Programming Methodology II

Lecture Notes

These notes may be copied for students who are taking

COEN 244 Programming Methodology II

Engineering and computer science
Concordia University, Montreal, Quebec

Contents

List of Figures

1 Getting Started

Although this course will cover programming practices in general, the focus will be on C++. Why not Java?

- C++ is more complex than Java. If you can write good programs in C++ well, you can transfer this skill to Java; the converse is not true.
- Although Java has achieved widespread popularity during the last ten years, there is still a need for C++ programmers.[1]
- Many games and most software for the telecommunications industry is written in C++. Companies that use C++ include (see also the course web page): Adobe products; Alias/Wavefront products (e.g., Maya); Amazon; Google; JPL/NASA; and Microsoft.
- There are many Java jobs and many Java programmers. There are not quite so many C++ jobs and there are very few good unemployed C++ programmers. Strong C++ programmers can find interesting and well-paid jobs.

It is a cliché to say that software is becoming ubiquitous. However, it is noteworthy that programs are getting larger:

Entity	Lines of code
Cell phone:	2×10^6
Car:	20×10^6
Telephone exchange:	100×10^6
Civil aircraft:	10^9
Military aircraft:	6×10^9

The crucial problem for all aspects of software development is ***scalability***. Approaches and techniques that do not work for millions of lines of code are not useful. C++ scales well.

1.1 `Hello, world!`

C++ is the result of a long hisory of developing programming languages: 2

1965: BCPL (Martin Richards)

1968: B (Ken Thompson @ Bell Labs)
($8K \times 18b$ PDP/7)

1969: C (Ken Thompson and Dennis Ritchie)
($64K \times 12b$ PDP/11)

1979: C with classes (Bjarne Stroustrup)

1983: C++ named

[1] It is hard to obtain reliable data, but once recent article said that current software development is 45% Java, 40% C++, and 15% other.

1990: templates and exceptions

1992: Microsoft C++ compiler

1993: RTTI, namespaces

1994: ANSI/ISO Draft Standard

1995: Java

The diagram on the following page gives a more detailed history of the development of C++ and related programming languages.

C++ has strengths: [3]

- Low-level systems programming
- High-level systems programming
- Embedded code
- Numeric/scientific computation
- Games programming
- General application programming

and weaknesses:

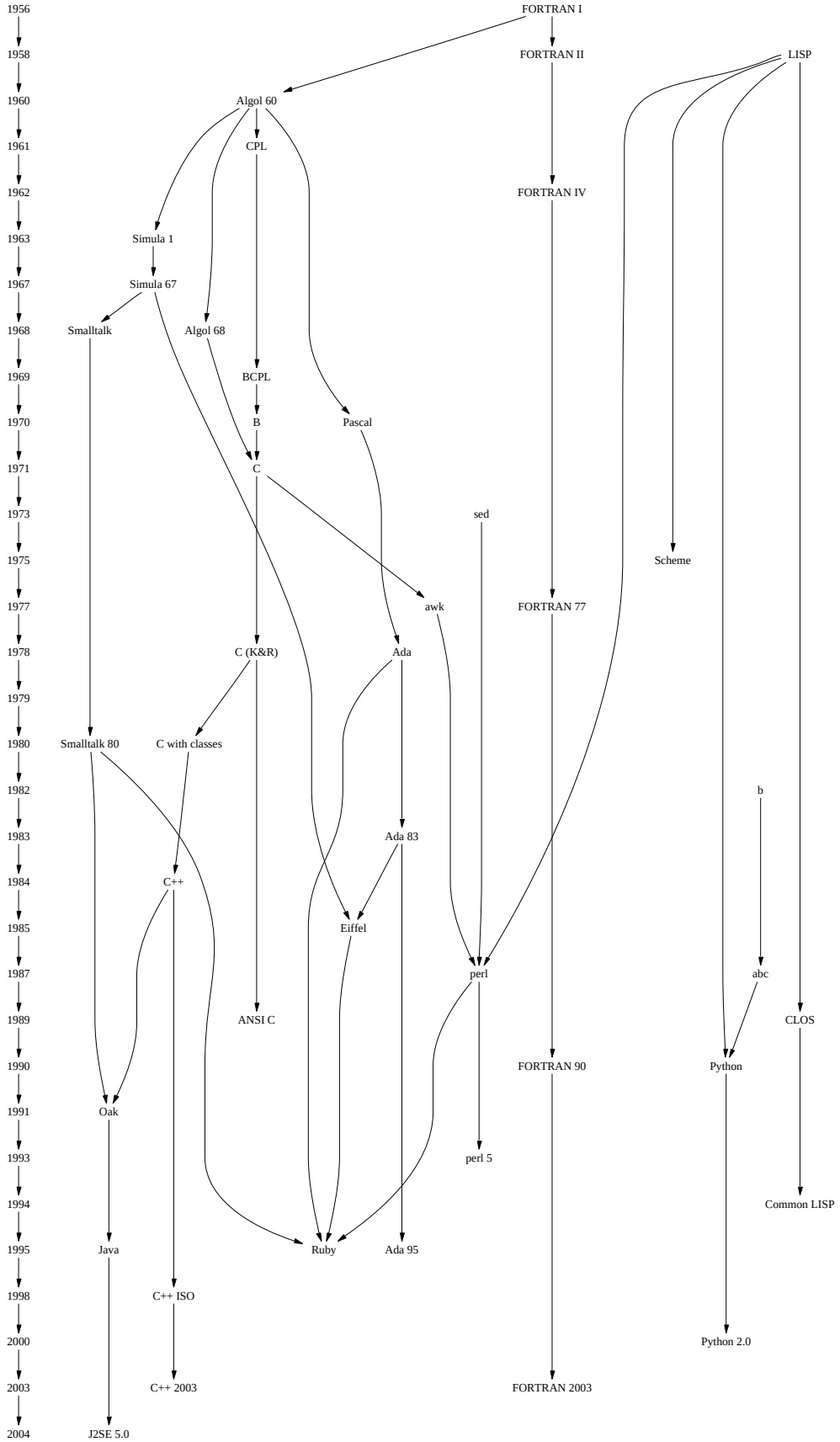
- Legacy of C
- Insecurities
- Complexity
- No standard GUI library

Even the lead designer of C++, Bjarne Stroustrup, does not claim that C++ is the dieal programming language: [4]

> *C makes it easy to shoot yourself in the foot.*
>
> *C++ makes it harder, but when you do, it blows away your whole leg.*

The C Programming Language (Kernighan and Ritchie 1978), the "classic" text for C, appeared in 1978. Its first example program has set the standard for all successors: [5]

```
main()
{
    printf("hello, world\n");
}
```


1956
1958
1960
1961
1962
1963
1967
1968
1969
1970
1971
1973
1975
1977
1978
1979
1980
1982
1983
1984
1985
1987
1989
1990
1991
1993
1994
1995
1998
2000
2003
2004
FORTRAN I
FORTRAN II
FORTRAN IV
FORTRAN 77
FORTRAN 90
FORTRAN 2003
LISP
Algol 60
CPL
Simula 1
Simula 67
Smalltalk
Algol 68
BCPL
B
Pascal
C
C (K&R)
Ada
Smalltalk 80
C with classes
Ada 83
C++
Eiffel
ANSI C
C++ ISO
C++ 2003
Oak
Java
J2SE 5.0
sed
awk
perl
perl 5
Ruby
Ada 95
Scheme
b
abc
CLOS
Python
Python 2.0
Common LISP

```
#include <iostream>

int main()
{
   std::cout << "Hello, world!" << std::endl;
   return 0;
}
```

Figure 1: Hello.1

Since then, C has evolved considerably: it has even changed its name, to C++. The recommended text book for this course, *Accelerated C++*, begins with the version of the "hello, world" program shown in Figure 1 (Koenig and Moo 2000, page 1).

A modern C++ compiler should compile both programs, because C++ is an extension of C.[2] To be more precise, it was originally the intention that C++ and C should be compatible, but that goal was abandoned when it became clear that C++ would suffer if it was followed rigorously: it is possible to write programs that are legal C but not legal C++. An unfortunate side-effect of this compatibility is that there is now a mixture of C and C++ programming styles, ranging from "classic C" style to "modern C++" style. In this course, we will emphasize the modern style.

The important features of Figure 1 are:

- `#include <iostream>` informs the compiler that the program uses components from the ***stream library*** to perform input and output. (`#include` is similar to Java's `import`.)
- `int main()` introduces a function that is usually called the ***main program***. Compilers are quite flexible about the declaration of this function. We will see later, for example, that `main` may take arguments. The initial `int` is important, however: it indicates that the function will return an integer that the operating system may use to choose the next action.
- `std::cout` is the new version of the old "`printf`" (or `System.out.print` in Java). The first part, `std`, refers to a ***namespace*** in which `cout` and other names are declared. `cout` (pronounced "see-out") is the name of the ***standard output stream***; when a program runs, text written to `cout` appears on the console window. The `::` is a ***scoping operator*** that says, in effect, "find the name `cout` in the namespace `std`".
- `<<` is an ***operator***, analogous to + or ×. In this case, its right operand consists of stuff to be written to standard output.
- `std::endl` is the new version of `\n`: it is short for "end line" and outputs a return character. (We can still use `\n`, however.)
- `return 0` sends zero back to the operating system, indicating that the program terminated successfully.

In general, we also note that:

[2] Stroustrup claims that "every example in *The C Programming Language* (2nd Edition) (Kernighan and Ritchie 1978) is also a C++ program.". Of course, this does ***not*** imply that every program that a C compiler compiles will also be accepted by a C++ compiler.

- Statements in C++ are terminated with a semicolon (“;”).
- Blocks of code are delimited by braces (“{” and “}”).

C has a reputation for being cryptic. The following code for copying a string (Kernighan and Ritchie 1978, page 101) is often quoted: 6

```
char *s;
char *t;
....
while (*s++ = *t++)
    ;
```

This works for several rather subtle reasons: the post-increment operator ++ has an effect and also returns a value; the assignment operator = returns a value; C strings are terminated by the special value '\0' (also known as the “null character”); and the null character, interpreted as a boolean, represents false. The loop terminates because a C string, by convention, is terminated with a null character ('\0').

This style of programming was fine for the time. Ritchie and Thompson designed C for systems programming on minicomputers of the early 1970s. They did most of their work on a DEC PDP/11 with 64 Kb of RAM and they wanted a small and simple compiler. The while statement above can be compiled into efficient code without any further optimization.

By way of contrast, here is the string copy implemented in “C-for-dummies” style. Note the explicit termination test, comparing s[i] to '\0'. The do-while loop is used because the terminator must be copied to the output string.

```
int i = 0;
do
{
   s[i] = t[i];
   i = i + 1;
} while (s[i] != '\0');
```

An experienced C programmer would probably prefer the library call strcpy(s, t) to either of these versions.

From a modern perspective, however, the while statement is unnecessarily cryptic and potentially ***inefficient*** because it places unnecessary constraints on what the compiler can do. Accurately implementing what the programmer has written requires the use of two registers, separately incremented, and each containing a pointer. At each cycle, the code must test for a null character.

In contrast, a C++ programmer would write something like this: 7

```
std::string s;
std::string t;
....
s = t;
```

In this version, strings are instances of a standard class `string` rather than "pointers to characters". The copying operation is performed by the library function that implements assignment ("="). This function can be written to be optimal for the architecture for which the library is written: for example, it might use "block moves" rather than copying characters one at a time. It might even avoid copying altogether (see Figure 2). The code does not depend on any particular representation for strings. For example, it does not require a null terminating character. In general, a C++ programmer will tend to work at a ***higher level of abstraction*** than a traditional C programmer. 8

Figure 2 shows the time required to copy a 44-character string[3] in the four different ways described above. All times are in microseconds. The results are surprising: `strcpy` is 34 times faster than `string` assignment with VC++, but 27 times ***slower*** with gcc. Perhaps Microsoft have not implemented the new libraries very well! It seems likely that gcc avoids copying the string until it becomes necessary (i.e., it uses the "copy on write" pattern). The standard class `string` is widely considered to be an inefficient mess, so perhaps we should not worry too much about these results.

Operation	VC++ 7.1	gcc 3.2.3
`char *s, *t; ... strcpy(s, t)`	0.025	4.30
`char *s, *t; ... while(*s++ = *t++);`	0.191	2.99
C-for-dummies	0.200	4.30
`string s, t; ... s = t;`	0.856	0.16

Figure 2: Timing comparisons for string copying

Here is another, much more advanced, example of modern C++. Suppose we want to parse strings according to the following grammar: 9

$$
\begin{array}{lcl}
\mathit{expr} & = & \mathit{term}\ \texttt{`+'}\ \mathit{expr} \\
 & | & \mathit{term}\ \texttt{`-'}\ \mathit{expr} \\
 & | & \mathit{term} \\
\mathit{term} & = & \mathit{factor}\ \texttt{`*'}\ \mathit{term} \\
 & | & \mathit{factor}\ \texttt{`/'}\ \mathit{term} \\
 & | & \mathit{factor} \\
\mathit{factor} & = & \mathit{integer} \\
 & | & \texttt{`('}\ \mathit{expr}\ \texttt{`)'}
\end{array}
$$

Using the Boost Spirit library (Abrahams and Gurtovy 2005), we can write code corresponding to these productions as shown in Figure 3 follows (some additional surrounding code is needed to make everything work). Notice how close the C++ code is to the original grammar. 10

That C++ is compatible with C is both good and bad. It is good because at Bell Labs, where C++ was developed, a new language that could not compile existing code would almost certainly have not been accepted. It is bad because C is an old language with many features that would now be considered undesirable; compatibility meant that C++ had to incorporate most of these

[3] `"the quick brown fox jumped over the lazy dog"`.

```
expr =
      ( term[expr.val = _1] >> '+' >> expr[expr.val += _1] )
    | ( term[expr.val = _1] >> '-' >> expr[expr.val -= _1] )
    | term[expr.val = _1]
    ;

term =
      ( factor[term.val = _1] >> '*' >> term[term.val *= _1] )
    | ( factor[term.val = _1] >> '/' >> term[term.val /= _1] )
    | factor[term.val = _1]
    ;

factor =
      integer[factor.val = _1]
    | ( '(' >> expr[factor.val = _1] >> ')' )
    ;
```

Figure 3: Parsing with Boost/Spirit

undesirable features, even though it does quite a good job of covering them up. A well-trained and conscientious programmer can write secure and efficient programs in C++; an inexperienced programmer can create a real mess.

Figure 3 illustrates another aspect of modern C++. All competent C programmers are essentially equal: they all have a good understanding of the entire language and its standard libraries. With C++, however, a wide gulf separates programmers who can ***use*** the Spirit parser components — which is straightforward — from the programmers who can ***create*** the Spirit components; these programmers form a small minority of C++ programmers. (Parser generators are constructed by ***template metaprogramming***, which few C++ programmers understand, let alone use.)

1.2 Compiling C++

Old compilers, and even some new compilers, are not able to compile modern C++ programs. Figure 4 lists compilers that work in the left column and compilers that may not work in the right column. In particular, Microsoft VC++ 7.0 or later is fine for this course, but you may have problems with VC++ 6. 11

1.2.1 The compilation process

Compiling a program consists of a number of steps. Errors may occur at any step, and it is important to distinguish the different kinds of error. The compiler processes the program as a number of ***compilation units***. Each compilation unit corresponds to a source code file together with any other files `#include`d with it.

1. The compiler parses each compilation unit. This may produce ***syntax errors***.

   If we omit a semicolon in Figure 1, writing

Good compilers		Bad compilers	
Name	Version	Name	Version
Comeau	4.3.3	Borland	5.5.1
gcc	3.2.2	Borland	5.6.4
gcc	3.3.1	gcc	2.95.3
Intel	7.1	Microsoft Visual	6.0 SP 5
Intel	8.0	Microsoft Visual	7.0
Metrowerks CodeWarrior	8.3	HP aCC	03.55
Metrowerks CodeWarrior	9.2	Sun	5.6
Microsoft Visual	7.1		

Figure 4: Compilers for modern C++

```
cout << "Hello, world!" << endl
```

the compiler issues a syntax error:[4] 12

```
f:\Courses\COMP6441\src\Hello\hello.cpp(32):
   error C2143: syntax error : missing ';' before '}'
```

The "32" is the number of the line on which the error is detected; in this case, it is the line containing `}`, the line ***following*** the line with the error. However, the error message itself is quite helpful: it actually tells you what is wrong with your program.

