template.

```
template<class T>
class auto_ptr
{
public:
    ...
    // "transfer" from derived class
    template<class U>
    auto_ptr (auto_ptr<U> & pSrc)
    {
        _p = pSrc.release ();
    }

    // assignment of derived class
    template<class U>
    auto_ptr & operator= (auto_ptr<U> & pSrc)
    {
        if (this != &pSrc)
            _p = pSrc.release ();
        return *this;
    }
};
```

These new methods are templates themselves. They are parametrized by another type, U.

The new transfer constructor can be used to construct an auto_ptr of one type, parametrized by T, from an auto_ptr of another type, parametrized by U. So, in fact, this constructor is a template parametrized by two types, T and U.

The new overload of the assignment operator can accept an auto_ptr parametrized by an arbitrary type U. It's this override that's invoked in the statement:

```
pNode = pMultiNode; // <- Up-casting from MultiNode to Node
```

Here, pNode is of the type auto_ptr<Node> and pMultiNode is of the type auto_ptr<MultiNode>. The compiler will automatically try to instantiate the appropriate implementation of the method auto_ptr<Node>::operator= (auto_ptr<MultiNode> &). And here's where type safety kicks in. When the compiler generates the line from the template:

```
_p = pSrc.release ();
```

it has to assign a pointer to MultiNode returned from pSrc.release (), to a pointer to Node, _p. The conversion works because of the is-a relationhip between the two classes. Had we tried the opposite conversion:

```
pMultiNode = pNode; // <- Error! Down-casting
```

the compiler would generate an error, because it would fail in the conversion of Node * to MultiNode * during the instantiation of the appropriate template.

The bottom line is that the assignment of auto_ptrs will only work for those types for which the assignments of pointers works. Notice that the types don't have to be related by inheritance. You can, for instance, implicitly convert a pointer to Node to a void pointer. It follows than that you can convert auto_ptr<Node> to auto_ptr<void>.

If your compiler doesn't support member templates, don't despair! I have a workaround for you. Since you won't be able to do implicit auto_ptr conversion, you'll have to do it explicitly.

```
pNode = up_cast<Node> (pMultiNode);
```

I created the up_cast function template, parametrized by two types, To and From.

```
template<class To, class From>
auto_ptr<To> up_cast (auto_ptr<From> & from)
{
    return auto_ptr<To> (from.release ());
}
```

Notice that, when I invoke this template, I don't have to specify the second type, From. The compiler can figure it out by looking at the type of the argument (pMultiNode, in this case). As before, the type checking is done by the compiler when instantiating this template for the requested pair of types.

It's pretty straightforward to convert the rest of the Parser to use auto_ptrs.

To summarize: Memory resources can be safely kept and transferred from one point to another using auto_ptrs. You can easily transform and existing program to use Resource Management techniques. Just search your project for all calls to new and make sure the resulting pointers are safely tucked inside auto_ptrs.

Safe Containers

We are not done yet. There are still some parts of our program that are not 100% resource safe. Let's go back to one such place inside the class MultiNode.

```
class MultiNode: public Node
```

```
{
public:
    MultiNode (auto_ptr<Node> & pNode)
    {
        AddChild (pNode, true);
    }
    ~MultiNode ();
    void AddChild (auto_ptr<Node> & pNode, bool isPositive)
    {
        _aChild.push_back (pNode.release ());
        _aIsPositive.push_back (isPositive);
    }
protected:
    std::vector<Node*> _aChild;
    std::vector<bool>  _aIsPositive;
};
```

Look what happens inside AddChild. We call release on pNode and then immediately call push_back on a vector. We know that push_back might re-allocate the vector's internal array. A re-allocation involves memory allocation which, as we know, might fail. You see the problem? The pointer that we have just released from pNode will be caught naked in the middle of an allocation failure. This pointer will never get a chance to be properly deleted. We have a memory leak!

What is needed here is for the vector to be re-allocated before the pointer is released. We could force the potential reallocation by first calling

```
_aChild.reserve (_aChild.size () + 1);
```

But that's an awkward solution (besides, it requires a lot of vigilance on the part of the programmer). So let's rethink the problem.

The culprit here is the vector of pointers, which has no notion of ownership. Notice that the deallocation of nodes is done not in the vector's destructor but rather in the MultiNode's destructor. What we need is some type of a vector that really *owns* the objects stored in it through pointers.

Could we use a vector of auto_ptrs?

```
std::vector<auto_ptr<Node> > _aChild;
```

(Notice the space left between the two greater-than signs. If you remove this space, the compiler will confuse it with the right-shift operator. It's just another quirk of C++ syntax.) When we push_back an auto_ptr on such a vector, the re-allocation will definitely happen before the "transfer" assignment. Morover, we could get rid of the explicit MultiNode destructor, because the vector's destructor will destroy all its auto_ptrs which, in turn, will destroy all the nodes.

Although in principle this is a valid solution to our problem, in practice it's hardly acceptable. The trouble is that the vector's interface is totally unsuitable for storing objects with weird copy semantics. For instance, on a const vector, the array-access operator returns an object by value. This is fine, except that returning an auto_ptr by value involves a resource transfer. Similarly, if you try to iterate over such a vector, you might get a resource transfer every time you dereference an iterator. Unless you are very careful, you're bound to get some nasty surprizes from a vector of auto_ptrs.

At this point you might expect me to come up with some clever data structure from the standard library. Unfortunately, there are no ownership-aware containers in the standard library, so we'll have to build one ourselves. Fortunately--we have all the necessary tools. So let's start with the interface.

```
template <class T>
class auto_vector
{
public:
    explicit auto_vector (size_t capacity = 0);
    ~auto_vector ();
    size_t size () const;
    T const * operator [] (size_t i) const;
    T * operator [] (size_t i);
    void assign (size_t i, auto_ptr<T> & p);
    void assign_direct (size_t i, T * p);
    void push_back (auto_ptr<T> & p);
    auto_ptr<T> pop_back ();
};
```

Notice a rather defensive attitude in this design. I could have provided an array-access operator that would return an lvalue but I decided against it. Instead, if you want to set a value, you have to use one of the assign or assign_direct methods. My philosophy is that resource transfer shouldn't be taken lightly and, besides, it isn't needed all that often--an auto_vector is usually filled using the push_back method.

As far as implementation goes, we can simply use a dynamic array of auto_ptrs.

```
template <class T>
class auto_vector
{
    ~auto_vector () { delete []_arr; }
private:
    void grow (size_t reqCapacity);

    auto_ptr<T>    *_arr;
    size_t        _capacity;
    size_t        _end;
};
```

The grow method allocates a larger array of auto_ptr<T>, transfers all the items from the old array, swaps it in, and deletes the old array.

```
template <class T>
void auto_vector<T>::grow (size_t reqCapacity)
{
    size_t newCapacity = 2 * _capacity;
    if (reqCapacity > newCapacity)
        newCapacity = reqCapacity;
    // allocate new array
    auto_ptr<T> * arrNew = new auto_ptr<T> [newCapacity];
    // transfer all entries
    for (size_t i = 0; i < _capacity; ++i)
        arrNew [i] = _arr [i];
    _capacity = newCapacity;
    // free old memory
    delete []_arr;
    // substitute new array for old array
    _arr = arrNew;
}
```

The rest of the implementation of auto_vector is pretty straightforward, since all the complexity of resource management is built into auto_ptr. For instance, the assign method simply utilizes the overloaded assignment operator to both deallocate the old object and transfer the new object in one simple statement.

```
void assign (size_t i, auto_ptr<T> & p)
{
    assert (i < _end);
    _arr [i] = p;
}
```

The method assign_direct takes advantage of the reset method of auto_ptr:

```
void assign_direct (size_t i, T * p)
{
    assert (i < _end);
    _arr [i].reset (ptr);
}
```

The pop_back method returns auto_ptr by value, because it transfers the ownership away from auto_vector.

```
auto_ptr<T> pop_back ()
{
    assert (_end != 0);
    return _arr [--_end];
}
```

Indexed access to auto_vector is implemented in terms of the get method of auto_ptr, which returns a "weak" pointer.

```
T * operator [] (size_t i)
{
    return _arr [i].get ();
}
```

With the new auto_vector the implementation of MultiNode is not only 100% resource-safe, but also simpler. One more explicit destructor bites the dust!

```
class MultiNode: public Node
{
public:
    MultiNode (auto_ptr<Node> & pNode)
    {
        AddChild (pNode, true);
    }
    void AddChild (auto_ptr<Node> & pNode, bool isPositive)
    {
        _aChild.push_back (pNode);
        _aIsPositive.push_back (isPositive);
    }
protected:
    auto_vector<Node>  _aChild;
    std::vector<bool>  _aIsPositive;
};
```

How do we know when we're done? How do we know when our code is completely resource-safe? There is a simple set of tests, at least as far as memory resources are concerned, that will answer this question. We have to search our code for all occurrences of new and release and go through the following series of test:

- Is this a direct transfer to an auto_ptr?
- Or, are we inside a constructor of an object? Is the result of the call immediately stored within the object, with no exception prone code following it in the constructor?
- If so, is there a corresponding delete in the destructor of the object?
- Or, are we inside a method that immediately assigns the result of the call to a pointer owned by the object, making sure the previous value is deleted?

Notice that all these tests are local. We don't have to follow all possible execution paths--just the close vicinity of the calls, except for the occasional look at some destructors. It can't get any easier than that!

Iterators

Like any other standard container, our auto_vector needs iterators. I will not attempt to provide the full set of iterators--it's a tedious work best left to library implementors. All we'll ever need in our calculator is a forward-only constant iterator, so that's what I'll define.

An iterator is an abstraction of a pointer to an element of an array. It is also usually implemented as a pointer. It can be cheaply passed around and copied. Our auto_vector is implemented as an array of auto_ptr, so it's only natural that we should implement its iterator as a pointer to auto_ptr. It will be cheap to create, pass around and copy. Moreover, incrementing such an iterator is as easy as incrementing a pointer. The only thing that prevents us from just typedefing an auto_vector iterator to a pointer to auto_ptr is its dereferencing behavior. When you dereference a pointer to an auto_ptr, you get an auto_ptr with all its resource-transfer semantics. What we need is an iterator that produces a regular weak pointer upon dereferencing (and in the case of a constant iterator, a weak pointer to const). Operator overloading to the rescue!

```
template<class T>
class const_auto_iterator: public
    std::iterator<std::forward_iterator_tag, T const *>
{

    const_auto_iterator () : _pp (0) {}
    const_auto_iterator (auto_ptr<T> const * pp) : _pp (pp) {}
    bool operator != (const_auto_iterator<T> const & it) const
        { return it._pp != _pp; }
    const_auto_iterator operator++ (int) { return _pp++; }
    const_auto_iterator operator++ () { return ++_pp; }
    T const * operator * () { return _pp->get (); }
    T const * operator-> () { return _pp->get (); }
private:
    auto_ptr<T> const * _pp;
};
```

First of all, a well-bahaved iterator should inherit from the `std::iterator` template, so that it can be passed to various standard algorithms. This template is parametrized by an iterator tag (`forward_iterator_tag` in our case), which specifies the iterators's capabilities; and its value type (pointer to const `T`, in our case). (Strictly speaking, there is a third parameter, the difference type--the type that you get when you subtract one iterator from another, but this one correctly defaults to `ptrdiff_t`.)

We have two constructors--one default and one taking a pointer to a const `auto_ptr`. We have a not-equal operator. We have two increment operators. One of them overloads the post-increment and the other the pre-increment operator. To distinguish between the declarations of these two, you provide a dummy `int` argument to your declaration of the post-increment operator (yet another quirk of C++ syntax).

> In many cases the implementation of the pre-increment operator is simpler and more efficient then the corresponding implementation of the post-increment operator. It's easy to first increment something and then return it. Post-increment, on the other hand, could be tricky. In principle, you have to somehow remember the original state of the iterator, increment its internal variables and return an iterator that's created using the remembered state. Fortunately, when you're dealing with pointers, the compiler does all the work for you, so that the statement `return p++;` does the right things. But try implementing post-increment for a linked list iterator and you'll see what I mean. By the way, this is the reason why I'm biased towards using pre-increment in my code, unless post-increment is specifically called for.

Our iterator has two dereference operators, both returning a weak pointer to const. The second one is useful if you want to call a method implemented by class `T`.

Inside the class `auto_vector`, we provide a traditional `typedef` and two methods `begin ()` and `end ()` to support iteration. Notice again that the end iterator poits one beyond the last element of the array. Notice also how in both methods a pointer to `auto_ptr` is automatically converted to a `const_iterator`. That's because we haven't restricted the appropriate constructor of `const_auto_iterator` to be `explicit`.

```
typedef const_auto_iterator<T> const_iterator;
const_iterator begin () const { return _arr; }
const_iterator end () const { return _arr + _end; }
```

Finally, let's look at an example of how our iterator might be used in the `Calc` method of `SumNode`.

```
double SumNode::Calc () const
{
    double sum = 0.0;
    auto_vector<Node>::const_iterator childIt = _aChild.begin ();
    std::vector<bool>::const_iterator isPosIt = _aIsPositive.begin ();
    for (; childIt != _aChild.end (); ++childIt, ++isPosIt)
    {
        assert (isPosIt != _aIsPositive.end ());
        double val = childIt->Calc ();
        if (*isPosIt)
            sum += val;
        else
            sum -= val;
    }
    assert (isPosIt == _aIsPositive.end ());
    return sum;
}
```

I said "might," because it's not immediately obvious that this style of coding, using iterators, is more advantageous that the traditional array-index iteration; especially when two parallel arrays are involved. I had to use the comma sequencing operator to squeeze two increment operations into one slot in the for-loop header. (Expressions separated by commas are evaluated in sequence. The value of the sequence is equal to the value of its last expression.)

On the other hand, this code would be easier to convert if we were to re-implement MultiNode to use linked lists instead of vectors. That, however, seems rather unlikely.

Error Propagation

Now that our code is exception-safe, we should reconsider our error-handling policy. Look what we've been doing so far when we detected a syntax error. We set the parser's status to stError and returned a null pointer from whatever parsing method we were in. It so happened that all syntax errors were detected at the lowest level, inside Parser::Factor. However, both Parser::Term and Parser::Expr had to deal with the possibility of a null node coming from a lower-level parsing method. In fact, Parser::Factor itself had to deal with the possibility that the recursive call to Expr might return a null. Our code was sprinkled with error-propagation artifacts like this one:

```
if (pNode.get () == 0)
    return pNode;
```

Whenever there is a situation where an error has to be propagated straight through a set of nested calls, one should consider using exceptions. If we let Parser::Factor throw an exception whenever it detects a syntax error, we won't have to worry about detecting and propagating null pointers through other parsing methods. All we'll need is to catch this exception at the highest level--say in Parser::Parse.

```
class Syntax {};

Status Parser::Parse ()
{
    try
    {
        // Everything is an expression
        _pTree = Expr ();
        if (!_scanner.IsDone ())

    }
    catch (Syntax)
    {
        _status = stError;
    }
    return _status;
}
```

Notice that I defined a separate class of exceptions, Syntax, for propagating syntax errors. For now this class is empty, but its type lets me distinguish it from other types of exceptions. In particular, I don't want to catch bad_alloc exceptions in Parser::Parse, since I don't know what to do with them. They will be caught and dealt with in main.

Here's an example of code from Parser::Factor converted to use exceptions for syntax error reporting. Notice that we no longer test for null return from Expr (in fact we can assert that it's not null!).

```
if (_scanner.Token () == tLParen) // function call
{
    _scanner.Accept (); // accept '('
    pNode = Expr ();
    assert (pNode.get () != 0);
    if (_scanner.Token () == tRParen)
        _scanner.Accept (); // accept ')'
    else
        throw Syntax ();
    if (id != SymbolTable::idNotFound && id < _funTab.Size ())
    {
        pNode = auto_ptr<Node> (
            new FunNode (_funTab.GetFun (id), pNode));
    }
    else
    {
        cerr << "Unknown function \"";
        cerr << strSymbol << "\"" << endl;
        throw Syntax ();
    }
}
```

There is also some leftover code that dealt with symbol table overflows.

```
// Factor := Ident
if (id == SymbolTable::idNotFound)
{
    id = _symTab.ForceAdd (strSymbol);
    if (id == SymbolTable::idNotFound)
    {
        cerr << "Error: Too many variables\n";
        _status = stError;
    }
    if (id != SymbolTable::idNotFound)
        pNode = auto_ptr<Node> (new VarNode (id, _store));
}
```

Since we don't expect overflows, and memory allocation failures are dealt with by exceptions, we can simplify this code:

```
// Factor := Ident
if (id == SymbolTable::idNotFound)
{
    id = _symTab.ForceAdd (strSymbol);
}
pNode = auto_ptr<Node> (new VarNode (id, _store));
```

Conversion

By now you should be convinced that any program that doesn't follow the principles of Resource Management described in this chapter has little chance of ever getting anywhere near the state of being bug-free. So what should you do if you already have a project that's ridden with resource bugs? Do you have to convert the whole project all at once, just like we did in our little program? Fortunately, not! You can do the conversion one step at a time.

Start by finding all occurrences of new in the whole project. If this is too much, divide your project into smaller areas and convert each of them in turn.

- For these calls to new that occur in constructors, make sure there is a corresponding delete in the destructor. Consider embedding an auto_ptr in your class to hold the result of such new.
- Outside constructors, make sure the result of new is:
    - Immediately stored in a member variable that accepts the ownership--i.e., there is a matching delete in the destructor and the previous value is released prior to assignment, or
    - Immediately stored in an auto_ptr.
- If you had to create a local auto_ptr in the previous step, call its release method before passing it down or up the chain of ownership. This is a temporary solution that will allow you to work in small increments.
- Once you're done with new, search for all occurrences of release that you have introduced in previous steps. They mark those spots where resource transfer takes place.
    - If a resource is being passed down, change the signature of the called procedure to accept an auto_ptr (by reference or by value). If that procedure has to pass the ownership of the pointer still further, call release. You'll come back to this of release later.
    - If a resource is being passed up, return auto_ptr by value. Fix all the callers to accept the returned auto_ptr. If they have to pass the ownership of the pointer still further, call release.
    - If the ownership of a resource is being passed down to a container, consider converting that container to auto_vector.
- Whenever a pointer is converted to auto_ptr, the compiler will detect all the calls to delete that are no longer needed. Remove them.

This conversion procedure is recursive. At each step you might create some calls to release that will have to be fixed later. You are done converting when you can convince yourself that all the remaining calls to new and release are absolutely necessary and they pass the ownership directly into the member variables that know how to deal with it.

There are many other types of resources besides memory. They are usually obtained from the operating system and have to be returned back to it. They, too, should be encapsulated in smart objects. We'll see some examples later.

Conclusion

We started with a simple question: "What should a program do when memory allocation fails?" The answer turned out to be much more complex that anticipated. We had to go though exceptions, stack unwinding, error propagation, resource management, resource transfer, smart pointers, auto pointers, smart containers, auto iterators and a lot of really advanced features of C++. But once you've assimilated all the material presented in this chapter and used the techniques described in it, you will have graduated from the level of Weekend Programming to the new level of Industrial Strength Programming. Congratulations!

Making Use of the Standard Template Library

Programming in C++ is so much easier once you start using the Standard Library. If I were to start the whole calculator project from scratch, with the full use of the Standard Library, it would look a lot different. I didn't, though, for several reasons. First of all, the standard library is based on templates--especially the part called the Standard Template Library (STL) that contains all of the containers (most notably, vectors) and algorithms. So I had to explain first what templates were and what kind of problems they solved. Second of all, you wouldn't know how much work they could save you, unless you tried programming without them.

So what would the calculator look like if I had a free run of STL from the very beginning First of all, I wouldn't bother implementing the hash table. No, there is no ready-made implementation of a hash table in STL--although many compiler vendors will undoubtedly add one to their version of the Standard Library. There is, however, another useful data structure that, although not as efficient as the hash table, fulfills the same role. I'm talking about an associative array, known in STL as `std::map`.

An associative array is like an array, except that it can be indexed using objects of a type that is not necessarily integral.

You know how to define an array of strings. You index it with an integer and it gives you a string. An associative array can do the inverse--you may index it, for instance, with a string and it could give you an integer. Like this:

```
int id = assoc ["foo"];
```

This looks very much like what we need in our symbol table. Given the name of an identifier, give me its integer id.

You can easily implement an associative array using the STL's `map` template. A map is parameterized by two types--the type used to index it, called the **key**; and the **value** type. There is only one requirement for the key type--there must be a way to compare keys. Actually, it's not necessary to provide a method to compare keys for (in-) equality. What is important for the map is to be able to tell if one key is less than the other. By default, a map expects the key type to either be directly comparable or have the less-than operator overload.

The naive, and for most purposes incorrect, approach to implementing an associative array of strings would be to use `char const *` as the key type, like this:

```
std::map<char const *, int> dictionary;
```

The first problem is that there indeed *is* an operator less-than defined for this key type, but it's totally inappropriate. Consider this:

```
char const * str1 = "sin";
char const * str2 = "sin";
if (str1 < str2) // <- not what you'd expect!
    cout << str1 << " is less than " << str2 << endl;
```

The result of this comparison is meaningless, because we are comparing two pointers--we are numerically comparing two memory addresses. This has nothing to do with what we normally consider "string comparison." There is, however, a special function strcmp defined in **<cstring>** that compares strings represented as pointers to characters (there is also a case-insensitive version called stricmp). We can tell the map to use this function to do key comparison. Here's how it's done.

There is a version of the map that takes a third template argument--the predicate functor. A predicate is a function that responds with *true* or *false* to a particular question. In our case the question would be, "Is one string less than the other?" A functor is a class that defines a function-call operator. Here's an example of a functor class that's suitable for our purposes:

```
class LessThan
    : public std::binary_function<char const *, char const *, bool>
{
public:
    bool operator () (char const * str1,
                      char const * str2) const
    {
        return strcmp (str1, str2) < 0;
    }
};

std::map<char const *, int, LessThan> dictionary;
```

Notice that the function strcmp returns a number less than zero when the first string precedes the second one lexicographically (i.e., it would precede it in a dictionary--let's forget for a moment about human-language-specific issues). Similarly, it returns a number greater than zero when the first string follows the other; and zero when they're equal.

The way you could use a functor object, other than passing it as a template argument to a map, would be to treat it just like a function (or function pointer).

```
char const * str1 = "cos";
char const * str2 = "sin";
LessThan lessThan;
if (lessThan (str1, str2))
    cout << str1 << " is less than " << str2 << endl;
```

In fact, a functor is even more efficient than a function pointer, because it may be inlined. In the example above, the implicit call to lessThan.operator () will be inlined, according to its declaration in LessThan.

If you think this is too complicated--having to define a predicate functor to do such a simple thing as string comparison in a map--you're right! This is because we are trying to fit a square peg in a round hole. The equivalent of a square peg is the low level data structure that we are insisting on using to represent strings. The Standard Library has a perfectly round peg for this purpose--it's called a string.

And that brings us to the second problem with the naive implementation of the symbol table: the keys could disappear from under the map. When you enter a new association--a pair (key, value)--in the map, the map has to remember both the key and the value. If you use a pointer to character as a key, the map will store the pointer, but you'll have no idea where the actual characters are stored and for how long. Somebody could call the map with a character string that's allocated on the stack and which will disappear as soon as the caller returns. We want the map to *own* the keys! And no, we can't use auto_ptrs as keys. Standard library containers use value semantics for all its data--they get confused by objects that exhibit transfer semantics. And that's the second reason to use strings.

A standard string not only has value semantics, but it also comes with a lot of useful methods--one of them being operator less-than. As a matter of fact, string and map are a perfect match. Implementing an associative array of strings is a no-brainer. So let's do it first, and ask detailed questions later. Here's the declaration of the new symbol table:

```
#include <map>
#include <string>

class SymbolTable
{
public:
    enum { idNotFound = -1 };
    SymbolTable () : _id (0) {}
    int ForceAdd (char const * str);
    int Find (char const * str) const;
private:
    std::map<std::string, int> _dictionary;
    int _id;
};
```

Notice that we are still clinging to the old char const * C-style strings in the symbol table interface. That's because I want to show you how you can start converting old legacy code without having to rewrite everything all at once. The two representations of strings can co-exist quite peacefully. Of course, we'll have to make some conversions at the boundaries of the two worlds. For instance, the method ForceAdd starts by creating a standard string from the C-string argument. It then stores the corresponding integer id under this string in the associative array.

```
int SymbolTable::ForceAdd (char const * str)
{
    std::string name (str);
    _dictionary [name] = _id;
    return _id++;
}
```

Here, the mere act of accessing a map using a key will create an entry, if one wasn't there.

> Don't get fooled by the syntactic simplicity of accessing a map. The time it takes to index an associative array that's implemented as a standard map is not constant, like it is with regular arrays or hash tables, but it depends on its size (number of elements). The good news is that it's not linear--the array is not searched element-by-element. The recommended implementation of an STL map is in terms of a balanced tree that can be searched in logarithmic time. Logarithmic access is pretty good for most applications.

There is also a more explicit way of adding entries to a map by calling its insert method. You insert a pair (key, value) using a standard pair template. For instance, ForceAdd may be also implemented this way:

```
int SymbolTable::ForceAdd (char const * str)
{
    std::string name (str);
    std::pair<std::string, int> p (str, _id);
    _dictionary.insert (p);
    return _id++;
}
```

The Find method of the symbol table could be simply implemented by returning _dictionary [str], if it weren't for the fact that we sometimes want to know whether the name wasn't there. We have to use a more explicit approach.

```
int SymbolTable::Find (char const * str) const
{
    std::map<std::string, int>::const_iterator it;
    it = _dictionary.find (str);
    if (it != _dictionary.end ())
        return it->second;
    return idNotFound;
}
```

The find method of map returns an iterator. This iterator could either point to the end of the dictionary, or to a pair (key, value). A pair has two public data members, first and second, with obvious meanings.

Notice also that I called find with a C-style string. It was possible, because the standard string has a constructor that accepts a C-string. This constructor can be used for implicit conversions.

We can use this property of strings to further simplify the implementation of the symbol table. We can change the signature of its methods to accept strings rather than C-strings.

```
int ForceAdd (std::string const & str);
int Find (std::string const & str) const;
```

This will work without any changes to the callers, because of the implicit conversions. The trick is that both methods take const references to strings. The compiler will have no trouble creating temporary strings at the point of call and passing out const references. This would not work, however, if symbol table methods required non-const references (we already talked about it, but this point is worth repeating--the compiler will refuse to create a non-const reference to a temporary object).

There is an interesting twist on this topic--we may pass strings by value.

```
int ForceAdd (std::string str);
int Find (std::string str) const;
```

This would work, because, as I mentioned, strings have value semantics. They behave as if every copy were a deep copy. However, since memory allocation and copying is rather expensive, a good implementation of a standard string uses all kinds of clever tricks to avoid, or at least postpone, these operations. Let's talk about this in some more detail.

Reference Counting and Copy-On-Write

First, let's make sure we understand what is meant by value semantics as applied to strings. Here's a simple implementation of a value string. Notice the copy constructor and the assignment operator.

```
class StringVal
{
public:
    StringVal (char const * cstr = 0)
        :_buf (0)
    {
        if (cstr != 0)
            Init (cstr);
    }
    StringVal (StringVal const & str)
        :_buf (0)
    {
        if (str.c_str () != 0)
            Init (str.c_str ());
    }
    ~StringVal ()
    {
        delete _buf;
    }

    char const * c_str () const { return _buf; }
    void Upcase ()
private:
    void Init (char const * cstr)
    {
        assert (cstr != 0);
        _buf = new char [strlen (cstr) + 1];
        strcpy (_buf, cstr);
    }
private:
    char  * _buf;
};
```

When overloading the assignment operator, we have to take care of a few special cases. For instance, we should do noting if the source is equal to the target, as in `str = str`. Then we have to be smart about null strings. Finally, we have to reallocate the buffer if there isn't enough space in it for the new string.

```
StringVal & StringVal::operator= (StringVal const & str)
{
    if (this != & str)
    {
        char const * cstr = str.c_str ();
        if (cstr == 0)
        {
            delete _buf;
            _buf = 0;
        }
        else
        {
            int len = strlen (cstr);
            if (_buf == 0 || strlen (_buf) < len)
            {
                delete _buf;
                Init (cstr);
            }
            else

        }
    }
    return *this;
}
```

I added the Upcase method to demonstrate what happens when a string is modified.

```
void StringVal::Upcase ()
{
    if (_buf)
    {
        int len = strlen (_buf);
        for (int i = 0; i < len; ++i)
            _buf [i] = toupper (_buf [i]);
    }
}
```

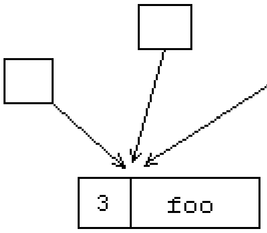
In particular, we can create two copies of the same string and, when we Upcase one of them, the other should remain unchanged.

```
StringVal str1 ("foo");
StringVal str2 (str1); // copy
str2.Upcase ();
cout << str1.c_str () << endl;
cout << str2.c_str () << endl;
```

Semantically, this is how we want our string to behave. Performance-wise, we might not be too happy with this implementation. Consider, as an exercise, how many memory allocations and string copies are made when, for instance, a StringVal is returned by value:

```
StringVal ByValue ()
{
    StringVal ret ("foo");
    return ret;
}

StringVal str;
str = ByValue ();
```

To save on allocating and copying, we might consider a scheme where multiple copies of the same string would internally share the same representation object--they'd have a pointer to the same buffer. But then the question of ownership arises. Who's supposed to delete the buffer? We have to somehow keep track of how many shared owners of the buffer there are at any point in time. Then, when the last owner disappears, we should delete the buffer.



Multiple object sharing the same reference-counted representation.

The best way to implement it is to divide the responsibilities between two classes of objects. One class encapsulates the reference counted shared object--in our case this object will also hold the string buffer. The other class deals with the ownership issues--it increments and decrements the shared object's reference count--in our case this will be the "string" object. When it discovers that the reference count of the shared object has gone down to zero (nobody else references this object) it deletes it.

You might think of reference count as a type of resource--it is acquired by incrementing it and it must subsequently be released by decrementing it. Like any other resource, it has to be encapsulated. Our "string" object will be such an encapsulator. In fact, since reference count is a common type of resource, we will build the equivalent of a smart pointer to deal with reference counted objects.

```
template <class T>
class RefPtr
{
public:
    RefPtr (RefPtr<T> const & p)
    {
        _p = p._p;
        _p->IncRefCount ();

    ~RefPtr ()
    {
        Release ();
    }
    RefPtr const & operator= (RefPtr const & p)
    {
        if (this != &p)
        {
            Release ();
            _p = p._p;
            _p->IncRefCount ();
        }
        return *this;
    }
protected:
    RefPtr (T * p) : _p (p) {}
    void Release ()
    {
        if (_p->DecRefCount () == 0)
            delete _p;
    }
    T * _p;
};
```

Notice that the reference-counted type T must provide at least two methods, IncRefCount and DecRefCount. We also tacitly assume that it is created with a reference count of one, before being passed to the protected constructor of RefPtr.

Although it's not absolutely necessary, we might want the type T to be a descendant of a base class that implements reference counting interface.

```
class RefCounted
{
public:
    RefCounted () : _count (1) {}
    int GetRefCount () const { return _count; }
    void IncRefCount () const { _count++; }
    int DecRefCount () const { return --_count; }
private:
    mutable int _count;
};
```

Notice one interesting detail, the methods IncRefCount and DecRefCount are declared const, even though they modify the object's data. You can do that, without the compiler raising an eyebrow, if you declare the relevant data member mutable. We do want these methods to be const (or at least one of them, IncRefCount) because they are called on const objects in RefPtr. Both the copy constructor and the assignment operator take const references to their arguments, but they modify their reference counts. We decided not to consider the updating of the reference count a "modification" of the object. It will make even more sense when we get to the copy-on-write implementation.

Just for the demonstration purposes, let's create a reference-counted string representation using our original StringVal. Normally, one would do it more efficiently, by combining the reference count with the string buffer.

```
class StringRep: public RefCounted
{
public:
    StringRep (char const * cstr)
        :_string (cstr)

    char const * c_str () const { return _string.c_str (); }
    void Upcase ()
    {
        _string.Upcase ();
    }
private:
    StringVal _string;
};
```

Our actual string class is built on the base of RefPtr which internally represents string data with StringRep.

```
class StringRef: public RefPtr<StringRep>
{
```

```
public:
    StringRef (char const * cstr)
        : RefPtr<StringRep> (new StringRep (cstr))
    {}
    StringRef (StringRef const & str)

    {}
    char const * c_str () const { return _p->c_str (); }
    void Upcase ()
    {
        _p->Upcase ();
    }
};
```

Other than in the special C-string-taking constructor, there is no copying of data. The copy constructor just increments the reference count of the string-representation object. So does the (inherited) assignment operator. Consequently, "copying" and passing a StringRef by value is relatively cheap. There is only one tiny problem with this implementation. After you call Upcase on one of the copies of a StringRef, all other copies change to upper case.

```
StringRef strOriginal ("text");
StringRef strCopy (strOriginal);
strCopy.Upcase ();
// The original will be upper-cased!
```

There is, however, a way to have a cake and eat it, too. It's called copy-on-write, or COW for short. The idea is to share a single representation between multiple copies, as long as they don't want to make any modifications. Every modifying method would have to make sure that its modifications are not shared. It would check the reference count of its representation and, if it's greater than one, make a private copy. This way, the copying is delayed as long as possible. Passing by value is as cheap as in the case of shared representation, but modifications are no longer shared between copies.

```
class StringCow: public RefPtr<StringRep>
{
public:
    StringCow (char const * cstr)
        : RefPtr<StringRep> (new StringRep (cstr))
    {}
    StringCow (StringCow const & str)
        : RefPtr<StringRep> (str)
    {}
    char const * c_str () const { return _p->c_str (); }
    void Upcase ()
    {
        Cow ();
        _p->Upcase ();
    }
private:
    void Cow ()
    {
        if (_p->GetRefCount () > 1)
        {
            // clone it
            StringRep * rep = new StringRep (_p->c_str ());
            Release ();
            _p = rep;
        }
    }
};
```

The beauty of this implementation is that, from the point of view of the user, StringCow behaves exactly like StringVal, down to the const-reference-taking copy constructor and assignment operator. Except that, when it comes to passing around by value, its performance is superior.

There is a good chance that your standard library implements string using some version of copy-on-write.

End of Restrictions

Now that we are no longer scared of passing strings by value, we might try some more code improvements. For instance, we can replace this awkward piece of code:

```
int Scanner::GetSymbolName (char * strOut, int lenBuf)
{
    assert (lenBuf > maxSymLen);
    assert (_lenSymbol < lenBuf);

    strncpy (strOut, &_buf [_iSymbol], _lenSymbol);
    strOut [_lenSymbol] = 0;
    return _lenSymbol;
}
```

When you use strings, you don't have to worry about sending in appropriately sized buffers. You let the callee create a string of the correct size and return it by value.

```
std::string Scanner::GetSymbolName ()
{
    return std::string (&_buf [_iSymbol], _lenSymbol);
```

And that's what it looks like on the receiving side.

```
std::string strSymbol = _scanner.GetSymbolName ();
int id = _symTab.Find (strSymbol);
...
cerr << "Unknown function \"";
cerr << strSymbol << "\"" << endl;
```

Notice that the string strSymbol is a local variable with limited scope. It will disappear when the flow of control leaves the scope and it will deallocate whatever resources it owns (or at least it will decrement their reference count).