2. The compiler checks the semantic correctness of the program. For example, it checks that each function is called with arguments of an appropriate type. A program may have ***semantic errors***.

   If we omit the "`l`" from "`endl`", the compiler issues two semantic errors: 13

   ```
   f:\Courses\COMP6441\src\Hello\hello.cpp(31):
      error C2065: 'end' : undeclared identifier
   f:\Courses\COMP6441\src\Hello\hello.cpp(31):
      error C2593: 'operator <<' is ambiguous
   ```

   Syntax errors and semantic errors are collectively call ***compile-time errors***.

3. The compiler generates object code for the compilation unit.

4. When each compilation unit has been controlled, the ***linker*** is invoked. The linker attempts to link all of the object code modules together to form an ***executable*** (or, occasionally, a library). This may produce ***link-time errors***.

   Suppose that we declare and use a function `f` but forget to define it. The compiler expects that `f` will be defined somewhere else and so does not generate a semantic error. We do not get an error until the linker discovers that there is no definition: 14

[4] The error messages were produced by VC++. Other compilers produce similar, but not identical, diagnostics.

```
Hello error LNK2019:
   unresolved external symbol "void __cdecl f(int)"
   (?f@@YAXH@Z) referenced in function _main
```

The mysterious string `?f@@YAXH@Z` is the name that the compiler has given to the function `f`. The conversion from `f` to `?f@@YAXH@Z` is called ***name mangling***.

5. When the program has been linked, it can be ***executed*** or, more simply, ***run***. It may run correctly or it may generate ***run-time errors***.

   If you write `cout` twice by mistake, the compiler accepts the program without any fuss:

   ```
   cout << cout << "Hello, world!" << endl;
   ```

   However, when the program runs, it displays: 15

   ```
   0045768CHello, world!
   ```

   Output like this can be disconcerting for beginners: the program has displayed the address of the object `cout` in hexadecimal!

Although you don't really want any errors at all, you should prefer compile-time and link-time errors to run-time errors, if only because your customers will never see them. Fortunately, C++ compilers perform thorough syntactic and semantic checking (much better than C compilers) and will catch many of your errors. 16

> ***An event that occurs during compilation is called static. An event that occurs during execution is called dynamic.***

1.2.2 Compiling in practice

There are a number of platforms available for C++ development. The following are provided on university lab computers:

Windows: Microsoft Visual Studio 6.0: The version of C++ provided by VS6 is obsolete. In particular, templates are not implemented correctly. You can use VS6 for simple programming tasks, but it is not adequate for this course.

Windows: Microsoft Visual Studio .NET Professional 2003: VS 2003 provides a better version of C++ (V7) that is usable for this course. It is not the most recent version of VS.NET and versions are not compatible.

For example, suppose that you have V8 at home. You will not be able to run programs developed at home on school machines — when you try do do so, Windows reports *"This application has failed to start because the application configuration is incorrect. Reinstalling the application may fix this problem"*. If you develop something at school under V7 and take it home, VS will tell you that the project must be converted to V8 — meaning that you can't work on it and then take it back to school!

A further problem with all versions of VS is that clutter your project with large quantities of Microsoft code that you probably don't understand, don't want, and don't need in you project. Appendix C (page 272 of these notes) explains how to prevent this.

Linux: gcc (Gnu C Compiler) is a command-line compiler, not an IDE. For simple programs with all of the source code in one file, just use "`gcc`" (defaults to C) or "`g++`" (defaults to C++) as a command. For more complex programs, it's best to write a `make` file.

Linux also provides a variety of tools that are useful for C++ development, including: `cvs` (for version control); Doxygen (for documentation); and `wxGTK`, an "easy-to-use API for writing GUI applications on multiple platforms".

If you are working on your home computer, you have various options.

Visual Studio: You can buy the educational version of Visual C++ for a moderate price.[5]

Alternatively, you can download the free "Express" version from Microsoft. The Express edition is limited in various ways but is good enough for this course.

An advantage of VS is that many libraries are provided in pre-compiled form for it. To install them, you just download the package and store various files in appropriate places.

Code::Blocks is a free IDE for C++ that can be used with various compilers. The standard download package includes MinGW, which is the GNU compiler for Windows.

Code::Blocks is a more friendly and easy-to-use IDE than VS. Its main disadvantage is libraries: it comes with the standard C++ libraries but, if you want to use any other libarries, you will have to compile them and install them yourself.

See the course web page for links to these and other free products.

1.3 `Hello, world!` — The Details

Here is the first C++ example program again: 17

```
#include <iostream>

int main()
{
   std::cout << "Hello, world!" << std::endl;
   return 0;
}
```

"`#include`" is a ***compiler directive***. Strictly speaking, it is not part of the program, but instead is an instruction telling the compiler what to do. In this case, the directive tells the compiler to make the part of the ***standard library*** that deals with streams accessible to the program.

Streams are one of the media that C++ programs used to communicate. The communication is usually external — data is read to or from a device such as a keyboard, screen disk, or network — but may also be internal to the program (that is, to or from memory). A stream is not the same as a file, although a file may be read or written with a stream.

[5] Recent (Dec 06) prices at the university computer store were $69 for VS 2005 Standard and $145 for VS 2005 Pro.

An ***output stream*** typically allows the program to send data ***to*** an external device, such as a window. An ***input stream*** typically allows the program to receive data ***from*** an external device, such as a keyboard. The library component `iostream` provides both: "`i`" for input and "`o`" for output.

The program itself consists of a single ***function definition***. The function's name is `main` and its ***return type*** is `int`. The word `int` is a ***keyword*** of C++, meaning that we cannot use it for anything other than its intended application. In fact, `int` is a ***type*** — the type of ***signed integers***. A value i of type `int` is typically represented using 32 bits and satisfies $-2^{31} \leq i < 2^{31} - 1$. A few modern C++ systems provide 64-bit `int`s.

The ***body*** of the function consists of a sequence of ***statements*** (two in this example) enclosed in braces (`{ ... }`). The last character of each statement is a semi-colon (`;`).

The return type of the function (`int`) is related to its last statement: `return 0`. The program behaves ***as if*** the function `main` is invoked by the operating system, executes its two statements, and returns the integer 0 as its result. The operating system interprets 0 as "normal termination". We can return other values if we want to tell the operating system that something went wrong.

The only thing left to discuss is the line beginning with `std::cout`. This illustrates an important general principle of C++. The text

```
std::cout << "Hello, world!" << std::endl
```

is an ***expression*** that yields a value. By putting a semicolon at the end of this expression, we discard the value and turn the expression into a ***statement***.

Expressions consist of ***operands*** and ***operators***. For example, $x + 5$ is an expression with two operands (x and 5) and one operator, $+$. We say that $+$ is a ***binary operator*** because it takes two operands: x is its ***left operand*** and 5 is its ***right operand***. An expression can be ***evaluated*** to yield a ***result***. If x has the value 2, then evaluating $x + 5$ yields 7.

In C++, `<<` is a binary operator, usually called "insert". Its left operand is an output stream; its right operand may be a ***string***, a ***manipulator***, and various other things. Its result is a stream. When an expression contains more than one insertion operator, they are evaluated fom left to right. The first step in evaluating

```
std::cout << "Hello, world!" << std::endl
```

is to evaluate

```
std::cout << "Hello, world!"
```

This has the effect of appending the string `"Hello, world!"` to the standard output stream and yields the updated output stream, say `uos`. The next step is to evaluate

```
uos << std::endl
```

which has the effect of appending a new line to the standard output stream (and a bit more, discussed below) and yields the updated output stream. In the program, this expression is followed by a semicolon, which throws away the final value. `cout` is a persistent object, however, and it still exists in its updated state.

There is a small, but significant, difference between the statements

```
std::cout << "Hello, world!" << std::endl;
```

and

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

which could also be written as

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

The difference is that `std::endl` writes the `\n` and also ***flushes the output buffer***. What does this mean?

It would be inefficient for a program to go through all of the operations of transferring data for every character in a stream. To save time, the program stores characters in a temporary area called a ***buffer*** and performs the actual transfers only when the buffer is full. If the program is writing data to a file, this behaviour is undetectable to the user. But, if the program is writing to a window, buffering makes a lot of difference. If we use `\n`, a significant amount of output may be generated by the program before we actually see it displayed. Using `std::endl` ensures that each line of output will be displayed as soon as it has been computed.

Here is yet another way of writing "Hello, world" followed by end-of-line:

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

When C++ sees a quote (`"`) at the end of a string, followed by white space (blanks, tabs, and line breaks), followed by a quote at the beginning of a string, it erases quotes and the white space and treats the result as a single string. This is useful for writing long strings or strings that run over several lines.

1.4 Namespaces

As we noted above, we write `std::cout` to tell the compiler that the name `cout` comes from the namespace `std`. The name `cout` is a ***simple name*** (or just a ***name***), `std::cout` is a ***qualified name***, and `std::` is a ***qualifier***. "`::`" is called the ***scope operator***.

We can use this convention but, as programs get longer and more complex, the frequent appearance of "`std::`" becomes irritating, as well as being tedious to type. An alternative is to say to the compiler "When I write `cout`, I mean `std::cout`". This is the purpose of the `using` directive, as shown in Figure 5. In general, we write one `using` directive at the beginning of the source file for each name that we intend to use. 18

We can go further, saying to the compiler "When I write any name that belongs to `std`, take it from there". The form of the `using` directive that does this is shown in Figure 6. Note that, in both Figure 5 and Figure 6, we use `cout` and `endl` without the qualifier `std::`. 19

Some people will tell you that the point of namespaces is to make the source of names explicit; they will tell you that Figure 1 is the best way to write programs, Figure 5 is acceptable, and Figure 6 is bad — the kind of code produced by lazy programmers who should be fired. This attitude is ***wrong***. The actual situation is more complex.

```
#include <iostream>

using std::cout;
using std::endl;

int main()
{
    cout << "Hello, world!" << endl;
}
```

Figure 5: Hello.2

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello, world!" << endl;
}
```

Figure 6: Hello.3

Namespaces were introduced into C++ to prevent name clashes in programs that use more than one library. Suppose that you are writing a program that uses a library `Cowboy` and another library `Artist`. Both libraries provide a function called `draw`. When you try to use `draw`, the compiler complains that it cannot tell which library to take it from. To avoid this problem, the libraries wrap their names in namespaces — perhaps `namespace cowboy` and `namespace artist`. The programmer can then write `cowboy::draw` or `artist::draw`, depending on which function is needed.

Most of the time, however, name clashes do ***not*** occur. In particular, it is very unlikely that a library writer would use a well-known name such as `cout`. Consequently, it is quite acceptable to use the form of Figure 6, while being aware that it may one day be necessary to use explicit qualification when a name clash actually occurs.

1.5 Strings

20

Figure 7 (Koenig and Moo 2000, page 9) shows a program that asks for the user's name and then greets the user. Running it produces a dialogue like this: 21

```
What is your name? Nebuchadnezzar
Hi, Nebuchadnezzar!
```

Much of this program should already be familiar because it is similar to `Hello, world!`. The new features are the class `string` and the input stream `cin`.

The directive `#include <string>` informs the compiler that we will be using class `string`. Since the names provided by `string` are included in the namespace `std`, we can write simply "`string`" in the program rather than "`std::string`".

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    cout << "What is your name? ";
    string name;
    cin >> name;
    cout << "Hi, " << name << "!" << endl;
    return 0;
}
```

Figure 7: Greeting.1

The second statement of the main program,

```
string name;
```

is both a ***declaration*** and a ***definition***. (We will discuss the distinction between declarations and definitions in detail later. For now, note that all definitions are also declarations, but a declaration is not necessarily a definition.) It introduces a new variable, `name`, into the program. The type of `name` is `string`. We could also say that `name` is a new ***object***, and an ***instance*** of class `string`.

The value of a `string` object is a string of characters, such as `"Hello, world!"`. After the declaration, the value of `name` is actually the ***empty string***, written `""`.

There are various ways of putting characters into a string. In this program, the statement

```
cin >> name;
```

reads data from the ***standard input stream***, `cin` (pronounced "see-in"). By default, `cin` gets data from the keyboard. In other words, whatever text you enter in response to the prompt `"What is your name?"` gets stored in the variable `name`.

Actually, this description is not quite accurate: `cin` reads only until it encounters white space (a blank, tab, or ENTER) and then stops. This explains the following dialogue:

```
What is your name? King Kong
Hi, King!
```

After the program has stored a value in `name`, it can use this value, as in the second `cout` statement.

There is a subtlety in Figure 7 that is worth noting. On the basis of the discussion at the end of Section 1.3, you would be right to wonder why the statement

```
cout << "What is your name? ";
```

produces any output at all. Why is the data not left in the buffer until `endl` is output in the second `cout` statement? The answer is that the streams `cin` and `cout` are ***linked***. Any use of `cin` causes the buffer for `cout` to be flushed. This ensures that programs that implement a dialog in which the user must respond to displayed text will work as expected.

Memory management is an important aspect of C++ programming. In Figure 7, memory for the string `name` is allocated on the run-time stack when the definition `string name` is evaluated. The class `string` manages memory when characters are put into the string (e.g., by `cin`) or the string is changed in other ways. At the final closing brace ("`}`") of the program, any memory used by `name` is de-allocated.

A definition, of a variable or other named entity, occurs within a ***scope***. The scope consists either of the closest enclosing braces or, if there are no enclosing braces, the entire source file. In the latter case, we say that the definition is "at file scope". Uses of the name must follow its definition. In other words, we cannot refer to a variable earlier in the program text than its definition: this is called the "definition before use rule". The name ceases to be accessible at the end of its scope.

> ***The definition of a name must textually precede its use.***

In many cases, there are actions connected with the definition and the end of the scope. In Figure 7, which is typical, the object `name` is ***constructed*** at the point of its definition and ***destroyed*** at the end of the program. What actually happens is this: when an object definition is processed, a ***constructor*** for the object is called and, at the end of a scope, the ***destructors*** for all stack-allocated objects are called.

Some C++ programmers (and books) use the abbreviations "ctor" and "dtor" for constructor and destructor, respectively.

1.6 A Pretty Frame

The greeting produced by Figure 7 is dull and boring. The next version, in Figure 8 (Koenig and Moo 2000, page 12), produces dialogues like this: 22

```
Please enter your first name: Ferdinand

**********************
*                    *
* Hi there, Ferdinand! *
*                    *
**********************
```

23

The new feature in this program is `const string`. There are four definitions of the form 24

```
const string ⟨name⟩ .... ;
```

The keyword `const` tells the compiler that the values of the names are not going to change. We cannot declare `name` as `const` because its initial value (`""`) gets changed when we use `cin` to copy characters into it.

When we introduce a `const` name, we must provide a value for it in the same declaration. The value can be a simple value, as in this example:

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
cout << "Please enter your first name: ";
string name;
cin >> name;

const string greeting = "Hi there, " + name + "!";
const string spaces(greeting.size(), ' ');
const string second = "* " + spaces + " *";
const string first(second.size(), '*');

cout << endl;
cout << first << endl;
cout << second << endl;
cout << "* " << greeting << " *" << endl;
cout << second << endl;
cout << first << endl;

return 0;
}
```

Figure 8: Greeting.2

```
const string WELCOME = "Welcome to COMP 6441!";
```

Figure 8 shows several other ways of providing an initial value using features of class `string`:

- `greeting = "Hi there, " + name + "!"`

  The operator `+` concatenates strings (i.e., joins them together). After `name` has received the value `"Ferdinand"`, the definition gives `greeting` the value `"Hi there, Ferdinand!"`.

  Although `greeting` is `const` and `name` is not `const`, we can use `name` as part of the value of `greeting`. The `const` qualifier says that `greeting` will not be changed later; it does not say that the value is known at compile-time.

- `spaces(greeting.size(), ' ')`

  The expression `greeting.size()` makes use of a ***member function*** of class `string`. Member functions are called with the "dot notation":

  ⟨*object name*⟩ `.` ⟨*function name*⟩ `(` ⟨*parameters*⟩ `)`

  In this case, the function name is `size` and it returns the size of (i.e., the number of characters in) the object `greeting`. Thus `greeting.size()` is a ***number*** (in this case, it is 20).

Class `string` has a constructor which expects two arguments: an numeric value (type `int`) and a character value (type `char`). By writing `spaces(n, ' ')` we are saying: construct an instance of class `string` with name `spaces` that contains `n` characters where each character is `' '` (that is, a blank).

Thus the effect of this particular definition is to define a constant `string` object `spaces` with value "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣". (␣ denotes a blank.)

- `second = "* " + spaces + " *"`

  The definition of `second` uses the definition of `spaces` and the concatenation operator, `+`.

- `first(second.size(), '*')`

  The definition of `first` is similar to that of `spaces`. It has the same number of characters as `second`, but each character is an asterisk, `'*'`. Note that C++ distinguishes between ***strings***, which have zero or more characters between double quotes (`"...."`) and ***characters*** (instances of type `char`), which have exactly one character between single quotes (`'.'`).

Having defined the strings `greeting`, `spaces`, `second`, and `first`, the program uses them to generate the desired output.

Although this program is trivial, it illustrates two important points about programming in general:

Avoid unnecessary assumptions.

The program is not written for people with five-letter names but for people with names of any (reasonable!) length.

Make the program do the work.

The strings needed for the display are ***computed*** (as much as possible). This makes it easy to generate a display whose size matches the name of the user.

It is not necessary to call `cout` once for each line of output. The six calls of `cout` in Figure 8 could be replaced by a single call:

```
cout <<
   endl <<
   first << endl <<
   second << endl <<
   "* " << greeting << " *" << endl <<
   second << endl <<
   first << endl;
```

2 Testing and Looping

The control flow of a program is determined by tests and loops. In this section, we review the main C++ structures for branching and looping and discuss some aspects of loop design.

Engineering is the application of theory (mathematics, physics, etc.) to practice. Programming has not yet reached the maturity of engineering because we do not yet have adequate theories. We can prove that trivial programs satisfy a specification, but the techniques do not scale.

Nevertheless, we should do what we can. It is feasible, for example, to use systematic techniques, rather than guesswork, to code loops, and these techniques are introduced in Section 2.3.

2.1 Conditions

A ***condition*** is an expression whose value is either ***true*** or ***false***. (This assumes a two-valued logic. There are other logics with more than two values. For example, the value of a formula in a three-valued logic might be ***true***, ***false***, or ***unknown***.)

C++ has a standard type, `bool`, with exactly two values, `true` and `false`. Type `bool` also provides operators, as shown in this table: 26

Operation	Logic Symbol	C++ operator
conjunction	$\wedge$	`&&`
disjunction	$\vee$	`||`
negation	$\neg$	`!`

C++ also has operators that work on all of the individual bits of their operands in parallel, shown in the next table. Do not confuse these with the boolean operators!

Operation	C++ operator
complement	$\sim$
shift left	`<<`
shift right	`>>`
and	`&`
exclusive or	`^`
inclusive or	`|`

The Boolean operators `&&` and `||` are ***lazy***; this means that they do not evaluate their right operands unless they need to. In detail, `p && q` is evaluated like this:

1. Evaluate `p`. If it is `false`, yield `false`.
2. Otherwise, evaluate `q` and yield the result.

Similarly, `p || q` is evaluated like this:

1. Evaluate `p`. If it is `true`, yield `true`.
2. Otherwise, evaluate `q` and yield the result.

Lazy evaluation has both good and bad consequences:

- It is more efficient, because the right operand is not evaluated unnecessarily.
- It enables us to write tests such as 27

```
if (y != 0 && x / y <= MAX_RATIO)
    ....
```

- In C++, conjunction (and) and disjunction (or) are not commutative! That is, `p && q` is not always equivalent to `q && p`. However, they cannot yield different truth values: at worst, one would succeed and the other would fail.

C++ also provides ***comparison operators***, which compare two operands and yield a boolean value. They are `<` (less than), `<=` (less than or equal to), `==` (equal to), `!=` (not equal to), `>` (greater than), and `>=` (greater than or equal to. Do not confuse `==` with `=` (the assignment operator). These work with operands of most types but may not always make sense.

How to read C++:

`x == y`	$\equiv$	"X equals Y"
`a = b`	$\equiv$	"A gets B" or "A becomes B" or "A is assigned B"

In addition to the type `bool`, C++ inherits some baggage from C. In C, and therefore in C++:

- 0 is considered `false`
- Any non-zero value is considered `true`

This convention has several consequences:

- A condition such as `counter != 0` , in which `counter` is an integer, has the same truth-value as `counter`. Many C++ programmers therefore use the simpler form, `counter`, in a context where a truth value is expected.
- Expressions that are `true`, when considered as truth values, are not necessarily equal. For example, suppose we want to express the fact that two counters are either both zero or both non-zero. We could express this condition as

  ```
  (counter1 == 0) == (counter2 == 0)
  ```

  or, equivalently, as

  ```
  (counter1 != 0) == (counter2 != 0)
  ```

  Using the abbreviation above, we might be tempted to simplify this to

  ```
  counter1 == counter2
  ```

  But this expression has a different meaning!

> ***Write conditions carefully. Prefer complete expressions to abbreviations.***

The convention also suggests the question: what ***is*** zero in C++? The answer is that there are many things that are considered to be zero — and are therefore also considered to be `false` in a condition:

- Integers of type `short`, `int`, and `long`, with value 0
- Floating point numbers of type `float`, `double`, and `long double`, with value 0.0
- The character `'\0'`
- Any null pointer (we will discuss pointers later)
- The first element of an enumeration
- The `bool` value `false`
- And perhaps others . . . .

2.2 Conditional Expressions

A ***conditional expression*** is an expression with a boolean value. Conditional expressions are often called ***tests***, for short.

Programs typically evaluate conditions using `if` statements, which have one of two forms 28

```
if ( ⟨condition⟩ )
   ⟨statement⟩

if ( ⟨condition⟩ )
   ⟨statement⟩
else
   ⟨statement⟩
```

and work as you would expect them to.

The ⟨*statement*⟩s in an `if` statement can be simple or compound. This example illustrates both possibilities: 29

```
if (angle < PI)
{
   cout << "Still going round ...";
   angle += 0.01;
}
else
{
   angle = 0;
}
```

The braces around the final assignment, `angle = 0`, are not essential. We could alternatively have written

```
....
else
   angle = 0;
```

Whether you include the braces or not is a matter of taste and preference. Including them adds two lines to the code (or one line if you put the left brace on the same line as `else`). But including them makes it easy to add another line later — which is often necessary — and avoids the error of adding a line but forgetting to add the braces.

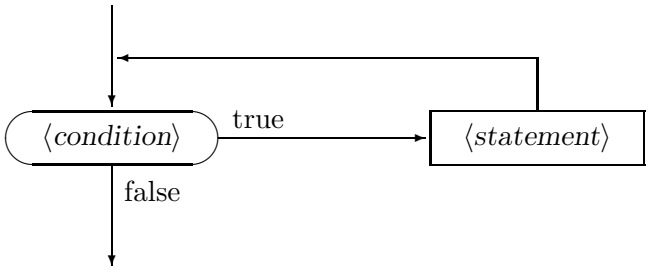
2.3 Loops

C++ provides three looping constructs. In the order in which you should consider using them, they are: `for`; `while`; and `do`/`while`.

The `while` loop has this structure 30

```
while ( ⟨condition⟩ )
   ⟨statement⟩
```

and it works like this:



An important feature of the `while` loop is that the loop body may not be executed at all. This is a very useful feature and, when writing a `while` loop, you should always check that its behaviour when the condition is initially `false` is correct.

A loop of this form 31

```
⟨initialize⟩
while ( ⟨condition⟩ )
{
   ⟨action⟩
   ⟨step⟩
}
```

should usually be written more concisely in this (almost) equivalent form:

```
for ( ⟨initialize⟩ ;  ⟨condition⟩ ; ⟨step⟩ )
{
   ⟨action⟩
}
```

For example, instead of writing 32

```
int i = 0;
while ( i < MAX )
{
   doSomething(i);
   ++i;
}
```

write this:

```
for (int i = 0; i < MAX; ++i)
{
   doSomething(i);
}
```

It is a good idea to get into the habit of using pre-increment (`++i`) rather than post-increment (`i++`). For integers, it doesn't make much difference, but `++` and `--` are overloaded for other types for which the difference is more significant. The reason for preferring pre-increment is that `i++` may force the compiler to generate a temporary variable, whereas `++i` does not.

The `do`/`while` loop is used only when you want to evaluate condition ***after*** performing the loop body: 33

```
do
   ⟨statement⟩
while ( ⟨condition⟩ )
```

2.4 Example: Computing the Frame

The program in Figure 9 uses `if`, `while`, and `for` statements. Figure 10 shows an example of its 34
use. The outer loop, formatting the rows of the display, is a `for` loop, because exactly one row is 35
processed during each iteration. This pattern does not work for the inner loop, because progress is not always one column at a time. The `if` statements make decisions about what text to output, and they all have complex conditions with `&&` and `||` operators.

This program is easier to modify than Figure 8: to change the size of the frame, all we have to do is change numbers in the `const` declarations. In fact, just changing the value of `pad` will change the spacing all around the greeting.

> ***Design programs so that a few easily changed parameters change the behaviour of the program in a consistent way.***

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
   cout << "Please enter your first name: ";
   string name;
   cin >> name;
   const string greeting = "Hello, " + name + "!";
   const int pad = 1;
   const int rows = pad * 2 + 3;
   const string::size_type cols = greeting.size() + pad * 2 + 2;
   cout << endl;
   for (int r = 0; r != rows; ++r)
   {
      string::size_type c = 0;
      while (c != cols)
      {
         if (r == pad + 1 && c == pad + 1)
         {
            // We are positioned for the greeting.
            cout << greeting;
            c += greeting.size();
         }
         else
         {
            if (r == 0 || r == rows - 1 ||
                c == 0 || c == cols - 1)
               // We are on a border.
               cout << "*";
            else
               cout << " ";
         ++c;
         }
      }
      cout << endl;
   }
   return 0;
}
```

Figure 9: Greeting.3

```
Please enter your first name: Wilberforce

**********************
*                    *
* Hello, Wilberforce! *
*                    *
**********************
```

Figure 10: A dialogue with Greeting.3

The original program (Koenig and Moo 2000, page 29) contains a number of comments: 36

```
// say what standard-library names we use
using std::cin;         using std::endl;
....
// ask for the person's name
cout << "Please enter your first name: ";
....
// read the name
string name;
cin >> name;
```

These comments are acceptable, but only because this program appears in an introductory text book. In general, comments like this should not be written unless:

- They provide information that is not obvious from the code
- They make the code more readable without adding noise to it

There is one comment in this program which might serve a purpose:

```
// the number of blanks surrounding the greeting
const int pad = 1;
```

Without the comment, the meaning of `pad` would not be obvious, although it is not hard to guess its meaning by reading the next few lines of code. But any comment that is provided to explain the role of a variable raises an immediate question: could we eliminate the need for the comment by choosing a better name?

In this case, we could replace `pad` by `spaceAroundGreeting`, or some such name. Then the comment would be unnecessary.

Of course, it takes longer to type `spaceAroundGreeting` than `pad`. But the time programmers take to type a name a few times (five times for this program) is negligible compared to the time maintainers take to figure out what they meant.

Another problem with Figure 9 is the mysterious numbers 2 and 3: 37

```
const int rows = pad * 2 + 3;
const string::size_type cols = greeting.size() + pad * 2 + 2;
```

Although there is a comment explaining the meaning of `pad`, there is no comment explaining the formulas `pad * 2 + 3` and `pad * 2 + 2`. Again, we can excuse the authors in this case, because the explanation appears in the book (Koenig and Moo 2000, page 18–22). In production code, however, these numbers should be accompanied by explanatory comments or even defined as constants.

We could write the definitions in a way that shows how the values are obtained. This requires more typing but should not make any difference to the compiled code:

```
const int rows = 1 + pad + 1 + pad + 1;
const string::size_type cols = 1 + pad + greeting.size() + pad + 1;
```

Why does the program use `int` as the type of `rows` — which seems quite natural — and the curious expression `string::size_type` as the type of `cols`? This type is used because it is the type returned by the function `string::size()`. It is an integer type, but we don't know which one (probably `unsigned long` but possibly something else). If we declare "`const int cols`", the compiler issues a warning: 38

```
frame.cpp(15) : warning C4267: 'initializing' :
   conversion from 'size_t' to 'const int', possible loss of data
```

We don't want warning messages when we compile, and one way to get rid of this message is to use the correct type.

> ***If the compiler issues warning errors, revise your code until they disappear.***

It is not always easy to eliminate warnings, but the effort is worthwhile. If it doesn't save your time now, it may save a maintainer's time later.

2.5 Counting

In everyday life, if we have N objects and want to identify them by numbers, we assign the numbers $1, 2, 3, \ldots, N$ to them. Another way to look at this is to say that the set of numbers forms a ***closed interval*** which we can write as either $1 \leq i \leq N$ or $[1, N]$.

Programmers are different. Given N objects, they number them $0, 1, 2, \ldots, N-1$. This set of numbers forms a ***semi-closed*** (or ***semi-open*** interval which we can write as either $0 \leq i < N$ or $[0, N)$. The advantages of semi-closed intervals include:

- They reduce the risk of "off-by-one" or "fencepost" errors.

  To see why off-by-one errors are called "fencepost errors", consider the length of a fence with N posts spaced 10 feet apart. The length of the fence is $10(N-1)$ feet: see Figure 11. 39

- The closed interval $[M, N]$ has $N - M + 1$ elements; the semi-closed interval $[M, N)$ has $N - M$ elements, which is easier to remember and calculate.

- In particular, the closed interval $[M, N]$ is empty if $M > N$, which many people find counter-intuitive. The semi-closed interval $[M, N)$ is empty if $M = N$, which is easier to remember and check.

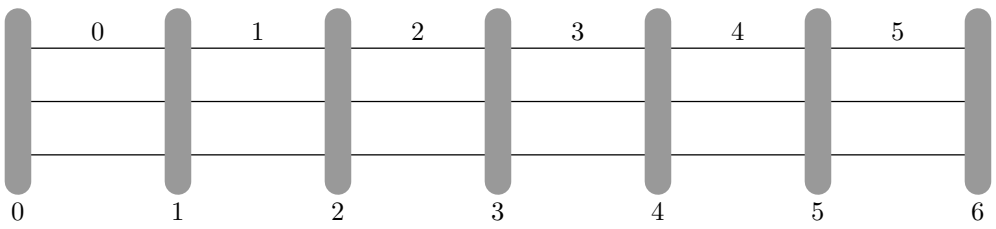


Figure 11: Fencepost numbering

- Semi-closed intervals are easier to join together. Compare

$$[A, B),\ [B, C),\ [C, D), \ldots$$

to

$$[A, B-1],\ [B, C-1],\ [C, D-1], \ldots$$

- The index of the first element of an array A in C++ is 0. The address of the I'th element of the array is $\&A + sI$ where $\&A$ is the address of the array and s is the size of one element.

Typical C++ loops do ***not*** have the form

```
for (int i = M; i <= N; ++i) ....
```

in which `M` is often 1. Instead, they have the form

```
for (int i = M; i < N; ++i) ....
```

in which `M` is often zero. Note that, in the first case, `N` is the last element processed but, in the second case, `N` is the first element ***not*** processed. In fact, we will see later that there are good reasons for writing the termination condition as `!=` rather than `<=`:

```
for (int i = M; i != N; ++i) ....
```

> ***Start a range with the index of the first item to be processed; end the range with the index of the first item <u>not</u> processed. The first index of a range is often 0.***

2.6 Loop Design

We discuss the design of loops with the assumption that they are going to be `while` loops. If they later turn out to have the appropriate pattern, we can convert them to `for` loops. With experience, we learn to recognize loops that have the `for`-loop pattern in advance and avoid the conversion step.