By the way, as you can see, a string may be sent to the standard output (or standard error) and it will be printed just like a c-string. Actually, it gets even better than that. You can read text from the standard input directly into a string. The beauty of it is that, because a string can dynamically resize itself, there is practically no restriction on the size of acceptable text. The string will accommodate as much text as the user wishes to type into it. Of course, there has to be a way terminate the string. Normally, the standard input will stop filling a string once it encounters any whitespace character. That means you may read one word at a time with simple code like this:

```
std::string str;
std::cin >> str;
```

In our program, we have a slightly different requirement. We want to be able to read one line at a time. The standard library has an appropriate function to do that, so let's just use it directly in main.

```
cerr << "> ";  // prompt
std::string str;
std::getline (cin, str);
Scanner scanner (str);
```

> Bug Alert! The standard library that comes with VC++ 6.0 has a bug that makes getline expect two newlines rather than one, as line terminator. A fix for this and many other bugs is available on the Internet at www.dinkumware.com.

Let's stop here for a moment. We have just removed the last built-in limitation from our program. There is no longer any restriction on the length of the input line, or on the length of an identifier. The file **params.h** is gone!

How did that happen? Well, we started using dynamic data structures. Is our program more complex because of that? No, it's not! In fact it's simpler now. Is there any reason to introduce limitations into programs? Hardly!

Now let's follow the fate of the input string from main and into the scanner. It makes little sense to store a naked c-string in the scanner. So let's rewrite it to use a standard string instead.

```
class Scanner
{
public:
    Scanner (std::string const & buf);
    bool IsDone () const { return _iLook == std::string::npos; }
    bool IsEmpty () const { return _buf.length () == 0; }
    EToken  Token () const { return _token; }
    EToken  Accept ();
    std::string GetSymbolName ();
    double Number ();
private:
    void EatWhite ();
    typedef std::string::size_type size_type;

    std::string const & _buf;
    size_type           _iLook;
    EToken              _token;
    double              _number;
    size_type           _iSymbol;
    size_type           _lenSymbol;
    static char         _whiteChars [];
};
```

The natural way of marking the fact that we are beyond the end of string is to use a special index value, `string::npos`. This is the value that is returned by string's various "find" methods when end of string is reached. For instance, we can use one such method, `find_first_not_of`, to skip whitespace in our buffer.

```
char Scanner::_whiteChars [] = " \t\n\r";

void Scanner::EatWhite ()
{
    _iLook = _buf.find_first_not_of (_whiteChars, _iLook);
}
```

The method `find_first_not_of` takes a null-terminated array of characters to be skipped (in our case the array contains a space, a tab, a newline and a carriage return) and the optional starting index which defaults to zero. It returns the index of the first occurrence of acharacterr that is not in the skip list. If no such character is found, it returns `string::npos`.

By the way, the value `string::npos` is guaranteed to be greater than any valid index, as long as you are comparing the same integral types. That's why we made sure we use the same type, `size_type`, for our index as the one used internally by the string itself.

`GetSymbolName` is returning a substring of the buffer.

```
std::string Scanner::GetSymbolName ()
{
    return _buf.substr (_iSymbol, _lenSymbol);
}
```

An alternative would be to use the string constructor that takes the source string, the starting offset and the length.

The rest of the implementation of the scanner works with almost no change. We only have to make sure that, at the end of `Accept`, we set the position to `string::npos` after the buffer has been consumed.

```
if (_iLook == _buf.length ())
    _iLook = std::string::npos;
```

In particular, code like this, although not the most elegant, will work with strings as well as it used to work with straight character arrays:

```
char * p;
_number = strtod (&_buf [_iLook], &p);
_iLook = p - &_buf [0];
```

This is not the preferred way of writing a program, if you've decided to use the standard library from the onset of the project. But I wanted to show you that it's quite possible to start the conversion in a legacy program and not have to go all the way at once.

Exploring Streams

So what is the preferred way; if we really wanted to use the standard library to its full advantage? I guess we wouldn't bother reading the line from the standard input into a string. We would just pass the stream directly to the scanner. After all, we don't need random access to the input line--we are parsing it more or less one character at a time. Except in rare cases, we don't have to go back in the string to re-parse it (there are some grammars that require it--ours doesn't). And when we do, it's only by one character.

Let's start from the top. We can create the scanner, passing it the standard input stream as an argument. By the way, the type of cin is std::istream.

```
cerr << "> ";  // prompt
Scanner scanner (cin);
```

Here's the new definition of the Scanner class.

```
class Scanner
{
public:
    explicit Scanner (std::istream & in);
    bool IsDone () const { return _token == tEnd; }
    bool IsEmpty () const { return _isEmpty; }
    EToken    Token () const { return _token; }
    void    Accept ();
    std::string GetSymbolName ();
    double Number ();
private:
    void ReadChar ();

    std::istream  & _in;
    int            _look;    // lookahead character
    bool           _isEmpty;
    EToken         _token;
    double         _number;
    std::string    _symbol;
};
```

I did a little reorganizing here. I'm keeping a lookahead character in _look. I also decided to have a Boolean flag _isEmpty, to keep around the information that the stream was empty when the scanner was constructed (I can no longer look back at the beginning of input, once Accept has been called). I changed the test for IsDone to simply compare the current token with tEnd. Finally, I needed a string to keep the name of the last symbolic variable read from the input.

Here's the constructor of the Scanner:

```
Scanner::Scanner (std::istream & in)
    : _in (in)
{
    Accept ();
    _isEmpty = (Token () == tEnd);
}
```

The Accept method needs a little rewriting. Where we used to call EatWhite (), we now call ReadChar (). It skips whitespace as before, but it also initializes the lookahead character to the first non-white character. Since the lookahead has been consumed from the input stream, we don't have to do any incrementing after we've recognized it in Accept.

```
void Scanner::Accept ()
```

```
{
    ReadChar ();

    switch (_look)
    {
    case '+':
        _token = tPlus;
        // no incrementing
        break;
    ...
    }
}
```

This is the implementation of ReadChar:

```
void Scanner::ReadChar ()
{
    _look = _in.get ();
    while (_look == ' ' || _look == '\t')
        _look = _in.get ();
}
```

I had to rethink the handling of the end of input. Before, when we used getline to read input, we actually never had to deal with a newline. By definition, getline eats the newline and terminates the string appropriately (i.e., the c-string version appends a null, the std::string version updates the internal length). The get method, on the other hand, reads every character as is, including the newline. So I let the scanner recognize a newline as the end of input.

```
    case '\n': // end of input
    case '\r':
    case EOF: // end of file
        _token = tEnd;
        break;
```

Incidentally, I did some thinking ahead and decided to let the scanner recognize the end of file. The special EOF value is returned by the get () method when it encounters the end of file. This value is not even a character (that's why get is defined to return an int, rather than char).

How can a standard input stream encounter an end of file? There's actually more than one way it may happen. First, you may enter it from the keyboard--in DOS it's the combination Ctrl-Z. Second, the program might be called from the command line with redirected input. You may create a text file, say **calc.txt**, filled with commands for the calculator and then call it like this:

```
calc < calc.txt
```

The operating system will plug the contents of this file into the program's standard input and execute it. You'll see the results of calculations flashing on your standard output. That is unless you redirect it too, like this:

```
calc < calc.txt > results.txt
```

Then you'll only see only the standard error (including the prompts) flashing before your eyes, and the file **results.txt** will be filled with results of your calculations.

Let's continue with our rewrite of the scanner. Here's what we do when we recognize a number:

```
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
case '.':
{
    _token = tNumber;
    _in.putback (_look);
    _in >> _number; // read the whole number
    break;
}
```

Reading a floating-point number from the standard input is easy. The only complication arises from the fact that we've already read the first character of the number--our lookahead. So before we read the whole number, we have to put our lookahead back into the stream. Don't worry, this is a simple operation. After all, the input stream is buffered. When you call get, the character is simply read from a buffer (unless the buffer is empty--in that case the system replenishis it by actually reading the input). Ungetting a character just means putting it back into that buffer. Input streams are implemented in such a way that it's always possible to put back one character.

When reading an identifier, we do a slight variation of the same trick.

```
default:
    if (isalpha (_look) || _look == '_')
    {
        _token = tIdent;
        _symbol.erase (); // erase string contents
        do {
            _symbol += _look;
            _look = _in.get ();
        } while (isalnum (_look));
        _in.putback (_look);
    }
    else
        _token = tError;
    break;
```

We don't have to putback a lookahead at the beginning of reading an identifier. Instead, we have to putback the last character, the one that is not part of the identifier, so that the next call to ReadChar () can see it.

Haven't we lost some generality by switching from a string to a stream in our implementation of the scanner? After all, you can always convert a stream to a string (e.g., using getline ()). Is the opposite possible? Not to worry! Converting a string into a stream is as easy. The appropriate class is called istringstream and is defined in the header **<sstream>**. Since istringstream inherits from istream, our scanner won't notice the difference. For instance, we can do this:

```
std::istringstream in ("sin (2 * pi / 3)");
Scanner scanner (in);
```

We have just skimmed the surface of the standard library and we've already found a lot of useful stuff. It really pays to study it, rather than implement your own solutions from scratch.

Code Review 7: Serialization and Deserialization

The Calculator Object

Look at main: There are too many objects there. The symbol table, the function table and the store. All three objects have the same lifespan--the duration of the program execution. They have to be initialized in particular order and all three of them are passed to the constructor of the parser. They just scream to be combined into a single object called--you guessed it--the Calculator. Embedding them in the right order inside this class will take care of the correct order of initialization.

```
class Calculator
{
    friend class Parser;
public:
    Calculator ()
        : _funTab (_symTab),
          _store (_symTab)
    {}

private:
    Store & GetStore () { return _store; }
    PFun GetFun (int id) const { return _funTab.GetFun (id); }
    bool IsFunction (int id) const { return id < _funTab.Size (); }
    int AddSymbol (std::string const & str)
    {
        return _symTab.ForceAdd (str);
    }
    int FindSymbol (std::string const & str) const
    {
        return _symTab.Find (str);
    }

    SymbolTable     _symTab;
    Function::Table _funTab;
    Store           _store;
};
```

Of course, now we have to make appropriate changes (read: simplifications) in main and in the parser. Here are just a few examples--in the declaration of the parser:

```
class Parser
{
public:
    Parser (Scanner & scanner, Calculator & calc);
    ...
private:
    ...
    Scanner         & _scanner;
    auto_ptr<Node>    _pTree;
    Status            _status;
    Calculator      & _calc;

};
```

and in its implementation.

```
// Factor := Ident
if (id == SymbolTable::idNotFound)
{
    id = _calc.AddSymbol (strSymbol);
}
pNode = auto_ptr<Node> (new VarNode (id, _calc.GetStore ()));
```

Have you noticed something? We just went ahead and made another major top-level change in our project, just like this! In fact it was almost trivial to do, with just a little help from the compiler. Here's the prescription.

Start in the spot in main where the symbol table, function table and store are defined (constructed). Replace them with the new object, calculator. Declare the class for Calculator and write a constructor for it. Now, if you are really lazy and tired of thinking, fire off the compiler. It will immediately tell you what to do next: You have to modify the constructor of the parser. You have to pass it the calculator rather than its three separate parts. At this point you might notice that it will be necessary to change the class declaration of the Parser to let it store a reference to the Calculator. Or, you could run the compiler again and let it remind you of it. Next, you will notice all the compilation errors in the implementation of Parser. You can fix them one-by-one, adding new methods to the Calculator as the need arises. The whole procedure is so simple that you might ask an intern who has just started working on the project to do it with minimal supervision.

The moral of this story is that it's never too late to work on the improvement of the high level structure of the project. The truth is that you rarely get it right the first time. And, by the way, you have just seen the method of top-down program modification. You start from the top and let the compiler lead you all the way down to the nitty-gritty details of the implementation. That's the third part of the top-down methodology which consists of:

- Top-down design
- Top-down implementation and
- Top-down modification.

I can't stress enough the importance of the top-down methodology. I have yet to see a clean, well written piece of code that was created bottom-up. You'll hear people saying that some things are better done top-down, others bottom-up. Some people will say that starting from the middle and expanding in both directions is the best way to go. Take all such statements with a very big grain of salt.

It is a fact that bottom-up development is more natural when you have no idea what you're doing-- when your goal is not to write a specific program, but rather to play around with some "neat stuff." It's an easy way, for instance, to learn the interface to some obscure subsystem that you might want to use. Bottom-up development is also preferable if you're not very good at design or if you dislike just sitting there and thinking instead of coding. It is a plus if you enjoy long hours of debugging or have somebody else (hopefully not the end user!) to debug your code.

Finally, if you embrace the bottom-up philosophy, you'll have to resign yourself to never being able to write a professionally looking piece of code. Your programs will always look to the trained eye like those electronics projects created with Radio Shack parts, on breadboards, with bent wires sticking out in all directions and batteries held together with rubber bands.

The real reason I decided to finally get rid of the top level mess and introduce the Calculator object was to simplify the job of adding a new piece of functionality. Every time the management asks you to add new features, take the opportunity to sneak in a little rewrite of the existing code. The code isn't good enough if it hasn't been rewritten at least three times. I'm serious!

By rewriting I don't mean throwing it away and starting from scratch. Just take your time every now and then to improve the structure of each part of the project. It will pay off tremendously. It will actually shorten the development cycle. Of course, if you have stress-puppy managers, you'll have a hard time convincing them about it. They will keep running around shouting nonsense like "if it ain't broken, don't fix it" or "if we don't ship it tomorrow, we are all dead." The moment you buy into that, you're doomed! You'll never be able to do anything right and you'll be spending more and more time fixing the scaffolding and chasing bugs in some low quality temporary code pronounced to be of the "ain't broken" quality. Welcome to the maintenance nightmare!

So here we are, almost at the end of our project, when we are told that if we don't provide a command to save and restore the state of the calculator from a file, we're dead. Fortunately, we can add this feature to the program without much trouble and, as a bonus, do some more cleanup.

Command Parser

We'll go about adding new functionality in an orderly fashion. We have to provide the user with a way to input commands. So far we've had a hack for inputting the *quit* command--an empty line was interpreted as "quit." Now that we want to add two more commands, *save* and *restore*, we can as well find a more general solution. I probably don't have to tell you that, but...

Whenever there are more than two special cases, you should generalize them.

The calculator expects expressions from the user. Let's distinguish commands from expressions by prefixing them with an exclamation sign. Exclamation has the natural connotation of commanding somebody to do something. We'll use a prefix rather than a suffix to simplify our parsing. We'll also make *quit* a regular command; to be input as "!q". We'll even remind the user of this command when the calculator starts.

```
cerr << "\n!q to quit\n";
```

The new Scanner method *IsCommand* simply checks for the leading exclamation sign. Once we have established that a line of text is a command, we create a simple CommandParser to parse and execute it.

```
if (!scanner.IsEmpty ())
{
    if (scanner.IsCommand())
    {
        CommandParser parser (scanner, calc);
        status = parser.Execute ();
    }
    else
    {
        Parser  parser (scanner, calc);
        status = parser.Parse ();
        if (status == stOk)
        {
            double result = parser.Calculate ();
            cout << result << endl;
        }
        else
        {
            cerr << "Syntax error\n";
        }
    }
}
```

Here's the new class, `CommandParser`,

```
class CommandParser
{
    enum ECommand
    {
        comSave,
        comLoad,
        comQuit,
        comError
    };
public:
    CommandParser (Scanner & scanner, Calculator & calc);
    Status Execute ();
private:
    Status Load (std::string const & nameFile);
    Status Save (std::string const & nameFile);


    Scanner &    _scanner;
    Calculator & _calc;
    ECommand     _command;
};
```

and this is how it parses a command.

```
CommandParser::CommandParser (Scanner & scanner, Calculator & calc)
: _scanner (scanner),
  _calc (calc)
{
    assert (_scanner.IsCommand());
    _scanner.Accept ();
    std::string name = _scanner.GetSymbolName ();
    switch (name [0])
    {
    case 'q':
```

```
    case 'Q':
        _command = comQuit;
        break;
    case 's':
    case 'S':
        _command = comSave;
        break;
    case 'l':
    case 'L':
        _command = comLoad;
        break;
    default:
        _command = comError;
        break;
    }
}
```

Notice that we use the Scanner method GetSymbolName to retrieve the command string.

The *load* and *save* commands require an argument, the file name. We retrieve it from the scanner using, again, the method SymbolName.

```
Status CommandParser::Execute ()
{
    scanner.AcceptCommand ();
    std::string nameFile;
    switch (_command)
    {
    case comSave:
        nameFile = _scanner.GetSymbolName ();
        return Save (nameFile);
    case comLoad:
        nameFile = _scanner.GetSymbolName ();
        return Load (nameFile);
    case comQuit:
        cerr << "Good Bye!" << endl;
        return stQuit;
    case comError:
        cerr << "Error" << endl;
        return stError;
    }
    return stOk;
}
```

We use the new method, AcceptCommand, to accept the command and read the following string. The string, presumably a file name, must be terminated by a whitespace. Notice that we can't use the regular Accept method of the Scanner, because it will only read strings that have the form of C++ identifiers. It would stop, for instance, after reading a dot, which is considered a perfectly valid part of a file name. (If we were stricter, we would even make provisions for file names with embedded spaces. We'd just require them to be enclosed in quotation marks.)

```
void Scanner::AcceptCommand ()
{
    ReadChar ();
    _symbol.erase ();
    while (!isspace (_look))
    {
        _symbol += _look;
        _look = _in.get ();
    }
}
```

As usual, we should provide simple stubs for the Load and Save methods and test our program before proceeding any further.

Serialization and Deserialization

We often imagine data structures as two- or even three-dimensional creatures (just think of a parsing tree, a hash table, or a multi-dimensional array). A disk file, on the other hand, has a one-dimensional structure--it's linear. When you write to a file, you write one thing after another--serially. Hence the name *serialization*. Saving a data structure means transforming a multi-dimensional idea into its one-dimensional representation. Of course, in reality computer memory is also one-dimensional. Our data structures are already, in some manner, serialized in memory. Some of them, like multi-dimensional arrays, are serialized by the compiler, others are fit into linear memory with the use of pointers. Unfortunately, pointers have no meaning outside the context of the currently running instance of the program. You can't save pointers to a file, close the program, start it again, read the file and expect the newly read pointers to point to the same data structures as before.

In order to serialize a data structure, you have to come up with a well-defined procedure for walking it, i.e., visiting every single element of it, one after another. For instance, you can walk a simple linked list by following the *next* pointers until you hit the end of the list. If the list is circular, you have to remember the initial pointer and, with every step, compare it with the *next* pointer. A binary tree can be walked by walking the left child first and the right child next (notice that it's a recursive prescription). For every data structure there is at least one deterministic procedure for walking it, but the procedure might be arbitrarily complicated.

Once you know how to walk a data structure, you know how to serialize it. You have a prescription for how to visit every element of the structure, one after another--a serial way of scanning it. At the bottom level of every data structure you find simple, built-in types, like int, char, long, etc. They can be written to a file following a set of simple rules--we'll come back to this point in a moment. If you know how to serialize each basic element, you're done.

Serializing a data structure makes sense only if we know how to restore it--*deserialize* it from file to memory. Knowing the original serialization procedure helps--we can follow the same steps when when we deserialize it; only now we'll *read* from file and *write* to memory, rather than the other way around. We have to make sure, however, that the procedure is unambiguous. For instance, we have to know when to stop reading elements of a given data structure. We must know where the end of a data structure is. The clues that were present during serialization might not be present on disk. For instance, a linked list had a null pointer as *next* in its last element. But if we decide not to store pointers, how are we to know when we have reached the end of the list? Of course, we may decide to store the pointers anyway, just to have a clue when to stop. Or, even better, we could store the count of elements in front of the list.

The need to know sizes of data structures before we can deserialize them imposes additional constraints on the order of serialization. When we serialize one part of the program's data, all other parts are present in memory. We can often infer the size of a given data structure by looking into other data structures. When deserializing, we don't have this comfort. We either have to make sure that these other data structures are deserialized first, or add some redundancy to the serialized image, e.g., store the counts multiple times. A good example is a class that contains a pointer to a dynamically allocated array and the current size of the array. It really doesn't matter which member comes first, the pointer or the count. However, when serializing an object we must store the count first and the contents of the array next. Otherwise we won't be able to allocate the appropriate amount of memory or read the correct number of entries.

Another kind of ambiguity might arise when storing polymorphic data structures. For instance, a binary node contains two pointers to `Node`. That's not a problem when we serialize it--we can tell the two children to serialize themselves by calling the appropriate virtual functions. But when the time comes to deserialize the node, how do we know what the real type of each child was? We have to know that before we can even start deserializing them. That's why the serialized image of any polymorphic data structure has to start with some kind of code that identifies the class of the data structure. Based on this code, the deserializer will be able to call the appropriate constructor.

Let's now go back to our project and implement the (de-) serialization of the Calculator's data structures. First we have to create an output file. This file will be encapsulated inside a serial stream. The stream can accept a number of basic data types, long, double; as well as some simple aggregates, like strings; and write them to the file.

Notice that I didn't mention the most common type--the integer. That's because the size of the integer is system dependent. Suppose that you serialize a data structure that contains integers and send it on a diskette or through e-mail to somebody who has a version of the same program running on a different processor. Your program might write an integer as two bytes and their program might expect a four-byte or even eight-byte integer. That's why, when serializing, we convert the system-dependent types, like integers, to system-independent types like longs. In fact, it's not only the size that matters--the order of bytes is important as well.

> There are essentially two kinds of processors, the ones that use the Big Endian and the ones that use the Little Endian order (some can use either). For instance, a `short` or a `long` can be stored most-significant-byte-first or least-significant-byte-first. The Intel(tm) family of processor stores the least significant byte first--the Little Endian style--whereas the Motorola(tm) family does the opposite. So if you want your program to inter-operate between Wintel and Macintosh (tm), you'll have to take the order of bytes into account when you serialize. Of course, if you're *not* planning on porting your program between the two camps, you may safely ignore one of them.
>
> Anyway, in most cases you *should* take precautions against variable size types and convert integers or enumerations to fixed-size types.

It would be great to be able to assume that once you come up with the on-disk format for your program, it will never change. In real life it would be very naïve. Formats change and the least you can do to acknowledge it is to refuse to load a format you don't understand.

Always store a version number in your on-disk data structures.

In order to implement serialization, all we have to do is to create a stream, write the version number into it and tell the calculator to serialize itself. By the way, we are now reaping the benefits of our earlier combining several objects into the Calculator object.

```
const long Version = 1;

Status CommandParser::Save (std::string const & nameFile)
{
    cerr << "Save to: \"" << nameFile << "\"\n";
    Status status = stOk;
    try
    {
        Serializer out (nameFile);
        out.PutLong ( Version );
        _calc.Serialize (out);
    }
    catch (char* msg)
    {
        cerr << "Error: Save failed: " << msg << endl;
        status = stError;
    }
    catch (...)
    {
        cerr << "Error: Save failed\n";
        status = stError;
    }
    return status;
}
```

When deserializing, we follow exactly the same steps, except that now we read instead of writing and deserialize instead of serializing. And, if the version number doesn't match, we refuse to load.

```
Status CommandParser::Load (std::string const & nameFile)
{
    cerr << "Load from: \"" << nameFile << "\"\n";
    Status status = stOk;
    try
    {
        DeSerializer in (nameFile);
        long ver = in.GetLong ();
        if (ver != Version)
            throw "Version number mismatch";
        _calc.DeSerialize (in);
    }
    catch (char* msg)
    {
        cerr << "Error: Load failed: " << msg << endl;
        status = stError;
    }
    catch (...)
    {
        cerr << "Error: Load failed\n";
        // data structures may be corrupt
        throw;
    }
    return status;
}
```

There are two objects inside the Calculator that we'd like to save to the disk--the symbol table and the store--the names of the variables and their values. So that's what we'll do.

```
void Calculator::Serialize (Serializer & out)
{
    _symTab.Serialize (out);
    _store.Serialize (out);
}
```

```
void Calculator::DeSerialize (DeSerializer & in)
{
    _symTab.DeSerialize (in);
    _store.DeSerialize (in);
}
```

The symbol table consists of a dictionary that maps strings to integers plus a variable that contains the current id. And the simplest way to walk the symbol table is indeed in this order. To walk the standard map we will use its iterator. First we have to store the count of elements, so that we know how many to read during deserialization. Then we will iterate over the whole map and store pairs: string, id. Notice that the iterator for `std::map` points to a `std::pair` which has `first` and `second` data members. According to our previous discussion, we store the integer id as a `long`.

```
void SymbolTable::Serialize (Serializer & out) const
{
    out.PutLong (_dictionary.size ());
    std::map<std::string, int>::const_iterator it;
    for (it = _dictionary.begin (); it != _dictionary.end (); ++it)
    {
        out.PutString (it->first);
        out.PutLong (it->second);
    }
    out.PutLong (_id);
}
```

The deserializer must read the data in the same order as they were serialized: first the dictionary, then the current id. When deserializing the map, we first read its size. Then we simply read pairs of strings and longs and add them to the map. Here we treat the map as an associative array. Notice that we first clear the existing dictionary. We have to do it, otherwise we could get into conflicts, with the same id corresponding to different strings.

```
void SymbolTable::DeSerialize (DeSerializer & in)
{
    _dictionary.clear ();
    int len = in.GetLong ();
    for (int i = 0; i < len; ++i)
    {
        std::string str = in.GetString ();
        int id = in.GetLong ();
        _dictionary [str] = id;
    }
    _id = in.GetLong ();
}
```

Notice that for every serialization procedure we immediately write its counterpart--the deserialization procedure. This way we make sure that the two match.

The serialization of the store is also very simple. First the size and then a series of pairs (double, bool).

```
void Store::Serialize (Serializer & out) const
{
    int len = _aCell.size ();
    out.PutLong (len);
    for (int i = 0; i < len; ++i)
    {
        out.PutDouble (_aCell [i]);
        out.PutBool (_aIsInit [i]);
    }
}
```

When deserializing the store, we first clear the previous values, read the size and then read the pairs (double, bool) one by one. We have a few options when filling the two vectors with new values. One is be to push them back, one by one. Since we know the number of entries up front, we could reserve space in the vectors up front, by calling the method reserve. Here I decided to resize the vectors instead and then treat them as arrays. The resizing fills the vector of doubles with zeroes and the vector of bool with false (these are the default values for these types).

There is an important difference between reserve and resize. Most standard containers have either one or both of these methods. Reserve makes sure that there will be no re-allocation when elements are added, e.g., using push_back, up to the reserved *capacity*. This is a good optimization, in case we know the required capacity up front. In the case of a vector, the absence of re-allocation also means that iterators, pointers or references to the elements of the vector won't be suddenly invalidated by internal reallocation.

Reserve, however, does not change the *size* of the container. Resize does. When you resize a container new elements are added to it. (Consequently, you can't resize containers that store objects with no default constructors or default values.)

- reserve--changes capacity but not size
- resize--changes size

capacity method. And, of course, you get its size by calling size.

```
void Store::DeSerialize (DeSerializer & in)
{
    _aCell.clear ();
    _aIsInit.clear ();
    int len = in.GetLong ();
    _aCell.resize (len);
    _aIsInit.resize (len);
    for (int i = 0; i < len; ++i)
    {
        _aCell [i] = in.GetDouble ();
        _aIsInit [i] = in.GetBool ();
    }
}
```

Finally, let's have a look at the implementation of the deserializer stream. It is a pretty thin layer on top of the output stream.

```
#include <fstream>
using std::ios_base;

const long TruePattern = 0xfab1fab2;
const long FalsePattern = 0xbad1bad2;

class DeSerializer
{
public:
    DeSerializer (std::string const & nameFile)
        : _stream (nameFile.c_str (), ios_base::in | ios_base::binary)
    {
        if (!_stream.is_open ())
            throw "couldn't open file";
    }
    long GetLong ()
    {
        if (_stream.eof())
            throw "unexpected end of file";
        long l;
        _stream.read (reinterpret_cast<char *> (&l), sizeof (long));
        if (_stream.bad())
            throw "file read failed";
        return l;
    }
    double GetDouble ()
    {
        double d;
        if (_stream.eof())
            throw "unexpected end of file";
        _stream.read (reinterpret_cast<char *> (&d), sizeof (double));
        if (_stream.bad())
            throw "file read failed";
        return d;
    }
    std::string GetString ()
    {
        long len = GetLong ();
        std::string str;
        str.resize (len);
        _stream.read (&str [0], len);
        if (_stream.bad())
            throw "file read failed";
        return str;
    }
    bool GetBool ()
    {
        long b = GetLong ();
        if (_stream.bad())
            throw "file read failed";
        if (b == TruePattern)
            return true;
        else if (b == FalsePattern)
            return false;
        else
            throw "data corruption";
    }
private:
    std::ifstream _stream;
};
```

Several interesting things happen here. First of all: What are these strange flags that we pass to `ifstream::open ()`? The first one, `ios_base::in`, means that we are opening the file for input. The second one, `ios_base::binary`, tells the operating system that we don't want any carriage return-linefeed translations.

What is this carriage return-linefeed nonsense? It's one the biggest blunders of the DOS file system, that was unfortunately inherited by all flavors of Windows. The creators of DOS decided that the system should convert single character '\n' into a pair '\r', '\n'. The reasoning was that, when you print a file, the printer interprets carriage return, '\r', as the command to go back to the beginning of the current line, and line feed, '\n', as the command to move down to the next line (not necessarily to its beginning). So, to go to the beginning of the next line, a printer requires two characters. Nowadays, when we use laser printers that understand Postscript and print wysywig documents, this whole idea seems rather odd. Even more so if you consider that an older operating system, Unix, found a way of dealing with this problem without involving low level file system services.

Anyway, if all you want is to store bytes of data in a file, you have to remember to open it in the "binary" mode, otherwise you might get unexpected results. By the way, the default mode is `ios_base::text` which does the unfortunate character translation.

Another interesting point is that the method `ifstream::read` reads data to a character buffer--it expects `char *` as its first argument. When we want to read a long, we can't just pass the address of a long to it--the compiler doesn't know how to convert a `long *` to a `char *`. This is one of these cases when we *have to* force the compiler to trust us. We want to split the long ito its constituent bytes (we're ignoring here the big endian/little endian problem). A reasonably clean way to do it is to use the `reinterpret_cast`. We are essentially telling the compiler to "reinterpret" a chunk of memory occupied by the long as a series of chars. We can tell how many chars a long contains by applying to it the operator sizeof.

This is a good place to explain the various types of casts. You use

- const_cast--to remove the const attribute
- static_cast--to convert related types
- reinterpret_cast--to convert unrelated types

(There is also a dynamic_cast, which we won't discuss here.)

Here's an example of const_cast:

```
char const * str = "No modify!";
char * tmp = const_cast<char *> (str);
tmp [0] = 'D';
```

To understand static_cast, think of it as the inverse of implicit conversion. Whenever type T can be implicitly converted to type U (in other words, T is-a U), you can use static_cast to perform the conversion the other way. For instance, a char can be implicitly converted to an int:

```
char c = '\n';
int i = c; // implicit conversion
```

Therefore, when you need to convert an int into a char, use static_cast:

```
int i = 0x0d;
char c = static_cast<char> (i);
```

Or, if you have two classes, Base and Derived: public Base, you can implicitly convert pointer to Derived to a pointer to Base (Derived is-a Base). Therefore, you can use static_cast to go the other way:

```
Base * bp = new Derived; // implicit conversion
Derived * = static_cast<Base *> (bp);
```

You should realize that casts are dangerous and should be used very judiciously. Try to avoid casting at all costs. Serialization and deserialization are special in this respect, since they require low level manipulation of types.

Finally, notice the strange way we store Boolean values. A Boolean value really requires only one bit for its storage. But, since we don't want to split bytes (or even longs, for that matter), we'll use some redundancy here. We could, in principle store the value true as one and false as zero. However, it will cost us the same to write a zero as to write an arbitrary value. The difference is that zeros are much more common in files than, say, *0xbad1bad2*. So when I read back the value *0xbad1bad2* and I expect a Boolean, I feel reassured that I'm reading sensible data and not some random garbage. This is only one of the ways of using redundancy for consistency checking.

The output serializing stream is the mirror image of DeSerializer.

```
class Serializer
{
public:
    Serializer (std::string const & nameFile)
        : _stream (nameFile.c_str (), ios_base::out |
ios_base::binary)
    {
        if (!_stream.is_open ())
```

```
            throw "couldn't open file";
    }
    void PutLong (long l)
    {
        _stream.write (reinterpret_cast<char *> (&l), sizeof (long));
        if (_stream.bad())
            throw "file write failed";
    }
    void PutDouble (double d)
    {
        _stream.write (reinterpret_cast<char *> (&d), sizeof 
(double));
        if (_stream.bad())
            throw "file write failed";
    }
    void PutString (std::string const & str)
    {
        int len = str.length ();
        PutLong (len);
        _stream.write (str.data (), len);
        if (_stream.bad())
            throw "file write failed";
    }
    void PutBool (bool b)
    {
        long l = b? TruePattern: FalsePattern;
        PutLong (l);
        if (_stream.bad ())
            throw "file write failed";
    }
private:
    std::ofstream _stream;
};
```

There is a shortcut notation combining assignment with a conditional. The following code:

```
long l = b? TruePattern: FalsePattern;
```

is equivalent to:

```
long l;
if (b)
    l = TruePattern;
else
    l = FalsePattern;
```

The ternary (meaning, three-argument) operator A? B: C first evaluates A. If A is true, it evaluates and returns B, otherwise it evaluates and returns C. A piece of trivia: unlike in C, in C++ the ternary operator returns an l-value, so it can be used on the left-hand-side of the assignment. Not that I would recommend this style!

There is an even more obscure operator in C++, the comma sequencing operator. The expression A, B first evaluates A, then evaluates and returns B. The evaluation of A is therefore a side effect of the whole operation. Most often the comma operator is used to combine two expressions where one is expected, like in this double loop:

```
for (int i = 0, j = 0; i < maxI && j < maxJ; ++i, ++j)
```

By the way, the first comma separates the declarations (complete with initialization) of two variables of the same type. It's the second comma, between ++i and ++j, that is the sequencing operator.

Notice how protective we are when reading from or writing to a file. That's because our program doesn't have full control of the disk. A write can fail because we run out of disk space. This can happen at any time, because we are not the only client of the file system--there are other applications and system services that keep allocating (and presumably freeing) disk space. Reading is worse, because we're not even sure what to expect in the file. Not only may a read fail because of a hardware problem (unreadable disk sector), but we must be prepared for all kinds of sabotage. Other applications could have gotten hold of our precious file and truncated, edited or written all over it. We can't even be sure that the file we are trying to parse has been created by our program. The user could have mistakenly or maliciously pass to our program the name of some executable, a spreadsheet or autoexec.bat.

We already have the first line of defense against such cases of mistaken identity or downright corruption--the version number. The first four bytes we read from the file must match our current version number or we refuse to load it. The error message we display in such a case is a bit misleading. A much better solution would be to spare a few additional bytes and stamp all our files with a magic number. Many people use their initials for the magic number in the hope that one day they'll be able to say to their children or grandchildren, "You see these bytes at the beginning of each file of this type? These are your mom's (dad's, gramma's, grampa's) initials." Provided the application or the system survives that long and is not widely considered an example of bad software engineering.