We design `while` loops using the following schema (the numbers are for reference, not part of the code):

```
1       ⟨initialize⟩
2       //  I
3       while (C)
4       {
5           //  I ∧ C
6           ⟨body⟩
7           //  I
8       }
9       //  I ∧ ¬ C
```

- The comment on line 2 says that, after initialization, the condition I is true. I is the ***invariant*** of the loop.
- The comment on line 5 says that, at the start of the body of the loop, the loop invariant I and the loop condition C are both true. This provides an ***assumption*** that we can use in coding the loop.
- The comment on line 7 says that, after the body of the loop has been executed, the invariant I is still true (this is what being an ***invariant*** means).
- The comment on line 9 says that, when the loop exits, the invariant I is true but the loop condition C is false.

> Good programmers instinctively know what the invariant should be in a `while` loop.
> — Joel Spolsky, *Joel on Software*, page 164.

2.6.1 Counting Lines

As a simple example of loop design, we consider the problem of writing `rows` lines of output, as in Figure 9. We use a counter `r` to count the number of lines written and, following the convention for initializing counter, we will initialize it to zero. We will make the minor change of writing `ROWS` rather than `rows`, to indicate that `ROWS` is a constant and to improve readability.

Here is a suitable invariant: *r lines have been written*. Note that initializing r to zero makes the invariant true, because we haven't written any lines yet.

If we have printed *ROWS* lines, there is no more to do. Consequently, the condition for the `while` loop is `r != ROWS` and the code begins: 41

```
1       int r = 0;
2       // r lines have been written
3       while (r != ROWS)
4       {
5           // r lines have been written and r != ROWS
```

The body of the loop must generate one line of output. We don't care (for this exercise) what that output will be, so we will just write a `cout` statement. The body must also count the lines produced. Thus the code continues:

```
6           cout << .... << endl;
6.1         // r+1 lines have been written
6.2         ++r;
7           // r lines have been written
8       }
9       // r lines have been written and not (r != ROWS)
```

Line 6 generates one line of output. This ***invalidates*** the invariant, as the comment on line 6.1 shows. Incrementing `r` makes the invariant valid again. We note that an invariant is not ***always*** true, but is true at certain well-defined points in the program. Also, whenever we perform an action that invalidates the invariant, we must perform another action (`++r` in this case) that makes it valid again.

Line 9 can be simplified as follows: 42

```
    r lines have been written and not (r != ROWS)
⇒   r lines have been written and r == ROWS
⇒   ROWS lines have been written
```

which is exactly what we needed.

Additional points:

- If we had written the `while` condition as `r < ROWS`, this reasoning would lead to a different conclusion:

  ```
      r lines have been written and not (r < ROWS)
  ⇒   r lines have been written and r >= ROWS
  ```

  That is, we could claim only that the code generates ***at least*** `ROWS` lines of output. This is correct, but it is less precise than the original conclusion, which is that the code generates ***exactly*** `ROWS` lines of output. One advantage of `!=` over `<` as a `while` condition is that it gives us a more precise conclusion. (This advantage was first pointed out by Dijkstra (1976, page 56n). We will discuss other advantages later, in connection with the STL.)

- This is an example of the situation mentioned above: the final code matches the pattern of the `for` statement and we can write the solution with a `for` loop. The invariant still applies: 43

  ```
  for (int r = 0; r != ROWS; ++r)
     // r lines have been written
     cout << .... endl;
  // ROWS lines have been written
  ```

- In addition to the invariant, which expresses something that does not change, we need something in the loop body that ***does*** change. Otherwise, the loop condition would never be satisfied and the loop would never terminate. In this example, the thing that changes is obviously the row counter; in other cases, it might not be so obvious.

- Suppose that we start counting from 1. A plausible invariant is: "`r` is the next line to be written", but this turns out not to be an invariant, because it is not true after we have written the last line. An invariant that works is "`r-1` lines have been written", which yields

```
1       int r = 1;
2       // r-1 lines have been written
3       while (r <= ROWS)
4       {
5           // r-1 lines have been written and r <= ROWS
6           cout << .... << endl;
6.1         // r lines have been written
6.2         ++r;
7           // r-1 lines have been written
8       }
9       // r-1 lines have been written and not (r <= ROWS)
```

Figure 12: Starting from 1

the following code in Figure 12. The code is correct, but it is more complicated and error-prone than the solution that counts from zero. As previously mentioned, the last line implies only that ***at least*** `ROWS` lines have been written rather than ***exactly*** `ROWS` lines have been written. 44

2.6.2 Finding Roots by Bisection

Suppose that f is a continuous real-valued function, $A < B$, and $f(A) \leq 0$ and $f(B) > 0$. Then a fundamental theorem of real analysis says that there must be a value of x such that $f(x) = 0$ and $A \leq x < B$; that is, a ***root*** (or ***zero***) of f. Using the bisection method, we find a sequence of approximations to x by halving the interval $[A, B)$ yielding smaller intervals $[a, b)$. Figure 13 illustrates the process. 45

Part of the invariant is $f(a) \leq 0 \wedge f(b) > 0$. This ensures that there is a root of f in $[a, b)$. To be complete, we will also require $a < b$. We cannot expect to find the root exactly, so we will stop when the interval is sufficiently small; specifically, when $b - a < \varepsilon$. This suggests that the loop condition should be $\neg(b - a < \varepsilon)$, which is equivalent to $b - a \geq \varepsilon$. Thus we have: 46

```
double a = A;
double b = B;
// I ≡ f(a) ≤ 0 ∧ f(b) > 0 ∧ a < b
while (b - a >= eps)

   ....

// I ∧ b − a < ε
```

Note that the final comment, obtained by and'ing the invariant and the negation of the `while` condition, is what we need: there is a root within a small interval.

To complete the loop body, we find the midpoint of the interval $[a, b)$, which is at $m = (b - a)/2$. If $f(m) > 0$, there must be a root between a and m. If $f(m) \leq 0$, there must be a root between m and b. We can write the code below. Note carefully how the `if` statement maintains the invariant. 47

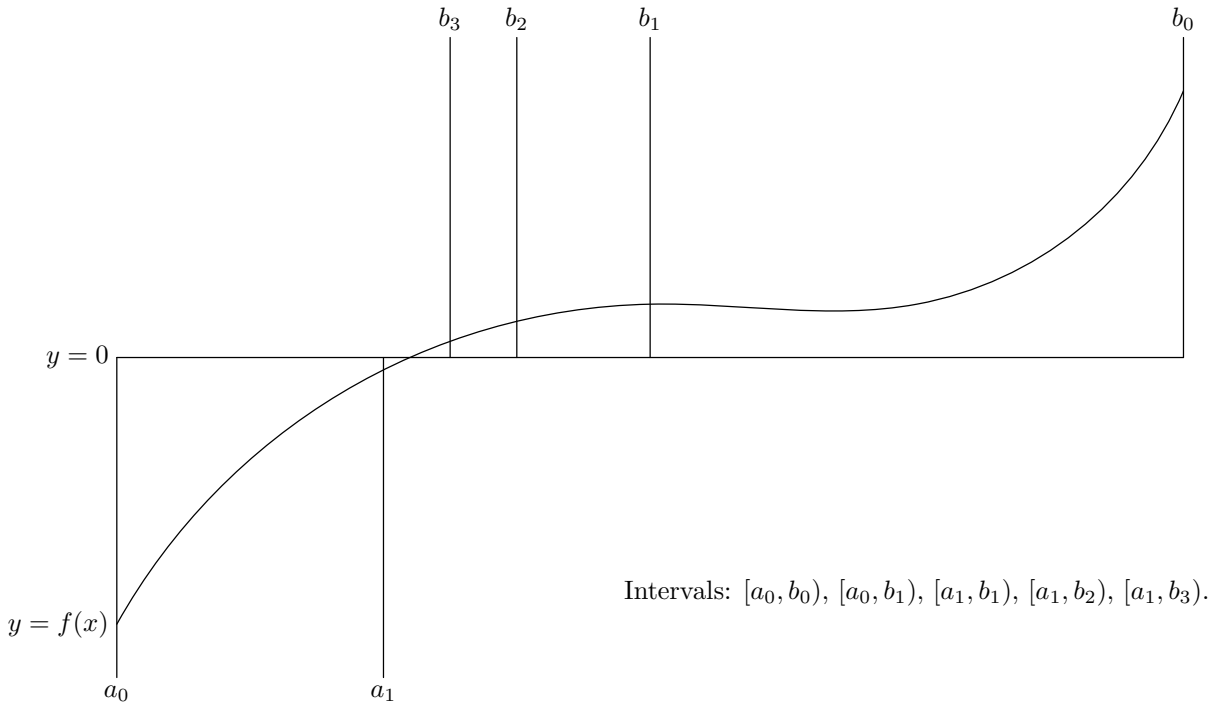


Figure 13: Finding zeroes by bisection

```
double a = A;
double b = B;
//  I  ≡  f(a) ≤ 0  ∧  f(b)  >  0  ∧  a  <  b
while (b - a >= tol)
{
   double m = 0.5 * (a + b);
   if (f(m) > 0)
      b = m;
   else
      a = m;
   //  I
}
//  I  ∧  b  −  a  <  ε
```

Figure 14 on page 31 expands this idea into a complete function and a test program. The `assert` statement is discussed in Section 2.7. When this program is run, it displays:

```
pi = 3.14159
e  = 2.71828
Assertion failed: a < b && f(a) <= 0 && f(b) > 0 &&
                  "solve: precondition violation",
  file f:\courses\comp6441\src\bisect\bisect.cpp, line 13
```

```
#include <iostream>
#include <cassert>
#include <cmath>

using namespace std;

double bisect(
    double f(double),
    double a,
    double b,
    double tol = 1e-6 )
{
    if (a > b)
        return bisect(f, b, a, tol);

    assert(a < b && f(a) <= 0 && f(b) > 0 &&
          "bisect: precondition violation");

    while (b - a > tol)
    {
        // a < b && f(a) <= 0 && f(b) > 0
        double m = 0.5 * (a + b);
        if (f(m) > 0)
            b = m; // f(b) > 0
        else
            a = m; // f(a) <= 0
    }
    return 0.5 * (a + b);
}

double logm(double x)
{
    return log(x) - 1;
}

int main()
{
    cout << "pi = " << 0.5 * bisect(sin, 4, 8) << endl;
    cout << "e  = " << bisect(logm, 0.5, 3) << endl;
    cout << "e  = " << bisect(logm, 3, 3) << endl;
}
```

Figure 14: Finding zeroes by bisection

2.6.3 Maximum Subsequence

The following problem arises in pattern recognition: find the maximum contiguous subvector of a one-dimensional vector (see (Bentley 1986, pp. 69–80)). The problem is trivial if all of the values in the vector are positive, because the maximum subvector is just the whole vector. If some of the values are negative, the problem is interesting. For example, given the vector 51

$$31 \quad -41 \quad 59 \quad 26 \quad -53 \quad 58 \quad 97 \quad -93 \quad -23 \quad 84$$

our algorithm should find the subvector

$$59 \quad 26 \quad -53 \quad 58 \quad 97$$

We assume that an empty subvector has sum 0. This implies that the maximum subvector can never be negative because, if we had a negative subvector, we could always replace it with the (larger) adjacent empty subvector.

There is an obvious solution: we can simply sum ***all possible*** subvectors and note which one has the largest sum. We assume that the given vector has N elements. We need three nested loops: one to choose the first element of the subvector, one to choose the last element, and one to sum the elements in between. Figure 15 shows a solution based on this idea. It uses `max`, a standard 52
C++ library function that returns the greater of its two arguments.

```
int maxSoFar = 0;
for (int i = 0; i < N; ++i)
{
    for (int k = i; k < N; ++k)
    {
        int sum = 0;
        for (int j = i; j <= k; ++j)
            sum += v[j];
        maxSoFar = max(maxSoFar, sum);
    }
}
```

Figure 15: Maximum subvector: first attempt

The algorithm of Figure 15 has complexity $\mathcal{O}(N^3)$.[6] It is not efficient and, by inspecting it carefully, we can see how to do better. During each cycle of the outer loop, we can use a single loop to sum all subvectors starting at that point. This gives the second version of the algorithm, with two nested loops and complexity $\mathcal{O}(N^2)$, shown in Figure 16. 53

Many, perhaps most, programmers would give up at this point and simply assume that quadratic complexity is the best that can be achieved. But, being more persistent, we will seek a better solution using invariants.

Here is a useful, general technique for solving problems with one-dimensional vectors: process the vector one element at a time, maintaining and updating as much information as is needed to proceed to the next step. Specifically, suppose we are about to process element `i`. What useful information can we obtain?

[6] Section 2.8 explains the notation $\mathcal{O}(N^3)$.

```
int maxSoFar = 0;
for (int i = 0; i < N; ++i)
{
    int sum = 0;
    for (int j = i; j < N; ++j)
    {
        sum += v[j];
        maxSoFar = max(maxSoFar, sum);
    }
}
```

Figure 16: Maximum subvector: second attempt

The first point to notice is that if we have a subvector "ending here", we can update it simply by adding `v[i]` to it. This will give us a new subvector "ending here" that we can keep if it is bigger than anything we have already seen.

The second point to notice is that we can remember the value of the largest subvector seen "so far" (this corresponds to what "we have already seen" in the previous sentence). The invariant that we need is:

$$\texttt{maxEndingHere} = \text{the largest subvector that ends here}$$
$$\texttt{maxSoFar} \equiv \text{the largest subvector we have seen so far}$$

To make the invariant true initially, we set both variables to zero. When we examine `v[i]`, we note that `maxEndingHere` will not get smaller if `v[i]` > 0. Figure 17 shows the final version of the algorithm. This version requires time proportional to the length of the sequence. This is much better than the first version, which required time proportional to the ***cube*** of the length of the sequence.

54

```
int maxSoFar = 0;
int maxEndingHere = 0;
for (int i = 0; i < N; ++i)
{
    maxEndingHere = max(maxEndingHere + v[i], 0);
    maxSoFar = max(maxSoFar, maxEndingHere);
}
```

Figure 17: Maximum subvector: an efficient solution

2.7 Assertions

The program in Figure 14 contains an ***assertion***:

55

```
assert(a < b && f(a) <= 0 && f(b) > 0 &&
       "bisect: precondition violation");
```

As with most programming constructs, there are three things that are useful to know about assertions: syntax (what do we write?); semantics (what happens?); and pragmatics (when and why do we use assertions?).

Syntax Any code unit that uses assertions must contain either the (preferred) new-style directive

```
#include <cassert>
```

or the old-style directive

```
#include <assert.h>
```

The assert statement itself has the form

`assert(` ⟨*condition*⟩ `);`

Semantics

- When the program executes, the ⟨*condition*⟩ is evaluated.
- If the ⟨*condition*⟩ yields `true`, or anything equivalent to `true`, the `assert` statement has ***no effect***.
- If the ⟨*condition*⟩ yields `false`, or anything equivalent to `false` (i.e., any kind of zero), the program is terminated with an error message.

The precise form of the error message depends on the compiler. In general, it will contain the text of the ⟨*condition*⟩, the name of the file in which the `assert` statement occurs, and the line number of the statement within the file.

Pragmatics Here is a reliable guide to the use of assertions:

> ***If an assertion fails, there is a logical error in the program.***

Think of `assert(`C`)` as saying "I (the programmer) believe that C should always be true at this point in the program". Then the failure of an assertion implies that the programmer's belief was mistaken, which further implies that there was something wrong with the reasoning and therefore something wrong with the program.

Assertions are useful for expressing preconditions, postconditions, and invariants. 56

- A ***precondition*** is a condition that should be true on entry to a function. It imposes an obligation on the caller to ensure that the arguments passed to the function are appropriate. The assertion in Figure 14 is of this form.
- A ***postcondition*** is a condition that should be true when a function returns. It is a promise by the function to the caller that the function has done its job correctly.
- An ***invariant*** is a condition that should be true at certain, well-defined points in the program. For example, a loop invariant.

A useful trick for assertions is to append "`&&` ⟨*string*⟩" to the condition, in which ⟨*string*⟩ describes what has gone wrong. This does not change the value of the condition, because any string is considered to be non-zero and therefore `true`, but it may improve the diagnostic issued by the compiler.

For example, when the assertion 57

```
assert( a < b && f(a) <= 0 && f(b) > 0 &&
        "bisect: precondition violation");
```

in the program of Figure 14, the run-time system displays

```
Assertion failed: a < b && f(a) <= 0 && f(b) > 0 &&
                  "solve: precondition violation",
file f:\courses\comp6441\src\bisect\bisect.cpp, line 13
```

Assertions can be disabled by writing "`#define NDEBUG`" ***before*** "`#include <cassert>`". If the assertions were not failing, the only effect of this will be to save a few nanoseconds of execution time. Since conditions that were evaluated with `NDEBUG` undefined are no longer evaluated with `NDEBUG` defined, it is important that:

> ***Asserted conditions must not have side-effects.***

Exceptions provide another way of recovering from errors; we will discuss them later in the course (Section 9.2).

2.8 Oh Notation

It is useful to have a concise way of describing how long a program or algorithm takes to run. The conventional way of doing this is to use "big-oh" notation.

The time taken to compute something usually depends on the size of the input. We will use n to stand, in a general way, for this size. For example, n might be the number of characters to be read, or the number of nodes of a graph to be processed. If the time required is ***independent*** of n, we say that the ***time complexity*** is $\mathcal{O}(1)$. If the time required increases linearly with n, we say that the complexity is $\mathcal{O}(n)$.

Formally, $\mathcal{O}(\cdot)$ defines a ***set*** of functions: 58

$$g(n) \quad \in \quad \mathcal{O}(f(n))$$

if and only if there are constants A and M such that, for all $N > M$, $g(N) < A\, f(N)$.

Big-oh notation does two things: it singles out the dominant term of a complicated expression, and it ignores constant factors. For example,

$$\begin{aligned} n^2 \quad &\in \quad \mathcal{O}(n^2) \\ \text{and} \qquad 1000000n^2 \quad &\in \quad \mathcal{O}(n^2) \end{aligned}$$

because we can chose $A = 1000001$ in the definition above. Also

$$n^3 + 100000n^2 + 100000n + 100000 \quad \in \quad \mathcal{O}(n^3)$$

because, for large enough n, the first term dominates the others. By similar reasoning

$$n + 10^{-10}n^2 \quad \in \quad \mathcal{O}(n^2)$$

even though the second term looks very small.

Informally, we don't say "is a member of $\mathcal{O}(n^2)$" but, less precisely, "the complexity is $\mathcal{O}(n^2)$" (or whatever the complexity actually is). 59

Figure 18 shows a small part of the ***complexity hierarchy***. Each lines defines a set of functions that is a proper subset of the set defined on the next line. So, for example, $\mathcal{O}(n) \subset \mathcal{O}(n \log n)$ (all linear functions are log-linear), and so on.

Very few algorithms are $\mathcal{O}(1)$. We use this set to describe operations that have a constant time bound. For example, "reading a character" is (or at least should be) $\mathcal{O}(1)$. Logarithmic and linear algorithms are good. Log-linear algorithms are acceptable: sorting, for example, is log-linear[7]. Polynomial algorithms, $\mathcal{O}(n^k)$ with $k > 2$, tend to be useful only for small sizes of problem.

Exponential and factorial algorithms are useless except for very small problems. A problem whose best known solution has exponential or factorial complexity is called ***intractable***. Such problems must be solved by looking for approximations rather than exact results. One of the the best-known intractable problems is TSP: the "travelling salesperson problem". A salesperson must make a certain number of visits and the problem is to find an ordering of the visits that minimizes some quantity, such as cost or distance. The only known way to find an exact solution is to try all possible routes and note the minimal route. If n visits are required, the number of possible paths is $\mathcal{O}(n!)$, making the exact solution infeasible if n is in the hundreds or even thousands.

Note that TSP is typical of problem descriptions. We are not really interested at all in travelling salespersons and, in any case, their actual problems (which might involve 20 visits at most) are easily solved. But TSP represents the generic problem of finding a minimal path in a weighted graph, and many practical problems can be put into this abstract form. One example is: find the quickest path for a drilling machine that has to drill several thousand holes in a printed-circuit board. 60

Figure 19 shows the progress that has been made in solving three-dimensional elliptic partial differential equations. Equations of this kind must be solved for VLSI simulation, oil prediction, reactor simulation, airfoil simulation, and other significant problems. The difference between $\mathcal{O}(N^7)$ and $\mathcal{O}(N^3)$ corresponds to a factor of N^4, or a million times for a problem for which $N = 100$.

It is often claimed that hardware has improved more rapidly than software. For some problems, however, the improvements in software have been just as dramatic as those for hardware. Putting the two together, a modern supercomputer can solve differential equations more than a trillion ($10^6 \times 10^6 = 10^{12}$) times faster than was possible in 1945.

[7] Provided that you don't use bubblesort.

Function	Condition	Description
$\mathcal{O}(1)$		constant
$\mathcal{O}(\log n)$		logarithmic
$\mathcal{O}(\sqrt{n})$		square-root
$\mathcal{O}(n)$		linear
$\mathcal{O}(n \log n)$		log-linear
$\mathcal{O}(n^2)$		quadratic
$\mathcal{O}(n^3)$		cubic
$\mathcal{O}(n^k)$	$k > 1$	polynomial
$\mathcal{O}(a^n)$	$a > 1$	exponential
$\mathcal{O}(n!)$		factorial

Figure 18: Part of the complexity hierarchy

Method	Year	Complexity
Gaussian elimination	1945	$\mathcal{O}(N^7)$
SOR iteration (suboptimal parameters)	1954	$\mathcal{O}(N^5)$
SOR iteration (optimal parameters)	1960	$\mathcal{O}(N^4 \log N)$
Cyclic reduction	1970	$\mathcal{O}(N^3 \log N)$
Multigrid	1978	$\mathcal{O}(N^3)$

Figure 19: Solving three-dimensional elliptic partial differential equations (adapted from ***Numerical Methods, Software, and Analysis***, by John Rice (McGraw-Hill, 1983))

3 Batches of Data

This section covers roughly the same material as Chapter 3 of (Koenig and Moo 2000). However, the ordering of topics is different and some of the examples are abstracted (for example, we discuss the mean/median of a general set of numbers rather than student marks) and there is some additional material (for example, reading from streams other than `cin`).

3.1 Lvalues and Rvalues

The names "Lvalue" and "Rvalue" are derived from the assignment statement 61

```
v = e;
```

Although the assignment looks symmetrical, it is not. When it is executed, the right side, `e`, must yield a ***value*** and the left side, `v`, must yield a ***memory address*** in which the value of `e` can be stored.

> ***Anything which can appear on the left of = (that is, anything that can represent an address) is called an Lvalue.***

> ***Anything which can appear on the right of = (that is, anything that can yield a value) is called an Rvalue.***

"Lvalue" and "Rvalue" are quite often written without capitals, as "lvalue" and "rvalue". We use initial capitals in these notes for clarity and emphasis.

All Lvalues are Rvalues because, having obtained an address, we can find the Rvalue stored at that address. The operation of obtaining an Rvalue from an Lvalue is called ***dereferencing***. But there are Rvalues that are not Lvalues. Lvalues include: simple variables (`x`); array components (`a[i]`); fields of objects (`o.f`); and a few other more exotic things. Rvalues that are not Lvalues include literals (for example, `67`, `"this is a string"`, `true`) and expressions.

3.2 Passing Arguments

In the function definition

```
double sqr(double x) { return x * x; }
```

the list `(double x)` is a ***parameter list*** and `x` is a ***parameter***. When we call the function, as in

```
cout << sqr(2.71828) << endl;
```

the expression 2.71828 is an ***argument*** that is ***passed*** to the function.

Some authors say "formal parameter" instead of "parameter" and "actual parameter" instead of "argument". We will use the shorter (and more correct) terms ***parameter*** and ***argument***. ("Parameter" can be roughly translated from Greek as "unknown quantity". When we write `sqr` above, we don't know the value of `x`, so it is reasonable to call `x` a parameter.)

Functions have parameters. When a function is called, arguments are passed to it.

There are a number of ways of passing arguments in C++. Three of them suffice for most applications. The program in Figure 20 includes examples of these three. When run, it produces the following output:[8]

62 63 64

```
Pass by value: 65
Pass by reference: 66
Pass by constant reference: 55
```

Pass by value: the parameter is unqualified. For example, `double x`. When the function is called, the run-time system makes a copy of the argument and passes the copy to the function. The function can change the value of its parameter but these changes have no effect in the calling environment. This explains the output "`65`": the function increments its parameter `k`, but the argument `pbv` remains unchanged.

Pass by reference: the parameter is qualified by `&`, which is read as "reference to". When the function is called, the run-time system passes a ***reference*** to the argument to the function. Pass by reference is usually implemented by passing an address. When the parameter is used in the function body, it is effectively a synonym for the argument. Any change made to the parameter is also made to the argument. This explains the output "`66`": the function increments its parameter `k`, and the argument `pbr` is also incremented.

An argument to be passed by reference must be an Lvalue. The call `passByReference(5)` does not compile because 5 is not an Lvalue.

Pass by constant reference: the parameter is qualified by `const &`, which is read "const reference to". The run-time system passes an address, but the compiler ensures that this address is used only as an Rvalue. Changing the value of the parameter in the function body is not allowed.

Use the following rules when deciding how to pass an argument: 65

1. ***Pass small arguments by value.***

   Addresses are used for passing by reference. An address is 4 bytes or, on some modern machines, 8 bytes. If an argument is smaller, or not much bigger, than an address, it is usually passed by value. In practice, this means that the standard types (`bool`, `char`, `short`, `int`, `float`, `double`, etc.) are usually passed by value, and user-defined types (that is, instances of classes) are usually passed by reference.

2. ***Pass large objects by constant reference.***

   "Constant reference" is usually considered to be the "default" mode for C++. (Default is in quotes because constant reference is not the compiler's default: it must be selected explicitly by the user.) It should be used for large objects, including instances of user-defined classes, except when there is a good reason not to use it.

[8] These modes are sometimes referred to as "call by value", " call by reference", etc. They mean the same thing but "pass" seems more precise than "call".

```
#include <iostream>

using namespace std;

void passByValue(int k)
{
    ++k;
    cout << k;
}

void passByReference(int & k)
{
    ++k;
    cout << k;
}

void passByConstantReference(const int & k)
{
//  ++k;                                Not allowed!
    cout << k;
}

int main()
{
    cout << "Pass by value: ";
    int pbv = 5;
    passByValue(pbv);
    cout << pbv;

    cout << endl << "Pass by reference: ";
    int pbr = 5;
    passByReference(pbr);
    cout << pbr;
//  passByReference(5);                 Not allowed!

    cout << endl << "Pass by constant reference: ";
    int pbcr = 5;
    passByConstantReference(pbcr);
    cout << pbcr << endl;
}
```

Figure 20: Passing arguments

3. ***Pass by reference.***

   Pass by reference should be used ***only*** when changes made by the function must be passed back to the caller. The usual way of returning results is to use a `return` statement. However, `return` can return only one value and it is sometimes necessary to return more than one value. In these and similar situations, reference parameters may be used to return several items of information.

```
int main()
{
    cout << "Enter observations, terminate with ^D:" << endl;
    int obsCount = 0;
    double sum = 0;
    double observation;
    while (cin >> observation)
    {
        sum += observation;
        ++obsCount;
    }
    cout <<
        "The mean of " << obsCount <<
        " observations is " << sum / obsCount <<
        endl;
    return 0;
}
```

Figure 21: Finding the mean

3.3 Reading Data

66

Figure 21 shows a program that reads a list of numbers, computes their mean, and displays it. This is what the console window looks like after running this program: 67

```
Enter observations, terminate with ^Z:
2.6 9.32 5.67 2.1 6.78 5.1 ^Z
The mean of 6 observations is 5.26167
```

The character `^Z` (Ctrl–Z on the keyboard) is used to indicate the end of an input stream reading from the keyboard (i.e., `cin`).

The important new feature of this program is the loop controlled by

```
while (cin >> observation)
```

The condition raises two questions: first, why does it work? Second, why do other, seemingly more natural constructions, ***not*** work?

Here is why it works:

1. The expression `cin >> observation` returns an updated value of `cin`. Note that this is ***not*** a boolean value (`true` or `false`). Consequently,

   ```
   if (cin >> observation) ....
   ```

   is equivalent to

   ```
   cin >> observation;
   if (cin) ....
   ```

2. When `cin` appears in a condition context, the compiler attempts to convert it into something that can be considered boolean. By means of a technical trick, this boolean value is `true` if the stream is in a "good" state and `false` if the stream is in a "bad" state.

3. If the stream is in a good state, then the operation `cin >> observation` must have succeeded and a value for `observation` was read successfully.

4. If the stream is in a bad state, the operation failed and the value of `observation` is undefined.

5. Looking at the way in which `while (cin >> observation)` is used in the program above, we see that everything works out nicely: the program ***either*** successfully reads a value and processes it ***or*** does not manage to read a value and terminates the loop.

The "technical trick" mentioned in step 2, in case you are interested, is as follows. Since the compiler cannot interpret an instance of `istream` (the class of `cin`) as a boolean, it looks for a conversion that would enable it to do so. Class `istream` provides a conversion from `istream` to `void*`, the type of pointers that point to nothing in particular. If the stream is in a good state, the result of this conversion is a non-null pointer (probably, but not necessarily, the address of the stream object), which is considered `true`. If the stream is in a bad state, the conversion yields a null pointer, which is zero, and is therefore considered `false`.

There are several reasons why a stream can get into a bad state. The most likely reason, and the one we expect here, is that the program has reached the end of the stream (indicated, in this example, by the `^Z` key). Another reason is that a disk has failed, although obviously this does not apply to `cin`.

```
void v1()
{
   vector<int> v;
   int k;
   while (cin >> k)
   {
      v.push_back(k);
   }
   for (vector<int>::const_iterator it = v.begin(); it != v.end(); ++it)
      cout << *it << ' ';
   cout << endl;
}
```

Figure 22: Testing end-of-file

The second question we have to answer is: why do other approaches not work? Problems arise because the normal input mode skips blanks to find the next datum. We will use the program in Figure 22 to explore the potential problems:

This program works correctly whether or not there is white space between the last dsatum and the end of the file. In this example and subsequent examples, end-of-file is indicated by `^Z`. Note the blank after `4` in the second example.

```
1 2 3 4^Z
1 2 3 4

1 2 3 4 ^Z
1 2 3 4
```

Class `stream` provides a function `eof` ("end-of-file") that yields `false` except at the end of the stream, where it returns `true`. A function that returns a boolean value is called a ***predicate***; `eof` is therefore a predicate. We can test for end-of-file *before* attempting to read the input: 68

```
while (!cin.eof())
{
   cin >> k;
   v.push_back(k);
}
```

This is what happens:

```
1 2 3 4^Z
1 2 3 4

1 2 3 4 ^Z
1 2 3 4 4
```

If there is no white space after the last datum, this code works correctly. If there is a blank, the following sequence of events occurs:

- The stream reads the last datum, `4`, correctly.
- Since the next character is a blank, `cin.eof()` is `false`.
- The stream reads the blank, finds no integer, and puts the stream into a "bad" state. The value of `k` is not changed.
- The value in `k`, which is still `4`, is stored in the vector *again*.

Another possibility is to test for end-of-file *after* reading: 69

```
while (true)
{
   cin >> k;
   if (cin.eof())
      break;
   v.push_back(k);
}
```

This code gives the following results:

```
1 2 3 4^Z
1 2 3

1 2 3 4 ^Z
1 2 3 4
```

If there is no blank after `4` then, after this datum has been read, `cin.eof()` is `true`. The loop terminates and `4` is not stored in the vector. When there is a blank, the program works correctly.