In-Memory (De-) Serialization

Serialization of data structures is not necessarily related to their storage in files. Sometimes you just want to store some data structure in a chunk of memory, especially if you want to pass it to another application. Programs can talk to each other and pass data through shared memory or other channels (Windows clipboard comes to mind). You might also want to send data in packets across the network. These are all situations in which you can't simply pass pointers embedded in your data. You have to change the format of data.

The serialization procedure is the same, whether the output goes to a file or to memory. In fact, if your data structure is serializable (it has the `Serialize` and `DeSerialize` methods), all you might need to do in order to serialize it to memory is to change the implementation of `Serializer` and `DeSerializer`. Even better, you might make these classes abstract--turn methods `PutLong`, `PutDouble`, `PutBool` and `PutString` to pure virtual--and provide two different implementations, one writing to a file and one writing to memory. You can do the same with the deserializer.

There is one big difference between a file and a chunk of memory--the file grows as you write to it, a chunk of memory has fixed size. You have two choices--you can either grow your memory buffer as needed, or you can calculate the required amount of memory up front and pre-allocate the whole buffer. As it turns out, calculating the size of a serializable data structure is surprisingly easy. All you need is yet another implementation of the Serializer interface called the counting serializer. The counting serializer doesn't write anything, it just adds up the sizes of various data types it is asked to write.

```
class CountingSerializer: public Serializer
{
public:
    CountingSerializer ()
        : _size (0) {}
    int GetSize () const { return _size; }
    void PutLong (long l)
    {
        _size += sizeof (long);
    }
    void PutDouble (double d)
    {
        _size += sizeof (double);
    }
    void PutString (std::string const & str)
    {
        _size += sizeof (long); // count
        _size += str.length ();
    }
    void PutBool (bool b)
    {
        _size += sizeof (long);
    }
private:
    int _size;
};
```

For instance, if you wanted to calculate the size of the file or memory buffer required for the serialization of a calculator, you'd call its Serialize method with a counting serializer.

```
CountingSerializer counter;
_calc.Serialize (counter);
int size = counter.GetSize ();
```

Remember that, in order for this to work, all methods of Serializer *must* be virtual.

Multiple Inheritance

In order to make a class serializable, you have to add to it two methods, Serialize and DeSerialize, and implement them. It makes sense, then, to create a separate abstract class--a pure interface--to abstract this behavior.

```
class Serializable
{
public:
    virtual void Serialize (Serializer & out) const = 0;
    virtual void DeSerialize (DeSerializer & in) = 0;
};
```

All classes that are serializable, should inherit from the `Serializable` interface.

```
class Calculator: public Serializable
class SymbolTable: public Serializable
class Store: public Serializable
```

What's the advantage of doing that? After all, even when you inherit from `Serializable`, you still have to add the declaration of the two methods to you class and you have to provide their implementation. Suppose that a new programmer joins your group and he (or she) has to add a new class to the project. One day he sends you email asking, "How do I make this class serializable?" If this functionality is abstracted into a class, your answer could simply be, "Derive your class from Serializable." That's it! No further explanation is necessary.

There is however a catch. What if your class is already derived from some other class? Now it will have to inherit from that class *and* from Serializable. This is exactly the case in which multiple inheritance can be put to work. In C++ a class may have more than one base class. The syntax for multiple inheritance is pretty straightforward:

```
class MultiDerived: public Base1, public Base2
```

Suppose, for instance, that you were not satisfied with treating `std::string` as a simple type, known to the `Serializer`. Instead, you'd like to create a separate type, a serializable string. Here's how you could do it, using multiple inheritance:

```
using std::string;

class SerialString: public string, public Serializable
{
public:
    SerialString (std::string const & str): string (str) {}
    void Serialize (Serializer & out) const;
    void DeSerialize (DeSerializer & in);
};
```

Multiple inheritance is particularly useful when deriving from abstract classes. This kind of inheritance deals with interface rather than implementation. In fact, this is exactly the restriction on multiple inheritance that's built into Java. In Java you can inherit only from one full-blown class, but you can add to it multiple inheritance from any number of interfaces (the equivalent of C++ abstract classes). In most cases this is indeed a very reasonable restriction.

Transactions

Imagine using a word processor. You are editing a large document and, after working on it for several hours, you decide to save your work. Unfortunately, there is not enough disk space on your current drive and the save fails. What do you expect to happen?

Option number one is: the program gives up and exits. You are horrified--you have just lost many hours of work. You try to start the word processor again and you have a heart attack--the document is corrupted beyond recovery. Not only have you lost all recent updates, but you lost the original as well.

If horrors like this don't happen, it is because of transactions. A transaction is a series of operations that move the program from one well defined state to another. A transaction must be implemented in such a way that it either completely succeeds or totally fails. If it fails, the program must return to its original state.

In any professionally written word processor, saving a document is a transaction. If the save succeeds, the on-disk image of the document (the file) is updated with the current version of the document and all the internal data structures reflect this fact. If the save fails, for whatever reason, the on-disk image of the documents remains unchanged and all the internal data structures reflect that fact. A transaction cannot succeed half-way. If it did, it would leave the program and the file in an inconsistent, corrupted, state.

Let's consider one more word-processing scenario. You are in the middle of editing a document when suddenly all lights go out. Your computer doesn't have a UPS (Uninterrupted Power Supply) so it goes down, too. Five minutes later, the electricity is back and the computer reboots. What do you expect to happen?

The nightmare scenario is that the whole file system is corrupt and you have to reformat your disk. Of course, your document is lost forever. Unfortunately, this is a real possibility with some file systems. Most modern file systems, however, are able to limit the damage to a single directory. If this is not good enough for you (and I don't expect it is), you should look for a recoverable file system. Such systems can limit the damage down to the contents of the files that had been open during the crash. It does it by performing transactions whenever it updates the file system metadata (e.g., directory entries). Asking anything more from a file system (e.g., transacting all writes) would be impractical--it would slow down the system to a crawl.

Supposing you have a recoverable file system, you should be able to recover the last successfully saved pre-crash version of your document. But what if the crash happened during the save operation? Well, if the save was implemented as a transaction it is guaranteed to leave the persistent data in a consistent state--the file should either contain the complete previously saved version or the complete new version.

Of course, you can't expect to recover the data structures that were stored in the volatile memory of your computer prior to the crash. That data is lost forever. (It's important to use the auto-save feature of your word processor to limit such losses). That doesn't mean that you can't or shouldn't transact operations that deal solely with volatile data structures. In fact, every robust program must use transactions if it is to continue after errors or exceptions.

What operations require transactions?

Any failure-prone action that involves updating multiple data structures might require a transaction.

Transient Transactions

A transient, or in-memory, transaction does not involve any changes to the persistent (usually on-disk) state of the program. Therefore a transient transaction is not robust in the face of system crashes or power failures.

We have already seen examples of such transactions when we were discussing resource management. A construction of a well implemented complex data structure is a transaction--it either succeeds or fails. If the constructor of any of the sub-objects fails and throws an exception, the construction of the whole data structure is reversed and the program goes back to the pre-construction state (provided all the destructors undo whatever the corresponding constructors did). That's just one more bonus you get from using resource management techniques.

There are however cases when you have to do something special in order to transact an operation. Let's go back to our word processor example. (By the way, the same ideas can be applied to the design of an editor; and what programmer didn't, at one time or another, try to write his or her own editor.) Suppose that we keep text in the form of a list of paragraphs. When the user hits return in the middle of a paragraph, we have to split this paragraph into two new ones. This operation involves several steps that have to be done in certain order:

- Allocate one new paragraph.
- Allocate another new paragraph.
- Copy the first part of the old paragraph into the first new paragraph.
- Copy the second part of the old paragraph into the second new paragraph.
- Plug the two new paragraphs in the place of the old one.
- Delete the old paragraph.

The switch--when you plug in the new paragraphs--is the most sensitive part of the whole operation. It is performed on some master data structure that glues all paragraphs into one continuous body of the document. It is also most likely a dynamic data structure whose modifications might fail--the computer might run out of memory while allocating an extension table or a link in a list. Once the master data structure is updated, the whole operation has been successful. In the language of transactions we say "the transaction has committed." But if the crucial update fails, the transaction aborts and we have to unroll it. That means we have to get the program back to its original state (which also means that we refuse to split the paragraph).

What's important about designing a transaction is to make sure that

- All operations that proceed the commit are undoable in a safe manner (although the operations themselves don't have to--and usually aren't--safe).
- The commit operation is safe.
- All operations that follow it are also safe.

Operations that involve memory allocation are not safe--they may fail, e.g., by throwing an exception. In our case, it's the allocation of new paragraphs that's unsafe. The undo operation, on the other hand, is the deletion of these paragraphs. We assume that deletion is safe--it can't fail. So it is indeed okay to do paragraph allocation before the commit.

The commit operation, in our case, is the act of plugging in new paragraphs in the place of the old paragraph. It is most likely implemented as a series of pointer updates. Pointer assignment is a safe operation.

The post-commit cleanup involves a deletion, which is a safe operation. Notice that, as always, we assume that destructors never throw any exceptions.

The best way to implement a transaction is to create a transaction object. Such an object can be in one of two states: committed or aborted. It always starts in the aborted state. If its destructor is called before the state is changed to committed, it will unroll all the actions performed under the transaction. Obviously then, the transaction object has to keep track of what's already been done--it keeps the *log* of actions. Once the transaction is committed, the object changes its state to committed and its destructor doesn't unroll anything.

Here's how one could implement the transaction of splitting the current paragraph.

```
void Document::SplitCurPara ()
{
    Transaction xact;
    Paragraph * para1 = new Paragraph (_curOff);
    xact.LogFirst (para1);
    Paragraph * para2 = new Paragraph (_curPara-->Size () - _curOff);
    xact.LogSecond (para2);
    Paragraph * oldPara = _curPara;
    // May throw an exception!
    SubstCurPara (para1, para2);
    xact.Commit ();
    delete oldPara;
    // destructor of xact executed
}
```

This is how the transaction object is implemented.

```
class Transaction
{
public:
    Transaction () : _commit (false), _para1 (0), _para2 (0) {}
    ~Transaction ()
    {
        if (!_commit)
        {
            // unroll all the actions
            delete _para2;
            delete _para1;
        }
    }
    void LogFirst (Paragraph * para) { _para1 = para; }
    void LogSecond (Paragraph * para) { _para2 = para; }
    void Commit () { _commit = true; }
private:
    bool        _commit;
    Paragraph * _para1;
    Paragraph * _para2;
};
```

Notice how carefully we prepare all the ingredients for the transaction. We first allocate all the resources and log them in our transaction object. The new paragraphs are now owned by the transaction. If at any point an exception is thrown, the destructor of the Transaction, still in its non-committed state, will perform a rollback and free all these resources.

Once we have all the resources ready, we make the switch--new resources go into the place of the old ones. The switch operation usually involves the manipulation of some pointers or array indexes. Once the switch has been done, we can commit the transaction. From that point on, the transaction no longer owns the new paragraphs. The destructor of a committed transaction usually does nothing at all. The switch made the document the owner of the new paragraphs and, at the same time, freed the ownership of the old paragraph which we then promptly delete. All simple transactions follow this pattern:

- Allocate and log all the resources necessary for the transaction.
- Switch new resources in the place of old resources and commit.
- Clean up old resources.

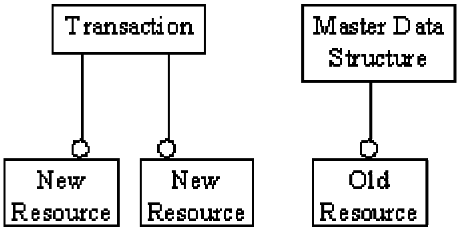


Figure: Prepared transaction. The transaction owns all the new resources. The master data structure owns the old resources.

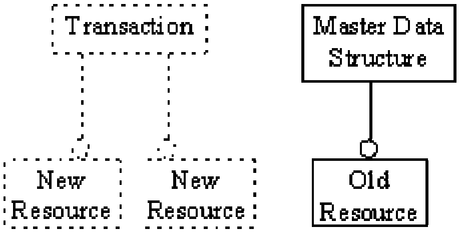


Figure: Aborting a transaction. The transaction's destructor frees the resources.

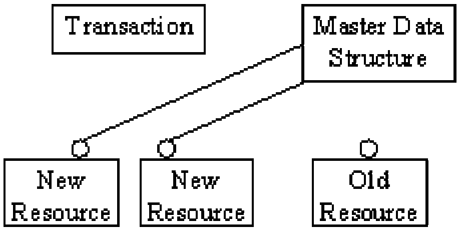


Figure 3--11 The switch. The master data structure releases the old resources and takes the ownership of the new resources.

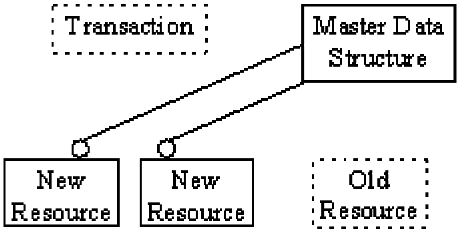


Figure 3--12 The cleanup. Old resources are freed and the transaction is deleted.

Persistent Transactions

When designing a persistent transaction--one that manipulates persistent data structures--we have to think of recovering from such disasters as system crashes or power failures. In cases like those, we are not so much worried about in-memory data structures (these will be lost anyway), but about the persistent, on-disk, data structures.

A persistent transaction goes through similar stages as the transient one.

- Preparation: New information is written to disk.
- Commitment: The new information becomes current, the old is disregarded.
- Cleanup: The old information is removed from disk.

A system crash can happen before or after commitmentment (I'll explain in a moment why it can't happen during the commit). When the system comes up again, we have to find all the interrupted transactions (they have to leave some trace on disk) and do one of two things: if the transaction was interrupted before it had a chance to commit, we must unroll it; otherwise we have to complete it. Both cases involve cleanup of some on-disk data. The unrolling means deleting the information written in preparation for the transaction. The completing means deleting the old information that is no longer needed.

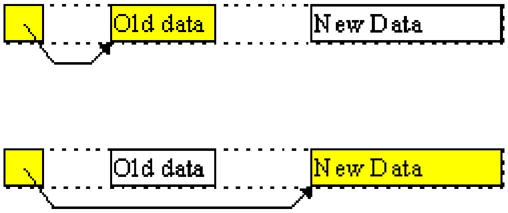


Figure: The Switch. In one atomic write the on-disk data structure changes its contents.

The crucial part of the transaction is, of course, commitmentment. It's the "flipping of the switch." In one atomic operation the new information becomes current and the old becomes invalid. An atomic operation either succeeds and leaves a permanent trace on disk, or fails without leaving a trace. That shouldn't be difficult, you'd say. How about simply writing something to a file? It either succeeds or fails, doesn't it?

Well, there's the rub! It doesn't! In order to understand that, we have to delve a little into the internals of a file system. First of all, writing into a file doesn't mean writing to disk. Or, at least, not immediately. In general, file writes are buffered and then cached in memory before they are physically written to disk. All this is quietly done by the runtime (the buffering) and by the operating system (the caching) in order to get reasonable performance out of your machine. Disk writes are so incredibly slow in comparison with memory writes that caching is a must.

What's even more important: the order of physical disk writes is not guaranteed to follow the order of logical file writes. In fact the file system goes out of its way to combine writes based on their physical proximity on disk, so that the magnetic head doesn't have to move too much. And the physical layout of a file might have nothing to do with its contiguous logical shape. Not to mention writes to different files that can be quite arbitrarily reordered by the system, no matter what your program thinks.

Thirdly, contiguous writes to a single file may be split into several physical writes depending on the disk layout set up by your file system. You might be writing a single 32-bit number but, if it happens to straddle sector boundaries, one part of it might be written to disk in one write and the other might wait for another sweep of the cache. Of course, if the system goes down between these two writes, your data will end up partially written. So much for atomic writes.

Now that I have convinced you that transactions are impossible, let me explain a few tricks of trade that make them possible after all. First of all, there is a file system call, `Flush`, that makes 100% sure that the file data is written to the disk. Not atomically, mind you--Flush may fail in the middle of writing a 32--bit number. But once Flush succeeds, we are guaranteed that the data is safely stored on disk. Obviously, we have to flush the new data to disk before we go about committing a transaction. Otherwise we might wake up after a system crash with a committed transaction but incomplete data structure. And, of course, another flush must finish the committing a transaction.

How about atomicity? How can we atomically flip the switch? Some databases go so far as to install their own file systems that support atomic writes. We won't go that far. We will assume that if a file is small enough, the writes are indeed atomic. "Small enough" means not larger than a sector. To be on the safe side, make it less than 256 bytes. Will this work on every file system? Of course, not! There are some file systems that are not even recoverable. All I can say is that this method will work on NTFS--the Windows NT(tm) file system. You can quote me on this.

We are now ready to talk about the simplest implementation of the persistent transaction--the three file scheme.

The Three-File Scheme

An idealized word processor reads an input file, lets the user edit it and then saves the result. It's the save operation that we are interested in. If we start overwriting the source file, we're asking for trouble. Any kind of failure and we end up with a partially updated (read: corrupted!) file.

So here's another scheme: Write the complete updated version of the document into a separate file. When you are done writing, flush it to make sure the data gets to disk. Then commit the transaction and clean up the original file. To keep permanent record of the state of the transaction we'll need one more small file. The transaction is committed by making one atomic write into that file.

So here is the three-file scheme: We start with file A containing the original data, file B with no data and a small 1-byte file S (for Switch) initializedcontaintain a zero. The transaction begins.

- Write the new version of the document into file B.
- Flush file B to make sure that the data gets to disk.
- Commit: Write 1 into file S and flush it.
- Empty file A.

The meaning of the number stored in file S is the following: If its value is zero, file A contains valid data. If it's one, file B contains valid data. When the program starts up, it checks the value stored in S, loads the data from the appropriate file and empties the other file. That's it!

Let's now analyze what happens if there is a system crash at any point in our scheme. If it happens before the new value in file S gets to the disk, the program will come up and read zero from S. It will assume that the correct version of the data is still in file A and it will empty file B. We are back to the pre-transaction state. The emptying of B is our *rollback*.

Once the value 1 in S gets to the disk, the transaction is committed. A system crash after that will result in the program coming back, reading the value 1 from S and assuming that the correct data is in file B. It will empty file A, thus completing the transaction. Notice that data in file B is guaranteed to be complete at that point: Since the value in S is one, file B must have been flushed successfully.

If we want to start another *save* transaction after that, we can simply interchange the roles of files A and B and commit by changing the value in S from one to zero. To make the scheme even more robust, we can choose some random (but fixed) byte values for our switch, instead of zero and one. In this way we'll be more likely to discover on-disk data corruption--something that might always happen as long as disks are not 100% reliable and other applications can access our files and corrupt them. Redundancy provides the first line of defense against data corruption.

This is how one might implement a *save* transaction.

```
class SaveTrans
{
    enum State
    {
        // some arbitrary bit patterns
        stDataInA = 0xC6,
        stDataInB = 0x3A
    };
public:
    SaveTrans ()
        : _switch ("Switch"), _commit (false)
    {
        _state = _switch.ReadByte ();
        if (_state != stDataInA && state != stDataInB)
            throw "Switch file corrupted";
        if (_state == stDataInA)
        {
            _data.Open ("A");
            _backup.Open ("B");
        }
        else
        {
            _data.Open ("B");
            _backup.Open ("A");
        }
    }
    File & GetDataFile () { return _data; }
    File & GetBackupFile () { return _backup; }
    ~SaveTrans ()
    {
        if (_commit)
            _data.Empty ();
        else
            _backup.Empty ();
    }
    void Commit ()
    {
        State otherState;
        if (_state == stDataInA)
            otherState = stDataInB;
        else
            otherState = stDataInA;

        _backup.Flush ();
        _switch.Rewind ();
        _switch.WriteByte (otherState);
        _switch.Flush ();
        _commit = true;
    }
private:
    bool    _commit;
    File    _switch;
    File    _data;
    File    _backup;
    State   _state;
};
```

This is how this transaction might be used in the process of saving a document.

```
void Document::Save ()
{
    SaveTrans xact;
    File &file = xact.GetBackupFile ();
    WriteData (file);
    xact.Commit ();
}
```

And this is how it can be used in the program initialization.

```
Document::Document ()
{
    SaveTrans xact;
    File &file = xact.GetDataFile ();
    ReadData (file);
    // Don't commit!
    // the destructor will do the cleanup
}
```

The same transaction is used here for cleanup. Since we are not calling `Commit`, the transaction cleans up, which is exactly what we need.

The Mapping-File Scheme

You might be a little concerned about the performance characteristics of the three-file scheme. After all, the document might be a few megabytes long and writing it (and flushing!) to disk every time you do a save creates a serious overhead. So, if you want to be a notch better than most word processors, consider a more efficient scheme.

The fact is that most of the time the changes you make to a document between saves are localized in just a few places . Wouldn't it be more efficient to update only those places in the file instead of rewriting the whole document? Suppose we divide the document into "chunks" that fit each into a single "page." By "page" I mean a power-of-two fixed size subdivision. When updating a given chunk we could simply swap a page or two. It's just like swapping a few tiles in a bathroom floor--you don't need to re-tile the whole floor when you just want to make a small change around the sink.

Strictly speaking we don't even need fixed size power-of-two pages, it just makes the flushes more efficient and the bookkeeping easier. All pages may be kept in a single file, but we need a separate "map" that establishes the order in which they appear in the document. Now, if only the "map" could fit into a small switch file, we would perform transactions by updating the map.

Suppose, for example, that we want to update page two out of a ten-page file. First we try to find a free page in the file (we'll see in a moment how transactions produce free pages). If a free page cannot be found, we just extend the file by adding the eleventh page. Then we write the new updated data into this free page. We now have the current version of a part of the document in page two and the new version of the same part in page eleven (or whatever free page we used). Now we atomically overwrite the map, making page two free and page eleven take its place.

What if the map doesn't fit into a small file? No problem! We can always do the three-file trick with the map file. We can prepare a new version of the map file, flush it and commit by updating the switch file.

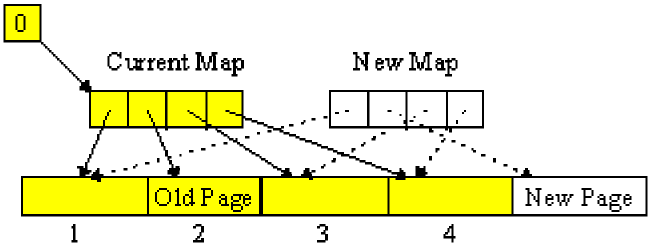


Figure: The Mapping File Scheme: Before committing.

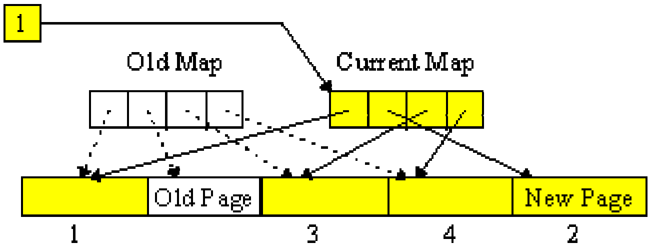


Figure: The Mapping File Scheme: After committing.

This scheme can be extended to a multi-level tree. In fact several databases and even file systems use something similar, based on a data structure called a B-tree.

Overloading operator new

Both `new` and `delete` are considered operators in C++. What it means, in particular, is that they can be overloaded like any other operator. And just like you can define a class-specific `operator=`, you can also define class-specific operators `new` and `delete`. They will be automatically called by the compiler to allocate and deallocate objects of that particular class. Moreover, you can overload and override global versions of `new` and `delete`.

Class-specific new

Dynamic memory allocation and deallocation are not cheap. A lot of programs spend the bulk of their time inside the heap, searching for free blocks, recycling deleted blocks and merging them to prevent heap fragmentation. If memory management is a performance bottleneck in your program, there are several optimization techniques that you might use.

Overloading `new` and `delete` on a per-class basis is usually used to speed up allocation/deallocation of objects of that particular class. There are two main techniques--caching and bulk allocation.

Caching

The idea behind caching is that recycling is cheaper than manufacturing. Suppose that we wanted to speed up additions to a hash table. Every time an addition is performed, a new link is allocated. In our program, these links are only deallocated when the whole hash table is destroyed, which happens at the end of the program. Imagine, however, that we are using our hash table in another program, where it's either possible to selectively remove items from the hash table, or where there are many hash tables created and destroyed during the lifetime of the program. In both cases, we might speed up average link allocation time by keeping around the links that are currently not in use.

A `FreeList` object will be used as storage for unused links. To get a new link we call its `NewLink` method. To return a link back to the pool, we call its `Recycle` method. The pool of links is implemented as a linked list. There is also a `Purge` method that frees the whole pool.

```
class Link;

class FreeList
{
public:
    FreeList () : _p (0) {}
    ~FreeList ();
    void Purge ();
    void * NewLink ();
    void Recycle (void * link);
private:
    Link * _p;
};
```

Class `Link` has a static member `_freeList` which is used by the overloaded class-specific operators `new` and `delete`. Notice the assertion in operator `new`. It protects us from somebody calling this particular operator for a different class. How could that happen? Operators new and delete are inherited. If a class derived from Link didn't override these operators, new called for the derived class would return an object of the wrong size (base-class size).

```
class Link
{
    friend class FreeList;
public:
    Link (Link * pNext, int id)
    : _pNext (pNext), _id (id) {}

    Link *  Next () const { return _pNext; }
    int     Id () const { return _id; }
    // allocator
    void * operator new (size_t size)
    {
        assert (size == sizeof (Link));
        return _freeList.NewLink ();
    }
    void operator delete (void * mem)
    {
        if (mem)
            _freeList.Recycle (mem);
    }
    static void Purge () { _freeList.Purge (); }
private:
    static    FreeList _freeList;

    Link *  _pNext;
    int     _id;
};
```

Inside List::Add the creation of a new Link will be translated by the compiler into the call to the class-specific operator new followed by the call to its constructor (if any). The beauty of this method is that no changes to the implementation of List are needed.

```
class List
{
public:
    List ();
    ~List ()
    {
        while (_pHead != 0)
        {
            Link * pLink = _pHead;
            _pHead = _pHead->Next();
            delete pLink;
        }
    }
    void Add (int id)
    {
        Link * pLink = new Link (_pHead, id);
        _pHead = pLink;
    }
    Link const * GetHead () const { return _pHead; }
private:
    Link * _pHead;
};
```

A hash table contains an array of Lists which will all internally use the special-purpose allocator for its links.

After we are done with the hash table, we might want to purge the memory stored in the private allocator. That would make sense if, for instance, there was only one hash table in our program, but it allowed deletion as well as addition of entries. On the other hand, if we wanted our pool of links to be shared between multiple hash tables, we wouldn't want to purge it every time a hash table is destroyed.

```
class HTable
{
public:
    explicit HTable (int size): _size(size)
    {
        _aList = new List [size];
    }

    ~HTable ()
    {
        delete [] _aList;
        // release memory in free list
        Link::Purge (); // optional
    }

    // ...
private:
    List * _aList;
    int    _size;
};
```

Notice: Purge is a *static* method of Link, so we don't need an instance of a Link in order to call it.

In the implementation file, we first have to define the static member _freeList of the class Link. Static data is automatically initialized to zero.

```
FreeList Link::_freeList;
```

The implementation of FreeList is pretty straightforward. We try to reuse Links, if possible; otherwise we call the global operator new. Since we are allocating raw memory, we ask for sizeof (Link) bytes (chars). When we delete this storage, we cast Links back to their raw form. Deleting a Link as a Link would result in a (second!) call to its destructor. We don't want to do it here, since destructors for these Links have already been called when the class-specific delete was called.

```
void * FreeList::NewLink ()
{
    if (_p != 0)
    {
        void * mem = _p;
        _p = _p->_pNext;
        return mem;
    }
    else
    {
        // use global operator new
        return ::new char [sizeof (Link)];
    }
}

void FreeList::Recycle (void * mem)
{
    Link * link = static_cast<Link *> (mem);
    link->_pNext = _p;
    _p = link;
}

FreeList::~FreeList ()
{
    Purge ();
}

void FreeList::Purge ()
{
    while (_p != 0)
    {
        // it was allocated as an array of char
        char * mem = reinterpret_cast<char *> (_p);
        _p = _p->Next();
        ::delete [] mem;
    }
}
```

Notice all the casting we have to do. When our overloaded new is called, it is expected to return a void pointer. Internally, however, we either recycle a Link from a linked-list pool, or allocate a raw chunk of memory of the appropriate size. We don't want to call ::new Link, because that would have an unwanted side effect of calling Link's constructor (it will be called anyway after we return from operator new).

Our delete, on the other hand, is called with a void pointer, so we have to cast it to a Link in order to store it in the list.

Purge deletes all as if they were arrays of chars--since that is how they were allocated. Again, we don't want to delete them as Links, because Link destructors have already been called.

As usually, calls to global operators new and delete can be disambiguated by prepending double colons. Here, they ar not strictly necessary, but they enhance the readability.

Bulk Allocation

{

Another approach to speeding up allocation is to allocate in bulk and thus amortize the cost of memory allocation across many calls to operator new. The implementation of Links, Lists and HashTables is as before, except that a new

class, LinkAllocator is used in place of FreeList. It has the same interface as FreeList, but its implementation is more involved. Besides keeping a list of recycled Links, it also has a separate list of blocks of links. Each block consists of a header of class Block and a block of 16 consecutive raw pieces of memory each the size of a
Link.

```
class Link;

class LinkAllocator
{
    enum { BlockLinks = 16 };
    class Block
    {
    public:
        Block * Next () { return _next; }
        void SetNext (Block * next) { _next = next; }
    private:
        Block * _next;
    };
public:
    LinkAllocator () : _p (0), _blocks (0) {}
    ~LinkAllocator ();
    void Purge ();
    void * NewLink ();
    void Recycle (void * link);
private:
    Link  * _p;
    Block * _blocks;
};
```

This is how a new Link is created:

```
void * LinkAllocator::NewLink ()
{
    if (_p == 0)
    {
        // use global operator new to allocate a block of links
        char * p = ::new char [sizeof (Block) + BlockLinks * sizeof
(Link)];
        // add it to the list of blocks
        Block * block = reinterpret_cast<Block *> (p);
        block->SetNext (_blocks);
        _blocks = block;
        // add it to the list of links
        p += sizeof (Block);
        for (int i = 0; i < BlockLinks; ++i)
        {
            Link * link = reinterpret_cast<Link *> (p);
            link->_pNext = _p;
            _p = link;
            p += sizeof (Link);
        }
    }
    void * mem = _p;
    _p = _p->_pNext;
    return mem;
}
```

The first block of code deals with the situation when there are no unused links in the `Link` list. A whole block of 16 (`BlockLinks`) `Link`-sized chunks is allocated all at once, together with some room for the `Block` header. The `Block` is immediately linked into the list of blocks and then chopped up into separate `Link`s which are added to the `Link` list. Once the `Link` list is replenished, we can pick a `Link` from it and pass it out.

The implementation of `Recycle` is the same as before--the links are returned to the `Link` list. `Purge`, on the other hand, does bulk deallocations of whole blocks.

```
void LinkAllocator::Purge ()
{
    while (_blocks != 0)
    {
        // it was allocated as an array of char
        char * mem = reinterpret_cast (_blocks);
        _blocks = _blocks->Next();
        ::delete [] mem;
    }
}
```

Only one call in 16 to `new Link` results in actual memory allocation. All others are dealt with very quickly by picking a ready-made `Link` from a list.

Array new

Even though class `Link` has overloaded operators `new` and `delete`, if you were to allocate a whole array of `Link`s, as in `new Link [10]`, the compiler would call global `new` to allocate enough memory for the whole array. It would *not* call the class-specific overload. Conversly, deleting such an array would result in the call to global operator `delete`--not it's class-specific overload.

Since in our program we never allocate arrays of `Link`s, we have nothing to worry about. And even if we did, global new and delete would do the right thing anyway.

However, in the unlikely case when you actually *want* to have control over array allocations, C++ provides a way. It let's you overload operators `new[]` and `delete[]`. The syntax and the signatures are analogous to the overloads of straight `new` and `delete`.

```
void * operator new [] (size_t size);
void operator delete [] (void * p);
```

The only difference is that the size passed to `new[]` takes into account the total size of the array plus some additional data used by the compiler to distinguish between pointers to objects and arrays of objects. For instance, the compiler has to know the number of elements in the array in order to be able to call destructors on all of them when `delete []` is called.

All four operators `new`, `delete`, `new[]` and `delete[]` are treated as static members of the class that overloads them (i.e., they don't have access to `this`).

Global new

Unlike class-specific new, global new is usually overloaded for debugging purposes. In some cases, however, you might want to overload global new and delete permanently, because you have a better allocation strategy or because you want more control over it.

In any case, you have a choice of overriding global new and delete or adding your own special versions that follow a slightly different syntax. Standard operator new takes one argument of type size_t. Standard delete takes one orgument of type void *. You can define your own versions of new and delete that take additional arguments of arbitrary types. For instance, you can define

```
void * operator new (size_t size, char * name);
void operator delete (void * p, char * name);
```

and call the special new using this syntax:

```
Foo * p = new ("special") Foo;
```

Unfortunately, there is no way to call the special delete explicitly, so you have to be sure that standard delete will correctly handle memory allocated using your special new (or that delete is never called for such objects).

So what's the use of the overloaded delete with special arguments? There is actually one case in which it will be called--when an exception is thrown during object construction. As you might recall, there is a contract implicit in the language that if an exception happens during the construction of an object, the memory for this object will be automatically deallocated. It so happens that during object's construction the compiler is still aware of which version of operator new was called to allocate memory. It is therefore able to generate a call to the corresponding version of delete, in case an exception is thrown. After the successful completion of construction, this information is no longer available and the compiler has no means to guess which version of global delete is appropriate for a given object.

Once you have defined an overloaded version of new, you can call it explicitly, by specifying additional argument(s). Or you can substitute all calls to new in your code with the overloaded version using macro substitution.

Macros

We haven't really talked about macros in this book--they are a part of standard C++, but their use is strongly discouraged. In the old times, they were used in place of the more sophisticated C++ features, such as inline functions and templates. Now that there are better ways of getting the same functionality, macros are fast becoming obsolete. But just for completeness, let me explain how they work.

> Macros are obnoxious, smelly, sheet-hogging bedfellows for several reasons, most of which are related to the fact that they are a glorified text-substitution facility whose effects are applied during preprocessing, before any C++ syntax and semantic rules can even begin to apply.
>
> Herb Sutter

A macro works through literal substitution. You may think of macro expansion as a separate process performed by the compiler before even getting to the main task of parsing C++ syntax. In fact, in older compilers, macro expansion was done by a separate program, the preprocessor.