3.4 Reading from a file

```
#include <iostream>
#include <fstream>
#include <string>

int main()
{
    cout << "Enter file name: ";
    string fileName;
    cin >> fileName;
    ifstream fin(fileName.c_str());

    int obsCount = 0;
    double sum = 0;
    double observation;
    while (fin >> observation)
    {
        sum += observation;
        ++obsCount;
    }
    fin.close();
    cout <<
        "The mean of " << obsCount <<
        " observations is " << sum / obsCount <<
        endl;
}
```

Figure 23: Finding the mean from a file of numbers

One of the nice features of C++ stream is that they make most kinds of input and output look the same to the programmer. The program in Figure 23 asks the user for a file name, reads a list of numbers from the file, and displays the mean. Note the similarities between this program and Figure 21. Other points of note include:

70 71

- The directive "`#include <fstream>`" is required for any program that uses input or output files.

- The type `ifstream` is the type of input streams. The variable `fin` is an instance of this type.
- The constructor for `ifstream` needs a file name. Curiously, it cannot accept the file name as a `string`; instead, it requires a C style string, of type `const char*`. Consequently, we have to use the conversion function `c_str` to convert the `string` to a `const char*`. Calling the constructor with a file name has the effect of opening the file.
- After the file has been opened, we use `fin` in exactly the same way that we used `cin` in Figure 21.
- When the program has finished reading data from the file, it calls `fin.close()` to close the stream. This step is not strictly necessary, because the destructor, called at the end of the scope, would close the stream if it was still open. Nevertheless, it is good practice to explicitly close a stream that is no longer required by the program.
- The operations of constructing and opening a file can be separated. Instead of 72

  ```
  ifstream fin(fileName.c_str());
  ```

  we could have written:

  ```
  ifstream fin;
  // ....
  fin.open(fileName.c_str());
  ```

- Whenever a program tries to open a file for input, there is a possibility that the file does not exist. As above, we can use the stream object as a boolean to check whether the stream was opened successfully:

  ```
  ifstream fin;
  fin.open(fileName.c_str());
  if (!fin)
  {
      cerr << "Failed to open " << fileName << endl;
      return;
  }
  ```

- Another possibility would be to trigger an assertion failure when a file cannot be opened:

  ```
  ifstream fin;
  fin.open(fileName.c_str());
  assert(fin);
  ```

  However, this would go against the recommendations of Section 2.7. There are many reasons why opening a file might fail, and the failure does not imply that there is a ***logical*** fault in the program.

3.4.1 Writing to a file

Writing is very similar to reading. The class for output files is `ofstream`. We use the insert operator `<<` to write data to the file. The methods `open` and `close` work in the same way as they do for input files. Figure 24 shows a very simple program that writes to an output file. 73

```
int main()
{
    ofstream fout("randomnumbers.txt");
    for (int n = 0; n < 50; ++n)
        fout << rand() << '\n';
    fout.close();
    return 0;
}
```

Figure 24: Writing random numbers to a file

3.4.2 Stream States

In general, the states of an input stream are ***good***, ***bad***, and ***end-of-file***. These are not mutually exclusive. If the state is ***good***, it cannot also be either ***bad*** or ***end-of-file***. However, if the state is ***bad***, it may or may not be ***end-of-file***. The input statement `cin >> n`, where `n` is an integer, for example, will put the stream into a ***bad*** state if the next character in the stream is not a digit or "−". But reading can continue after calling `cin.clear()` to clear the bad state.

This section provides a brief overview of stream states. For details, consult a reference work, such as (Langer and Kreft 1999, pages 31–35).

The state of the stream is represented by four bits; Figure 25 shows their names and meanings. 74
Here are some examples of how the bits can get set:

- The program wants to read an integer and the next character in the stream is `'x'`. After the input operation, `failbit` is set and the stream position is unchanged.
- The program wants to read an integer. The only characters remaining in the file are "white space" (blanks, tabs, and newlines). After the input operation, `failbit` and `eofbit` are both set and the stream is positioned at end-of-file.
- The program wants to read an integer. The only characters remaining in the file are digits. After the input operation, `eofbit` is set and the stream is positioned at end-of-file.
- The program wants to read data from a disk file but the hardware (or operating system) reports that the disk is unreadble. After the operation, `badbit` is set and the stream cannot be used again.

Name	Meaning
`goodbit`	The stream is in a "good" state — nothing's wrong
`eofbit`	The stream is positioned at end-of-file — no more data can be read
`failbit`	An operation failed but recovery is possible
`badbit`	The stream has "lost integrity" and cannot be used any more

Figure 25: Stream bits and their meanings

75

Figure 26 shows how to test the stream bits. These functions are members of the stream classes. For example, if `fin` is an input file stream, then `fin.bad()` tests its `badbit`. There are several things to note about theses functions:

- The first five return `bool` values — that is, `true` or `false`.
- `fail()` returns `true` if `failbit` is set ***or if*** `badbit` ***is set***.
- `operator!()` is called by writing `!` before the stream name, as in

```
if (!fin)
    // failbit is set or badbit is set
else
    // file is OK
```

- `operator void*()` is called by writing the stream name in a context where an expression is expected. For example:

```
if (fin)
    // file is OK
else
    // failbit is set or badbit is set
```

- `operator void*()` is also called when we write, for example

```
if (fin >> data)
```

Function	Value/Effect
`bool good()`	None of the error flags are set
`bool eof()`	`eofbit` is set
`bool fail()`	`failbit` is set or `badbit` is set
`bool bad()`	`badbit` is set
`bool operator!()`	`failbit` is set or `badbit` is set
`operator void*()`	Null pointer if `fail()` and non-null pointer otherwise
`void clear()`	Set `goodbit` and clear the error bits

Figure 26: Reading and setting the stream bits

Output streams use the same state bits, but it is not often necessary to use them. An output stream is always positioned at end-of-file, ready for the next write. Output operations fail only when something unusual happens, such as a disk filling up with data.

3.4.3 Summary of file streams

The stream class hierarchy is quite large. For practical purposes, there are four useful kinds of stream: 76

`ifstream`: input file stream

`ofstream`: output file stream

`istringstream`: input string stream

`ostringstream`: output string stream

We will discuss string streams later. For each of these kinds of stream, there is another with "`w`" in front (for example, `wifstream`). These are streams of "wide" (16–bit or Unicode) characters.

File streams are opened by providing a file or path name. The name can be passed to the constructor or to the `open` method. When a file is no longer needed, the `close` method should be called to close it.

The extract operator `>>` reads date from an input file. The insert operator `<<` writes data to an output file. The right operand of an extractor must be a Lvalue. The right operand of an inserter is an Rvalue.

The right operand of an extractor or inserter may also be a ***manipulator***. Manipulators may extract or insert data, but they are usually used to control the state of the stream. We have already seen `endl`, which writes a new line and then flushes the buffer of an output stream.

Although `endl` is defined in `iostream`, most other manipulators are not. They are defined in `iomanip` and so we have to write `#include <iomanip>` in order to use them. Here is a selection of commonly used manipulators for output streams:

`left`: Start left-justifying output data (appropriate for strings) 77

`right`: Start right-justifying output data (appropriate for numbers)

`setprecision(`n`)`: Put n digits after the decimal point for `float` and `double` data

`setw(`n`)`: Write the next field using at least n character positions 78

`fixed`: Use fixed-point format for `float` and `double` data (for example, `3.1415926535`)

`scientific`: Use "scientific" format for `float` and `double` data (for example, `1.3e12`)

For example, the program shown is Figure 27 displays: 79 80

```
Alice          41    1169699.780
Boris        6500    3014133.560
Ching        1478    7915503.960
Daoust       4464    1605672.250
```

```
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <cstdlib>

using namespace std;

int main()
{
    vector<string> names;
    names.push_back("Alice");
    names.push_back("Boris");
    names.push_back("Ching");
    names.push_back("Daoust");

    for ( vector<string>::const_iterator it = names.begin();
          it != names.end();
          ++it)
    {
        cout << left << setw(8) << *it;
        cout << right << setw(10) << rand() % 10000;
        cout << fixed << setprecision(3) << setw(15) <<
                rand() * (rand() / 100.0);
        cout << endl;
    }
    return 0;
}
```

Figure 27: Output manipulators

3.5 Storing Data: STL Containers

For a few applications, such as computing an average, it is sufficient to read values; we do not have to store them. For most applications, it is useful or necessary to store values as we read them. The program in Figure 28 shows one way of doing this. 81

The new feature in this program is

```
vector<double> observations;
```

This declaration introduces `observations` as an instance of the class `vector<double>`. The ***template class*** `vector` is one of the ***containers*** provided by the Standard Template Library (STL). A template class is ***generic*** — it stands for many possible classes — and must be instantiated by providing a type. In this case, the type is `double`, giving us a `vector` of `double`s. When we use a vector, we must also write

```
#include <vector>
```

```
int main()
{
    cout << "Enter file name: ";
    string fileName;
    cin >> fileName;
    ifstream fin(fileName.c_str());

    vector<double> observations;
    double obs;

    while (fin >> obs)
    {
        observations.push_back(obs);
    }
    fin.close();

    for ( vector<double>::size_type i = 0;
          i != observations.size();
          ++i )
        cout << observations[i] << '\n';
}
```

Figure 28: Storing values in a vector

The relation between a generic class and an instantiated class is analogous to the relation between a function and a function application: 82

	Generic	Application
Function	log	$\log x$
Class	`vector`	`vector<double>`

There are various things we can do with a vector. The operation v.`push_back`(x) inserts the value x at the back end of the vector v. We do not have to specify the initial size of the vector, and we do not have to worry about how much stuff we put into it; storage is allocated automatically as the vector grows to the required size.

The argument for `push_back` is passed ***by value***. This means that the vector gets its own copy of the argument, which is usually what we want.

The method `size` returns the number of elements in the vector. As with `string`, the type of the size is not `int` but `vector<double>::size_type`. This is the type that we use for the controlled variable `i` in the final `for` loop of the program.

A vector can be subscripted, like an array. If v is a vector, $v[i]$ is the i'th element. In the program, `observations[i]` gives the `i`'th element of the vector of observations.

Vector indexes are not checked!

This means that, when we write `observations[20]`, for example, there is no check that this element exists. If it doesn't exist, the result will be garbage. Even worse, if we use this expression on the left of an assignment, we can write into any part of memory!

`<sermon>` This is an astonishing oversight. Here is Tony Hoare (1981):

> . . . . we asked our customers whether they wished us to provide an option to switch off these [array index] checks in the interests of efficiency on production runs. Unanimously, they urged us not to — they already know how frequently subscript errors occur on production runs where failure to detect them could be disastrous. I note with fear and horror that even in 1980, language designers and users have not learned this lesson. In any respectable branch of engineering, failure to observe such elementary precautions would have long ago been against the law.

In this example, the language was Algol 60 and the computer was an Elliott 503. The 503 weighed several tons, had 8K words of memory, and needed 7 μsec to add two numbers.[9]

Another 25 years have passed, and it is more than 40 years since Hoare's customers wanted subscript checking. Today's computers are around 10,000 times faster than the 503, and programmers are still concerned about the time taken to check subscripts. A high proportion of viruses, worms, phishers, and other kinds of malicious software exploit precisely this loophole.
`</sermon>`

Fortunately, there are safer ways of accessing the elements of a vector than using subscripting. One way is to use the function `at`. The call `array.at(i)` has the same effect as `array[i]` but checks that the index is within range and throws an exception otherwise.

Iterators are another alternative to subscripts and, in many cases, a better one. An iterator is an object that keeps track of the objects in a container and provides facilities for accessing them. The type of the iterator that we need is

```
vector<double>::const_iterator
```

Two of its values are `observations.begin()`, which refers to the first element of the vector `observations`, and `observations.end()`, which refers to the first element ***not*** in the vector — that is, one past the end. Iterators have increment (`++`) and decrement (`--`) operators. Iterators also have pointer-like behaviour: dereferencing an iterator yields an element of the container.

Putting all these things together, we can write the following loop to access the elements in the vector `observations`: 83

```
for (  vector<double>::const_iterator i = observations.begin();
       i != observations.end();
       ++i )
    cout << *i << '\n';
```

This code has two significant advantages over the original version:

- Using the iterator and, specifically, the function `end`, ensures that we access exactly the elements that are stored.

[9] `http://members.iinet.net.au/~daveb/history.html`.

```
f:\Courses\COMP6441\src\means\means.cpp(148): error C2784:
'bool std::operator <(const
    std::basic_string<_Elem,_Traits,_Alloc> &,const _Elem *)' :
could not deduce template argument for
'const std::basic_string<_Elem,_Traits,_Ax> &' from
'std::list<_Ty>::const_iterator'
        with
        [
            _Ty=double
        ]
f:\Courses\COMP6441\src\means\means.cpp(148): error C2784:
'bool std::operator <(const
    std::list<_Ty,_Alloc> &,const std::list<_Ty,_Alloc> &)' :
could not deduce template argument for
'const std::list<_Ty,_Ax> &' from
'std::list<_Ty>::const_iterator'
        with
        [
            _Ty=double
        ]
f:\Courses\COMP6441\src\means\means.cpp(148): error C2784:
'bool std::operator <(const
    std::list<_Ty,_Alloc> &,const std::list<_Ty,_Alloc> &)' :
could not deduce template argument for
'const std::list<_Ty,_Ax> &' from
'std::list<_Ty>::const_iterator'
        with
        [
            _Ty=double
        ]
```

Figure 29: Complaints from the compiler

- This code can be used with other kinds of container.

For example, if we replace each occurrence of `vector` by `list`:

```
list<double> observations;
....
for (  list<double>::const_iterator i = observations.begin();
   i != observations.end();
   ++i )
cout << *i << '\n';
```

the program compiles and runs with exactly the same effect.

However, if we replace `!=` in by `<` , the compiler complains — see Figure 29.[10] The problem is that list iterators, unlike vector iterators, provide equality (`==` and `!=`) but not ordering (`<`, etc.). 84

In this case, the compiler diagnostic starts with `bool std::operator <` and, since introducing `<` was the only change we made to the program, it is not hard to figure out that it is this operator

[10]The compiler actually produces 22 messages of this form; only the first three are shown here.

that caused the problem. Unfortunately, the STL can produce even worse error messages that can be very hard to interpret.[11]

Sorting the vector of observations is very easy. Only one extra line of code is required: 85

```
sort(observations.begin(), observations.end());
```

However, `sort` is not a part of `vector`; it is one of the algorithms provided by the STL. Consequently, we also need the directive

```
#include <algorithm>
```

3.5.1 Summary of STL

The STL provides containers, iterators, algorithms, function objects, and adaptors.

Containers are data structures used to store collections of data of a particular type. The operations available for a container, and the efficiency of the operations, depend on the underlying data structure. For example, `vector` provides array-like behaviour: elements can be accessed randomly, but inserting or deleting elements may be expensive. In contrast, a `list` can be accessed sequentially but not randomly, and provides efficient insertion and deletion.

The containers also include `set` for unordered data and `map` for key/value pairs without duplicates. The containers `multiset` and `multimap` are similar but allow duplicates.

Iterators provide access to the elements stored in containers. They are used to ***traverse*** containers (i.e., visit each element in turn) and to specify ***ranges*** (i.e., groups of consecutive elements in a container).

Algorithms provide standard operations on containers. There are algorithms for finding, searching, copying, swapping, sorting, and many other applications.

Function objects are objects that behave as functions. Function objects are needed in the STL because the compiler can sometimes select an appropriate object in a context where it could not select an appropriate function. However, there are also other uses of function objects.

Adaptors allow interface modification and increase the flexibility of the STL. Suppose we want a stack. There are several ways to implement a stack: we could use a vector, or a list, or some other type. The STL might provide a class for each combination (`StackVector`, `StackList`) and perhaps `Stack` as a default.