There are two major types of macros. The first type simply substitutes one string with another, in the code that logically follows it (by *logically* I mean that, if the macro is defined in an include file, it will also work in the file that includes it, and so on). Let me give you an example that might actually be useful. Let's define the following macro in the file dbnew.h

```
#define new new(__FILE__, __LINE__)
```

This macro will substitute all occurrences of `new` that logically follow it with the string `new (__FILE__, __LINE__)`. Moreover, the macro preprocessor will then substitute all occurrences of the special pre-defined symbol `__FILE__` with the full name of the source file in which it finds it; and all occurrences of `__LINE__` with the appropriate line number. So if you have a file c:\test\main.cpp with the contents:

```
#include "dbnew.h"
int main ()
{
    int * p = new int;
    return 0;
}
```

it will be pre-processed to produce the following code:

```
int main ()
{
    int * p = new ("c:\test\main.cpp", 4) int;
    return 0;
}
```

Now you can use your own overloaded operator `new`, for instance to trace all memory allocation. Here's a simple example of such implementation.

```
void * operator new (size_t size, char const * file, int line)
{
    std::cout << file << ": " << line << std::endl;
    return ::new char [size];
}
```

Notice that we have to make sure that our macro is not included in the file that defines this overload. Otherwise both occurrences of "new" would be substituted by "new(__FILE__, __LINE)" resulting in incorrect code.

The second type of macro also works through textual substitution, but it behaves more like an inline function-- it takes arguments. And again, since macro expansion works outside of the C++ compiler, there is no type checking and a possibility of unexpected side effects. A classic example is the `max` macro:

```
#define max(a, b) (((a) > (b))? (a): (b))
```

Notice the parentesis paranoia--a characteristic feature of macros. Programmers learned to put parentheses around macro parameters, because they might be expressions containing low precedence operators. Consider, for instance, what happens when you call

```
c = max (a & mask, b & mask)
```

Without the parentheses around parameters in the definition of `max`, the preprocessor would expand it into

```
c = a & mask >  b & mask? a & mask: b & mask;
```

which, because of operator precedence rules, would be interpreted as:

```
c = (a & (mask > b) & mask)? (a & mask): (b & mask;)
```

The result of this calculation would most likely be erroneous.

Things get even worse, when you call a macro with expressions that have side effects. Consider for instance the expansion of `max (a++, b++)`:

```
(((a++) > (b++))? (a++): (b++))
```

One of the variables will be incremented twice, the other once. This is probably not what the programmer expected.

By the way, there is one more gotcha--notice that I didn't put a space between.

Tracing Memory Leaks

{

A more interesting application of this technique lets you trace unreleased allocations, a.k.a. memory leaks. The idea is to store information about each allocation in a global data structure and dump its contents at the end of the program. Overloaded operator `delete` would remove entries from this data structure.

Since operator delete has only access to a pointer to previously allocated memory, we have to be able to reasonably quickly find the entry based on this pointer. A map keyed by a pointer comes to mind immediately. We'll call this global data structure a Tracer

```
class Tracer
{
private:
    class Entry
    {
    public:
        Entry (char const * file, int line)
            : _file (file), _line (line)
        {}
        Entry ()
            : _file (0), _line (0)
        {}
        char const * File () const { return _file; }
        int Line () const { return _line; }
    private:
        char const * _file;
        int _line;
    };
    class Lock
    {
    public:
        Lock (Tracer & tracer)
            : _tracer (tracer)
        {
            _tracer.lock ();
        }
        ~Lock ()
        {
            _tracer.unlock ();
        }
    private:
        Tracer & _tracer;
    };
    typedef std::map<void *, Entry>::iterator iterator;
    friend class Lock;
public:
    Tracer ();
    ~Tracer ();
    void Add (void * p, char const * file, int line);
    void Remove (void * p);
    void Dump ();

    static bool Ready;
private:
    void lock () { _lockCount++; }
    void unlock () { _lockCount--; }
private:

    std::map<void *, Entry> _map;
    int _lockCount;
};
```

We have defined two auxillary classes, Tracer::Entry which is used as the value for the map, and Tracer::Lock which is used to temporary disable tracing. They are used in the implementation of Tracer::Add and Tracer::Remove.

The method Add adds a new entry to the map, but only when tracing is active. Notice that it disables tracing when accessing the map--we don't want to trace the allocations inside the map code.

```
void Tracer::Add (void * p, char const * file, int line)
{
    if (_lockCount > 0)
        return;
    Tracer::Lock lock (*this);
    _map [p] = Entry (file, line);
}
```

The method Remove makes the same preparations as Add and then searches the map for the pointer to be removed. If it's found, the whole entry is erased.

```
void Tracer::Remove (void * p)
{
    if (_lockCount > 0)
        return;
    Tracer::Lock lock (*this);

    iterator it = _map.find (p);
    if (it != _map.end ())
    {
        _map.erase (it);
    }
}
```

Finally, at the end of the program, the method Dump is called from the destructor of Tracer to display all the leaks.

```
Tracer::~Tracer ()
{
    Ready = false;
    Dump ();
}

void Tracer::Dump ()
{
    if (_map.size () != 0)
    {
        std::cout << _map.size () << " memory leaks detected\n";
        for (iterator it = _map.begin (); it != _map.end (); ++it)
        {
            char const * file = it->second.File ();
            int line = it->second.Line ();
            std::cout << file << ", "  << line << std::endl;
        }
    }
}
```

Notice: if your implementation of standard library cannot deal with standard output after the termination of main (), read the next section, Debug Output.

Since we are overloading global operators new and delete, the Tracer has to be a global object too.

```
extern Tracer NewTrace;
```

Notice that this might lead to some problems, if there are other global objects that allocate memory in their constructors. The order of construction of global objects residing in different files is undefined. If a memory-allocating global object is constructed before the construction of NewTracer, we're in trouble. That's why I introduced a static Boolean flag, Tracer::Ready, which is originally set to false.

```
bool Tracer::Ready = false;
```

The constructor of Tracer sets this flag to true and Tracer::Dump sets it back to false.

```
Tracer::Tracer ()
: _lockCount (0)
{
    Ready = true;
}
```

The implementation of the overloaded new is straightforward.

```
void * operator new (size_t size, char const * file, int line)
{
    void * p = malloc (size);
    if (Tracer::Ready)
        NewTrace.Add (p, file, line);
    return p;
}
```

Notice that we use the low level memory allocating function malloc, rather than calling operator ::new. That's because we are going to overload the regular new as well.

There must be a corresponding overload of delete, to be used during exception unwinding.

```
void operator delete (void * p, char const * file, int line)
{
    if (Tracer::Ready)
        NewTrace.Remove (p);
    free (p);
}
```

Since we used malloc for memory allocation, we have to use free for deallocation.

For completeness, we also override the regular global new, in case there are parts of our code outside of the reach of macro substitution (for instance, parts of the standard library).

```
void * operator new (size_t size)
{
    void * p = malloc (size);
    if (Tracer::Ready)
        NewTrace.Add (p, "?", 0);
    return p;
}
```

Finally, we have to override the global delete in order to trace *all* deallocations.

```
void operator delete (void * p)
{
    if (Tracer::Ready)
        NewTrace.Remove (p);
    free (p);
}
```

Since we only want the tracing to be enabled in the debug version of our program, we'll enclose the definition of our macro in conditional compilation directives.

```
#if !defined NDEBUG
#include "debugnew.h"
```

```
#define new new(__FILE__, __LINE__)

#endif
```

Most compilers define the flag NDEBUG (no debug) when building the release (non-debugging) version of the program. The file debugnew.h contains, among others, the declaration of overloaded operators new and delete.

Similarly, we have to make sure that the implementation of overloaded new and delete is also compiled conditionally. That's because the decision as to which version of new and delete will be called from your program is done by the linker. If the linker doesn't find an implementation of these operators in your code, it will use the ones privided by the runtime library. Otherwise it will call your overrides throughout.

Finally, we add the definition of the global object NewTrace to main.cpp. The destructor of this object will dump memory leaks after the end of main ().

```
#if !defined NDEBUG
Tracer NewTrace;
#endif
```

Debug Output

An even better idea is to redirect the dump to the debugger. (An additional advantage of doing that is to bypass potential library bugs that prevent standard output after main ()). There is a function OutputDebugString, declared in <windows.h>, which outputs strings to the debug window, if you are running the program under the debugger. We format the string using std::stringstream.

```
if (_map.size () != 0)
{
    OutputDebugString ("*** Memory leak(s):\n");
    for (iterator it = _map.begin (); it != _map.end (); ++it)

        char const * file = it->second.File ();
        int line = it->second.Line ();
        int addr = reinterpret_cast (it->first);
        std::stringstream out;
        out << "0x" << std::hex << addr << ": "
            << file << ", line " << std::dec << line << std::endl;
        OutputDebugString (out.str ().c_str ());
    }
    OutputDebugString ("\n");
}
```

If your standard library doesn't handle even that, try bypassing integer output by using a low-level conversion routine, itoa (integer-to-ascii).

```
    char buffer1 [10];
    char buffer2 [8];
    out << "0x" << itoa (addr, buffer1, 16) << ": "
        << file << ", line " << itoa (line, buffer2, 10) << std::endl;
```

Placement new

There is one particular overload of new that is part of the standard library. It's called *placement new* (notice: sometimes all overrides of new that take extra arguments are called *placement new*) and it takes one additional argument--a void pointer. It is used whenever the memory for the object has already been allocated or reserved by other means. The argument could be a pointer to some static memory or to a chunk of pre-allocated raw dynamic memory. Placement new does not allocate memory--it uses the memory passed to it (it's your responsibility to make sure the chunk is big enough) and calls the appropriate constructor.

For instance, in our earlier example with bulk allocation, we could use placement new to create a Block object using memory that's been allocated as an array of bytes.

```
char * p = ::new char [sizeof (Block) + BlockLinks * sizeof (Link)];
Block * block = new (p) Block (_blocks);
_blocks = block;
```

It makes sense now to have a constructor of Block that initializes the pointer to next. In fact, the method SetNext is no longer needed.

```
LinkAllocator::Block::Block (Block * next) : _next (next) {}
```

The standard library defines a corresponding placement delete which does absolutely nothing, but is required in case the constructor throws an exception. Since placement new doesn't allocate any memory, it's an error to delete the object created by it. Of course, the *raw* memory that's been passed to placement new has to be dealt with appropriately. In our example, it's the Purge method that frees raw memory.

By the way, there is also an *array* placement operator new[] and the corresponding delete[]. It is left as an exercise for the user to use it for converting memory following the Block header to an array of Links (what kind of a constructor would you add to Link for that purpose?).

Windows Techniques

Hello Windows!

Controlling Windows through C++

Painting

Windows Application

Windows Techniques

Introduction

No serious programmer can ignore Windows--or, more generally, a window-based programming environment. I knew from the beginning that this book would have to include a section on Windows programming. At some point I had several chapters written, complete with a multi-threaded application that painted a three-dimensional animated polygon mesh. Then I tossed it all.

The problem is that teaching Windows programming is a huge undertaking. It calls for a separate book, maybe even a multi-volume compendium. So if I wanted to include an introduction to Windows in this book, I had to be very selective. I had to focus on a few important issues and give the foundations on which the reader could build his or her understanding of Windows programming.

From the point of view of C++ programming methodologies, Windows programming is essentially a virgin territory. There are a few books that use a limited subset of C++ to demonstrate Windows API (Application Programmmer's Interface--the name given to the set of library functions that give the programmer access to the Windows operating system). There are a few commercial libraries that develop their own dialects--again, based on a subset of C++. But the attempts at providing a comprehensive strategy of dealing with a window-based environment using modern C++ are essentially non-existent.

A few years ago I started posting little Windows tutorials and snippets of code at my company's web site. Nothing sophisticated, just a few tips at how to encapsulate some basic Windows functionality. I was amazed at the positive response I got from the programming community. Judging from the number of visits and the tenor of e-mail messages, both from beginning as well as experienced Windows programmers, I must have fulfilled a significant demand.

So here's the plan. I'll start with a small diatrybe against some existing solutions. Then I'll proceed with the introduction to the Windows programming paradigm. I'll show you how to write some simple programs to get you started programming Windows. Then I'll get to the main topic--how to best encapsulate Windows API using the object-oriented paradigm in C++. Even though it's not my ambition to write a complete Windows library, I'll try to explain a lot of the techniques that could be used in writing one.

Of Macros and Wizards

Let's start by talking about how *not* to write a library.

First of all, do not try to create an obscure dialect of C++. By a dialect I mean a system of macros that generate C++ code. For instance, would you recognize this as a fragment of a C++ program?

```
IMPLEMENT_DYNAMIC(CMyThing, CMyObject)
BEGIN_MESSAGE_MAP(CMyThing, CMyObject)
    ON_WM_PAINT()
    ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()
```

If this were the only way to write Windows programs in C++, I'd say C++ is just not the right language for the task. Instead of creating a new dialect using a C++ macro preprocessor, let's invent a new language altogether--a language that is more suitable for Windows and has a semblance of a half-decent programming language. In fact I toyed with this idea, but decided to first give C++ a chance.

Another language escape mechanism used by library writers are "wizards"--little programmer-friendly gadgets that generate C++ code. In principle, generating code using GUI (Graphical User Interface) controls is not such a bad idea. If well thought out, it is equivalent to introducing some sort of a higher-level programming language, in which programming might involve the manipulation of UI gadgets. There is some research into "visual" languages, and this might just be the thing for visually intensive Windows applications.

However, if you want to invent a new language--visual or otherwise--you better be a pretty good language designer. Throwing together a few ad hoc wizards is no substitute for a programming paradigm. It might in fact do more damage than good. One major, unspoken, requirement of any programming language is that your programming environment doesn't discard your source code.

Imagine a C++ compiler deleting your source files after producing machine code.

And this is exactly what most wizards do! Once they generate the target C++ code, your input is lost. Because *input* in this case consists of the strings you entered in edit controls, the checkboxes you checked, the buttons you pushed, etc. These actions are all part of the "visual" language the wizard implements. If they are not remembered, there is no way you (or, for that matter, anybody else) can make incremental modifications. You either have to redo the whole procedure from scratch, or abandon the visual language altogether and modify the generated code by hand. At this point the whole advantage of the wizard is lost. No matter how good the wizard is, the code it generates is not human-friendly.

So as long as we don't have two-way wizards that can reverse-engineer C++ code into its visual form and let you not only create, but also *maintain* visual code, we have little choice but to use the programming language at hand--in our case, C++.

Programming Paradigm

A Windows program, like any other interactive program, is for the most part input-driven. However, the input of a Windows program is conveniently pre-digested by the system. When the user hits the keyboard or moves the mouse, Windows intercepts such an event, pre-processes it and dispatches it to the appropriate user program. The program gets all its messages from Windows. It may do something about them, or not; in the latter case it lets Windows do the "right" thing (do the default processing).

When a Windows program starts for the first time, it registers a ***Windows class*** (it’s not a C++ class) with the system. Through this class data structure it gives the system a pointer to a callback function called the ***Window Procedure***. Windows will call this procedure whenever it wants to pass a message to the program, to notify it of interesting events. The name "callback" means just this: we don't call Windows, Windows will call us back.

The program also gets a peek at every message in the ***message loop*** before it gets dispatched to the appropriate Windows Procedure. In most cases the ***message loop*** just forwards the message back to Windows to do the dispatching.

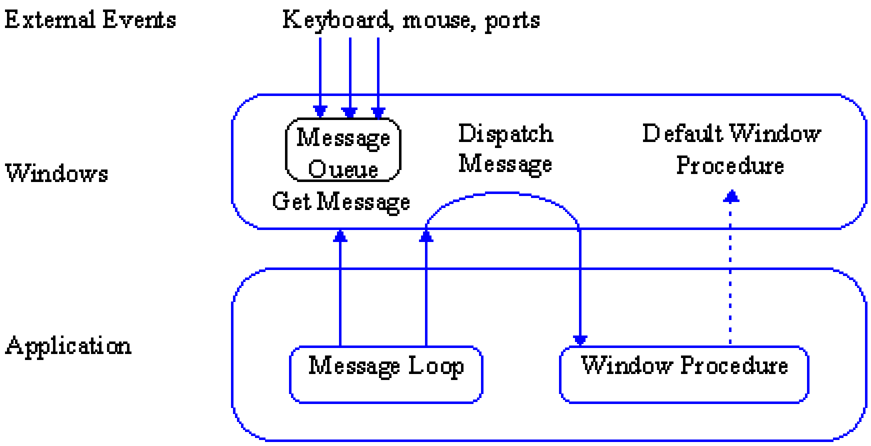


Figure. Input driven Windows paradigm

Windows is a multi-tasking operating system. There may be many programs running at the same time. So how does Windows know which program should get a particular message? Mouse messages, for instance, are usually dispatched to the application that created the window over which the mouse cursor is positioned at a given moment (unless an application "captures" the mouse).

Most Windows programs create one or more ***windows*** on the screen. At any given time, one of these windows has the ***focus*** and is considered ***active*** (its title bar is usually highlighted). Keyboard messages are sent to the window that has the focus.

Events such as resizing, maximizing, minimizing, covering or uncovering a window are handled by Windows, although the concerned program that owns the window also gets a chance to process messages for these events. There are dozens and dozens of types of messages that can be sent to a Windows program. Each program handles the ones that it's interested in.

Windows programs use Windows services to output text or graphics to the screen. Windows not only provides high level graphical interface, but it separates the program from the actual graphical hardware. In this sense Windows graphics is device independent.

It is very easy for a Windows program to use standard Windows controls. ***Menus*** are easy to create, so are message boxes. Dialog boxes are more general--they can be designed by a programmer using a ***Dialog Editor***, icons may be created using an Icon Editor. List boxes, edit controls, scroll bars, buttons, radio buttons, check boxes, etc., are all examples of built-in ready to use controls that make Windows programs so attractive and usable.

All this functionality if available to the programmer through Windows API. It is a (very large) set of C functions, typedefs, structures and macros whose declarations are included in <windows.h> and whose code is linked to your program through a set of libraries and DLLs (dynamic-load libraries).

Hello Windows!

The simplest Windows program does nothing more but to create a window with the title "Hello Windows!" It is definitely more complicated than the Kernighan and Ritchie's "Hello world!" and more complicated than our first "Hello!" C++ program. However, what we are getting here is much more than the simple old-fashioned teletype output. We are creating a window that can be moved around, resized, minimized, maximized, overlapped by other windows, etc. It also has a standard system menu in the upper left corner. So let's not complain too much!

In Windows, the main procedure is called `WinMain`. It must use the `WINAPI` calling convention and it is called by the system with the following parameters:

- `HINSTANCE hInst`--the handle to the current instance of the program
- `HINSTANCE hPrevInst`--obsolete in Win32
- `LPSTR cmdParam`--a string with the command line arguments
- `int cmdShow`--a flag that says whether to show the main window or not.

Notice the strange type names. You'll have to get used to them--Windows is full of typedefs. In fact, you will rarely see an `int` or a `char` in the description of Windows API. For now, it's enough to know that `LPSTR` is in fact a typedef for a `char *` (the abbreviation stands for Long Pointer to STRing, where string is a null terminated array and "long pointer" is a fossil left over from the times of 16-bit Windows).

In what follows, I will consequently prefix all Windows API functions with a double colon. A double colon simply means that it's a globally defined function (not a member of any class or namespace). It is somehow redundant, but makes the code more readable.

```
int WINAPI WinMain
   (HINSTANCE hInst, HINSTANCE hPrevInst, LPSTR cmdParam, int cmdShow)
{
    char className [] = "Winnie";
    WinClassMaker winClass (WinProcedure, className, hInst);
    winClass.Register ();
    WinMaker maker (className, hInst);
    Window win = maker.Create ("Hello Windows!");
    win.Display (cmdShow);

    // Message loop
    MSG  msg;
    int status;
    while ((status = ::GetMessage (& msg, 0, 0, 0)) != 0)
    {
        if (status == -1)
            return -1;
        ::DispatchMessage (& msg);
    }
    return msg.wParam;
}
```

First, we create a Window class and register it. Then we create a window with the caption "Hello Windows!" and display it (or not, depending on the flag we were given). Finally, we enter the message loop that waits for a message from Windows (the `::GetMessage` API) and then lets the system dispatch it. The message will come back to us when Windows calls our Window procedure, WinProcedure. For the time being we don’t have to worry about the details of the message loop. The program normally terminates when ::GetMessage returns zero. The wParam of the last message contains the return code of our program.

> The ::GetMessage function is an interesting example of three-state logic. It is defined to return the type BOOL, which is a typedef for int, not bool (in fact, there was no bool in C++, not to mention C, when Windows was first released). The documentation, however, specifies three types of returns, non-zero, zero and -1 (I am not making it up!). Here's the actual excerpt from the help file:
>
> - If the function retrieves a message other than WM_QUIT, the return value is **nonzero**.
> - If the function retrieves the WM_QUIT message, the return value is **zero**.
> - If there is an error, the return value is **-1**.

The prototypes of all Windows APIs and data structures are in the main inlcude file <windows.h>.

The class WinClassMaker encapsulates the WNDCLASS data structure as well as the ::RegisterClass API.

```
class WinClassMaker
{
public:
    WinClassMaker (WNDPROC WinProcedure,
                   char const * className,
                   HINSTANCE hInst);
    void Register ()
    {
        if (::RegisterClassEx (&_class) == 0)
            throw "RegisterClass failed";
    }
private:
    WNDCLASSEX _class;
};
```

In the constructor of WinClassMaker we initialize all parameters to some sensible default values. For instance, we default the mouse cursor to an arrow, the brush (used by Windows to paint our window’s background) to the default window color.

We have to provide the pointer to the Window procedure, the name of the class and the handle to the instance that owns the class.

```
WinClassMaker::WinClassMaker
    (WNDPROC WinProcedure, char const * className, HINSTANCE hInst)
{
    _class.lpfnWndProc = WinProcedure;// window procedure: mandatory
    _class.hInstance = hInst;         // owner of the class: mandatory
    _class.lpszClassName = className; // mandatory
    _class.cbSize = sizeof (WNDCLASSEX);
    _class.hCursor = ::LoadCursor (0, IDC_ARROW);
    _class.hbrBackground = reinterpret_cast<HBRUSH> (COLOR_WINDOW +
1);
    _class.style = 0;
    _class.cbClsExtra = 0;
    _class.cbWndExtra = 0;
    _class.hIcon = 0;
    _class.hIconSm = 0;
    _class.lpszMenuName = 0;
}
```

Class WinMaker initializes and stores all the parameters describing a particular window.

```
class WinMaker
{
public:
    WinMaker (char const * className, HINSTANCE hInst);
    HWND Create (char const * title);
private:
    HINSTANCE    _hInst;        // program instance
    char const  *_className;     // name of Window class
    DWORD        _style;        // window style
    DWORD        _exStyle;      // window extended style
    int          _x;             // horizontal position of window
    int          _y;             // vertical position of window
    int          _width;         // window width
    int          _height;        // window height
    HWND         _hWndParent;    // handle to parent or owner window
    HMENU        _hMenu;         // handle to menu, or child-window id
    void *       _data;          // pointer to window-creation data
};
```

The constructor of WinMaker takes the title of the window and the name of its Window class. The title will be displayed in the title bar, the class name is necessary for Windows to find the window procedure for this window. The rest of the parameters are given some reasonable default values. For instance, we let the system decide the initial position and size of our window. The style, WS_OVERLAPPEDWINDOW, is the most common style for top-level windows. It includes a title bar with a system menu on the left and the minimize, maximize and close buttons on the right. It also provides for a "thick" border that can be dragged with the mouse in order to resize the window.

```
WinMaker::WinMaker (char const * className, HINSTANCE hInst)
  : _style (WS_OVERLAPPEDWINDOW),
    _exStyle (0),
    _className (className),
    _x (CW_USEDEFAULT), // horizontal position of window
    _y (0),             // vertical position of window
    _width (CW_USEDEFAULT), // window width
    _height (0),        // window height
    _hWndParent (0),    // handle to parent or owner window
    _hMenu (0),         // handle to menu, or child-window identifier
    _data (0),          // pointer to window-creation data
    _hInst (hInst)
{}
```

All these paramters are passed to the ::CreateWindowEx API that creates the window (but doesn't display it yet).

```
HWND WinMaker::Create (char const * title)
{
    HWND hwnd = ::CreateWindowEx (
        _exStyle,
        _className,
        title,
        _style,
        _x,
        _y,
        _width,
        _height,
        _hWndParent,
        _hMenu,
        _hInst,
        _data);

    if (hwnd == 0)
        throw "Window Creation Failed";
    return hwnd;
}
```

`Create` returns a handle to the successfully created window. We will conveniently encapsulate this handle in a class called `Window`. Other than storing a handle to a particular window, this class will provide interface to a multitude of Windows APIs that operate on that window.

```
class Window
{
public:
    Window (HWND h = 0) : _h (h) {}
    void Display (int cmdShow)
    {
        assert (_h != 0);
        ::ShowWindow (_h, cmdShow);
        ::UpdateWindow (_h);
    }
private:
    HWND _h;
};
```

To make the window visible, we have to call `::ShowWindow` with the appropriate parameter, which specifies whether the window should be initially minimized, maximized or regular-size. `::UpdateWindow` causes the contents of the window to be refreshed.

The window procedure must have the following signature, which is hard-coded into Windows:

```
LRESULT CALLBACK WinProcedure
    (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam);
```

Notice the calling convention and the types of parameters and the return value. These are all typedefs defined in windows.h. `CALLBACK` is a predefined language-independent calling convention (what order the parameters are pushed on the stack, etc.). `LRESULT` is a type of return value. `HWND` is a handle to a window, `UINT` is an unsigned integer that identifies the message, `WPARAM` and LPARAM are the types of the two parameters that are passed with every message.

WinProcedure is called by Windows every time it wants to pass a message to our program. The window handle identifies the window that is supposed to respond to this message. Remember that he same Window procedure may service several instances of the same Windows class. Each instance will have its own window with a different handle, but they will all go through the same procedure.

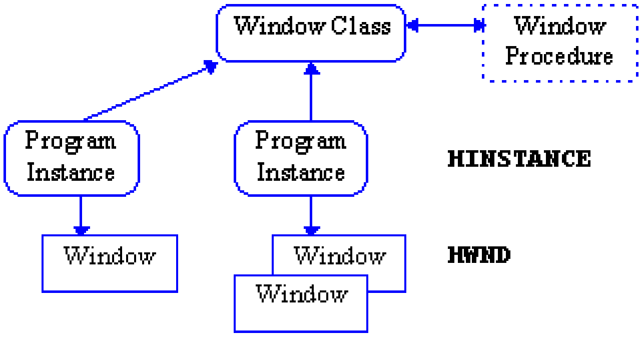


Figure. The relationship between instances of the same program, their windows and the program's Window class and procedure.

The message is just a number. Symbolic names for these numbers are defined in windows.h. For instance, the message that tells the window to repaint itself is defined as

```
#define WM_PAINT         0x000F
```

Every message is accompanied by two parameters whose meaning depends on the kind of message. (In the case of WM_PAINT the parameters are meaningless.)

To learn more about window procedure study the help files that come with your compiler.

Here's our minimalist implementation of Window procedure.

```
LRESULT CALLBACK WinProcedure
    (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_DESTROY:
            ::PostQuitMessage (0);
            return 0;
    }
    return ::DefWindowProc (hwnd, message, wParam, lParam );
}
```

It doesn't do much in this particular program. It only handles one message, WM_DESTROY, that is sent to the window when it is being destroyed. At that point the window has already been closed--all we have to do is to terminate WinMain. We do it by posting the final *quit* message. We also pass the return code through it--zero in our case. This message will terminate the message loop and control will be returned to WinMain.

Encapsulation

A lot of Windows APIs take a large number of arguments. For instance, when you create a window, you have to be able to specify its position, size, style, title, and so on... That's why CreateWindowEx takes 12 arguments. RegisterClassEx also requires 12 parameters, but they are combined into a single structure, WNDCLASSEX. As you can see, there is no consistency in the design of Windows API.

Our approach to encapsulating this type of APIs is to create a C++ class that combines all the arguments in one place. Since most of these arguments have sensible defaults, the constructor should initialize them appropriately. These parameters that don't have natural defaults are passed as arguments to the constructor. The idea being that, once an object of such a class is constructed, it should be usable without further modification. However, if modifications are desired, they should be done by calling appropriate methods.

For instance, if we are to develop the class WinMaker into a more useful form, we should add methods such as SetPosition and SetSize that override the default settings for _x, _y, _width and _height.

Let's analyze the two classes, WinClassMaker and WinMaker in this context. WinClassMaker encapsulates the API RegisterClassEx. The argument to this API is a structure which we can embed directly into our class. Three of the ten parameters--window procedure, class name and program instance--cannot be defaulted, so they are passed in the constructor. The window background color is normally defaulted to whatever the current system setting is--that's the COLOR_WINDOW constant. The mouse cursor in most cases defaults to whatever the system considers an arrow--that's the IDC_ARROW constant. The size of the structure must be set to sizeof (WNDCLASSEX). The rest of the parameters can be safely set to zero. We don't have to do anything else before calling WinClassMaker::Register. Of course, in a more sophisticated program, we might want to modify some of these settings, and we would most certainly add methods to do that. We'll talk about it later.

In a similar way, the API CreateWindowEx is encapsulated in the class WinMaker. The non-defaultable parameters are the class name, the program instance and the title of the window. This time, however, we might want to call WinMaker::Create multiple times in order to create more than one window. Most likely these windows would have different titles, so we pass the title as an argumet to Create.

To summarize, in the main procedure, your program creates a window of a particular class and enters the message processing loop. In this loop, the program waits idly for a message from Windows. Once the message arrives, it dispatches is back to Windows. The system then calls your Window procedure with the same message. You either process it yourself, or use the default processing. After returning from Window procedure, control goes back to the message loop and the whole process repeats itself. Eventually a "quit" message is posted and the loop ends.

Controlling Windows through C++

Creating a Namespace

Before we start developing our Windows library any further, we should make an important naming decision. As you might have noticed, I employed a convention to use the prefix `Win` for all classes related to Windows. Such naming conventions used to make sense in the old days. Now we have better mechanisms, not only to introduce, but also to enforce naming conventions. I'm talking about namespaces.

It's easy to enclose all definitions of Windows-related classes, templates and functions in one namespace which we will conveniently call `Win`. The net result, for the user of our library, will be to have to type `Win::Maker` instead of `WinMaker`, `Win::Dow` instead of `Window`, etc. It will also free the prefix `Win` for the use in user-defined names (e.g., the client of our library will still be able to define his or her own class called `WinMaker`, without the fear of colliding with library names).

Also, in preparation for further development, let's split our code into multiple separate files. The two classes `Win::ClassMaker` and `Win::Maker` will each get a pair of header/implementation files. The `Win::Procedure` function will also get its own pair, since the plan is to make it part of our library.

Model-View-Controller

The biggest challenge in Windows programming is to hide the ugliness of the big switch statement that forms the core of a window procedure. We'll approach this problem gradually.

> You have to understand that the basic structure of Windows API was established in pre-historic times, when computers were many orders of magnitude slower than they are now, and when using C instead of assembly was a risk not many PC companies were willing to take. C++ didn't even exist and strong typing was a fringe idea. A switch statement was the fastest way to dispatch a message to the appropriate chunk of code crammed under the corresponding `case:` label. It didn't matter that programs were unreliable and buggy, as long as they were reasonably fast. The feedback loop between user input and screen output had to be as tight as possible, otherwise the computer would seem sluggish and unresponsive.
>
> Things are different now. Processors are on the verge of breaking the 1GHz barrier--a billion clock ticks per second. It means that a computer may easily execute hundreds of thousands of instructions in the time between the user pushes the mouse and the cursor moves on the screen. The mere act of processing user input--deciding what to do with it--is no longer a bottleneck.

The first step is to abstract the user interface of the program from the engine that does the actual work. Traditionally, the UI-independent part is called the *model*. The UI part is split into two main components, the *view* and the *controller*.

The view's responsibility is to display the data to the user. This is the part of the program that draws pictures or displays text in the program's window(s). It obtains the data to be displayed by querying the model.

The controller's responsibility is to accept and interpret user input. When the user types in text, clicks the mouse button or selects a menu item, the controller is the first to be informed about it. It converts the raw input into something intelligible to the model. After notifying the model it also calls the view to update the display (in a more sophisticated implementation, the model might selectively notify the view about changes).

The model-view-controller paradigm is very powerful and we'll use it in our encapsulation of Windows. The problem is, how do we notify the appropriate controller when a window procedure is notified of user input? The first temptation is to make the Controller object global, so that a window procedure, which is a global function, can access it. Remember however that there may be several windows and several window procedures in one program. Some window procedures may be shared by multiple windows. What we need is a mapping between window handles and controllers. Each window message comes with a window handle, HWND, which uniquely identifies the window for which it was destined.

The window-to-controller mapping can be done in many ways. For instance, one can have a global map object that does the translation. There is, however, a much simpler way--we can let Windows store the pointer to a controller in its internal data structures. Windows keeps a separate data structure for each window. Whenever we create a new window, we can create a new Controller object and let Windows store the pointer to it. Every time our window procedure gets a message, we can retrieve this pointer using the window handle passed along with the message.

The APIs to set and retrieve an item stored in the internal Windows data structure are SetWindowLong and GetWindowLong. You have to specify the window whose internals you want to access, by passing a window handle. You also have to specify *which* long you want to access--there are several pre-defined longs, as well as some that you can add to a window when you create it. To store the pointer to a controller, we'll use the long called GWL_USERDATA. Every window has this long, even a button or a scroll bar (which, by the way, are also windows). Moreover, as the name suggests, it can be used by the user for whatever purposes.

We'll be taking advantage of the fact that a pointer has the same size as a long--will this be true in 64-bit Windows, I don't know, but I strongly suspect.

There is a minor problem with the Get/SetWindowLong API: it is typeless. It accepts or returns a long, which is not exactly what we want. We'd like to make it type-safe. To this end, let's encapsulate both functions in templates, parametrized by the type of the stored data.

```
namespace Win
{
    template <class T>
    inline T GetLong (HWND hwnd, int which = GWL_USERDATA)
    {
        return reinterpret_cast<T> (::GetWindowLong (hwnd, which));
    }

    template <class T>
    inline void SetLong (HWND hwnd, T value, int which = GWL_USERDATA)
    {
        ::SetWindowLong (hwnd, which, reinterpret_cast<long> (value));
    }
}
```

In fact, if your compiler supports member templates, you can make GetLong and SetLong methods of Win::Dow.

```
namespace Win
{
    class Dow
    {
    public:
        Dow (HWND h = 0) : _h (h) {}
        template <class T>
        inline T GetLong (int which = GWL_USERDATA)
        {
            return reinterpret_cast<T> (::GetWindowLong (_h, which));
        }
        template <class T>
        inline void SetLong (T value, int which = GWL_USERDATA)
        {
            ::SetWindowLong (_h, which, reinterpret_cast<long>
(value));
        }

        void Display (int cmdShow)
        {
            assert (_h != 0);
            ::ShowWindow (_h, cmdShow);
            ::UpdateWindow (_h);
        }
    private:
        HWND _h;
    };
}
```

Notice the use of default value for the which argument. If the caller calls any of these functions without the last argument, it will be defaulted to GWL_USERDATA.

Are default arguments already explained???

We are now ready to create a stub implementation of the window procedure.

```
LRESULT CALLBACK Win::Procedure (HWND hwnd,
                                 UINT message,
                                 WPARAM wParam,
                                 LPARAM lParam)
{
    Controller * pCtrl = Win::GetLong<Controller *> (hwnd);
    switch (message)
    {
    case WM_NCCREATE:
        {
            CreateData const * create =
                reinterpret_cast<CreateData const *> (lParam);
            pCtrl = static_cast<Controller *> (create->GetCreationData
());
            pCtrl->SetWindowHandle (hwnd);
            Win::SetLong<Controller *> (hwnd, pCtrl);
        }
        break;
    case WM_DESTROY:
        // We're no longer on screen
        pCtrl->OnDestroy ();
        return 0;
    case WM_MOUSEMOVE:
        {
            POINTS p = MAKEPOINTS (lParam);
            KeyState kState (wParam);
            if (pCtrl->OnMouseMove (p.x, p.y, kState))
                return 0;
        }
    }
    return ::DefWindowProc (hwnd, message, wParam, lParam);
}
```

We initialize the GWL_USERDATA slot corresponding to hwnd in one of the first messages sent to our window. The message is WM_NCCREATE (Non-Client Create), sent before the creation of the non-client part of the window (the border, the title bar, the system menu, etc.). (There is another message before that one, WM_GETMINMAXINFO, which might require special handling.) We pass the pointer to the controller as window creation data. We use the class Win::CreateData, a thin encapsulation of Windows structure CREATESTRUCT. Since we want to be able to cast a pointer to CREATESTRUCT passed to us by Windows to a pointer to Win::CreateData, we use inheritance rather than embedding (you can inherit from a struct, not only from a class).

```
namespace Win
{
    class CreateData: public CREATESTRUCT
    {
    public:
        void * GetCreationData () const { return lpCreateParams; }
        int GetHeight () const { return cy; }

        int GetX () const { return x; }
        int GetY () const { return y; }
        char const * GetWndName () const { return lpszName; }
    };
}
```

The message WM_DESTROY is important for the top-level window. That's where the "quit" message is usually posted. There are other messages that might be sent to a window after WM_DESTROY, most notably WM_NCDESTROY, but we'll ignore them for now.

I also added the processing of WM_MOUSEMOVE, just to illustrate the idea of message handlers. This message is sent to a window whenever a mouse moves over it. In the generic window procedure we will always unpack message parameters and pass them to the appropriate handler.

There are three parameters associated with WM_MOUSEMOVE, the x coordinate, the y coordinate and the state of control keys and buttons. Two of these parameters, x and y, are packed into one LPARAM and Windows conveniently provides a macro to unpack them, MAKEPOINTS, which turns lParam into a structure called POINTS. We retrieve the values of x and y from POINTS and pass them to the handler.

The state of control keys and buttons is passed inside WPARAM as a set of bits. Access to these bits is given through special bitmasks, like MK_CONTROL, MK_SHIFT, etc., provided by Windows. We will encapsulate these bitwise operations inside a class, Win::KeyState.

```
class KeyState
{
public:
    KeyState (WPARAM wParam): _data (wParam)
    {}
    bool IsCtrl () const { return (_data & MK_CONTROL) != 0; }
    bool IsShift () const { return (_data & MK_SHIFT) != 0; }
    bool IsLButton () const { return (_data & MK_LBUTTON) != 0; }
    bool IsMButton () const { return (_data & MK_MBUTTON) != 0; }
    bool IsRButton () const { return (_data & MK_RBUTTON) != 0; }
private:
    WPARAM    _data;
};
```

The methods of Win::KeyState return the state of the control and shift keys and the state of the left, middle and right mouse buttons. For instance, if you move the mouse while you press the left button and the shift key, both IsShift and IsLButton will return true.

In WinMain, where the window is created, we initialize our controller and pass it to Win::Maker::Create along with the window's title.

```
        TopController ctrl;
        win.Create (ctrl, "Simpleton");
```

This is the modified Create. It passes the pointer to Controller as the user-defined part of window creation data--the last argument to CreateWindowEx.

```
HWND Maker::Create (Controller & controller, char const * title)
{
    HWND hwnd = ::CreateWindowEx (
        _exStyle,
        _className,
        title,
        _style,
        _x,
        _y,
        _width,
        _height,
        _hWndParent,
        _hMenu,
        _hInst,
        &controller);

    if (hwnd == 0)
        throw "Internal error: Window Creation Failed.";
    return hwnd;
}
```

To summarize, the controller is created by the client and passed to the Create method of Win::Maker. There, it is added to the creation data, and

Windows passes it as a parameter to WM_NCREATE message. The window procedure unpacks it and stores it under GWL_USERDATA in the window's internal data structure. During the processing of each subsequent message, the window procedure retrieves the controller from this data structure and calls its appropriate method to handle the message. Finally, in response to WM_DESTROY, the window procedure calls the controller one last time and unplugs it from the window.

Now that the mechanics of passing the controller around are figured out, let's talk about the implementation of Controller. Our goal is to concentrate the logic of a window in this one class. We want to have a generic window procedure that takes care of the ugly stuff--the big switch statement, the unpacking and re-packing of message parameters and the forwarding of the messages to the default window procedure. Once the message is routed through the switch statement, the appropriate Controller method is called with the correct (strongly-typed) arguments.

For now, we'll just create a stub of a controller. Eventually we'll be adding a lot of methods to it--as many as there are different Windows messages.

The controller stores the handle to the window it services. This handle is initialized inside the window procedure during the processing of WM_NCCREATE. That's why we made Win::Procedure a friend of Win::Controller. The handle itself is protected, not private--derived classes will need access to it. There are only two message-handler methods at this point, OnDestroy and OnMouseMove.

```
namespace Win
{
    class Controller
    {
        friend LRESULT CALLBACK Procedure (HWND hwnd,
                        UINT message, WPARAM wParam, LPARAM lParam);

        void SetWindowHandle (HWND hwnd) { _h = hwnd; }
    public:
        virtual ~Controller () {}
        virtual bool OnDestroy ()
            { return false; }
        virtual bool OnMouseMove (int x, int y, KeyState kState)
            { return false; }
    protected:
        HWND  _h;
    };
}
```

You should keep in mind that Win::Controller will be a part of the library to be used as a base class for all user-defined controllers. That's why all message handlers are declared virtual and, by default, they return false. The meaning of this Boolean is, "I handled the message, so there is no need to call DefWindowProc." Since our default implementation doesn't handle any messages, it always returns false.

The user is supposed to define his or her own controller that inherits from Win::Controller and overrides some of the message handlers. In this case, the only message handler that has to be overridden is OnDestroy--it must close the application by sending the "quit" message. It returns true, so that the default window procedure is not called afterwards.

```
class TopController: public Win::Controller
{
public:
    bool OnDestroy ()
    {
        ::PostQuitMessage (0);
        return true;
    }
};
```

To summarize, our library is designed in such a way that its client has to do minimal work and is protected from making trivial mistakes. For each class of windows, the client has to create a customized controller class that inherits from our library class, `Win::Controller`. He implements (overrides) only those methods that require non-default implementation. Since he has the prototypes of all these methods, there is no danger of misinterpreting message parameters. This part--the interpretation and unpacking--is done in our Win::Procedure. It is written once and for all, and is thoroughly tested.

This is the part of the program that is written by the client of our library. In fact, we will simplify it even more later.

Is it explained that the result of assignment can be used in an expression?

```
#include "Class.h"
#include "Maker.h"
#include "Procedure.h"
#include "Controller.h"

class TopController: public Win::Controller
{
public:
    bool OnDestroy ()
    {
        ::PostQuitMessage (0);
        return true;
    }
};

int WINAPI WinMain
   (HINSTANCE hInst, HINSTANCE hPrevInst, LPSTR cmdParam, int cmdShow)
{
    char className [] = "Simpleton";
    Win::ClassMaker winClass (className, hInst);
    winClass.Register ();
    Win::Maker maker (className, hInst);
    TopController ctrl;
    Win::Dow win = maker.Create (ctrl, "Simpleton");
    win.Display (cmdShow);

    MSG  msg;
    int status;
    while ((status = ::GetMessage (& msg, 0, 0, 0)) != 0)
    {
        if (status == -1)
            return -1;
        ::DispatchMessage (& msg);
    }
    return msg.wParam;
}
```

Notice that we no longer have to pass window procedure to class maker. Class maker can use our generic Win::Procedure implemented in terms of the interface provided by our generic Win::Controller. What will really distinguish the behavior of one window from that of another is the implementation of a controller passed to Win::Maker::Create.

The cost of this simplicity is mostly in code size and in some minimal speed deterioration.

Let's start with speed. Each message now has to go through parameter unpacking and a virtual method call--even if it's not processed by the application. Is this a big deal? I don't think so. An average window doesn't get many messages per second. In fact, some messages are queued in such a way that if the window doesn't process them, they are overwritten by new messages. This is for instance the case with mouse-move messages. No matter how fast you move the mouse over the window, your window procedure will not choke on these messages. And if a few of them are dropped, it shouldn't matter, as long as the last one ends up in the queue. Anyway, the frequency with which a mouse sends messages when it slides across the pad is quite arbitrary. With the current processor speeds, the processing of window messages takes a marginally small amount of time.

Program size could be a consideration, except that modern computers have so much memory that a megabyte here and there doesn't really matter. A full blown Win::Controller will have as many virtual methods as there are window messages. How many is it? About 200. The full vtable will be 800 bytes. That's less than a kilobyte! For comparison, a single icon is 2kB. You can have a dozen of controllers in your program and the total size of their vtables won't even reach 10kB.

There is also the code for the default implementation of each method of Win::Controller. Its size depends on how aggressively your compiler optimizes it, but it adds up to at most a few kB.

Now, the worst case, a program with a dozen types of windows, is usually already pretty complex--read, large!--plus it will probably include many icons and bitmaps. Seen from this perspective, the price we have to pay for simplicity and convenience is minimal.

Exception Specification

What would happen if a Controller method threw an exception? It would pass right through our Win::Procedure, then through several layers of Windows code to finally emerge through the message loop. We could, in principle catch it in WinMain. At that point, however, the best we could do is to display a polite error message and quit. Not only that, it's not entirely clear how Windows would react to an exception rushing through its code. It might, for instance, fail to deallocate some resources or even get into some unstable state. The bottom line is that Windows doesn't expect an exception to be thrown from a window procedure.

We have two choices, either we put a try/catch block around the switch statement in Win::Procedure or we promise not to throw any exceptions from Controller's methods. A try/catch block would add time to the processing of every single message, whether it's overridden by the client or not. Besides, we would again face the problem, what to do with such an exception. Terminate the program? That seems pretty harsh! On the other hand, the contract not to throw exceptions is impossible to enforce. Or is it?!

Enter exception specifications. It is possible to declare what kind of exceptions can be thrown by a function or method. In particular, we can specify that no exceptions can be thrown by a certain method. The declaration:

```
virtual bool OnDestroy () throw ();
```

promises that `OnDestroy` (and all its overrides in derived classes) will not throw any exceptions. The general syntax is to list the types of exceptions that can be thrown by a procedure, like this:

```
void Foo () throw (bad_alloc, char *);
```

How strong is this contract? Unfortunately, the standard doesn't promise much. The compiler is only obliged to detect exception specification mismatches between base class methods and derived class overrides. In particular, the specification can be only made stronger (fewer exceptions allowed). There is no stipulation that the compiler should detect even the most blatant violations of this promise, for instance an explicit `throw` inside a method defined as `throw()` (throw nothing). The hope, however, is that compiler writers will give in to the demands of programmers and at least make the compiler issue a warning when an exception specification is violated. Just as it is possible for the compiler to report violations of `const`-ness, so it should be possible to track down violations of exception specifications.

For the time being, all that an exception specification accomplishes in a standard-compliant compiler is to guarantee that all unspecified exceptions will get converted to a call to the library function `unexpected ()`, which by default terminates the program. That's good enough, for now. Declaring all methods of `Win::Controller` as "throw nothing" will at least force the client who overrides them to think twice before allowing any exception to be thrown.

Cleanup

It's time to separate library files from application files. For the time being, we'll create a subdirectory "lib" and copy all the library files into it. However, when the compiler compiles files in the main directory, it doesn't know where to find library includes, unless we tell it. All compilers accept additional include paths. We'll just have to add "lib" to the list of additional include paths. As part of the cleanup, we'll also move the definition of `TopController` to a separate file, *control.h*.

Painting

Application Icon

Every Windows program must have an icon. When you browse into the directory where the executable is stored, Windows browser will display this program's icon. When the program is running, this icon shows up in the taskbar and in the upper-left corner of the program's window. If you don't provide your program with an icon, Windows will provide a default.

The obvious place to specify an icon for your application is in the window class of the top-level window. Actually, it's best to provide two icons at once, the large one and the small one, otherwise Windows will try to stretch or shrink the one icon you give it, often with un-esthetic results.

Let's add a SetIcons method to Win::ClassMaker and embed two icon objects in it.

```
class ClassMaker
{
public:
    ...
    void SetIcons (int id);
protected:
    WNDCLASSEX    _class;
    StdIcon       _stdIcon;
    SmallIcon     _smallIcon;
};
```

We'll get to the implementation of StdIcon and SmalIcon soon. First, let's look at the implementation of SetIcons. The images of icons are loaded from program resources.

```
void ClassMaker::SetIcons (int id)
{
    _stdIcon.Load (_class.hInstance, id);
    _smallIcon.Load (_class.hInstance, id);
    _class.hIcon = _stdIcon;
    _class.hIconSm = _smallIcon;
}
```

Program resources are icons, bitmaps, strings, mouse cursors, dialog templates, etc., that you can tack on to your executable. Your program, instead of having to search the disk for files containing such resources, simply loads them from its own executable. How do you identify resources when you want to load them? You can either give them names or integer ids. For simplicity (and efficiency), we will use ids. The set of your program's resources is identified by the instance handle that is passed to WinMain.

Lets start with the base class, Win::Icon. When you load an icon, you have to specify the resources where it can be found, the unique id of the particular icon, its dimensions in pixels (if the actual icon has different dimensions, Windows will stretch or shrink it) and some flags.

```
class Icon
{
public:
    Icon (HINSTANCE res,
          int id,
          int dx = 0,
          int dy = 0,
          unsigned flag = LR_DEFAULTCOLOR)
    {
        Load (res, id, dx, dy, flag);
    }
    ~Icon ();
    operator HICON () const { return _h; }
protected:
    Icon () : _h (0) {}
    void Load (HINSTANCE res,
            int id,
            int dx = 0,
            int dy = 0,
            unsigned flag = LR_DEFAULTCOLOR);
protected:
    HICON _h;
};
```

The API to load an icon is called LoadImage and can be also used to load other types of images. It's return type is ambiguous, so it has to be cast to HICON. Once the icon is no longer used, DestroyIcon is called.

```
void Icon::Load (HINSTANCE res, int id, int dx, int dy, unsigned flag)
{
    _h = reinterpret_cast<HICON> (
        ::LoadImage (res,
                     MAKEINTRESOURCE (id),
                     IMAGE_ICON,
                     dx, dy,
                     flag));
    if (_h == 0)
        throw "Icon load image failed";
}

Icon::~Icon ()
{

}
```

Notice that we can't pass the icon id directly to the API. We have to use a macro MAKEINTRESOURCE which does some cheating behind the scenes.

You see, LoadImage and several other APIs have to guess whether you are passing them a string or an id. Since these are C functions, they can't be overloaded. Instead, you have to trick them into accepting both types and then let them guess their real identity. MAKEINTRESOURCE mucks with the bits of the integer to make it look different than a pointer to char. (This is the kind of programming that was popular when Windows API was first designed.)

We can immediately subclass Icon to SmallIcon and StdIcon. Their constructors and Load methods are simpler--they don't require dimensions or flags.

```
class SmallIcon: public Icon
{
public:
    SmallIcon () {}
    SmallIcon (HINSTANCE res, int id);
    void Load (HINSTANCE res, int id);
};

class StdIcon: public Icon
{
public:
    StdIcon () {}
    StdIcon (HINSTANCE res, int id);
    void Load (HINSTANCE res, int id);
};
```

The Load methods are implemented using the parent class' Icon::Load method (you have to use the parent's class name followed by double colon to disambiguate--without it the compiler would understand it as a recursive call and the program would go into an infinite loop.

To find out what the correct sizes for small and standard icons are, we use the universal API, GetSystemMetrics that knows a lot about current system's defaults.

```
void SmallIcon::Load (HINSTANCE res, int id)
{
    Icon::Load (res, id,
            ::GetSystemMetrics (SM_CXSMICON),
            ::GetSystemMetrics (SM_CYSMICON));


void StdIcon::Load (HINSTANCE res, int id)
{
    Icon::Load (res, id,
            ::GetSystemMetrics (SM_CXICON),
            ::GetSystemMetrics (SM_CYICON));
}
```

There's one more thing: how does one create icons? There is a hard way and an easy way. The hard way is to have some kind of separate icon editor, write your own resource script that names the icon files and, using a special tool, compile it and link with the executable.

Just to give you an idea of what's involved, here are some details. Your resource script file, let's call it script.rc, should contain these two lines:

```
#include "resource.h"
IDI_MAIN   ICON   "main.ico"
```

IDI_MAIN is a constant defined in resource.h. The keyword ICON means that it corresponds to an icon. What follows is the name of the icon file, main.ico.

The header file, resource.h, contains the definitions of constants, for instance:

```
#define IDI_MAIN   101
```

Unfortunately, you can't use the sefer, C++ version of it,

```
 const int IDI_MAIN = 101;
```

A macro substitution results in exactly the same code as const int definition. The only difference is that, as is usual with macros, you forgo type checking.

The script file has to be compiled using a program called rc.exe (resource compiler) to produce a file script.res. The linker will then link such file with the rest of the object files into one executable.

Or, if you have an integrated development environment with a resource editor, you can create an icon in it, add it to your resources under an appropriate symbolic id, and let the environment do the work for you. (A graphical resource editor becomes really indispensable when it comes to designing dialogs.)

Notice that I'm using the same id for both icons. It's possible, because you can have two (or more) images of different size in the same icon. When you call LoadImage, the one with the closest dimensions is picked. Normally, you'd create at least a 32x32 and a 16x16 icon.

I have created a set of two icons and gave them an integer id IDI_MAIN (defined in resource.h). All I need now is to make one additional call in WinMain.

```
    Win::ClassMaker winClass (className, hInst);
    winClass.SetIcons (IDI_MAIN);
    winClass.Register ();
```

Finally, you might be wondering: if you add many icons to your program resources, which one is used by the system as the icon for the whole executable? The answer is, the one with the lowest numerical id.

Window Painting and the View Object

Just like with any other action in Windows, window painting is done in response to some external actions. For instance, your program may paint someting whenever a user moves a mouse or clicks a mouse button, it may draw characters in response to key presses, and so on. The part of the window that you normally paint is called the client area--it doesn't include the window borders, the title bar, the menu, etc.

But there is one more situation when Windows may ask your program to redraw a part or the whole client area of your window. That happens because Windows is lazy (or short of resources). Whenever another application (or sometimes your own menu) overlaps your program's window, the system simply throws away the part of the image that's occluded. When your window is finally uncovered, somebody has to redraw the discarded part. Guess who! Your program! The same thing happens when a window is minimized and then maximized again. Or when the user resizes the window.

Since, from the point of view of your application, these actions happen more or less randomly, you have to be prepared, at any time, to paint the whole client area from scratch. There is a special message, WM_PAINT, that Windows sends to you when it needs your assistance in repainting the window. This message is also sent the first time the window is displayed.

To illustrate painting, we'll extend our Windows program to trace mouse movements. Whenever the mouse moves, we'll draw a line connecting the new cursor position with the previous one. Buty before we do that, we'll want to add the second object from the triad Model-View-Controller to our program. The

View will take care of all painting operations. It will also store the last recorded position of the mouse.

```
class TopController: public Win::Controller
{
    ...
private:
    View _view;
};
```

The Canvas

All display operations are done in the context of a particular device, be it the screen, a printer, a plotter or something else. In the case of drawing to a window, we have to obtain a *device context* (DC) for this window's client area. Windows can internally create a DC for us and give us a handle to it. We use this handle for all window output operations. When done with the output, we must release the handle.

A DC is a resource and the best way to deal with it is to apply Resource Management methods to it. We'll call the generic owner of a DC, Canvas. We will have many different types of Canvas, depending on how the device context is created and disposed of. They will all, however, share the same functionality. For instance, we can call any Canvas object to draw a line or print some text. Let's make these two operations the starting point of our implementation.

```
namespace Win
{
    class Canvas
    {
    public:
        operator HDC ()
        {
            return _hdc;
        }
        void Line (int x1, int y1, int x2, int y2)
        {
            ::MoveToEx (_hdc, x1, y1, 0);
            ::LineTo (_hdc, x2, y2);
        }

        {
            ::TextOut (_hdc, x, y, buf, count);
        }
    protected:
        Canvas (HDC hdc) :_hdc (hdc) {}

        HDC  _hdc;
    };
}
```

`HDC` is Windows data structure, a handle to a device context.

Our generic class, `Canvas`, doesn't provide any public way to initialize this handle--this responsibility is left to derived classes. The member `operator HDC ()` provides implicit conversion from `Canvas` to `HDC`. It comes in handy when passing a `Canvas` object to an API that requires `HDC`.

In order to draw a line from one point to another, we have to make two API calls. The first one, `MoveToEx`, sets the "current position." The second, `LineTo`, draws a line from current position to the point specified as its argument (it also moves the current position to that point). Point positions are specified by two coordinates, x and y. In the default coordinate system, both are in units of screen pixels. The origin, corresponding to x = 0 and y = 0, is in the upper left corner of the client area of the window. The x coordinate increases from left to right, the y coordinate grows from top to bottom.

To print text, you have to specify where in the window you want it to appear. The x, y coordinates passed to `TextOut` tell Windows where to position the upper left corner of the string. This is different than printing to standard output, where the only control over placement was by means of newline characters. For a Windows device context, newlines have no meaning (they are blocked out like all other non-printable characters). In fact, the string-terminating null character is also meaningless to Windows. The string to be printed using `TextOut` doesn't have to be null-terminated. Instead, you are supposed to specify the count of characters you want printed.

So how and where should we obtain the device context? Since we want to do the drawing in response to every mouse move, we have to do it in the handler of the `WM_MOUSEMOVE` message. That means our Controller has to override the `OnMouseMove` virtual method of `Win::Controller`.

The type of `Canvas` that gets the DC from Windows *outside* of the processing of `WM_PAINT`, will be called `UpdateCanvas`. The pair of APIs to get and release a DC is `GetDC` and `ReleaseDC`, respectively.

```
class UpdateCanvas: public Canvas
{
public:
    UpdateCanvas (HWND hwnd)
        :   Canvas (::GetDC(hwnd)),
        _hwnd(hwnd)
    {}
    ~UpdateCanvas ()
    {
        ::ReleaseDC (_hwnd, _hdc);
    }
protected:
    HWND _hwnd;
};
```

We create the `Canvas` is in the appropriate `Controller` method--in this case `OnMouseMove`. This way the methods of `View` will work independent of the type of `Canvas` passed to them.

```
bool TopController::OnMouseMove
        (int x, int y, Win::KeyState kState) throw ()
{
    Win::UpdateCanvas canvas (_h);
    _view.MoveTo (canvas, x, y);
    return true;
}
```

We are now ready to implement the `View` object.

```
class View
{
public:
    View () : _x (0), _y (0) {}
    void MoveTo (Win::Canvas & canvas, int x, int y)
    {
        canvas.Line (_x, _y, x, y);
        _x = x;
        _y = y;
        PrintPos (canvas);
    }
private:
    void PrintPos (Win::Canvas & canvas)
    {
        std::string str ("Mouse at: ");
        str += ToString (_x);
        str += ", ";
        str += ToString (_y);
        canvas.Text (0, 0, &str [0], str.length ());
    }
private:
    int _x, _y;
};
```

The PrintPos method is interesting. The purpose of this method is to print "Mouse at:" followed by the x and y coordinates of the mouse position. We want the string to appear in the upper left corner of the client area, at coordinates (0, 0).

First, we have to format the string. In particular, we have to convert two numbers to their string representations. The formatting of numbers for printing is built into standard streams so we'll just use the capabilities of a string-based stream. In fact, any type that is accepted by a stream can be converted to a string using this simple template function:

```
#include <sstream>

template<class T>
inline std::string ToString (T & val)
{
    std::stringstream out;
    out << val;
    return out.str ();
}
```

Afterwards, we use operator += to concatenate the various strings. Finally, converting a string to a pointer-to-character, as required by the API, is done by taking the address of the first character in the string.

You can now run this simple program, move the mouse cursor over the client area and see for yourself that it indeed leaves a trail and that the mouse coordinates are printed in the upper left corner. You will also discover several shortcomings. For instance, try minimizing the window. After you maximize it again, all the previous traces disappear and so does the mouse-position indicator. Also, if you cover part of the window with some other application window, and then uncover it again, the restored area will be empty--mouse traces will be erased. The same with resizing!

Other minor annoyances are related to the fact that, when the cursor leaves the window it's position is not updated, and when it enters the window again, a spurious line is drawn from the last remembered position to the new entry point.

To round out the list of complaints, try moving the mouse towards the lower right corner and back to the upper left corner. The string showing mouse coordinates becomes shorter (fewer digits), but the trailing digits from previous strings are not erased.

Let's try to address these problems one by one.

The WM_PAINT Message

First of all, every well-behaved Windows application must be able to respond to the WM_PAINT message. Ours didn't, so after its window was occludded or minimized and then restored, it didn't repaint its client area. What should we do when Windows asks us to restore the client area? Obviously, the best solution would be to redraw the mouse trace and redisplay mouse coordinates. The trouble is that we don't remember the trace. So let's start with a simple fix--redisplaying the coordinates.

A new case has to be added to our generic window procedure:

```
case WM_PAINT:
    if (pCtrl->OnPaint ())
        return 0;
    break;
```

Strictly speaking, WM_PAINT comes with a WPARAM that, in some special cases, having to do with common controls, might be set to a device context. For now, let's ignore this parameter and concentrate on the common case.

The standard way to obtain a device context in response to WM_PAINT is to call the API BeginPaint. This device context has to be released by a matching call to EndPaint. The ownership functionality is nicely encapsulated into the PaintCanvas object:

```
class PaintCanvas: public Canvas
{
public:
    PaintCanvas (HWND hwnd)
        :   Canvas (::BeginPaint (hwnd, &_paint)),
        _hwnd (hwnd)
    {}
    ~PaintCanvas ()
    {
        ::EndPaint(_hwnd, &_paint);
    }
    int Top () const    { return _paint.rcPaint.top; }
    int Bottom () const { return _paint.rcPaint.bottom; }
    int Left () const   { return _paint.rcPaint.left; }
    int Right () const  { return _paint.rcPaint.right; }
protected:
    PAINTSTRUCT _paint;
    HWND        _hwnd;
};
```

Notice that BeginPaint gives the caller access to some additional useful information by filling the PAINTSTRUCT structure. In particular, it is possible to retrieve the coordinates of the rectangular area that has to be repainted. In many cases this area is only a small subset of the client area (for instance, after uncovering a small portion of the window or resizing the window by a small increment). In our unsophisticated application we won't make use of this additional info--we'll just repaint the whole window from scratch.

Here's our own override of the `OnPaint` method of the controller. It creates a `PaintCanvas` and calls the appropriate `View` method.

```
bool TopController::OnPaint () throw ()
{
    Win::PaintCanvas canvas (_h);
    _view.Paint (canvas);
    return true;
}
```

`View` simply calls its private method `PrintPos`. Notice that `View` doesn't distinguish between `UpdateCanvas` and `PaintCanvas`. For all it knows, it is being given a generic `Win::Canvas`.

```
void View::Paint (Win::Canvas & canvas)
{
    PrintPos (canvas);
}
```

What can we do about the varying size of the string being printed? We need more control over formatting. The following code will make sure that each of the two numbers is be printed using a fixed field of width 4, by passing the `std::setw (4)` manipulator to the stream. If the number following it in the stream contains fewer than 4 digits, it will be padded with spaces.

```
void PrintPos (Win::Canvas & canvas)
{
    std::stringstream out;
    out << "Mouse at: " << std::setw (4) << _x;
    out << ", " << std::setw (4) << _y;
    out << "    ";
    std::string s = out.str ();
    canvas.Text (0, 0, &s [0], s.length ());
}
```

You may notice that, in a fit of defensive programming, I appended four spaces to the string. Actually, without these spaces, we would still have a problem of a shrinking string. The fact that the number of characters is now fixed to 24, doesn't guarantee that the displayed string will always occupy the same area. That's because Windows uses proportional fonts, in which some characters a wider than others. In particular a space is usually narrower than a non-space character. Whence the fudge factor.

The `WM_PAINT` message is also sent to our program in response to the user resizing the window. This is one of the cases when it would make sense to repaint only the newly added area (if any). The coordinates of the fresh rectangle are after all available through `PaintCanvas`.

This is not always as simple as it sounds. For instance, for some applications, any resizing of a window should call for a total repaint. Think of a window that stretches the picture to fill the whole client area.

Yet even for such an application, outside of resizing, it might make sense to limit the repainting only to freshly uncovered areas. So is it possible to have a cake and eat it too?

The answer is in two style bits that can be set when registering a window class. These bits are CS_HREDRAW and CS_VREDRAW. The first one tells the Windows to ask for the complete redraw (i.e., invalidate the whole client area) whenever a horizontal size (width) of the window changes. The second one does the same for height. You can set both by combining them using binary or.

Invalidating an area not only sends a WM_PAINT message with the appropriate bounding rectangle to our window, but it also erases the area by overpainting it with the background brush. The background brush is set in the window class definition--our default has been the standard brush with COLOR_WINDOW. (If instead the background brush is set to 0, no erasing will be done by Windows--try it!).

The Model

If we want to be able to do a meaningful redraw, we must store the history of mouse moves. This can mean only one thing: we are finally ready to introduce the Model part of the triad Model-View-Controller. In general, you don't want to put too much intelligence or history into the View or the Controller. Mouse trace is our program's data--its I/O-independent part. You could even imagine a command-line version of the program. It wouldn't be able to display the line visually, and you'd have to input each point by typing in it's coordinates, but the Model would be the same.

I decided to use a new data structure from the standard library, the *deque* (pronounced "deck"). It works like a double-ended vector. You can push and pop items from both ends, and the methods push_front and pop_front work as efficiently as push_back and pop_back.

We don't want the history to grow beyond MAX_SIZE points. So when we add a new point to it, if it would cause the deque to exceede this count, we will pop the oldest point. In fact, this is the gist of the traditional queue, or LIFO (Last In First Out), data structure.

```
#include <deque>
#include <utility> // pair

class Model
{
    enum { MAX_SIZE = 200 };
public:
    typedef std::deque< std::pair<int, int> >::const_iterator iter;

    Model ()
    {
        AddPoint (0, 0);
    }
    void AddPoint (int x, int y)
    {
        if (_queue.size () >= MAX_SIZE)
            _queue.pop_front ();
        _queue.push_back (std::make_pair (x, y));
    }
    iter begin () const { return _queue.begin (); }
    iter end () const { return _queue.end (); }
private:
    std::deque< std::pair<int, int> > _queue;
};
```

As you can see, points are stored as std::pairs of integers. I didn't bother to create a special data structure for a two-dimensional point. The function make_pair comes in handy when you don't want to explicitly specify the types of the members of the pair. You simply let the compiler deduce them from the types of arguments--in this case both are integers. Were we to use a pair of shorts instead, we would have to use the more explicit construct:

```
_queue.push_back (std::pair<short, short> (x, y));
```

The controller must have access to both, the model and the view. In response to a mouse move, it adds a new point to the model and, as before, tells the view to move to a new position.

```
bool TopController::OnMouseMove
         (int x, int y, Win::KeyState kState) throw ()
{
    _model.AddPoint (x, y);
    Win::UpdateCanvas canvas (_h);
    _view.MoveTo (canvas, x, y);
    return true;
}
```

The repainting can now be done more intelligently (albeit wastefully--we still repaint the whole client area instead of just the rectangle passed to us in PaintCanvas). We obtain the iterator from the model and pass it to the view. The iterator gives the view access to the part of the trace that is still remembered.

```
bool TopController::OnPaint () throw ()
{
    Win::PaintCanvas canvas (_h);
    _view.Paint (canvas, _model.begin (), _model.end ());
    return true;
}
```

All the view has to do now is to connect the dots. We'll use a little trick from the <algorithm> section of the standard library. The for_each algorithm takes the starting iterator, the ending iterator and a functor.

We've already seen the use of a functor as a predicate for sorting. Here we'll make use of another type of functor that can operate on objects "pointed-to" by iterators. In our case, the iterator "points to" a pair of coordinates--what I mean is that, when it's dereferenced, it returns a reference to a std::pair<int, int>.

Our functor is called DrawLine and it draws a line from the last remembered position to the postion passed to it in a pair. We have to initialize it with the starting position and let the std::for_each template call it for each value from the iterator.

```
void View::Paint (Win::Canvas & canvas,
                  Model::iter end)
{
    PrintPos (canvas);
    if (beg != end)
    {
        DrawLine draw (canvas, *beg);
        ++beg;
        std::for_each (beg, end, draw);
    }
}
```

If you're curious to know how for_each is implemented, it's really simple:

```
template<class Iter, class Op>
inline Op for_each (Iter it, Iter end, Op operation)
{
    for (; it != end; ++it)
        operation (*it);
    return (operation);
}
```

Op here could be a pointer to a global function or an object of a class that overloads the function-call operator. This is the implementation of our functor:

```
class DrawLine
{
public:
    DrawLine (Win::Canvas & canvas, std::pair<int, int> const & p)
        : _canvas (canvas)
    {
        _canvas.MoveTo (p.first, p.second);
    }
    void operator () (std::pair<int, int> const & p)
    {
        _canvas.LineTo (p.first, p.second);
    }

    Win::Canvas & _canvas;
};
```

The two new methods of Canvas, MoveTo and LineTo, are easy to implement.

Capturing the Mouse

It doesn't make much sense to unconditionally draw the trace of the mouse. If you have ever used any Windows graphical editor, you know that you're supposed to press and hold a mouse button while drawing. Otherwise there would be no way to "lift the pen" in order to access some controls or start drawing in a different place.

We could in principle use the Win::KeyState argument to OnMouseMove and check if a button is pressed. But we can do better than that and solve one more problem at the same time. I'm talking about being able to follow the mouse outside of the window.

Normally, mouse messages are sent only to the window over which the mouse cursor hovers. But if your window "captures" the mouse, Windows will redirect *all* mouse messages to it--even when the cursor is outside of its area. Obviously, capturing a mouse requires some caution. As long as the mouse is captured, the user will not be able to click any controls or interact with any other application (e.g., to activate it). That's why it is customary only to capture the mouse while it's being dragged--and the drag has to originate in the capturing window. Dragging is done by pressing and holding a mouse button while moving the mouse.

There are three APIs related to mouse capture: `SetCapture`, `ReleaseCapture` and `GetCapture`. `SetCapture` takes the handle to the window that wants to capture the mouse. `ReleaseCapture` ends the capture and sends the `WM_CAPTURECHANGED` message to the captor. `GetCapture` returns the handle to the window that currently has the capture, in the current application (strictly speaking, in the current thread).