In fact, the STL separates abstract data types (such as `stack`) and representations (such as `vector` and `list`) and provides adaptors to fit them together. Thus we have: 86

`stack<T>:` stack of objects of type `T` with default implementation

`stack<T, vector<T> >:` stack of objects of type `T` with vector implementation

`stack<T, list<T> >:` stack of objects of type `T` with list implementation

[11] Some programmers write perl scripts to parse compiler diagnostics and pick out the key parts.

Assuming that the STL provides M container classes and N algorithms, it is tempting to assume that there are $M \times N$ ways of using it, because we should be able to apply each algorithm to each container. However, this is not in fact how the STL works. Instead:

- A container/algorithm combination works only if the algorithm is appropriate for the data structure used by the container.
- If the combination does work, its performance is guaranteed in terms of a complexity class, e.g., $\mathcal{O}(N)$.

We have already seen an example of this in Section 3.5. Suppose that `i` and `j` are iterators for a container and that `*i` and `*j` are the corresponding elements. We would expect `==` and `!=` to be defined for the iterators. It is also reasonable to expect `++`, because iterators are supposed to proved a way of stepping through the container. But what does `i < j` mean? Presumably, something like "`*i` appears before `*j` in the container". This is easy to evaluate if the container is a `vector`, because vectors are indexed by integers (or perhaps pointers), which can be compared. But evaluating `i < j` for a linked list is inefficient, because it requires traversing the list. This is why the iterator for a vector provides `<` but the iterator for list does not.

It is important to check that the algorithm you want to use works with the container that you are using. The penalty for not checking is weird error messages. For example, 87

```
void main()
{
    std::vector<int> v;
    std::stable_sort(v.begin(), v.end());
}
```

compiles correctly, but

```
void main()
{
    std::list<int> v;
    std::stable_sort(v.begin(), v.end());
}
```

produces the message 88

```
stl_algo.h: In function `void __merge_sort_loop<_List_iterator
  <int,int &,int *>, int *, int>(_List_iterator<int,int &,int *>,
  _List_iterator<int,int &,int *>, int *, int)':
stl_algo.h:1448:   instantiated from `__merge_sort_with_buffer
  <_List_iterator<int,int &,int *>, int *, int>(
   _List_iterator<int,int &,int *>, _List_iterator<int,int &,int *>, int *, int *)'
stl_algo.h:1485:   instantiated from `__stable_sort_adaptive<
  _List_iterator<int,int &,int *>, int *, int>(_List_iterator
  <int,int &,int *>, _List_iterator<int,int &,int *>, int *, int)'
stl_algo.h:1524:   instantiated from here
stl_algo.h:1377: no match for `_List_iterator<int,int &,int *> & -
  _List_iterator<int,int &,int *> &'
```

Why doesn't the STL generate more useful diagnostics? The reason is that it is based on templates. The compiler first expands all template applications and then tries to compile the resulting code. If the code contains errors, the compiler cannot trace back to the origin of those errors, saying perhaps "`list` does not provide `stable_sort`", but can only report problems with the code that it has.

Various objects and values associated with containers have types. These types may depend on the type of the elements for the container. For example, the type of an iterator for `vector<int>` may not be the same as the type of an iterator for `vector<double>`. Consequently, the container classes must provide the types we need. In fact, we have already seen expressions such as `vector<double>::const_iterator`, which is the type of `const` iterators for a vector of `double`s.

These type names can get quite long. It is common practice to use `typedef` directives to abbreviate them. A `typedef` has the form

89

`typedef` ⟨*type expression*⟩ ⟨*identifier*⟩

and defines ⟨*identifier*⟩ to be a synonym for ⟨*type expression*⟩. For example, after

```
typedef vector<double>::const_iterator vci;
```

we can write `vci` instead of the longer expression.

4 Application: Grading a class

The program developed in this section is similar, but not identical, to the grading program described in Chapter 4 of (Koenig and Moo 2000).

4.1 Problem Statement

Here is a statement of the problem that it solves.

90
91
92

> **Problem Statement:** The program reads a file of marks (e.g., Figure 30) and writes a report of grades (e.g., Figure 31). Each line of the marks file consists of a name, a mark for the midterm, a mark for the final, and marks for assignments. The number of assignments is not fixed; students do as many as they want to. The name is a single name with no embedded blanks.
>
> A line of the output file is similar to a line of the input file, but the first number is the total mark, computed as 20% of the midterm mark plus 60% of the final mark plus the median of the assignments. The output is written twice, once sorted by name, and once sorted by total mark.

```
Thomas  47 83 9 8 4 7 6 9 8
Georges 36 88 8 6 7 4 9 7 8 7
Tien    49 91 9 6 7 8 5 6 7
Lei     41 82 8 8 8 8
Oanh    45 76 9 9 8 9 9 8 8 9
Lazybones  31 45
Mohamad 39 99 8 6 7 9 5 6 9
Jane    36 64 7 5 8
```

Figure 30: Input for grading program

4.2 First version

93
94

Figure 32 shows the main program. A goal of the design is to use an object to store a student record and to put as much problem-specific information as possible into the corresponding class.

The first paragraph of the program asks the user for a file name and tries to open the file. The second paragraph declares the principal data object of the program, a vector of `Student`s. From this paragraph, we can tell that the `Student` class must provide a reading capability (`>>`) and a method `process` to compute the final mark.

It is important that the argument to `push_back` is passed by value. If it was passed by reference, each entry in the vector `classData` would refer to the same local variable, `stud`!

The last part of the program opens an output file and writes the data to it twice, first sorted by name and then sorted by total marks. From this section, we deduce that the `Student` class must provide two sorting functions, `ltNames` and `ltMarks`. We also need a free function `showClass` to write the class list.

```
Sorted by name:
Georges    67.5 36 88  8  6  7  4  9  7  8  7
Jane       52.6 36 64  7  5  8
Lazybones  33.2 31 45
Lei        65.4 41 82  8  8  8  8
Mohamad    74.2 39 99  8  6  7  9  5  6  9
Oanh       63.6 45 76  9  9  8  9  9  8  8  9
Thomas     67.2 47 83  9  8  4  7  6  9  8
Tien       71.4 49 91  9  6  7  8  5  6  7

Sorted by marks:
Lazybones  33.2 31 45
Jane       52.6 36 64  7  5  8
Oanh       63.6 45 76  9  9  8  9  9  8  8  9
Lei        65.4 41 82  8  8  8  8
Thomas     67.2 47 83  9  8  4  7  6  9  8
Georges    67.5 36 88  8  6  7  4  9  7  8  7
Tien       71.4 49 91  9  6  7  8  5  6  7
Mohamad    74.2 39 99  8  6  7  9  5  6  9
```

Figure 31: Output from grading program

The function `showClass` is straightforward; it is shown in Figure 33. The output stream `os` is passed by reference because the function will change it when writing. The student data is passed by constant reference because it will not be changed by the function. It uses an iterator to traverse the vector of marks data. The statement

95

```
os << *it << endl;
```

requires class `Student` to provide an inserter (`<<`).

96

Figure 34 shows the declaration for class `Student`. There are several points to note:

- In C++, the declaration of a class and the definitions of its functions are separate. The function definitions may be — and often are — in a different file.
- A class declaration may introduce functions as `friend`s. Although these functions are not member functions, they have access to the classes' private data.
- The declarations `public` and `private` introduce a *group* of declarations with the given accessibility.

  Some authors put the `private` declarations before the `public` declarations. This seems backwards: the users of a class need to know about only the `public` attributes and these should therefore appear *first*.[12]

[12] It would be even betetr if the `private` attributes could be hidden from users altogether. Although C++ does not allow that, documentation tools such as Doxygen can provide the required effect.

```
int main()
{
    string classFileName;
    cout << "Please enter class file name: ";
    cin >> classFileName;
    ifstream ifs(classFileName.c_str());
    if (!ifs)
    {
        cerr << "Failed to open " << classFileName << endl;
        return 1;
    }

    vector<Student> classData;
    Student stud;
    while (ifs >> stud)
    {
        stud.process();
        classData.push_back(stud);
    }

    ofstream ofs("grades.txt");
    sort(classData.begin(), classData.end(), ltNames);
    ofs << "Sorted by name:\n";
    showClass(ofs, classData);

    sort(classData.begin(), classData.end(), ltMarks);
    ofs << "\nSorted by marks:\n";
    showClass(ofs, classData);
}
```

Figure 32: The main program

```
void showClass(ostream & os, const vector<Student> & classData)
{
    for (  vector<Student>::const_iterator it = classData.begin();
           it != classData.end();
           ++it )
        os << *it << endl;
}
```

Figure 33: Function `showClass`

```
class Student
{
    friend ostream & operator<<(ostream & os, const Student & stud);
    friend istream & operator>>(istream & is, Student & stud);
    friend bool ltNames(const Student & left, const Student & right);
    friend bool ltMarks(const Student & left, const Student & right);
public:
    void process();
private:
    static string::size_type maxNameLen;
    string name;
    int midterm;
    int final;
    vector<int> assignments;
    double total;
};
```

Figure 34: Class `Student`

- Functions declared within a class are called *member functions*. Functions declared outside a class are called *free functions*.[13] Member functions can be called only with an object, as in `obj.fun()`. Free functions are called without an object, as in `fun()`.
- There is no constructor. We rely on the default constructor provided by the compiler; this constructor allocates space for the object but performs no other initialization. When we define a new instance of `Student`, we must ensure that all fields are correctly initialized.
- There is only one public method, `process`, which performs any necessary computation on the data read from the marks file.
- The private data includes the information that is read from the marks file (`name`, `midterm`, `final`, and `assignments`) and computed information, `total`.
- For formatting the output, we need to know the length of the longest name. This is an attribute of the class, not the object, and so it is declared as a `static` data member.
- We need methods for input (`>>`) and output (`<<`); these are declared as friends.
- We need comparison functions that will be used for sorting: `ltNames` orders by student's names, and `ltMarks` orders by student's total marks.

  There is an important design choice here. The four `friend` functions cannot be member functions, because of the way they are called. The alternatives are either to provide access functions to private data in the class or to declare these functions as `friend`s. Access functions should be avoided if possible, especially functions with write access, as would be required for `>>`. Although `friend` functions should be used only where necessary, they sometimes provide better encapsulation, as in this case.[14]

[13] Strictly speaking, Java does not have free functions. However, classes such as `Math` provide static functions that are effectively the same as free functions. In Java, you write `Math.sqrt(x)`, in C++, you write `sqrt(x)`.

[14] We will discuss the undesirability of access functions later in the cousre, when we address class design issues.

The next step is to complete the implementation of class `Student` by providing definitions for functions and initial values for static variables. The static data member must be initialized like this:

```
string::size_type Student::maxNameLen = 0;
```

This is the *only* way to initialize a static data member. It has the form of a declaration rather than an assignment (the type is included) and it appears at global scope, that is, outside any function.

The public function `process` has two tasks: it maintains the length of the longest name seen so far and it calculates the total mark. Calculating the total mark requires finding the median of the assignments. The median is meaningless for an empty vector, and the median function requires a non-empty vector as its argument (see Figure 36). Thus `process`, shown in Figure 35, calls `median` only if the student has done at least one assignment. 97

```
void Student::process()
{
    if (maxNameLen < name.size())
        maxNameLen = name.size();
    total = 0.3 * midterm + 0.6 * final;
    if (assignments.size() > 0)
        total += median(assignments);
}
```

Figure 35: Function `process` for class `Student`

In general a function should not perform two unrelated tasks, as `process` does. The rationale in this case is that `process` performs *all* of the processing that is needed for one student record. There might be more tasks to perform than just these two. An alternative would be to define two functions, one to update `maxNameLen` and the other to calculate `total`. These two functions would always be called together, so it makes sense to combine them into a single function.

As a general principle, it should always be possible to explain the purpose of a function with a one-line description. If you need three sentences to say what a function does, there's probably something wrong with it. We could describe the purpose of function `process` as "perform all calculations needed to generate the marks file".

Every function should have a complete, one-line description.

The median calculation is performed by the function in Figure 36. The main design issue for this function is how to pass the vector of scores. Since we have to sort the vector in order to find the median, we cannot pass it by constant reference. If we pass it by reference, the caller will get back a sorted vector. Although this does not matter much for this program, a function should not in general change the data it is given unless the caller needs the changed value. Consequently, we choose to pass the vector by value, incurring the cost of copying it. 98

Finally, note that `median` has a precondition: it does not accept an empty vector. The only use of `median` in this program is in the context

```
if (assignments.size() > 0)
    total += median(assignments);
```

It follows that, if the assertion fails, there is a logical error in the program.

Perhaps, after this program has been used for a few years, another programmer decides to extend it. The function `median` is called from another location, without the check for an empty vector. During testing, the assertion fails, and the programmer immediately sees the problem with the extension.

```
// Requires: scores.size() > 0.
double median(vector<int> scores)
{
    typedef vector<int>::size_type szt;
    szt size = scores.size();
    assert(size > 0);
    sort(scores.begin(), scores.end());
    szt mid = size / 2;
    return size % 2 == 0 ?
        0.5 * (scores[mid - 1] + scores[mid]) :
        scores[mid];
}
```

Figure 36: Finding the median

99

Figure 37 shows the comparison functions that we need for sorting. The parameter lists of these functions are determined by the requirements of the `sort` algorithm: there must be two parameters of the same type, both passed by constant reference. Since we have declared these functions as `friend`s of `Student`, they have access to `Student`'s private data members. The type of `name` is `string` and the type of `total` is `double`; both of these types provide the comparison operator `<`.

After sorting, the records will be arranged in increasing order for the keys. Names will be alphabetical: `Anne`, `Bo`, `Colleen`, `Dingbat`, etc. Records sorted by marks will go from lowest mark to highest mark. To reverse this order, putting the students with highest marks at the "top" of the class, all we have to do is change "`<`" to "`>`" in `ltMarks`. As a courtesy to readers, it would also be a good idea to change the name to `gtMarks`.

```
bool ltNames(const Student & left, const Student & right)
{
    return left.name < right.name;
}

bool ltMarks(const Student & left, const Student & right)
{
    return left.total < right.total;
}
```

Figure 37: Comparison functions

The compiler has to perform a number of steps to determine that these functions are called by the statements

```
sort(classData.begin(), classData.end(), ltNames);
sort(classData.begin(), classData.end(), ltMarks);
```

The reasoning goes something like this:

1. The type of the argument `classData.begin()` is `vector<Student>::const_iterator`
2. The compiler infers from this that the elements to be sorted are of type `Student`
3. The comparison functions must therefore have parameters of type `const & Student`
4. There are functions `ltNames` and `ltTypes` with parameters of the correct type

```
istream & operator>>(istream & ifs, Student & stud)
{
    if (ifs >> stud.name)
    {
        ifs >> stud.midterm >> stud.final;
        int mark;
        stud.assignments.clear();
        while (ifs >> mark)
            stud.assignments.push_back(mark);
        ifs.clear();
    }
    return ifs;
}
```

Figure 38: Extractor for class `Student`

100

Figure 38 shows the extractor (`>>`) for class `Student`. It is a bit tricky, because we rely on the failure management of input streams. The key problem is this: since students complete different numbers of assignments, how do we know when we have read all the assignments? The method we use depends on what follows the last assignment: it is either the name of the next student or the end of the file. If we attempt to read assignments as numbers, either of these will cause reading to fail. Consequently, we can use the following code to read the assignments:

```
while (ifs >> mark)
    stud.assignments.push_back(mark);
```

However, we must not leave the stream in a bad state, because this would prevent anything else being read. Therefore, when the loop terminates, we call

```
ifs.clear();
```

to reset the state of the input stream.

We assume that, if a student name can be read successfully, the rest of the record is also readable. If the name is *not* read successfully, the function immediately returns the input stream in a bad state, telling the user that we have encountered end of file.