We will add these APIs to our class `Win::Dow` representing a window handle.

```
namespace Win
{
    class Dow
    {
    public:
        Dow (HWND h = 0) : _h (h) {}
        void Init (HWND h) { _h = h; }
        operator HWND () const { return _h; }
        void CaptureMouse ()
        {
            ::SetCapture (_h);
        }
        void ReleaseMouse ()
        {
            if (HasCapture ())
                ::ReleaseCapture ();
        }
        bool HasCapture () const
        {
            return ::GetCapture () == _h;
        }
        // Window visibility
        void Show (int cmdShow = SW_SHOW)
        {
            ::ShowWindow (_h, cmdShow);
        }
        void Hide ()
        {
            ::ShowWindow (_h, SW_HIDE);
        }
        void Update ()
        {
            ::UpdateWindow (_h);
        }
        void Display (int cmdShow)
        {

            Update ();
        }
    private:
        HWND _h;
    };
}
```

The typical procedure for capturing a mouse goes like this:

- Capture mouse when the user presses the appropriate mouse button.
- While processing mouse-move messages, check if you have the capture. If so, implement dragging behavior.
- Release capture when the user releases the mouse button.
- Finish up dragging in response to WM_CAPTURECHANGED message.

It's important to wrap up dragging inside the WM_CAPTURECHANGED handler, rather than directly in response to button release. Your window may lose capture for unrelated reasons and it's important to clean up also in that case. (An example of externally initiated mouse capture change is when the system pops up an alert message.)

We will apply this procedure by overriding the handlers for WM_LBUTTONDOWN, WM_LBUTTONUP and WM_CAPTURECHANGED. (Notice that after all this talk, we leave the implementation of OnCaptureChange empty. That's because we don't have any dragging wrap-up.)

```
bool TopController::OnLButtonDown
        (int x, int y, Win::KeyState kState) throw ()
{
    _h.CaptureMouse ();
    _model.AddPoint (x, y, true); // starting point
    Win::UpdateCanvas canvas (_h);
    _view.MoveTo (canvas, x, y, false); // don't draw
    return true;
}

bool TopController::OnLButtonUp
        (int x, int y, Win::KeyState kState) throw ()
{
    // ReleaseMouse will send WM_CAPTURECHANGED
    _h.ReleaseMouse ();
    return true;
}

bool TopController::OnCaptureChanged
        (HWND hwndNewCapture) throw ()
{
    return true;
}
```

The implementation of OnLButtonDown (that's Left Button Down) has some interesting points. Since the user now has the option to "lift the pen," we must be able to draw (and re-draw) non-continuous lines. The MoveTo method of View must be able to shift current position without drawing a line and the Model has to somehow mark the starting point of a new line. That's the meaning of the two Boolean flags in OnLButtonDown. We'll come back to that. Now let's examine the new implementation of the OnMouseMove method.

```
bool TopController::OnMouseMove
        (int x, int y, Win::KeyState kState) throw ()
```

```
{
    Win::UpdateCanvas canvas (_h);
    if (_h.HasCapture ())
    {

        _view.MoveTo (canvas, x, y);
    }
    else
        _view.PrintPos (canvas, x, y);
    return true;
}
```

Notice that we're recording the mouse position only if we have capture, otherwise we only update the display of mouse coordinates (and, since we don't have the mouse capture, we stop it when the mouse leaves the window boundaries).

Repainting the window gets slightly more complicated, because of the possibility of having multiple disconnected lines. I chose to store a Boolean flag with each remembered point, and set it to `true` only for points that are starting a new line (see the `OnLButtonDown` implementation, above). We have now outgrown our simple representation of a point as a `std::pair`. In fact, to save space, I decided to store coordinates as `short`s (obviously, we'll have to rewrite this program when screen resolutions overcome the 32k pixels-per-line barrier).

```
class Model
{
public:

    {
    public:
        Point (int x = 0, int y = 0, bool isStart = false)
            : _x (static_cast<short> (x)),

              _isStart (isStart)
        {}
        int X () const { return _x; }
        int Y () const { return _y; }
        bool IsStart () const { return _isStart; }
    private:
        short _x;
        short _y;
        bool  _isStart;
    };

    typedef std::deque<Point>::const_iterator iter;
    ...
```

Notice the default values for all arguments of the `Point` contructor. It turns out that `Point` needs a default (no-arguments) constructor if we want to store it in a `deque`. The `deque` must be able to allocate and initialize new blocks of `Point`s when it grows internally. It does it by calling the default constructors.

The implementation of View::Paint doesn't really change much, except that PrintPos now takes the values of coordinates to be displayed (and also updates the remembered position).

```
void View::Paint (Win::Canvas & canvas,
                  Model::iter beg,
                  Model::iter end)
{
    PrintPos (canvas, _x, _y);
    if (beg != end)
    {
        DrawLine draw (canvas, *beg);
        ++beg;
        std::for_each (beg, end, draw);
    }
}
```

The relevant change is in the implementation of the DrawLine functor. It doesn't draw lines that lead to starting points of line segments. It just quietly moves the current position.

```
class DrawLine
{
public:
    DrawLine (Win::Canvas & canvas, Model::Point const & p)
        : _canvas (canvas)
    {
        _canvas.MoveTo (p.X (), p.Y ());
    }
    void operator () (Model::Point const & p)
    {
        if (!p.IsStart ())
            _canvas.LineTo (p.X (), p.Y ());
        else
            _canvas.MoveTo (p.X (), p.Y ());
    }
private:
    Win::Canvas & _canvas;
};
```

Adding Colors and Frills

Let's have some fun with our program. For instance, how does one draw colored lines? The LineTo API doesn't have any arguments that could be used to specify color.

It's the device context, or Canvas, that stores all the parameters used in drawing. We have to instruct the canvas to change the drawing color. And, like good citizens, we should change it back after we're done with our drawing.

Various drawing and printing modes are conveniently grouped in GDI objects (GDI means Graphical Device Interface, in Winspeak). The one that controls the drawing of lines is called a *pen*. You change the properties of the DC by *selecting* and object into it. Again, if you're a good citizen, you deselect it afterwards, usually by selecting back the one you have displaced.

This object selection mechanism is best encapsulated in various Resource Management objects. Let's start with the most common type of object, the *stock* *object*. Windows conveniently provides a whole bunch of most commonly used device context settings in the form of predefined objects. We'll see in a moment how well stocked the Windows storeroom is. Right now we want to have a nice encapsulator for such objects.

```
namespace Win
{
    class StockObject
    {
    public:
        StockObject (int type)
            : _obj (::GetStockObject (type))

        operator HGDIOBJ () const { return _obj; }
    protected:
        HGDIOBJ _obj;
    };

    class StockObjectHolder
    {
    public:
        StockObjectHolder (HDC hdc, int type)
          : _hdc (hdc)
        {
            _hObjOld = ::SelectObject (_hdc, StockObject (type));
        }

        ~StockObjectHolder ()
        {
            ::SelectObject (_hdc, _hObjOld);
        }
    private:
        HGDIOBJ  _hObjOld;
        HDC      _hdc;

}
```

The constructor of `Win::StockObjectHolder` takes a handle to device context (which means we can pass it any `Canvas` object) and selects a particular stock object into it. The `SelectObject` API returns the previous object of that type, which we diligently remember for later. We also remember the DC, so that, in the destructor, we can restore the previous object into its proper place in that DC.

That's the Resource Management part. The various types of stock objects are identified by predefined integer ids. Given such an id, we use `GetStockObject` to retrieve the proper object from the Windows storeroom. We'll mostly use `Win::StockObjectHolder` as a base class for specific stock object holders.

There are several stock pens that might be of use. For instance, there is a black pen and a white pen. We'll enclose all these in a new nemespace `Pen`, as embedded classes of a more general class `Pen::Holder`. This class, for the moment, will be otherwise empty--but that will change soon.

This kind of class embedding gives rise to a rather convenient naming convention. Not only do we have classes like `Pen::Holder::White` and

`Pen::Holder::Black`, but also a more general class `Pen::Holder`, that will soon makes sense in itself.

```
namespace Pen
{
    class Holder
    {
    public:
        class White : public Win::StockObjectHolder
        {
        public:
            White (HDC hdc): Win::StockObjectHolder (hdc, WHITE_PEN)
{}
        };

        class Black : public Win::StockObjectHolder
        {
        public:
            Black (HDC hdc): Win::StockObjectHolder (hdc, BLACK_PEN)
{}
        };
    }
}
```

For simplicity, I didn't nest the `Pen` namespace inside `Win`; so instead of `Win::Pen::Holder::Black`, you simply call it `Pen::Holder::Black`. If I were to design a commercial library, I'd probably nest the namespaces, to avoid naming conflicts.

The way to use a stock pen is to define a holder in a given scope. For instance to draw white lines, youd write code like this:

```
{
    Pen::Holder::White wp (canvas);
    // do the drawing
}
```

The change in pen color applies to all line-drawing operations between the definition of `wp` and the end of scope, unless overridden by another pen change. End of scope always restores the pre-scope pen (automatic destructors!).

Beware of a common error of defining a stock pen holder without naming it. The code:

```
{
    Pen::Holder::White (canvas);
    // do the drawing
}
```

is also valid, although it doesn't produce the desired result. The compiler will create an anonymous temporary object and then immediately destroy it, without waiting for the end of the scope.

If you want to use *real* colors, not just black and white, you have to create your own GDI objects. A color pen is created by calling `CreatePen` and destroyed using `DeleteObject`. You specify the color by splitting it into three components: red, green and blue. Each of them can take values between 0 and 255. It doesn't necessarily mean that your computer can display different colors for all 256x256x256 combinations. Unless you set your display mode to at least

24-bit color (True Color, in Winspeak) the system will attempt to find the closest available match.

```
namespace Pen
{
    class Color
    {
    public:
        Color (int r, int g, int b, int style = PS_SOLID)
        {
            _hPen = ::CreatePen (style, 0, RGB (r, g, b));
        }
        ~Color ()
        {
            ::DeleteObject (_hPen);
        }
        operator HPEN () { return _hPen; }
    private:
        HPEN    _hPen;
    };
}
```

Creating a colored pen doesn't, by itself, change the color of lines being drawn. You still have to *select* your pen into the device context. According to the principles of Resource Management, we will encapsulate such a selection in an object called `Pen::Holder`--yes, that's the same class we used for embedding the classes `White` and `Black`. Defining a *named* object of the type `Pen::Holder` will temporarily change the color of all subsequent line drawings.

```
namespace Pen
{
    class Holder
    {
    public:
        Holder (HDC hdc, HPEN hPen)
            : _hdc (hdc)
        {
            _hPenOld = reinterpret_cast<HPEN> (
                           ::SelectObject (_hdc, hPen));
        }
        ~Holder ()
        {
            ::SelectObject (_hdc, _hPenOld);
        }
    private:
        HDC     _hdc;
        HPEN    _hPenOld;
    public:
        class White : public Win::StockObjectHolder
        ...
        class Black : public Win::StockObjectHolder
        ...
    };
}
```

Notice that, since a `Win::Canvas` can be implicitly converted to `HDC`, and a `Pen::Color` to `HPEN`; the holder's constructor is normally called with a `Win::Canvas` and a `Pen::Color`.

The standard way to use pens and holders is to define and store your pens in the `View` object and then "hold" them for the duration of a particular drawing operation. Let's define two pen colors, dark red and light blue, for our drawings.

```
class View
{
public:
    View ()
        : _x (0), _y (0),
          _redPen (128, 0, 0),
          _bluePen (0, 0, 255)
    {}
    ...
private:
    int _x, _y;
    Pen::Color _redPen;
    Pen::Color _bluePen;
};
```

We'll use the blue pen for our regular drawing:

```
void View::MoveTo (Win::Canvas & canvas, int x, int y, bool visible)
{
    if (visible)
    {
        Pen::Holder ph (canvas, _bluePen);
        canvas.Line (_x, _y, x, y);
    }
    PrintPos (canvas, x, y);
}
```

and the red pen for re-drawing:

```
void View::Paint (Win::Canvas & canvas,
                  Model::iter beg,
                  Model::iter end)
{
    Pen::Holder ph (canvas, _redPen);
    PrintPos (canvas, _x, _y);
    if (beg != end)
    {
        DrawLine draw (canvas, *beg);
        ++beg;
        std::for_each (beg, end, draw);
    }
}
```

This is not what you would normally do--visually distinguishing between active drawing and passive redrawing--but here we are trying to learn something. Play with this program and observe what areas are being redrawn when you uncover parts of your window or resize it. You'll notice that the red lines appear only in the freshly uncovered areas, even though our algorithm seems to be repainting the whole client area in response to `WM_PAINT` messages.

Well, not everyting your program draws is displayed by Windows. For instance, you must be aware of the fact that parts of the lines that fall outside of the boundaries of the client area are never displayed (they would overwrite other windows!). Windows *clips* all drawings down to the client area. In fact, during the processing of WM_PAINT, Windows clips your drawings even further--down to the invalidated area. And even though in PaintCanvas, Windows sends us a single bounding rectangle for our convenience, the invalidated area may be more complicated (e.g., multiple rectangles) and the PaintCanvas will clip all subsequent drawings down to this area. If you really want to draw outside of this area, you have to create an additional UpdateCanvas and work with it.

But what should you do if, every time you draw, you want to use a different color? You can't possibly prepare all possible pens in the constructor of View! In that case, you are stuck with creating a pen on the spot, selecting it into the DC, and discarding it immediatly after deselecting. This use is common enough to call for a separate encapsulator, Pen::Holder::Instant.

```
namespace Pen
{
    class Holder
    {
    ...
    public:
    ...
        class Instant
        {
        public:
            Instant (HDC hdc, int r, int g, int b)
                : _hdc (hdc)
            {
                _hPen = ::CreatePen (PS_SOLID, 0, RGB (r, g, b));
                _hPenOld = reinterpret_cast<HPEN> (
                                   ::SelectObject (_hdc, _hPen));
            }
            ~Instant ()
            {
                ::SelectObject (_hdc, _hPenOld);
                ::DeleteObject (_hPen);
            }
        private:
            HDC     _hdc;
            HPEN    _hPen;
            HPEN    _hPenOld;
        };
    };
}
```

The same thing we've done with pens, you can do with brushes. A brush is used for filling areas--for instance if you draw a solid rectangle or ellipse. If you substitute the word Brush for Pen, you can almost directly copy the above code and use it for brushes. The only real difference is the use of the API CreateSolidBrush in place of CreatePen. There are other types of brushes besides solid--hatched and patterned--that you can use as well. Just provide the appropriate overloaded constructors for Brush::Color.

There is one special place where a brush is used--in the definition of a window class. This is the brush that is used for painting the window background.

For instance, Windows automatically paints a fresh background over all the invalidated areas, in response to BeginPaint (called in the constructor of our PaintCanvas). By the way, EndPaint does the validation of this area, so if you forget to call it, Windows will keep sending WM_PAINT messages forever.

To choose a non-default background color for the windows of a particular class, we have to create a brush and pass it to the WNDCLASSEX structure. The brush will be automatically destroyed when the application exits (or the class is discarded).

```
void ClassMaker::SetBkColor (int r, int g, int b)
{
    _class.hbrBackground = ::CreateSolidBrush (RGB (r, g, b));
}
```

We call this method in main, setting the window background to light blue.

```
    Win::ClassMaker winClass (className, hInst);
    winClass.SetIcons (IDI_MAIN);
    winClass.SetBkColor (128, 128, 255);
    winClass.Register ();
```

Finally, we can play similar games with fonts. We can change the color of the printed text, the color of its background and even the type of the font. This last option, in particular, might help us with our incredible shrinking string. Changing the font from *proportional* to *fixed* does the trick. In a fixed-pitch font, all characters, including spaces, are the same size. We'll be able to get rid of the spurious spaces at the end of our string.

```
namespace Font
{
    class Stock: public Win::StockObject
    {
    public:
        Stock (int type) : Win::StockObject (type) {}
        operator HFONT () const
        {
            return reinterpret_cast<HFONT> (_obj);
        }
    };

    class SysFixed: public Stock
    {
    public:
        SysFixed () : Stock (SYSTEM_FIXED_FONT) {}
    };

    class Holder
    {
    public:
        class Color
        {
        public:
            Color (HDC hdc, int r, int g, int b)
                : _hdc (hdc),
                  _oldColor (::SetTextColor (_hdc, RGB (r, g, b)))
            {}
            ~Color ()
            {
                ::SetTextColor (_hdc, _oldColor);
            }
        private:
            HDC            _hdc;
            COLORREF    _oldColor;
        };

        class Background
        {
        public:
            Background (HDC hdc, int r, int g, int b)
                : _hdc (hdc),
                  _oldColor (::SetBkColor (_hdc, RGB (r, g, b)))
            {}
            ~Background ()
            {
                ::SetBkColor (_hdc, _oldColor);
            }
        private:
            HDC            _hdc;
            COLORREF    _oldColor;
        };

        class SysFixed : public Win::StockObjectHolder
        {
        public:
            SysFixed (HDC hdc)
                : Win::StockObjectHolder (hdc, SYSTEM_FIXED_FONT)
                      {}
        };
    };
}
```

Equipped with all these tools, we can now print the position of the mouse in yellow on dark blue, using a fixed-pitch system font.

```
void View::PrintPos (Win::Canvas & canvas, int x, int y)
{
    ...
    Font::Holder::Color fc (canvas, 255, 255, 0); // yellow
    Font::Holder::Background bk (canvas, 0, 64, 128); // dk blue
    Font::Holder::SysFixed fix (canvas); // fixed pitch
    canvas.Text (0, 0, &s [0], s.length ());
}
```

Stock Objects

Here's a complete list of Windows stock objects.

- Brushes
    - WHITE_BRUSH
    - LTGRAY_BRUSH
    - GRAY_BRUSH
    - DKGRAY_BRUSH
    - BLACK_BRUSH

    - NULL_BRUSH
    - HOLLOW_BRUSH
    - DC_BRUSH
- Pens
    - WHITE_PEN
    - BLACK_PEN
    - NULL_PEN
    - DC_PEN
- Fonts
    - OEM_FIXED_FONT
    - ANSI_FIXED_FONT
    - ANSI_VAR_FONT
    - SYSTEM_FONT
    - DEVICE_DEFAULT_FONT
    - SYSTEM_FIXED_FONT
    - DEFAULT_GUI_FONT
- Misc.
    - DEFAULT_PALETTE

Windows Application

Since I can't possibly cover Windows programming to any depth within the confines of this book, I will now switch to a bird's-eye view of how one might approach the task of writing a Windows application. As before, I will concentrate on the two main challenges of Windows programming: how the program should work and look like, and how to encapsulate and categorize various APIs. The mere number of Windows APIs is so overwhelming that some kind of classification (with C++ classes and namespaces) is a must.

After reading this chapter, I strongly suggest browsing the source code and studying the details of the implementation.

Porting the Calculator to Windows

Taking a command-line application, like our symbolic calculator, and making a minimal port to Windows could be quite easy. We have to find a way, other than `std::cin`, to get user input; and to display the results in a window. Error display could be dealt with using message boxes. A single dialog box with two edit fields--one for input and one for output--would suffice.

But this kind of port is not only primitive--it doesn't even cover the whole functionality of our original application. In the command-line calculator, the user was at least able to see some of his or her previous inputs--the history of the interaction. Also, in Windows, commands like *load*, *save* or *quit* are usually disengaged from text-based input and are available through menus, buttons, toolbars or accellerator keys. Finally, there are certain possibilities specific to Windows that can enhance the functionality of our program and are expected by the user. An obvious example would be the display the contents of the calculator's memory.

Some of these features don't require substantial changes to the calculator engine. Disengaging commands from text input will actually make the parsing simpler--the `Scanner` will have to recognize fewer tokens and the `CommandParser` object will become unnecessary. Other features, such as the display of memory, will require deeper changes inside the `Calculator` object (which, by the way, will play the role of a Model).

User Interface

Whether it's porting or designing an application from scratch, the first step is to envision the user interface.

In our case, we obviously need an input field, where the user will type the expressions to be evaluated. We also need an output field, where the results of the calculations will appear. We'd also like to have one window for the history of user input, and another to display the contents of memory.

The design of the menu is a non-trivial problem. We know, more or less, what we will have there: Load, Save as well as the traditional About and Exit. We can also add one more command--Clear memory. The problem is how to organize these commands.

The traditional approach, which often leads to barely usable menu systems, is to follow the historical pattern. Menu commands are grouped under File, Edit, View, Tools, Window, Help, etc. Notice that some of these words are nouns, others verbs. Tradition also dictates arbitrarily that, for instance, Exit goes under File, About under Help, Search under Edit, Options under Tools, etc...

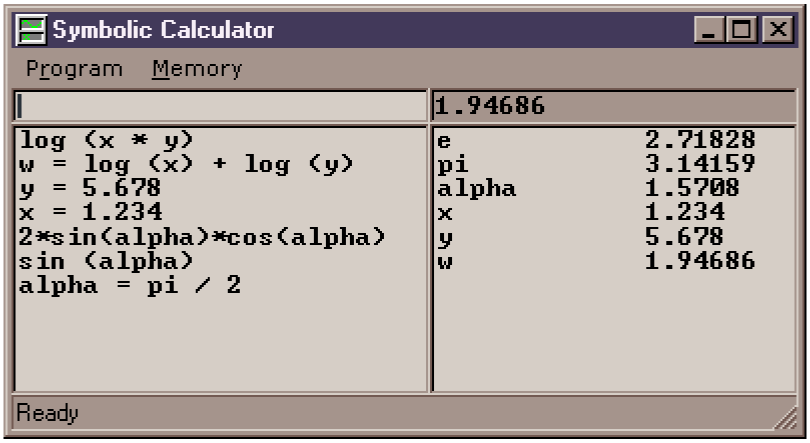
The more modern approach is to group commands under noun headings that describe objects upon which these command act. The headings are, as far as possible, ordered from general to specific. The first, most general menu item is Program. This is where you should put the commands About and Exit. The About command displays information about the Program, not the Help. Similarly, you Exit a Program, not a File.

The Program item is usually followed by such groups as Project, Document (or File); then View or Window, Edit and Search; then the elements of the particular display, such as All, Selection; and finally, Help. Even if the user is used to traditional groupings, he or she is likely to find this new arrangement more logical and easier to use.

In our calculator, we'll have the Program menu followed by the Memory menu. The operations one can apply to Memory are Clear, Save and Load. A more complete version would also have the Help menu.

Finally, it's nice to have a status bar at the bottom of the window to show the readiness of the calculator to accept user input, as well as little command help snippets when the user selects menu items.

Instead of showing you varius sketches, I'll present the final result--the actual screenshot of the calculator's user interface.



```
1.94686
```

```
log (x * y)
w = log (x) + log (y)
y = 5.678
x = 1.234
2*sin(alpha)*cos(alpha)
sin (alpha)
alpha = pi / 2
```

```
e              2.71828
pi             3.14159
alpha          1.5708
x              1.234
y              5.678
w              1.94686
```

Child Windows

The calculator's main window is tiled with various sub-windows, or child windows. Some of these children--like the title bar, the menu bar, minimize/maximize buttons, etc.--are created and managed by Windows. Others are a programmer's responsibility.

A *child* window is owned by its parent window. Since there is often some confusion between the meaning of a "child" window vs. an "owned" window, here's a short summary.

- Child window: lives in the coordinate system of its parent and its clipped to its boundaries. A child cannot have a menu, but it can have an integer id (that's why the `menu` member of the `CREATESTRUCT` doubles as a child id). To create a child window, specify `WS_CHILD` as window style and initialize the `hwndParent` member of `CREATESTRUCT` to its parent's window handle.
- Owned window: always appears in front (on top) of its owner, is hidden when the owner is minimized, and is destroyed when the owner is destroyed. The handle to the owner window is passed in the `hwndParent` member of `CREATESTRUCT`. An owned window doesn't have to be a child window--when it's not, it doesn't use the owner's coordinate system and is not clipped to its boundaries.

You will probably want all windows in your applications to be owned by the top-level window. If, furthermore, you'd like them to be constrained to the client area of the top-level (or some other) window, you'll create them as children of their owners. (Examples of owned, but non-child, windows are dialog boxes and message boxes, about which I'll talk later.)

I introduced a separate class, Win::ChildMaker, derived from Win::Maker, to encapsulate the creation of child windows. You can have a child window based on any window class, as long as you know the class' name. For our purposes, however, we'll be creating windows based on classes that are pre-defined (and pre-registered) by Windows.

There are several useful window classes corresponding to standard controls. Instead of writing and testing new window procedures, we can get a lot of standard functionality by re-using these controls. That's why I created a subclass of Win::ChildMaker called Win::ControlMaker that can be used as base for the specialized contol makers, such as Win::EditMaker, Win::ListBoxMaker and Win::StatusBarMaker.

Before going into any more detail about controls, let's talk some more about the management of child windows. I made them part of the View object, whose constuctor uses various control makers to create them. There are two edit controls, one of them read-only. There are two listboxes that are particularly suited for displaying lists. Finally, there is a status bar control.

The first thing our program has to take care of is the proper positioning of these child windows. This is usually done in response to the WM_SIZE message sent to the top-level window. The first such message arrives just before the child windows are displayed, so that's a great time to size and position them for the first time. Then, every time the user resizes the top-level window, the WM_SIZE message is sent again. The parameters of WM_SIZE are the width and the height of the client area. All we have to do is to partition that rectangle among all the children. After doing some simple arithmetic, we tell the children to move to their new positions.

Next, we have to take care of the focus. Whenever the calculator is active, we would like the keyboard input to go directly to the input edit control. Since this doesn't happen automatically, we have to explicitly pass keyboard focus to this child window whenever the calculator gets focus. Fortunately, the top-level window is notified every time this happens. The WM_SETFOCUS message is sent to it whenever the window becomes active, including the time of its creation.

Windows Controls

A Windows control is a window whose class and procedure--the major elements of behavior--are implemented by the system. Controls vary from extremely simple, like static text, to very complex, like ListView or TreeView. They are ubiquitous in dialog boxes (about which we'll talk soon), but they can also be created as stand-alone child windows. The application communicates with a control by sending it messages. Conversely, a control communicates with its parent by sending messages to it. In addition to common messages, each type of control has its own distinct set of messages it understands.

I defined a class Win::SimpleCtrl that combines the attributes common to all controls. Since a control is a window, Win::SimpleCtrl inherits from Win::Dow. Like all child windows, controls can be assigned numerical ids, so that their parent can distinguish between them (although it may also tell them apart by their window handles). If a control is created standalone, using an appropriate maker, we can initialize the corresponding SimpleCtrl object directly with its window handle. Otherwise, we initialize it with the parent's window handle and the child id (this pair is immediately translated to the window handle using the GetDlgItem API).

The simplest control is called *static text*. It displays text in a simple frame. The corresponding object, Win::StaticText, is the simplest descendant of Win::SimpleCtrl. The program can modify the text displayed in it by calling the SetText method that Win::StaticText inherits from Win::Dow.

A little more interesting is the *edit control*. Its read-only version behaves just like static text, except that you can copy text from it into the clipboard (by selecting it and pressing Ctrl-C). The full-blown edit control can be used not only to display text, but also to read user input. It also supports several editing functions, like delete, copy, cut, paste, etc. In fact, a multi-line edit control is the engine behind the Windows own Notepad.

The most important thing that an application may want to get from an edit control is the text that's been input there by the user. The class Win::Edit, another descendant of Win::SimpleCtrl, provides two different ways of doing that. The low-level method GetText takes a buffer and a count (you can use GetLen method to enquire about the length first). The high-level method GetText takes no arguments and returns a std::string.

Controls keep sending messages to their parent windows. The interesting ones are WM_COMMAND and, for the newer generation of controls, WM_NOTIFY. It so happens that, for some historical reason, WM_COMMAND is not only used by controls, but also by menus and accellerators. Our generic window procedure sorts them out and calls, respectively, OnControl or OnCommand. OnControl is passed the control's window handle, its numerical id and the id of the command. For instance, every time the user changes the text in an edit control, a command message is set to the parent, with the command id EN_CHANGE (applications usually ignore this message).

The main question with edit controls is when it's a good time to retrieve their text. The user must have some way of telling the program that the input is ready--in our case, that the whole expression has been entered. We could have added a special button "Evaluate" for the user to click on. But that's not what the user expects. He or she will most likely press the *Enter* key on the keyboard and expect the program to take it as a cue that the input is ready. Surprisingly, this very basic feature is not easy to implement. There is no simple way to tell the edit control to notify the parent when the Enter key is pressed.

So what are we to do? We'll have to use the back-door approach called *window subclassing*. We have to write our own window procedure and plug it into the edit control. Fortunately, our window procedure doesn't have to implement all the editing functionality from scratch. All it has to do is to intercept the few messages we're interested in and pass the rest to the original, Windows-defined procedure.

I have conveniently encapsulated window subclassing in two Win::Dow methods, SubClass and UnSubClass. The SubClass method takes a pointer to the SubController object. It substitutes a generic SubProcedure in place of the old window procedure. For every message, the SubProcedure first calls the SubController to process it. If the SubController doesn't process it (i.e., returns false), it calls the old window procedure. Notice how the procedures are chained together. Each of them either processes a message or calls the next, until the last one calls the default window procedure.

The subclassing of the edit control proceeds as follows. We define a subclass of Win::SubController, and call it EditController. It overrides only one method, OnKeyDown, and processes it only if the key code corresponds to the Enter key (VK_RETURN). On intercepting the Enter key, it sends a message to the parent window. I decided to use a standard control message, WM_CONTROL, which could be processed by the OnControl method of the parent window. I had to pack the wParam and lParam arguments to mimic the control messages sent by Windows. As the control id I chose the pre-defined constant IDOK (thus immitating the pressing of the OK button in dialog boxes).

To perform the subclassing, we call the SubClass method of the edit window, passing it the EditController. We have to override the OnControl method of our main controller to intercept the IDOK message and react appropriately. In fact that's where all the code from our old main went. We retrieve the string from the edit control, encapsulate it in a stream, create the scanner and the parser and evaluate the result. At this point we display the result and update the history and memory displays (in case the calculation changed memory).

Displaying history is very simple. We just tell the ListBox to add a line to its window containing the string that was input by the user. ListBox will than take care of storing the string, displaying it, as well as scrolling and re-painting when necessary. We don't have to do anyting else. That's the power of Window controls.

Displaying memory, on the other hand, is not so simple. That's because, in that case, the display has to reflect the state of a data structure that constantly changes. New variables are added and the values of old variables are modified as a result of user actions. Only the parser has any understanding of these actions. It decides whether an expression entered in the input window modifies the calculator's memory or not. Thus the display has to change in response to a change in the model.

There are two approaches in dealing with the model-view feedback loop. The shotgun approach assumes that every user action that may change the state of the model requires the refreshing of the particular part of the view. In this scheme, display changes are controller-driven. The controller tells the view to refresh itself, and the view queries the model for the changes to be displayed. The model has no dependency on the view, as it shouldn't.

The notification approach, on the other hand, assumes that the model will notify the view directly. A notification might simply tell the view to refresh itself, or it might provide the exact information about what and how should be changed. The problem with this scheme is that it introduces a circular dependency. The view depends on the model, because it has to know how to query for data to be displayed. The model depends on the view, because it has to know how to notify it about changes. Now, if you look back at our definition of the model, you'll find that it was supposed to be unaware of the details of the user interface. It seems like we'll have to abandon this nice subdivision of responsibilities and break the simple hierarchy of dependencies in favor of a more entangled (and complex) system.

Don't despair yet! I'll show you how you can have a cake of hierarchies and eat it too with notifications. The trick is to give the model only a very minimal glimpse of the view. All the model needs is a "notification sink," and object that expresses interest in being notified about certain events. The model will have no clue about how these notifications are used. Furthermore, it will have no knowledge of the existence of the class View.

A notification sink should be a separate class that will encapsulate only the notification interface. The model will have to know about this class, have access to an object of this class and call its methods. The view could simply inherit from the notification sink and implement its virtual methods. When the model sends a notification to its notification sink object, it really sends it to the view. But it still lives in a state of ignorance about the details of the UI.

The absence of the dependency loop in this scheme is best illustrated by the hierarchy of includes. The header file that defines the notification sink is at the top of the hierarchy. It has to be included in `view.h`, because class `View` inherits from class `NotificationSink`. Some of the files that constitute the implementation of the model (the `Calculator` object) will also have to include it, because they call methods of `NotificationSink`. `View.cpp` will have to include `calc.h`, because it needs data from the model. The important thing is that *no model file* will have to include `view.h`. The graph of dependencies is a DAG--a directed acyclic graph--the sign of a good design.

Now that you have an idea how to implement the optimal solution, you might be pleased to learn that in the implementation of our calculator I voted for the simpler, shotgun approach. The controller calls `View::RefreshMemory` every time a new expression is processed by the parser. As you might recall, this is done in response to the `IDOK` command sent by the edit control after detecting the enter key.

In order not to be really crude (and not to cause too much screen flicker), I chose to update the memory display line-by-line, and modify only these entries which have changed since the last refresh. So in our port to Windows, the changes to the calculator not only require a new interface for memory listing, but also some mechanism to mark (and unmark) individual memory entries. This snippet of code shows how it works:

```
void View::UpdateMemory ()
{
    int count = _memoryView.GetCount ();
    for (int i = 0; i < count; ++i)
    {
        int id = _memoryView.GetData (i);
        if (_calc.HasValueChanged (id))
        {
            _calc.ResetChange (id);
            std::string varStr = FormatMemoryString (id);
            _memoryView.ReplaceString (i, varStr);
            _memoryView.SetData (i, id);
        }
    }
    int iNew;
    while ((iNew = _calc.FindNewVariable ()) != idNotFound)
    {
        _calc.ResetChange (iNew);
        std::string varStr = FormatMemoryString (iNew);
        int i = _memoryView.AddString (varStr);
        _memoryView.SetData (i, iNew);
    }
}
```

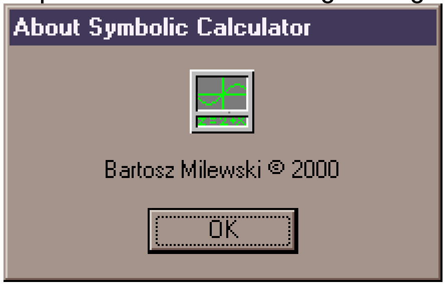
The member `_memoryView` represent the ListBox control that displays the contents of the calculator's memory.

Dialogs

The other, and indeed more common, application of Window controls is in the construction of dialog boxes. A dialog box is a pre-fabricated window that provides a frame for various controls specified by the programmer. The type of controls and their positions are usually described in the resource script (the same where we described the icon). Here's an example of such a description:

```
IDD_ABOUT DIALOG DISCARDABLE  0, 0, 142, 70
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION
CAPTION "About Symbolic Calculator"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON   "OK",IDOK,46,49,50,14
    CTEXT           "Bartosz Milewski © 2000",IDC_STATIC,10,33,121,14
    ICON            IDI_MAIN,IDC_STATIC,60,7,20,20
END
```

You don't really have to learn the language of resource scripts, because their creation is normally handled by the resource editor. The above script corresponds to the following dialog box:



The "About" dialog of the Symbolic Calculator.

This dialog contains three controls, a default push button, a static text and a static icon. The button's id is IDOK, the two static controls have both the same id, IDC_STATIC. These particular ids are predefined, but in general you are free to assign your own ids to controls, as long as you define symbols for them in the resource header (resource.h, in our case).

The Load and Save dialogs (displayed after selecting Load or Save from the Memory menu) are a little more involved. They both have an OK button and a CANCEL button. The Save dialog, moreover, contains an edit control and the Load dialog has a ListBox. These are the same controls we used in the main window, but here it's up to the dialog box to create and position them according to the script. Not only that, the dialog makes sure that when the user presses the enter key, the result is the same as if he clicked on the OK button. There is no need for us to subclass the edit control in a dialog. You can also understand now why I chose the constant IDOK as the command identifier for the subclassed edit control in the main window.

Dialog box takes care of one more thing, the focus. In our main window, we had a very simple rule--the focus always went to the input edit control. It's not so simple when you can have more than one potentially active control. Suppose that you, as the user, have activated one of the controls (i.e., one of several edit boxes in the same dialog, for instance by typing something into it). You may want to temporarily switch focus to a different application, and then go back to finish the typing. You'd expect the focus to return to the same edit control you activated before. The dialog is supposed to *remember* which control had been active before losing focus, and give the focus back to it upon being reactivated. A dialog takes care of it, transparently. In contrast, if you arrange a few controls in an arbitrary (non-dialog) window, you'd have to add special logic to its controller, to take care of correct focus restoration.

There's more! A dialog provides means of navigation between various controls using the keyboard. You can move focus from one group to another by pressing the tab key, and you can move within one group (e.g., of radio buttons) by pressing the arrow keys.

So why, you might ask, didn't I make the top-level window of the calculator a dialog? It would have solved several problems all at once. Indeed, there are some applications that use a dialog as their main window--the Windows calculator comes to mind. But I wanted to write a more mainline Windows program, with a menu, a status bar and a resizable main window. Taking all this into account, it was simpler to just put together a few controls and stick them into a resizable parent window.

A dialog provides a generic way of handling all the controls it owns. But if you want the dialog to do something useful, you have to intercept some of the communications between the controls and the dialog. The least you can do is to end the dialog when the user presses the OK (or CANCEL) button. If there is some data the user inputs through the dialog, you might want to retrive it from one (or more) of its controls and pass it on to the caller. You might also want to mediate between some of the controls. All this is possible through the use of a *dialog procedure*.

A dialog procedure is a simplified version of a window procedure. All messages that the dialog receives from its controls are first filtered by this user-defined procedure. The main difference between a dialog procedure and a window procedure is that a dialog procedure doesn't call DefWindowProc, but instead returns FALSE when it doesn't care about a particular message. If, on the other hand, it *does* process a message, it is supposed to return TRUE. Notice that these are Windows-defined Boolean values, not the C++ true and false.

That's the theory behind dialogs--now we have to come up with a way to encapsulate them into something easier to use and less error-prone. A dialog is like a function that uses UI to get its data. You "call" a dialog with some initial data (the arguments) and it returns a result, presumbly obtained from the user. We can always encapsulate the in- and out- data into some user-defined data structure. But we also have to define the active part, some kind of a controller that could be plugged into the generic dialog procedure. We can even combine the in/out data with this controller, so that it can operate on them in response to control messages.

Here's the simplest example of how one may use this abstraction in order to display the About dialog.

```
AboutCtrl ctrl;
Dialog::Modal dialog (_win, IDD_ABOUT, ctrl);
```

The dialog object takes three arguments, the owner window (here, it's the top-level window), the dialog id (to identify the script in the resources), and the controller object. The controller is derived from the library class, Dialog::ModalController. In this absolutely minimal implementation, it only overrides the OnCommand method to intercept the OK button press.

```
bool AboutCtrl::OnCommand (int ctrlId, int notifyCode) throw
(Win::Exception)
{
    if (ctrlId == IDOK)
    {
        EndOk ();
        return true;
    }
    return false;
}
```

The Save dialog has some more functionality. It contains a string, _path, that will store the path to be returned by the dialog. It has a Win::Edit object which it uses to communicate with the edit control present in the dialog box. The OK handler retrieves the string from the edit control and copies it into _path. The CANCEL handler just terminates the dialog by calling the method Dialog::ModalController::EndCancel.

```
bool SaveCtrl::OnCommand (int ctrlId, int notifyCode) throw
(Win::Exception)
{
    if (ctrlId == IDOK)
    {
        SetPath (_edit.GetText ());
        EndOk ();
        return true;
    }
    else if (ctrlId == IDCANCEL)
    {
        EndCancel ();
        return true;
    }
    return false;
}
```

The caller can distinguish between a successful input (OK was pressed) and an aborted input (CANCEL was pressed) by calling the method IsOk.

```
SaveCtrl ctrl;
Dialog::Modal dialog (_win, IDD_SAVE, ctrl);
if (dialog.IsOk ())
{
    std::string const & path = ctrl.GetPath ();
    Serializer out (path);
    _calc.Serialize (out);
}
```

The controller must also initialize the _edit object by providing it with the dialog window (which is the parent to this edit control) and a control id. This is done inside the OnInit method.

```
void SaveCtrl::OnInitDialog () throw (Win::Exception)
{
    _edit.Init (GetWindow (), IDC_EDIT);
}
```

Notice that we are using the same Win::Edit class that we used in the top-level window to encapsulate its edit control. The only difference is that here we don't use a *maker* to create the edit control. The dialog creates the control, and we can retrieve its window handle by calling GetDlgItem inside Win::Edit::Init.

The Load dialog is even more advanced. It has a ListBox control which is used to display the list of files in the current directory. It also has a static text control which is used by the ListBox to display the path to current directory. The ListDirectory method of the Win::ListBox takes care of the listing of the directory and the initialization of the static text.

```
void LoadCtrl::OnInitDialog () throw (Win::Exception)
{
    _listBox.Init (GetWindow (), IDC_LIST);

IDC_STATICPATH);
}
```

When the user clicks the OK button or double-clicks on an item, we try retrieve the full path of the selection from the ListBox. The method, GetSelectedPath, fills the buffer with data and returns true if the selection was a directory (not a file). If it's a directory, we change the current directory and re-initialize the ListBox. If it's a file, we end the dialog and let the caller retrieve the path from the buffer.

```
bool LoadCtrl::OnCommand (int ctrlId, int notifyCode) throw
(Win::Exception)
{
    if (ctrlId == IDOK || ctrlId == IDC_LIST && notifyCode ==
LBN_DBLCLK)
    {
        if (_listBox.GetSelectedPath (GetWindow (),
            GetBuffer (), GetBufLen ()))
        {
            // directory selected
            ChangeDirectory ();
        }
        else if (_listBox.IsSelection ())
            EndOk ();
        else
            EndCancel ();
        return true;
    }
    else if (ctrlId == IDCANCEL)
    {
        EndCancel ();
        return true;
    }
    return false;
}
```

I must admit that this type of UI for retrieving files is somehow obsolete. For one, it doesn't display long file names. Also, navigation between directories is not very intuitive. There is a Windows API, called GetOpenFileName, which has all the functionality of our Load dialog and a much better user interface. I chose the old-fashioned way (still used in some applications) only to illustrate the use of dialog boxes with non-trivial controls.

Commands and Menus

Pointers to members.

In the command-line version of our program we had to use a special parser to decipher user commands. In Windows it is much simpler. Commands are entered mainly by making menu selections. The obvious advantage of such an approach is that the user doesn't have to remember the exact name and syntax of each command. They are listed and nicely organized in a system of menus. Even if a program offers other means of entering commands, like keyboard shortcuts or toolbars, menus still play an important role in teaching a newcomer what functionality is available.

It's relatively easy to equip a Windows program with a menu. You can define it in the resource script, which can be made really easy by using a resource editor. You give the menu a name, which you then pass to the `Win::ClassMaker`, or to each individual `Win::Maker`.

Each menu item is given a unique id, called *command id*. This id becomes one of the arguments to the `WM_COMMAND` message, which is sent whenever the user selects a menu item. All you have to do is to implement the `OnCommand` method of the top-level controller to respond to these commands.

The obvious implementation of `OnCommand` would be to write a big `switch` statement with a `case` for each command. Did I hear "switch"? Can't we come up with something better? Can we hide the switch in the library, just like we did with the window procedure? Not really--unlike Windows messages which are defined up-front by Windows and are not supposed to change, each menu comes with its own completely different set of commands and command ids.

On the other hand, a menu command doesn't come with a variable number and type of arguments. In fact a menu command has no arguments--the arguments, if needed, are later picked up by dialog boxes. So why not organize all commands into an array of pointers to functions that take no arguments, just like we organized built-in functions in the calculator. There is only one problem in such a scheme--commands would have to be free functions. On the other hand, most of them need to have access to the model (the `Calculator`). We don't want to make the calculator a global object and give commands the knowledge of its name. Such implicit dependency through global objects is a sign of bad design and will cause maintenance naightmares in more complex programs.

How about creating a separate object, `Commander`, whose methods would be the commands? Such an object can be made a member of `TopController` and be initialized in the controller's `OnCreate` method. We could easily give it access to the `Calculator` without making it global. We could create a vector of pointers to `Commander` methods, and use it for dispatching menu commands. This is the same scheme that we used for dispatching function calls in the calculator.

But what is a *pointer to method*? Unlike a free function, a (non-static) method can only be called in the context of an object, which becomes the provider of the `this` pointer. Also, a definition of a pointer to method must specify the class, whose method it might point to. A pointer to a `Controller` method is a different beast altogether than a pointer to a `Commander` method. This requirement is reflected in the declaration of a pointer to member. For instance, a pointer to the `Commander` method, `pCmdMethod`, that takes no arguments and has no return value will be declared like this:

```
void (Commander::*pCmdMethod) ();
```

Such a pointer can be initialized to point to a particular method, e.g. Program_About, of the Commander.

```
pCmdMethod = &Commander::Program_About;
```

Given an object, _commander, of the class Commander, we can call its method through the pointer to member.

```
(_commander.*pCmdMethod) ();
```

Similarly, if _commander is a *pointer* rather than an object (or reference), the syntax changes to:

```
(_commander->*pCmdMethod) ();
```

Let's define all the commands as members of Commander and give them names corresponding to their positions in the menu. The definition and initialization of the command vector would look something like this:

```
typedef void (Commander::*CmdMethod) ();

const CmdMethod CmdTable [] =
{

    &Commander::Program_Exit,
    &Commander::Memory_Clear,
    &Commander::Memory_Save,
    &Commander::Memory_Load,
    0 // sentinel
};
```

So how would our CmdTable work with a menu system? What happens when the user selects a menu item? First, the message WM_COMMAND is sent to the generic window procedure, which calls the OnCommand method of our TopController. This method should somehow translate the command id (defined in the resource script together with the menu) to the appropriate index into the CmdTable, and execute the corresponding method.

```
(_commander->*CmdTable [idx]) ();
```

The translation from command id to an index is the weakest point of this scheme. In fact, the whole idea of defining your menu in the resource file is not as convenient as you might think. A reasonably complex application will require dynamic changes to the menu depending on the current state of the program. The simplest example is the Memory>Save item in the calculator. It would make sense for it to be inactive (grayed out) as long as there has been no user-defined variable added to memory. We could try to somehow re-activate this menu item when a variable is added to memory. But that would require the model to know something about the user interface--the menu. We could still save the day by making use of the notification sink. However, there is a better and more general approach--dynamic menus.

Dynamic Menus

First, let's generalize and extend the idea of a command table. We already know that we need there a pointer to member through which we can execute commands. We can also add another pointer to member through which we can quickly test the availability of a given command--this will enable us to dynamically gray out some of the items. A short help string for each command would be nice, too. Finally, I decided that it will be more general to give commands string names, rather than integer identifiers. Granted, searching through strings is slower than finding an item by id, but usually there aren't that many menu items to make a perceptible difference. Moreover, when the program grows to include not only menus, but also accelerators and toolbars; being able to specify commands by name rather than by offset is a great maintainability win.

So here's the definition of a command item, the building block of a command table.

```
namespace Cmd
{
    template <class T>
    class Item
    {
    public:
        char const * _name;            // official name
        void (T::*_exec)();            // execute command
        Status (T::*_test)() const;    // test commnad status
        char const * _help;            // help string
    };
}
```

If we want to reuse `Cmd::Item` we have to make it a template. The parameter of the template is the class of the particular commander whose methods we want to access.

This is how the client creates a command table and initializes it with appropriate strings and pointers to members.

```
namespace Cmd
{
    const Cmd::Item<Commander> Table [] =
    {
        { "Program_About",  &Commander::Program_About,
                            &Commander::can_Program_About,
          "About this program"},
        { "Program_Exit",   &Commander::Program_Exit,
                            &Commander::can_Program_Exit,
          "Exit program"},
        { "Memory_Clear",   &Commander::Memory_Clear,
                            &Commander::can_Memory_Clear,
           "Clear memory"},
        { "Memory_Save",    &Commander::Memory_Save,
                            &Commander::can_Memory_Save,
          "Save memory to file"},
        { "Memory_Load",    &Commander::Memory_Load,
                            &Commander::can_Memory_Load,
          "Load memory from file"},
        { 0, 0, 0}
    };
}
```

Here, Commander is the name of the commander class defined in the calculator.

Command table is used to initialize the actual command vector, Cmd::VectorExec, which adds functionality to this data structure. The relationship between Cmd::Table and Cmd::VectorExec is analogous to the relationship between Function::Array and Function::Table inside the calculator. As before, this scheme makes it very easy to add new items to the table--new commands to our program.

Cmd::VectorExec has to be a template, for the same reason Cmd::Items have. However, in order not to templatize everything else that makes use of this vector (in particular, the menu system) I derived it from a non-template class, Cmd::Vector, that defines a few pure virtual functions and some generic functionality, like searching commands by name using a map.

The menu provides acces to the command vector. In a dynamic menu system, we initialize the menu from a table. The table is organized hierarchicaly: menu bar items point to popup menus which contain commands. For instance, this is what the initialization table for our calculator menu looks like (notice that command that require further user input--a dialog--are followed by three dots):

```
namespace Menu
{
    const Item programItems [] =
    {
        {CMD,        "&About...", "Program_About"},
        {SEPARATOR,  0,           0},
        {CMD,        "E&xit",     "Program_Exit"},
        {END,        0,           0}
    };
    const Item memoryItems [] =
    {
        {CMD,        "&Clear",    "Memory_Clear"},
        {SEPARATOR,  0,           0},
        {CMD,        "&Save...",  "Memory_Save"},
        {CMD,        "&Load...",  "Memory_Load"},
        {END,        0,           0}
    };
    //---- Menu bar ----
    const BarItem barItems [] =
    {
        {POP,   "P&rogram", "Program",   programItems},
        {POP,   "&Memory",  "Memory",    memoryItems},
        {END,   0,          0,           0}
    };
}
```

Note that each item contains the display name with an embedded ampersand. This ampersand is translated by Windows into a keyboard shortcut (not to be confused with a keyboard accellerator). The ampersand itself is not displayed, but the letter following it will be underlined. The user will then be able to select a given menu item by pressing the key corresponding to that letter while holdnig the Alt key. All items also specify command names--for popup items, these are the same strings that were used in the naming of commands. Menu bar items are also named, but they don't have commands associated with them. Finally, menu bar items have pointers to the corresponding popup tables.

Similar tables can be used for the initialization of accelerators and toolbars.

The actual menu object, of the class Menu::DropDown, is created in the constructor of the View. It is initialized with the table of menu bar items, Menu::barItems, shown above; and a Cmd::Vector object (initialized using Cmd::Table). The rest is conveniently encapsulated in the library.

You might be interested to know that, since a menu is a resource (released using DestroyMenu API), the class Menu::Maker has transfer semantics. For instance, when we create a menu bar, all the popup menus are transfered to Menu::BarMaker, one by one.

But that's not the end of the story. We want to be able to dynamically activate or deactivate particular menu items. We already have Commander methods for testing the availability of particular commands--they are in fact accessible through the command vector. The question remains, what is the best time to call these methods? It turns out that Windows sends a message, WM_INITMENUPOPUP, right before opening up a popup menu. The handler for this message is called OnInitPopup. We can use that opportunity to manipulate the menu while testing for the availability of particular commands. In fact, since the library class Menu::DropDown has access to the command vector, it can implement the RefreshPopup method once and for all. No need for the client to write any additional code.

Displaying short help for each selected menu item is also versy simple. When the user moves the mouse cursor to a popup menu item, Windows sends us the message, WM_MENUSELECT, which we can process in the controller's method, OnMenuSelect. We just call the GetHelp method of the command vector and send the help string to the status bar.

Let's now review the whole task from the point of view of the client. What code must the client write to make use of our dynamic menu system? To begin with, he has to implement the commander, which is just a repository of all commands available in the particular program. Two methods must be implemented for each command: one to execute it and one to test for its availability.

The role of the commander is:

- if required, get data from the user, usually by means of a dialog box
- dispatch the request to the model for execution.

Once the commander is in place, the client has to create and statically initialize a table of commands. In this table all commands are given names and assigned short help strings. This table is then used in the initialization of the command vector.

The menu system is likewise initialized by a table. This table contains command names, display names for menu items and markers differentiating between commands, separators and bar items. Once the menu bar is ready, it has to be attached to the top-level window. However, don't try to attach the menu inside the constructor of View. Both View and Controller must be fully constructed before adding the menu. Menu attachment results in a series of messages sent to the top level window (most notably, to resize its client area), so the whole controller has to be ready to process them in an orderly manner.

Finally, the user must provide a simple implementations of OnInitPopup and, if needed, OnMenuSelect, to refresh a popup menu and to display short help, respectively.

Because major data structures in the menu system are initialized by tables, it is very easy to change them. For instance, reorganizing the menu or renaming menu items requires changes only to a single file--the one that contains the menu table. Modifying the behavior of commands requires only changes to the commander object. Finally, adding a new command can be done in three independent stages: adding the appropriate methods to the commander, adding an entry to the command table, and adding an item to the menu table. It can hardly be made simpler and less error-prone.

Fig 1. shows the relationships and dependencies between various elements of the controller.

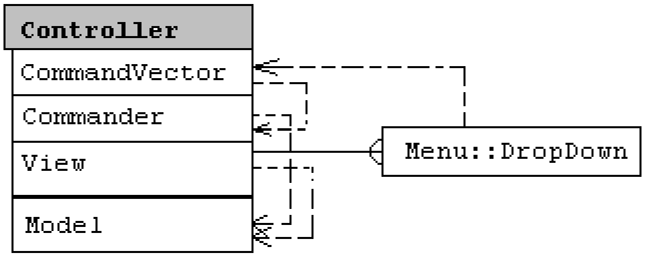


Fig 1. The relationships between various elements of the controller.

Because Commander doesn't have access to View, it has no direct way to force the refreshing of the display after such commands as Memory_Clear or Memory_Load. Again, we can only solve this problem by brute force (refresh memory display after every command) or some kind of notifications. I decided to use the most generic notification mechanism--sending a Windows message. In order to force the clearing of the calculator's memory display, the Commander sends a special user-defined message MSG_MEMCLEAR to the top-level window.

Remember, a message is just a number. You are free to define your own messages, as long as you assign them numbers that won't conflict with any messages used by Windows. There is a special identifier WM_USER which defines a number that is guaranteed to be larger than that of any Windows-specific message.

To process user-defined messages, I added a new handler, OnUserMessage, to the generic Win::Controller. This handler is called whenever the message is larger or equal to WM_USER.

One more change is necessary in order to make the menus work correctly. We have to expand the message loop to call TranslateMessage before DispatchMessage. TranslateMessage filters out these keyboard messages that have to be translated into menu shortcuts and turns them into WM_COMMAND messages. If you are also planning on adding keyboard accelerators (not to be confused with keyboard shortcuts that are processed directly by the menu system)--for instance, Ctrl-L to load memory--you'll have to further expand the message loop to call TranslateAccellerator.

Although we won't discuss *modeless* dialogs here, you might be interested to know that they also require a pre-processing step, the call to IsDialogMessage, in the message loop. It makes sense to stick all these accellerators and modeless dialog handles in a separate preprocessor object, of the class Win::MessagePrepro. It's method Pump enters the message loop and returns only when the top-level window is destroyed. One usually passes the preprocessor object to the top-level controller, to make it possible to dynamically switch accellerator tables or create and destroy modeless dialogs.

Exercises

1. In response to the user's double-clicking on an item in the history pane, copy the selected string into the edit control, so that the user can edit and re-execute it.
2. Add item "Function" to the menu bar. The corresponding popup menu should display the list of available built-in functions. When the user selects one, its name and the opening parentesis should be appended to the string in the edit control.
   Hint: This popup menu should not be initialized statically. It should use the function table from the calculator for its initialization.
3. Add keyboard accelerators for Ctrl-L and Ctrl-S for invoking the Load and Save commands, respectively. Use a statically initialized accelerator table. Pass this table, together with the command vector (for command name to command id translation) to the accellerator maker. The API to create an accelerator table is called `CreateAcceleratorTable`. Since an accellerator table is a resource (released via `DestroyAccelleratorTable`), you'll have to apply resource management in the design of your classes.
   To attach the accellerator, pass a reference to the message preprocessor from `WinMain` to `TopController`. After creating the accellerator, use the `MsgPrepro::SetKbdAccelerator` method to activate it.
   Hint: Remember to change the display string in menu items to include the accellerator key. For instance, the Load item should read, `"&Load...\tCtrl+L"` (the tab marker `\t` right-aligns the accellerator string).
4. Convert the Load command to use `GetOpenFileName` for browsing directories.

Software Project

About Software

- Complexity
- The Fractal Nature of Software
- The Living Project
- The Living Programmer

Design Strategies

- Top-Down Object-Oriented Design
- Model-View-Controller
- Documentation

Team Work

- Productivity
- Team Strategies

Implementation Strategies

- Global Decisions
- Top-Down Object-Oriented Implementation
- Inheriting Somebody Else's Code
- Multi-Platform Development
- Program Modifications
- Testing

About Software

Complexity

Dealing with complexity, the finite capacity of human mind, divide and conquer, abstraction.

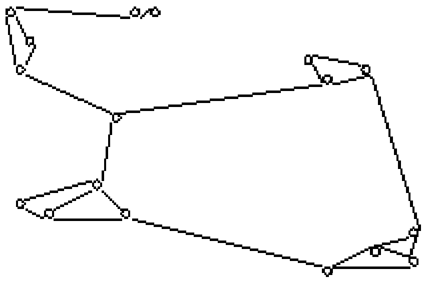
Dealing with complexity is the essence of software engineering. It is also the most demanding part of it, requiring both discipline and creativity. Why do we need special methodologies to deal with complexity? The answer is in our brains. In our immediate memory we can deal only with a finite and rather small number of objects--whatever type they are, ideas, images, words. The ballpark figure is seven plus/minus two, depending on the complexity of the objects themselves. Apparently in many ancient cultures the number *seven* was considered synonymous with *many*. There are many folk stories that start with "Long, long ago behind seven mountains, behind seven forests, behind seven rivers there lived..."

There are essentially two ways in which we human beings can deal with complexity. The divide-and-conquer method, and the abstraction method. The divide-and-conquer methods is based on imposing a tree-like structure on top of a complex problem. The idea is that at every node of the tree we have to deal with only a small number of branches, within the limits of our immediate memory. The traversal of the tree leaf-to-root or root-to-leaf requires only a logarithmic number of steps-- again, presumably within the limits of our immediate memory. For instance, the body of academic knowledge is divided into humanities and sciences (branching factor of 2). Sciences are subdivided into various areas, one of them being Computer Science, and so on.

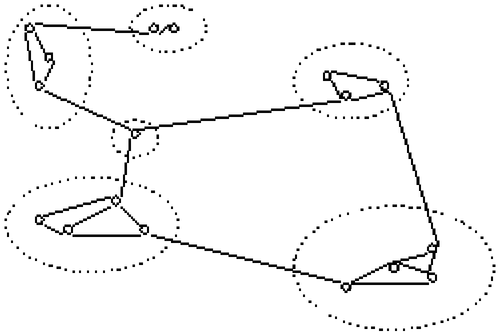
To understand Kernighan and Ritchie's book on C, the CS student needs only very limited education in humanities. On the other hand, to write a poem one is not required to program in C. The tree-like subdivision of human knowledge not only facilitates in-depth traversal and search, it also enables division of work between various teams. We can think of the whole humanity as one large team taking part in the enormous project of trying to understand the World.

Another very powerful tool developed by all living organisms and perfected by humans is abstraction. The word "abstraction" has the same root as subtraction. Abstracting means subtracting non-essential features. Think of how many features you can safely subtract from the description of your car before it stops being recognizable as a car. Definitely the color of the paint, the license plates, the windshield wipers, the capacity of the trunk, etc. Unconsciously the same process is applied by a bird when it creates its definition of a "predator." Abstraction is not 100% accurate: a crow may get scared by a scarecrow, which somehow falls within its abstract notion of a "predator."

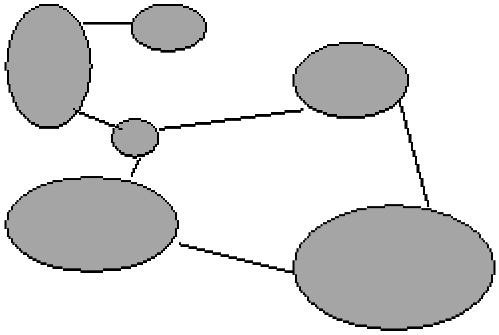
Division and abstraction go hand in hand in, what one can call, divide-and-abstract paradigm. A complex system can be visualized as a very large network of interconnected nodes. We divide this network into a few "objects"--subsets of nodes. A good division has the property that there are as few inter-object connections as possible. To describe the objects resulting from such a division we use abstraction. In particular, we can describe the objects by the way they connect to other objects (the interface). We can simplify their inner structure by subtracting as many inessential features as possible. At every stage of division, it should be possible to understand the whole system in terms of interactions between a few well abstracted objects. If there is no such way, we give up. The real miracle of our World is that large portions of it (maybe even everything) can be approached using this paradigm.



A complex system.

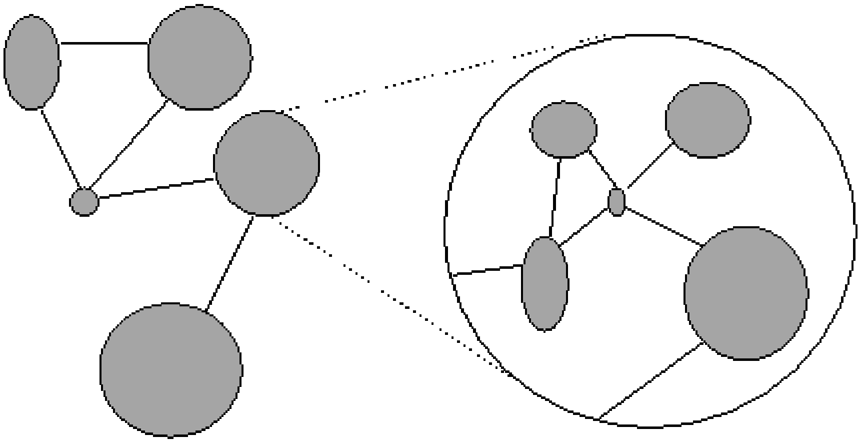


Abstracting objects out of a complex system.



The high level view of the complex system after abstracting objects.

This process is then repeated recursively by dividing objects into sub-objects, and so on. For every object, we first undo the abstraction by adding back all the features we have subtracted, divide it into sub-objects and use new abstractions to define them. An object should become understandable in terms of a few well abstracted sub-objects. In some way this recursive process creates a self-similar, fractal-like structure.

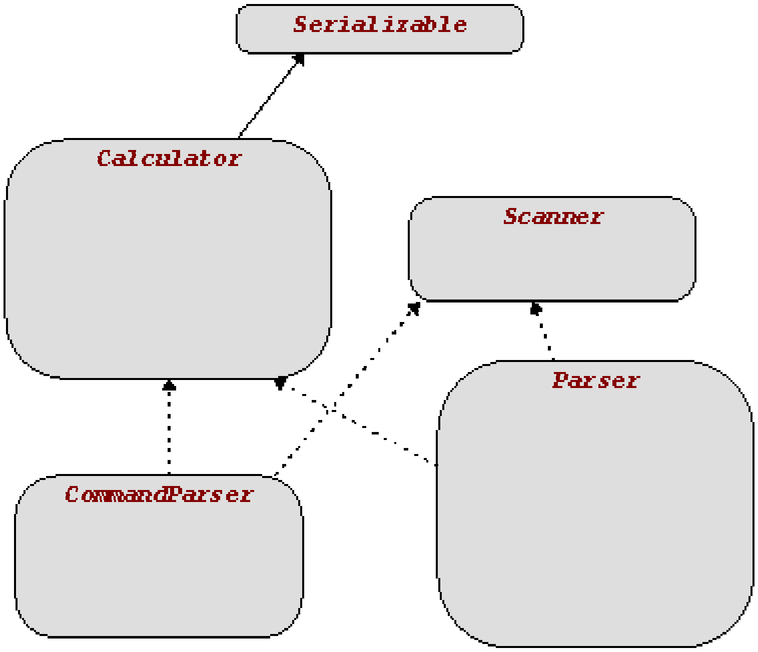


The fractal structure of a complex systems.

In software engineering we divide a large project into manageable pieces. In order to define, name and describe these pieces we use abstraction. We can talk about symbol tables, parsers, indexes, storage layers, etc. They are all abstractions. And they let us divide a bigger problem into smaller pieces.

The Fractal Nature of Software

Let me illustrate these ideas with the familiar example of the software project that we've been developing in the second part of the book--the calculator. The top level of the project is structured into a set of interrelated objects, Figure.



Top level view of the calculator project.

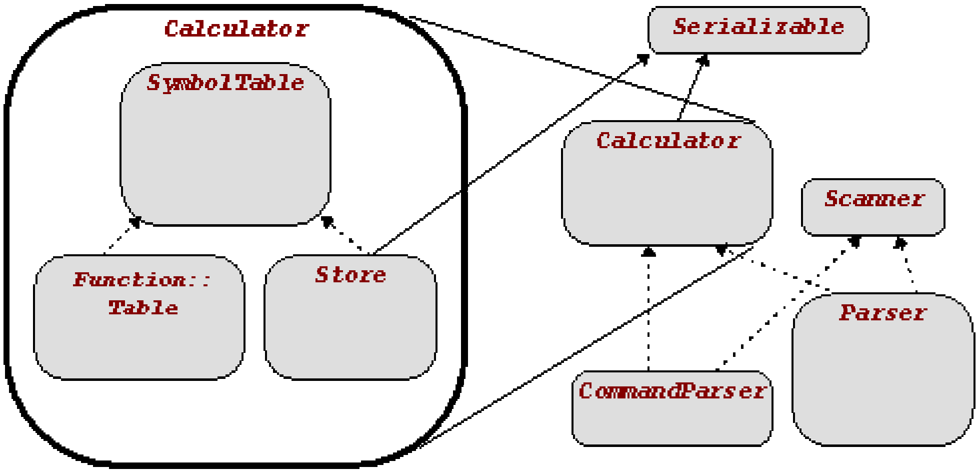
This system is closed in the sense that one can explain how the program works (what the function main does) using only these objects--their public interfaces and their functionality. It is not necessary to know *how* these object perform their functions; it is enough to know *what* they do.

So how does the program work? First, the Calculator is created inside main. The Calculator is Serializable, that means that its state can be saved and restored. Notice that, at this level, we don't need to know anything about the streams--they are black boxes with no visible interface (that's why I didn't include them in this picture).

Once the Calculator is created, we enter the loop in which we get a stream of text from the user and create a Scanner from it. The Scanner can tell us whether the user input is a command or not. If it is a command, we create a CommandParser, otherwise we create a Parser. Either of them requires access to both the Calculator and the Scanner. CommandParser can Execute a command, whereas Parser can Parse the input and Calculate the result. We then display the result and go back to the beginning of the loop. The loop terminates when CommandParser returns status stQuit from the Execute method.

That's it! It could hardly be simpler than that. It's not easy, though, to come up with such a nice set of abstraction on the first try. In fact we didn't! We had to go through a series of rewrites in order to arrive at this simple structure. All the techniques and little rules of thumb described in the second part of the book had this goal in mind.

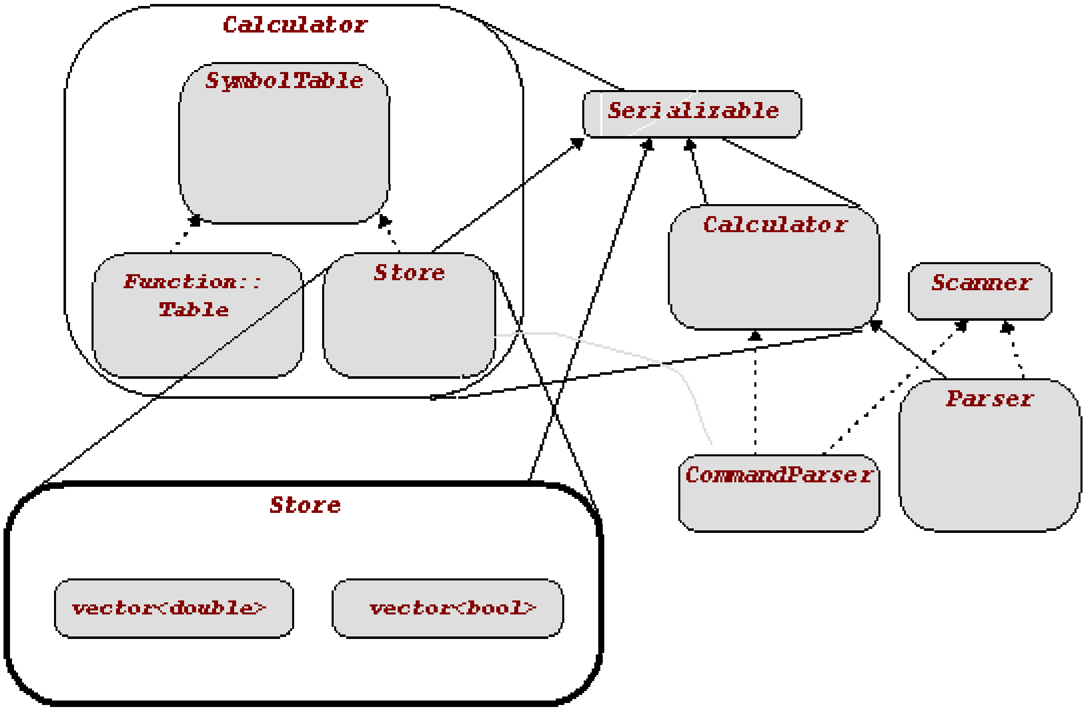
But let's continue the journey. Let's zoom-in into one of the top level components--the Calculator. Again, it can be described in terms of a set of interrelated objects, Figure.



The result of zooming-in on the Calculator.

And again, I could explain the implementation of all Calculator methods using only these objects (and a few from the level above).

Next, I could zoom-in on the Store object and see a very similar picture.



The result of zooming-in on Store.

I could go on like this, just like in one of these Mandelbrot set programs, where you can zoom-in on any part of the picture and see something that is different and yet similar. With a mathematical fractal, you can keep zooming-in indefinitely and keep seeing the same infinite level of detail. With a software project, you will eventually get to the level of plain built-in types and commands. (Of course, you may continue zooming-in into assembly language, microcode, gates, transistors, atoms, quarks, superstrings and further, but that's beyond the scope of this book.)

The Living Project

The lifetime of the project, cyclic nature of programming, the phases, open-ended design, the program as a living organism.

Every software project has a beginning. Very few have an end (unless they are cancelled by the management). You should get used to this kind of open-ended development. You will save yourself and your coworkers a lot of grief . Assume from the very beginning that:

- New features will be added,
- Parts of the program will be rewritten,
- Other people will have to read, understand, and modify your code,
- There will be version 2.0 (and further).

Design for version 2, implement for version 1. Some of the functionality expected in v. 2 should be stubbed out in v. 1 using dummy components.

The development of a software project consists of cycles of different magnitude. The longest scale cycle is the major version cycle. Within it we usually have one or more minor version cycles. The creation of a major version goes through the following stages:

- Requirement (or external) specification,

- Architectural design (or re-design),
- Implementation,
- Testing and bug fixing.

Time-wise, these phases are interlaced. Architectural design feeds back into the requirements spec. Some features turn out to be too expensive, the need for others arises during the design.

Implementation feeds back into the design in a major way. Some even suggest that the development should go through the cycles of implementation of throw-away prototypes and phases of re-design. Throwing away a prototype is usually too big a waste of development time. It means that too little time was spent designing and studying the problem, and that the implementation methodology was inadequate.

One is *not* supposed to use a different methodology when designing and implementing prototypes, scaffolding or stubs--as opposed to designing and implementing the final product. Not following this rule is a sign of hypocrisy. Not only is it demoralizing, but it doesn't save any development time. Quite the opposite! My fellow programmers and I were bitten by bugs or omissions in the scaffolding code so many times, and wasted so much time chasing such bugs, that we have finally learned to write scaffolding the same way we write production code. As a side effect, whenever the scaffolding survives the implementation cycle and gets into the final product (you'd be surprised how often that happens!), it doesn't lead to any major disasters.

Going back to the implementation cycle. Implementing or rewriting any major component has to be preceded by careful and detailed design or re-design. The documentation is usually updated in the process, little essays are added to the architectural spec. In general, the design should be treated as an open-ended process. It is almost always strongly influenced by implementation decisions. This is why it is so important to have the discipline to constantly update the documentation. Documentation that is out of sync with the project is useless (or worse than useless--it creates misinformation).

The implementation proper is also done in little cycles. These are the fundamental edit-compile-run cycles, well known to every programmer. Notice how testing is again interlaced with the development. The *run* part of the cycle serves as a simple sanity test.

At this level, the work of a programmer resembles that of a physician. The first principle-- never harm the patient--applies very well to programming. It is called "don't break the code." The program should be treated like a living organism. You have to keep it alive at all times. Killing the program and then resuscitating it is not the right approach. So make all changes in little steps that are self-contained and as much testable as possible. Some functionality may be temporarily disabled when doing a big "organ transplant," but in general the program should be functional at all times.

Finally, a word of caution: How *not* to develop a project (and how it is still done in many places). Don't jump into implementation too quickly. Be patient. Resist the pressure from the managers to have something for a demo as soon as possible. Think before you code. Don't sit in front of the computer with only a vague idea of what you want to do with the hope that you'll figure it out by trial and error. Don't write sloppy code "to be cleaned up later." There is a big difference between stubbing out some functionality and writing sloppy code.

The Living Programmer

Humility, simplicity, team spirit, dedication.

A programmer is a human being. Failing to recognize it is a source of many misunderstandings. The fact that the programmer interacts a lot with a computer doesn't mean that he or she is any less human. Since it is the computer that is supposed to serve the humans and not the other way around, programming as an activity should be organized around the humans. It sounds like a truism, but you'd be surprised how often this simple rule is violated in real life. Forcing people to program in assembly (or C for that matter) is just one example. Structuring the design around low level data structures like hash tables, linked lists, etc., is another example.

The fact that jobs of programmers haven't been eliminated by computers (quite the opposite!) means that being human has its advantages. The fact that some human jobs have been eliminated by computers means that being computer has its advantages. The fundamental equation of software engineering is thus

Human Creativity + Computer Speed and Reliability = Program

Trying to write programs combining human speed and reliability with computer creativity is a big mistake! So let's face it, we humans are slow and unreliable. When a programmer has to wait for the computer to finish compilation, something is wrong. When the programmer is supposed to writhe error-free code without any help from the compiler, linker or debugger, something is wrong. If the programmer, instead of solving a problem with paper and pencil, tries to find the combination of parameters that doesn't lead to a general protection fault by trial and error, something is badly wrong.

The character traits that make a good programmer are (maybe not so surprisingly) similar to those of a martial art disciple. Humility, patience, simplicity on the one hand; dedication and team spirit on the other hand. And most of all, mistrust towards everybody including oneself.

- Humility: Recognize your shortcomings. It is virtually impossible for a human to write error-free code. We all make mistakes. You should write code in anticipation of mistakes. Use any means available to men and women to guard your code against your own mistakes. Don't be stingy with assertions. Use heap checking. Take time to add debugging output to your program.
- Patience: Don't rush towards the goal. Have patience to build solid foundations. Design before you code. Write solid code for future generations.
- Simplicity: Get rid of unnecessary code. If you find a simpler solution, rewrite the relevant parts of the program to make use of it. Every program can be simplified. Try to avoid special cases.
- Dedication: Programming is not a nine-to-five job. I am not saying that you should work nights and weekends. If you are, it is usually a sign of bad management. But you should expect a lifetime of learning. You have to grow in order to keep up with the tremendous pace of progress. If you don't grow, you'll be left behind by the progress of technology.
- Team spirit: Long gone are the times of the Lone Programmer. You'll have to work in a team. And that means a lot. You'll have to work on your communication skills. You'll have to accept certain standards, coding conventions, commenting conventions, etc. Be ready to discuss and change some of the conventions if they stop making sense. Some people preach the idea that "A stupid convention is better than no convention." Avoid such people.
- Programmer's paranoia: Don't trust anybody's code, not even your own.

Design Strategies

Top-Down Object Oriented Design

Top level objects, abstractions and metaphors, components.

It is all too easy to start the design by coming up with such objects as hash tables, linked lists, queues, trees, and trying to put them together. Such an approach, bottom-up, implementation driven, should be avoided. A program that is built bottom-up ends up with a structure of a soup of objects. There are pieces of vegetables, chunks of meat, noodles of various kinds, all floating in some kind of broth. It sort of looks object oriented--there are "classes" of noodles, vegetables, meat, etc. However, since you rarely change the implementation of linked lists, queues, trees, etc., you don't gain much from their object-orientedness. Most of the time you have to maintain and modify the shapeless soup.

Using the top-down approach, on the other hand, you divide your program into a small number of interacting high-level objects. The idea is to deal with only a few types of objects--classes (on the order of seven plus/minus two--the capacity of our short-term memory!). The top-level objects are divided into the main actors of the program, and the communication objects that are exchanged between the actors. If the program is interactive, you should start with the user interface and the objects that deal with user input and screen- (or teletype-) output.

Once the top level objects are specified, you should go through the exercise of rehearsing the interactions between the objects (this is sometimes called going through use-case scenarios). Go through the initialization process, decide which object have to be constructed first, and in what state they should start. You should avoid using global objects at any level other than possibly the top level. After everything has been initialized, pretend that you are the user, see how the objects react to user input, how they change their state, what kind of communication objects they exchange. You should be able to describe the interaction at the top without having to resort to the details of the implementation of lower levels.

After this exercise you should have a pretty good idea about the interfaces of your top-level objects and the contracts they have to fulfill (that is, what the results of a given call with particular arguments should be). Every object should be clearly defined in as few words as possible, its functionality should form a coherent and well rounded abstraction. Try to use common language, rather than code, in your documentation, in order to describe objects and their interactions. Remember, center the project around humans, not computers. If something can be easily described in common language, it usually is a good abstraction.

For things that are not easily abstracted use a ***metaphor***. An editor might use a metaphor of a sheet of paper; a scheduler a metaphor of a calendar; a drawing program, metaphors of pencils, brushes, erasers, palettes, etc. The design of the user interface revolves around metaphors, but they also come in handy at other levels of design. Files, streams, semaphores, ports, pages of virtual memory, trees, stacks--these are all examples of very useful low-level metaphors.

The right choice of abstractions is always important, but it becomes absolutely crucial in the case of a large software project, where top-level objects are implemented by separate teams. Such objects are called ***components***. Any change to the component's interface or its contract, once the development started going full steam ahead, is a disaster. Depending on the number of components that use this particular interface, it can be a minor or a major disaster. The magnitude of such a disaster can only be measured in Richter scale. Every project goes through a few such "earthquakes"--that's just life!

Now, repeat the same design procedure with each of the top-level objects. Split it into sub-objects with well defined purpose and interface. If necessary, repeat the procedure for the sub-objects, and so on, until you have a pretty detailed design. Use this procedure again and again during the implementation of various pieces. The goal is to superimpose some sort of a self-similar, fractal structure on the project. The top level description of the whole program should be similar to the description of each of the components, its sub-components, objects, sub-objects, etc. Every time you zoom-in or zoom-out, you should see more or less the same type of picture, with a few self-contained objects collaborating towards implementing some well-defined functionality.

Model-View-Controller

Designing user interface, input driven programs, Model-View-Controller paradigm

Even the simplest modern-day programs offer some kind of interactivity. Of course, one can still see a few remnants of the grand UNIX paradigm, where every program was written to accept a one dimensional stream of characters from its standard input and spit out another stream at its standard output. But with the advent of the Graphical User Interface (GUI), the so-called "command-line interface" is quickly becoming extinct. For the user, it means friendlier, more natural interfaces; for the programmer it means more work and a change of philosophy.

With all the available help from the operating system and with appropriate tools at hand it isn't difficult to design and implement user interfaces, at least for graphically non-demanding programs. What is needed is a change of perspective. An interactive program is, for the most part, ***input-driven***. Actions in the program happen in response to user input. At the highest level, an interactive program can be seen a series of event handlers for externally generated events. Every key press, every mouse click has to be handled appropriately.

The object-oriented response to the interactive challenge is the Model-View-Controller paradigm first developed and used in Smalltalk. The ***Controller*** object is the focus of all external (and sometimes internal as well) events. Its role is to interpret these events as much as is necessary to decide which of the program objects will have to handle them. Appropriate messages are then sent to such objects (in Smalltalk parlance; in C++ we just call appropriate methods).

The ***View*** takes care of the program's visual output. It translates requests from other objects into graphical representations and displays them. In other words it abstracts the output. Drawing lines, filling areas, writing text, showing the cursor, are some of the many responsibilities of the View.

Centralizing input in the Controller and output in the View leaves the rest of the program independent from the intricacies of the input/output system (also makes the program easy to port to other environments with slightly different graphical capabilities and interfaces). The part of the program that is independent of the details of input and output is called the ***Model***. It is the hard worker and the brains of the program. In simple programs, the Model corresponds to a single object, but quite often it is a collection of top level objects. Various parts of the Model are activated by the Controller in response to external events. As a result of changes of state, the Model updates the View whenever it finds it appropriate.

As a rule, you should start the top-down design of an interactive program by establishing the functionality of the Controller and the View. Whatever happens prior to any user action is considered initialization of these components and the model itself. The M-V-C triad may also reappear at lower levels of the program to handle a particular type of control, a dialog box, an edit control, etc.

Documentation

Requirement Specification

Statement of purpose, functionality, user interface, input, output, size limitations and performance goals, features, compatibility.

The first document to be written before any other work on a project may begin is the Requirement Specification (also called an External Specification). In large projects the Requirement Spec might be prepared by a dedicated group of people with access to market research, user feedback, user tests, etc. However, no matter who does it, there has to be a feedback loop going back from the architects and implementers to the group responsible for the Requirement Spec.

The crucial part of the spec is the statement of ***purpose***--what the purpose of the particular software system is. Sometimes restating the purpose of the program might bring some new insights or change the focus of the design. For instance, describing a compiler as a program which checks the source file for errors, and which occasionally creates an object file (when there are no errors), might result in a competitively superior product.

The statement of purpose might also contain a discussion of the key metaphor(s) used in the program. An editor, for instance, may be described as a tool to manipulate lines of text. Experience however has shown that editors that use the metaphor of a sheet of paper are superior. Spreadsheet programs owe their popularity to another well chosen metaphor.

Then, a detailed description of the ***functionality*** of the program follows. In a word-processor requirement spec one would describe text input, ways of formatting paragraphs, creating styles, etc. In a symbolic manipulation program one would specify the kinds of symbolic objects and expressions that are to be handled, the various transformations that could be applied to them, etc. This part of the spec is supposed to tell the designers and the implementers what functionality to implement. Some of it is described as mandatory, some of it goes into the wish list.

The ***user interface*** and visual metaphors go next. This part usually undergoes most extensive changes. When the first prototype of the interface is created, it goes through more or less (mostly less) rigorous testing, first by developers, then by potential users. Sometimes a manager doesn't like the feel of it and sends programmers back to the drawing board. It is definitely more art than science, yet a user interface may make or break a product.

What compounds the problem is the fact that anybody may criticize user interface. No special training is required. And everybody has different tastes. The programmers that implement it are probably the least qualified people to judge it. They are used to terse and cryptic interfaces of their programming tools, grep, make, link, Emax or vi. In any case, designing user interface is the most frustrating and ungrateful job.

Behind the user interface is the ***input/output*** specification. It describes what kind of input is accepted by the program, and what output is generated by the program in response to this input. For instance, what is supposed to happen when the user clicks on the format-brush button and then clicks on a word or a paragraph in a document (the formatting should be pasted over). Or what happens when the program reads a file that contains a comma-separated lists of numbers. Or what happens when a picture is pasted from the clipboard.

Speed and size requirements may also be specified. The kind of processor, minimum memory configuration, and disk size are often given. Of course there is always conflict between the ever growing list of desired features and the always conservative hardware requirements and breathtaking performance requirements. (Features always win! Only when the project enters its critical phase, features get decimated.)

Finally, there may be some ***compatibility*** requirements. The product has to understand (or convert) files that were produced either by its earlier versions, or by competitors' products, or both. It is wise to include some compatibility features that will make future versions of the product easier to implement (version stamps are a must).

Architecture Specification

Top level view, crucial data structures and algorithms. Major implementation choices.

The architectural document describes how things work and why they work the way they work. It's a good idea to either describe theoretical foundations of the system, or at least give pointers to some literature.

This is the document that gives the top level view of the product as a program, as seen by the developers. All top level components and their interactions are described in some detail. The document should show it clearly that, if the top level components implement their functionality according to their specifications, the system will work correctly. That will take the burden off the shoulders of developers--they won't have to think about too many dependencies.

The architectural spec defines the major data structures, especially the persistent ones. The document then proceeds with describing major event scenarios and various states of the system.

The program may start with an empty slate, or it may load some history (documents, logs, persistent data structures). It may have to finish some transactions that were interrupted during the last session. It has to go through the initialization process and presumably get into some quiescent state.

External or internal events may cause some activity that transforms data structures and leads to state transitions. New states have to be described. In some cases the algorithms to be used during such activity are described as well. Too detailed a description of the implementation should be avoided. It becomes obsolete so quickly that it makes little sense to try to maintain it.

Once more we should remind ourselves that the documentation is a living thing. It should be written is such a way that it is easy to update. It has to have a sensible structure of its own, because we know that it will be changed many times during the implementation cycle. In fact it *should* be changed, and it is very important to keep it up-to-date and encourage fellow developers to look into it on a regular basis.

In the implementation cycle, there are times when it is necessary to put some flesh into the design of some important object that has only been sketched in the architectural spec. It is time to either expand the spec or write short essays on selected topics in the form of separate documents. In such essays one can describe non-trivial implementations, algorithms, data structures, programming notes, conventions, etc.

Team Work

Productivity

Communication explosion, vicious circle.

The life of a big-team programmer is spent

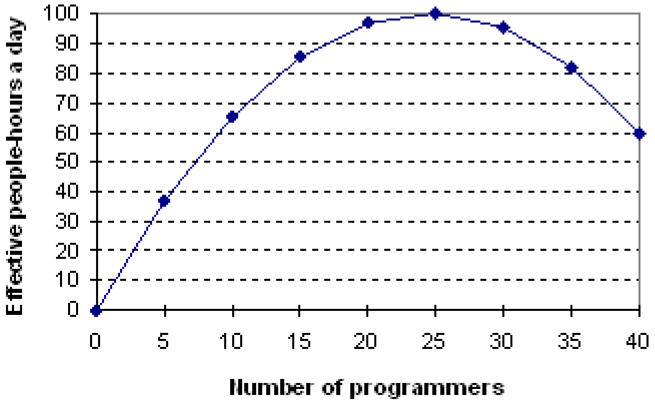
- Communicating with other programmers, attending meetings, reading documents, reading email, responding to it,
- Waiting for others to finish their job, to fix a bug, to implement some vital functionality, to finish building their component,
- Fighting fires, fixing build breaks, chasing somebody else's bugs,
- Staring at the computer screen while the machine is compiling, loading a huge program, running test suits, rebooting--and finally, when time permits--
- Developing new code.

There are n(n-1)/2 possible connections between n dots. That's of the order of O(n^2). By the same token, the number of possible ***interactions*** within a group of n programmers is of the order of O(n^2). The number of hours they can put out is of the order of O(n). It is thus inevitable that at some point, as the size of the group increases, its members will start spending all their time communicating. In real life, people come up with various communication-overload defense mechanisms. One defense is to ignore incoming messages, another is to work odd hours (nights, weekends), when there aren't that many people around to disturb you (wishful thinking!).

As a programmer you are constantly bombarded with information from every possible direction. They will broadcast messages by email, they will drop printed documents in your mailbox, they will invite you to meetings, they will call you on the phone, or in really urgent cases they will drop by your office or cubicle and talk to you directly.

If programmers were only to write and test code (and it used to be like this not so long ago) the market would be flooded with new operating systems, applications, tools, games, educational programs, and so on, all at ridiculously low prices. As a matter of fact, almost all public domain and shareware programs are written by people with virtually no communication overhead.

The following chart shows the results of a very simple simulation. I assumed that every programmer spends 10 minutes a day communicating with every other programmer in the group. The rest of the time he or she does some real programming. The time spent programming, multiplied by the number of programmers in the group, measures team productivity-- the effective work done by the group every day. Notice that, under these assumptions, the effective work peaks at about 25 people and then starts decreasing.



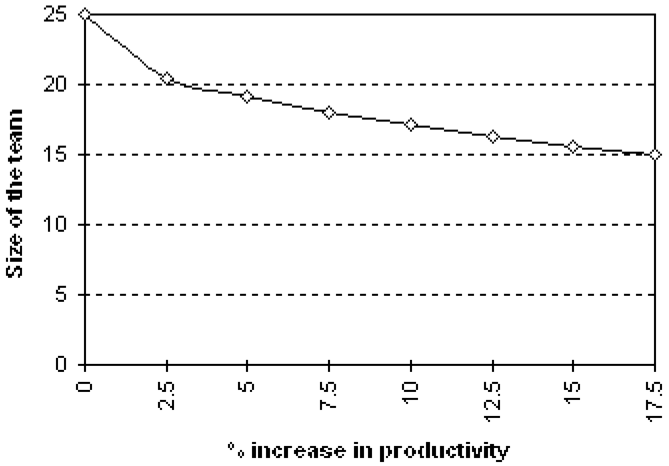
But wait, there's more! The more people you have in the group, the more complicated the dependency graph. Component A cannot be tested until component B works correctly. Component B needs some new functionality from C. C is blocked waiting for a bug fix in D. People are waiting, they get frustrated, they send more email messages, they drop by each other's offices.

Not enough yet? Consider the reliability of a system with n components. The more components, the more likely it is that one of them will break. When one component is broken, the testing of other components is either impossible or at least impaired. That in turn leads to more buggy code being added to the project causing even more breakages. It seems like all these mechanisms feed on each other in one vicious circle.

In view of all this, the team's productivity curve is much too optimistic.

The other side of the coin is that raising the productivity of a programmer, either by providing better tools, better programming language, better computer, or more help in non-essential tasks, creates a positive feedback loop that amplifies the productivity of the team. If we could raise the productivity of every programmer in a fixed size project, we could reduce the size of the team-- that in turn would lead to decreased communication overhead, further increasing the effective productivity of every programmer. Every dollar invested in programmer's productivity saves several dollars that would otherwise be spent hiring other programmers.

Continuing with our simple simulation--suppose that the goal is to produce 100,000 lines of code in 500 days. I assumed the starting productivity of 16 lines of code per day per programmer, if there were no communication overhead. The following graph shows how the required size of the team shrinks with the increase in productivity.



Notice that, when the curve turns more or less linear (let's say at about 15 programmers), every 3% increase in productivity saves us one programmer, who can then be used in another project.

Several things influence productivity:

- The choice of a programming language and methodology. So far it is hard to beat C++ and object oriented methodology. If the size or speed is not an issue, other specialized higher-level languages may be appropriate (Smalltalk, Prolog, Basic, etc.). The tradeoff is also in the initial investment in the education of the team.
- The choice of project-wide conventions. Such decisions as whether to use exceptions, how to propagate errors, what is the code sharing strategy, how to deal with project-wide header files, etc., are all very difficult to correct during the development. It is much better to think about them up front.
- The choice of a programming environment and tools.
- The choice of hardware for the development. RAM and disk space are of crucial importance. Local Area Network with shared printers and email is a necessity. The need for large monitors is often overlooked.

Team Strategies

In the ideal world we would divide work between small teams and let each team provide a clear and immutable interface to which other teams would write their code. We would couple each interface with a well defined, comprehensive and detailed contract. The interaction between teams would be reduced to the exchange of interface specifications and periodic updates as to which part of the contract has already been implemented.

This ideal situation is, to some extent, realized when the team uses externally created components, such as libraries, operating system API's (application programmer interfaces), etc. Everything whose interface and contract can be easily described is a good candidate for a library. For instance, the string manipulation library, the library of container objects, iostreams, etc., they are all well described either in on-line help, or in compiler manuals, or in other books. Some API's are not that easily described, so using them is often a matter of trial and error (or customer support calls).

In real world, things are more complicated than that. Yes, we do divide work between small teams and they do try to come up with some interfaces and contracts. However, the interfaces are far from immutable, the contracts are far from comprehensive, and they are being constantly re-negotiated between teams. All we can do is to try to smooth out this process as much as possible.

You have to start with a good design. Spend as much time as necessary on designing good hierarchical structure of the product. This structure will be translated into the hierarchical structure of teams. The better the design, the clearer the interfaces and contracts between all parts of the product. That means fewer changes and less negotiating at the later stages of the implementation.

Divide the work in clear correspondence to the structure of the design, taking into account communication needs. Already during the design, as soon as the structure crystallizes, assign team leaders to all the top level components. Let them negotiate the contracts between themselves. Each team leader in turn should repeat this procedure with his or her team, designing the structure of the component and, if necessary, assigning sub-components to leaders of smaller teams.

The whole design should go through several passes. The results of lower level design should serve as feedback to the design of higher level components, and eventually contribute to the design of the whole product. Each team writes its own part of the specification. These specifications are reviewed by other teams responsible for other parts of the same higher level component.

The more negotiating is done up front, during the design, the better the chances of smooth implementation. The negotiations should be structured in such a way that there are only a few people involved at a time. A "plenary" meeting is useful to describe the top level design of the product to all members of all teams, so that everybody knows what the big picture is. Such meetings are also useful during the implementation phase to monitor the progress of the product. They are *not* a good forum for design discussions.

> Contract negotiations during implementation usually look like this: Some member of team A is using one of the team B's interfaces according to his or her understanding of the contract. Unexpectedly the code behind the interface returns an error, asserts, raises an exception, or goes haywire. The member of team A either goes to team B's leader to ask who is responsible for the implementation of the interface in question, or directly to the implementor of this interface to ask what caused the strange behavior. The implementor either clarifies the contract, changes it according to the needs of team A, fixes the implementation to fulfill the contract, or takes over the tracing of the bug. If a change is made to component B, it has to be thoroughly tested to see if it doesn't cause any unexpected problems for other users of B.

During the implementation of some major new functionality it may be necessary to ask other teams to change or extend their interfaces and/or contracts. This is considered a ***re-design***. A re-design, like any other disturbance in the system, produces concentric waves. The feedback process, described previously in the course of the original design, should be repeated again. The interface and contract disturbance are propagated first within the teams that are immediately involved (so that they make sure that the changes are indeed necessary, and to try to describe these changes in some detail.), then up towards the top. Somewhere on the way to the top the disturbances of the design may get dumped completely. Or they may reach the very top and change the way top level objects interact. At that point the changes will be propagated downwards to all the involved teams. Their feedback is then bounced back towards the top, and the process is repeated as many times as is necessary for the changes to stabilize themselves. This "annealing" process ends when the project reaches a new state of equilibrium.

Implementation Strategies

Global Decisions

Error handling, exceptions, common headers, code reuse, debug output.

The biggest decision to be made, before the implementation can even begin, is how to handle errors and exceptions. There are a few major sources of errors

- Bug within the component,
- Incorrect parameters passed from other (trusted) component,
- Incorrect user input,
- Corruption of persistent data structures,
- System running out of resources,

Bugs are not supposed to get through to the final retail version of the program, so we have to deal with them only during development. (Of course, in practice most retail programs still have some residual bugs.) Since during development we mostly deal with debug builds, we can protect ourselves from bugs by sprinkling our code with assertions. Assertions can also be used to enforce contracts between components.

User input, and in general input from other less trusted parts of the system, must be thoroughly tested for correctness before proceeding any further. "Typing monkeys" tests have to be done to ensure that no input can break our program. If our program provides some service to other programs, it should test the validity of externally passed arguments. For instance, operating system API functions always check the parameters passed from applications. This type of errors should be dealt with on the spot. If it's direct user input, we should provide immediate feedback; if it's the input from an untrusted component, we should return the appropriate error code.

Any kind of persistent data structures that are not totally under our control (and that is always true, unless we are a file system; and even then we should be careful) can always get corrupted by other applications or tools, not to mention hardware failures and user errors. We should therefore always test for their consistency. If the corruption is fatal, this kind of error is appropriate for turning it into an exception. A common programming error is to use assertions to enforce the consistency of data structures read from disk. Data on disk should never be trusted, therefore all the checks must also be present in the retail version of the program.

Running out of resources-- memory, disk space, handles, etc.-- is the prime candidate for exceptions. Consider the case of memory. Suppose that all programmers are trained to always check the return value of operator new (that's already unrealistic). What are we supposed to do when the returned pointer is null? It depends on the situation. For every case of calling new, the programmer is supposed to come up with some sensible recovery. Consider that the recovery path is rarely tested (unless the test team has a way of simulating such failures). We take up a lot of programmers' time to do something that is as likely to fail as the original thing whose failure we were handling.

The simplest way to deal with out-of-memory situations is to print a message "Out of memory" and exit. This can be easily accomplished by setting our own out-of-memory handler (`_set_new_handler` function in C++). This is however very rarely the desired solution. In most cases we at least want to do some cleanup, save some user data to disk, maybe even get back to some higher level of our program and try to continue. The use of exceptions and resource management techniques (described earlier) seems most appropriate.

If C++ exception handling is not available or prohibited by managers, one is left with conventional techniques of testing the results of new, cleaning up and propagating the error higher up. Of course, the program must be thoroughly tested using simulated failures. It is this kind of philosophy that leads to project-wide conventions such as "every function should return a status code." Normal return values have then to be passed by reference or a pointer. Very soon the system of status codes develops into a Byzantine structure. Essentially every error code should not only point at the culprit, but also contain the whole history of the error, since the interpretation of the error is enriched at each stage through which it passes. The use of constructors is then highly restricted, since these are the only functions that cannot return a value. Very quickly C++ degenerates to "better C."

Fortunately most modern C++ compilers provide exception support and hopefully soon enough this discussion will only be of historical interest.

Another important decision to be made up front is the choice of project-wide debugging conventions. It is extremely handy to have progress and status messages printed to some kind of a debug output or log.

The choice of directory structure and build procedures comes next. The structure of the project and its components should find its reflection in the directory structure of source code. There is also a need for a place where project-wide header files and code could be deposited. This is where one puts the debugging harness, definitions of common types, project-wide parameters, shared utility code, useful templates, etc.

Some degree of code reuse within the project is necessary and should be well organized. What is usually quite neglected is the need for information about the availability of reusable code and its documentation. The information about what is available in the reusability department should be broadcast on a regular basis and the up-to-date documentation should be readily available.

One more observation-- in C++ there is a very tight coupling between header files and implementation files-- we rarely make modifications to one without inspecting or modifying the other. This is why in most cases it makes sense to keep them together in the same directory, rather than is some special include directory. We make the exception for headers that are shared between directories.

It is also a good idea to separate platform dependent layers into separate directories. We'll talk about it soon.

Top-Down Object Oriented Implementation

The implementation process should model the design process as closely as possible. This is why implementation should start with the top level components. The earlier we find that the top level interfaces need modification, the better. Besides, we need a working program for testing as soon as possible.

The goal of the original implementation effort is to test the flow of control, lifetimes and accessibility of top level objects as well as initialization and shutdown processes. At this stage the program is not supposed to do anything useful, it cannot be demoed, it is not a prototype. If the management needs a prototype, it should be implemented by a separate group, possibly using a different language (Basic, Smalltalk, etc.). Trying to reuse code written for the prototype in the main project is usually a big mistake.

Only basic functionality, the one that's necessary for the program to make progress, is implemented at that point. Everything else is stubbed out. Stubs of class methods should only print debugging messages and display their arguments if they make sense. The debugging and error handling harness should be put in place and tested.

If the program is interactive, we implement as much of the View and the Controller as is necessary to get the information flowing towards the model and showing in some minimal view. The model can be stubbed out completely.

Once the working skeleton of the program is in place, we can start implementing lower level objects. At every stage we repeat the same basic procedure. We first create stubs of all objects at that level, test their interfaces and interactions. We continue the descent until we hit the bottom of the project, at which point we start implementing some "real" functionality. The goal is for the lowest level components to fit right in into the whole structure. They should snap into place, get control when appropriate, get called with the right arguments, return the right stuff.

This strategy produces professional programs of uniform quality, with components that fit together very tightly and efficiently like in a well designed sports car. Conversely, the bottom-up implementation creates programs whose parts are of widely varying quality, put together using scotch tape and strings. A lot of programmer's time is spent trying to fit square pegs into round holes. The result resembles anything but a well designed sports car.

Inheriting Somebody Else's Code

In the ideal world (from the programmer's point of view) every project would start from scratch and have no external dependencies. Once in a while such situation happens and this is when real progress is made. New languages, new programming methodologies, new team structures can be applied and tested.

In the real world most projects inherit some source code, usually written using obsolete programming techniques, with its own model for error handling, debugging, use or misuse of global objects, goto's, spaghetti code, functions that go for pages and pages, etc. Most projects have external dependencies--some code, tools, or libraries are being developed by external groups. Worst of all, those groups have different goals, they have to ship their own product, compete in the marketplace, etc. Sure, they are always enthusiastic about having their code or tool used by another group and they promise continuing support. Unfortunately they have different priorities. Make sure your manager has some leverage over their manager.

If you have full control over inherited code, plan on rewriting it step by step. Go through a series of code reviews to find out which parts will cause most problems and rewrite them first. Then do parallel development, interlacing rewrites with the development of new code. The effort will pay back in terms of debugging time and overall code quality.

Multi-platform development

A lot of programs are developed for multiple platforms at once. It could be different hardware or a different set of APIs. Operating systems and computers evolve-- at any point in time there is the obsolete platform, the most popular platform, and the platform of the future. Sometimes the target platform is different than the development platform. In any case, the platform-dependent things should be abstracted and separated into layers.

Operating system is supposed to provide an abstraction layer that separates applications from the hardware. Except for very specialized applications, access to disk is very well abstracted into the file system. In windowing systems, graphics and user input is abstracted into APIs. Our program should do the same with the platform dependent services-- abstract them into layers. A layer is a set of services through which our application can access some lower level functionality. The advantage of layering is that we can tweak the implementation without having to modify the code that uses it. Moreover, we can add new implementations or switch from one to another using a compile-time variable. Sometimes a platform doesn't provide or even need the functionality provided by other platforms. For instance, in a non-multitasking system one doesn't need semaphores. Still one can provide a locking system whose implementation can be switched on and off, depending on the platform.

We can construct a layer by creating a set of classes that abstract some functionality. For instance, memory mapped files can be combined with buffered files under one abstraction. It is advisable that the implementation choices be made in such a way that the platform-of-the-future implementation be the most efficient one.

It is worth noticing that if the platforms differ by the sizes of basic data types, such as 16-bit vs. 32-bit integers, one should be extremely careful with the design of the persistent data structures and data types that can be transmitted over the wire. The fool proof method would be to convert all fundamental data types into strings of bytes of well defined length. In this way we could even resolve the Big Endian vs. Little Endian differences. This solution is not always acceptable though, because of the runtime overhead. A tradeoff is made to either support only these platforms where the sizes of shorts and longs are compatible (and the Endians are the same), or provide conversion programs that can translate persistent data from one format to another. In any case it is a good idea to avoid using `int`s inside data types that are stored on disk or passed over the wire.

Program Modifications

Modifications of existing code range from cosmetic changes, such as renaming a variable, to sweeping global changes and major rewrites. Small changes are often suggested during code reviews. The rule of thumb is that when you see too many local variables or objects within a single function, or too many parameters passed back and forth, the code is ripe for a new abstraction.

It is interesting to notice how the object oriented paradigm gets distorted at the highest and at the lowest levels. It is often difficult to come up with a good set of top level objects, and all too often the main function ends up being a large procedure. Conversely, at the bottom of the hierarchy there is no good tradition of using a lot of short-lived lightweight local objects. The top level situation is a matter of good or poor design, the bottom level situation depends a lot on the quality of code reviews. The above rule of thumb is of great help there. You should also be on the lookout for too much cut-and-paste code. If the same set of actions with only small modifications happens in many places, it may be time to look for a new abstraction.

Rewrites of small parts of the program happen, and they are a good sign of healthy development. Sometimes the rewrites are more serious. It could be abstracting a layer, in which case all the users of a given service have to be modified; or changing the higher level structure, in which case a lot of lower level structures are influenced. Fortunately the top-down object-oriented design makes such sweeping changes much easier to make. It is quite possible to split a top level object into more independent parts, or change the containment or access structure at the highest level (for example, move a sub-object from one object to another). How is it done? The key is to make the changes incrementally, top-down.

During the first pass, you change the interfaces, pass different sets of arguments-- for instance pass reference variables to those places that used to have direct access to some objects but are about to loose it. Make as few changes to the implementation as possible. Compile and test.

In the second pass, move objects around and see if they have access to all the data they need. Compile and test.

In the third pass, once you have all the objects in place and all the arguments at your disposal, start making the necessary implementation changes, step by step.

Testing

Testing starts at the same time as the implementation. At all times you must have a working program. You need it for your testing, your teammates need it for their testing. The functionality will not be there, but the program will run and will at least print some debugging output. As soon as there is some functionality, start regression testing.

Regression Testing

Develop a test suite to test the basic functionality of your system. After every change run it to see if you hadn't broken any functionality. Expand the test suite to include basic tests of all new functionality. Running the regression suite should not take a long time.

Stress Testing

As soon as some functionality starts approaching its final form, stress testing should start. Unlike regression testing, stress testing is there to test the limits of the system. For instance, a comprehensive test of all possible failure scenarios like out-of-memory errors in various places, disk failures, unexpected power-downs, etc., should be made.

The scaleability under heavy loads should be tested. Depending on the type of program, it could be processing lots of small files, or one extremely large file, or lots of requests, etc.