What happens if there is a format error in the input? Some markers, although they are asked to provide integer marks only, include fractions. Suppose that the input file contains this line:

```
ostream & operator<<(ostream & os, const Student & stud)
{
    os <<
        left << setw(static_cast<streamsize>(Student::maxNameLen)) <<
        stud.name << right <<
        fixed << setprecision(1) << setw(6) << stud.total <<
        setw(3) << stud.midterm <<
        setw(3) << stud.final;
    for (  vector<int>::const_iterator it = stud.assignments.begin();
           it != stud.assignments.end();
           ++it )
       os << setw(3) << *it;
    return os;
}
```

Figure 39: Inserter for class `Student`

```
Joe    45 76 9 9 8.5 9 9 8 8 9
```

The corresponding output file contains these lines:

```
Joe       63.6 45 76  9  9  8
.5         15.2  9  9  8  8  9
```

We see that Joe has lost all his assignment marks after 8.5 and we have a new student named "`.5`". It is clear that, if this was a production program, we would have to do more input validation.

The inserter (`<<`) in Figure 39 does not have these complications. The main points to note are: 101

- The manipulators:
  - `left` aligns text to the left
  - `right` (the default) aligns text to the right
  - `fixed` chooses fixed-point (as opposed to scientific) format for floating-point numbers
  - `setprecision(1)` requests one decimal digit after the decimal point
  - `setw(`n`)` requests a field width of N characters
- We use the longest name to align columns. The type of the variable `Student::maxNameLen` is `string::size_type` but the type expected by `setw` is `std::streamsize`. To avoid warnings from the compiler, we cast the type. Since the cast can be performed at compile time, we use a *static cast*:

  ```
  static_cast<streamsize>(Student::maxNameLen)
  ```

- We use an iterator to output the assignment marks.

All extractors and inserters follow the same pattern: 102

```
istream & operator>>(istream & is, T & val)
{
    // perform read operations for fields of T
    return is;
}

ostream & operator<<(ostream & os, const T & val)
{
    // perform write operations for fields of T
    return os;
}
```

In both cases, the function is passed a reference to a stream and returns a reference to a stream. In fact, of course, both references are to the same stream, but the state of the stream changes during the operation. The second argument for the extractor is passed by reference, because its value will be updated when the stream is read. The second argument for the inserter is passed by constant reference, because the inserter should not change it. When writing inserters and extractors, remember to return the updated stream.

Compiling this program requires the inclusions shown in Figure 40. The comments are not necessary, because experienced C++ programmers are familiar with these names. 103

```
#include <algorithm>  // sort
#include <cassert>    // assertions
#include <fstream>    // input and output file streams
#include <iomanip>    // stream manipulators
#include <iostream>   // input and output streams
#include <string>     // STL string class
#include <vector>     // STL vector class
```

Figure 40: Directives required for the grading program

4.3 Program Structure

A C++ program consists of *header files* and *implementation files*. The program is compiled as a collection of translation units. A *translation unit* normally consists of a single implementation file that may `#include` several header files.

Building a program is a process that consists of compiling each translation unit and linking the resulting object files. A *build* is the result of building. Some companies have a policy such as "daily build" to ensure that an application under development can always be compiled and passes basic tests.

4.3.1 Header Files

Header files contain declarations but not definitions. This implies that the compiler:

1. does not generate any code while processing a header file

2. may read a header more than once during a build

If a header file that contains a definition such as

```
int k;
```

is read more than once during a build, the linker will complain that `k` is redefined and will not link the program. This is why it is important not to put definitions into header files.

Although a header file may be read more than once during a build, a header file should be read once only during the compilation of a translation unit. Suppose that translation unit A includes headers for translation units B and C, and these units both include `utilities.h`. To prevent the compiler from reading `utilities.h` twice, we write it in the following way: 104

```
#ifndef UTILITIES_H
#define UTILITIES_H

// Declarations for utilities.h

#endif
```

This is the standard pattern for *all* header files. You do not have to use the exact name `UTILITIES_H`, but it is important to choose a name that is unique and has some obvious connection to the name of the header file. For example, *Accelerated C++* uses `GUARD_utilities_h`.

Header files generated by VC++.NET contain the directive `#pragma once`, which has the same effect.

In most cases, the guards are not logically necessary. Since header files contain only declarations, reading them more than once should not cause errors. Some header files, however, cannot be read twice, and these can cause problems if they don't have guards. A more important reason for using guards is efficiency: header files can be very long, and they may include other header files. Without the guards, the compiler may be forced to read thousands of lines of declarations that it has seen before.

Typically, a header file will contain declarations for types, constants, functions, and classes.

A header file should `#include` anything that the compiler needs in order to process it. For example, if a class has a data member of type `string`, its header file must contain

```
#include <string>
```

It is not a good idea to include `using` declarations in header files. A header file may be included in many translation units that may not want namespaces opened for them.

> ***Do not write*** `using` ***declarations in header files.***

4.3.2 Implementation Files

Implementation files contain definitions for the objects declared in header files. An implementation file is processed only once during a build. An implementation file should `#include` its own header file and header files for anything else that it needs.

As a general rule, the first `#include` directive should name the header file corresponding to the implementation file. For example, `utilities.cpp` would have the following structure: 105

```
#include "utilities.h"

// #includes for other components of this program

// #includes for library components

// definitions of the utilities
```

Header and implementation files create *dependencies*, which are discussed in Section 4.5 below. A header file may depend on other header files and an implementation file may depend on one or more header files. A file should never depend on an implementation file; in other words, you should never write

```
#include "something.cpp"
```

> ***Do not*** `#include` ***implementation files.***

When an implementation file `#include`s header files, the compiler obviously reads *all* of the files. Amongst other things, it checks that declarations in header files match definitions in implementation files. It is important to realize that the checking is not complete. For example, the header file 106

```
#ifndef CONFLICT_H
#define CONFLICT_H

double mean(double values[]);

#endif
```

and the implementation file

```
#include "conflict.h"
#include <vector>

double mean(std::vector<double> values)
{
    // ....
}
```

will *not* produce any error messages. Since C++ allows functions to be overloaded, it assumes that the two versions of `mean` are different functions and that the vector version will be declared somewhere else. If the program calls either version, the linker will produce an error message.

4.4 Structuring the grading program

We split the grading program of Section 4.2 into three translation units: 107

1. class `Student`
2. function `median`
3. the main program

The translation unit for function `median` is rather small, but it demonstrates the idea of splitting of generally useful functions in a larger application. For a small program such as this one, we could have put `Student` and `median` into the same implementation file.

4.4.1 Translation unit `Student`

108

Figure 41 shows the header file for class `Student`, `student.h`. There is an `#include` directive for each library type mentioned in the class declaration. The class declaration is unchanged from the original program. 109

Although function `showClass` is not a `friend` of class `Student`, it is closely related to the class; consequently, we put its declaration in `student.h`.

Figures 42 and 43 show the implementation file for class `Student`, `student.cpp`. The first `#include` directive includes `student.h`; this ensures that `student.h` does not depend on anything that it does not mention (if it did, the compiler would fail while reading it). Then we include `median.h` for this program, and finally the library types that we need. Since `student.h` includes `iostream`, `string`, and `vector`, we need only include `iomanip` for the output statements.

The implementation file `student.cpp`, shown in Figures 42 and 43 implements the member function of `Student`, `process`, and the `friend` functions. It also initializes the static data member `maxNameLen`. 110 111 112 113

4.4.2 Translation unit `median`

The header and implementation files for `median` are both short: see Figures Figure 44 and Figure 45. In a more typical application, other useful functions might be incorporated into a single translation unit. 114 115

4.4.3 Translation unit for the main program

The last step is to write an implementation file for the main program, `grader.cpp`. We do not need a header file (which would be called `grader.h`) because no other translation unit refers to anything in the main program. It is a good idea in general to avoid dependencies on the main program. See Figure 47. 116

This translation unit includes only the header files for other translation units that it needs — `student.h` in this case — and any library types. 117 118

```
#ifndef STUDENT_H
#define STUDENT_H

#include <iostream>
#include <string>
#include <vector>

class Student
{
    friend std::ostream & operator<<(std::ostream & os,
        const Student & stud);
    friend std::istream & operator>>(std::istream & is,
        Student & stud);
    friend bool ltNames(const Student & left, const Student & right);
    friend bool ltMarks(const Student & left, const Student & right);
public:
    void process();
private:
    // Data read from file
    std::string name;
    int midterm;
    int final;
    std::vector<int> assignments;

    // Data computed by process
    double total;
    static std::string::size_type maxNameLen;
};

void showClass(std::ostream & os,
    const std::vector<Student> & classData);

#endif
```

Figure 41: `student.h`: header file for class `Student`

4.5 Dependencies

119

Figure 46 shows the dependencies between the files of the grading program. Dependencies on libraries are not shown. File X *depends on* file Y if the compiler must read Y in order to compile X. In general:

- Implementation files depend on header files
- Header files may depend on other header files
- An implementation file *never* depends on another implementation file
- There must be no circular dependencies

```
#include "student.h"
#include "median.h"

#include <iomanip>

using namespace std;

string::size_type Student::maxNameLen = 0;

void Student::process()
{
    if (maxNameLen < name.size())
        maxNameLen = name.size();
    total = 0.3 * midterm + 0.6 * final;
    if (assignments.size() > 0)
        total += median(assignments);
}

ostream & operator<<(ostream & os, const Student & stud)
{
    os <<
        left << setw(static_cast<streamsize>(Student::maxNameLen)) <<
        stud.name << right <<
        fixed << setprecision(1) << setw(6) << stud.total <<
        setw(3) << stud.midterm <<
        setw(3) << stud.final;
    for (  vector<int>::const_iterator it = stud.assignments.begin();
           it != stud.assignments.end();
           ++it )
       os << setw(3) << *it;
    return os;
}

istream & operator>>(istream & ifs, Student & stud)
{
    if (ifs >> stud.name)
    {
        ifs >> stud.midterm >> stud.final;
        int mark;
        stud.assignments.clear();
        while (ifs >> mark)
            stud.assignments.push_back(mark);
        ifs.clear();
    }
    return ifs;
}
```

Figure 42: `student.cpp`: implementation file for class `Student`: first part

```
bool ltNames(const Student & left, const Student & right)
{
    return left.name < right.name;
}

bool ltMarks(const Student & left, const Student & right)
{
    return left.total < right.total;
}

void showClass(ostream & os, const vector<Student> & classData)
{
    for (  vector<Student>::const_iterator it = classData.begin();
           it != classData.end();
           ++it )
        os << *it << endl;
}
```

Figure 43: `student.cpp`: implementation file for class `Student`: second part

```
#ifndef MEDIAN_H
#define MEDIAN_H

#include <vector>

double median(std::vector<int> scores);

#endif
```

Figure 44: `median.h`: header file for function `median`

- Fewer dependencies are better (few dependencies = "low coupling")

Dependencies have an important effect on compilation time. A file with a high in-degree will trigger extensive recompilation when it is changed.

In Figure 46, a change to either `median.h` or `student.h` will cause two of the three implementation files to be recompiled. If `grader.cpp` depended on `median.h`, changing `median.h` would cause all three implementation files to be recompiled.

In a small program like the grader, the effect of dependencies on compilation is negligible. In large programs, the effect can be significant. Large programs require hours or even days to compile. Some header files are used by hundreds or even thousands of implementation files. A change to one of the header files can trigger hours of recompilation time.

An important component of large-scale C++ design is to reduce the dependencies between source files. We will discuss ways to do this as the course progresses.

```
#include "median.h"

#include <algorithm>
#include <cassert>

using namespace std;

// Requires: scores.size() > 0.
double median(vector<int> scores)
{
    typedef vector<int>::size_type szt;
    szt size = scores.size();
    assert(size > 0);
    sort(scores.begin(), scores.end());
    szt mid = size / 2;
    return size % 2 == 0 ?
        0.5 * (scores[mid - 1] + scores[mid]) :
        scores[mid];
}
```

Figure 45: `median.cpp`: implementation file for function `median`

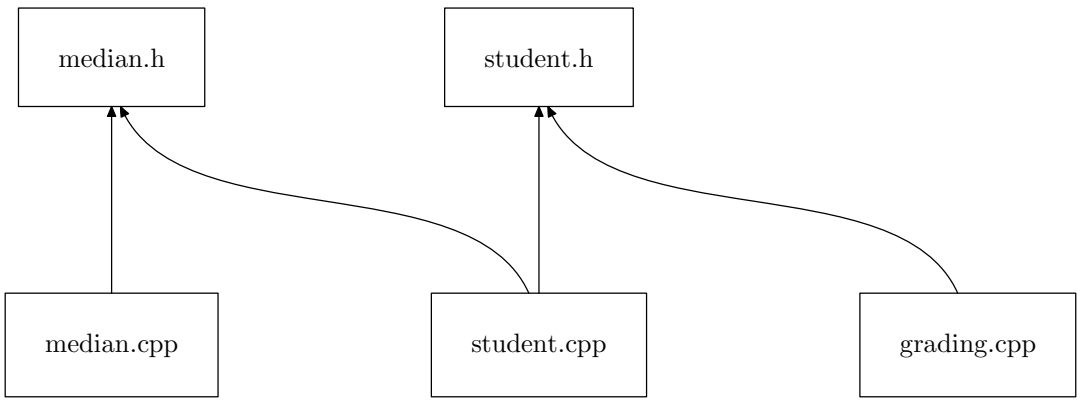


Figure 46: Dependencies in the grading program

```
#include "student.h"

#include <algorithm>
#include <fstream>
#include <string>
#include <vector>

using namespace std;

int main()
{
    // Attempt to open file provided by user
    string classFileName;
    cout << "Please enter class file name: ";
    cin >> classFileName;
    ifstream ifs(classFileName.c_str());
    if (!ifs)
    {
        cerr << "Failed to open " << classFileName << endl;
        return 1;
    }

    // Process the file
    vector<Student> classData;
    Student stud;
    while (ifs >> stud)
    {
        stud.process();
        classData.push_back(stud);
    }

    // Print reports
    ofstream ofs("grades.txt");
    sort(classData.begin(), classData.end(), ltNames);
    ofs << "Sorted by name:\n";
    showClass(ofs, classData);

    sort(classData.begin(), classData.end(), ltMarks);
    ofs << "\nSorted by marks:\n";
    showClass(ofs, classData);
}
```

Figure 47: `grader.cpp`: implementation file for main program

4.6 Strings and characters

Strings Class `string` provides a large number of functions in addition to those that we have already seen. See `http://www.cppreference.com/cppstring/all.html` for a complete reference.

The following operators work for strings: 120

```
<     <=     ==     !=     >=     >     +     <<     >>     =     []
```

Operator `+` concatenates strings. There are several overloads, allowing for all combinations of `char`, `C` strings, and strings. Operator `[]` provides indexing for strings.

Strings function as containers for characters, working in a similar way to the type `vector<char>`. Consequently, iterators, `push_back`, and similar functions work for strings. In particular, `string` provides `insert` to insert characters into a string, `erase` to remove characters from a string, and `replace` to replace a sequence of characters in a string.

There several functions for finding characters or substrings in strings. For example:
`find` `find_first_of` `find_first_not_of` `find_last_of` `find_last_not_of`

Characters The library `cctype` provides the same functionality as the `C` header file `ctype.h`. Although many of these functions should now be considered obsolete (e.g., `strcpy` and friends), others are still useful. In particular: 121

`isalpha(c)` returns `true` if `c` is a letter

`isdigit(c)` returns `true` if `c` is a digit

`isspace(c)` returns `true` if `c` is a blank, tab, or linebreak

`tolower(c)` returns the lower case equivalent of an upper case character and leaves other characters unchanged

`toupper(c)` returns the upper case equivalent of an lower case character and leaves other characters unchanged

5 Pointers and Iterators

5.1 Pointers and Dynamic Allocation

An important difference between C++ and Java is that, in C++, dynamic memory allocation and deallocation is *explicit* (done by code written by programmers) whereas in Java it is *implicit* (done by built-in system code). It is not quite true to say "Java does not have pointers", but it is true to say "Java does not allow direct access to pointers".

A pointer is a variable that contains the address of another variable.

Like all variables in C++, a pointer has a type. The type depends on the object addressed by the pointer. For example, a pointer that points to an `int` has type "pointer to `int`", written `int*`.

For every type T, there is a type $T*$ or pointer to T.

To assign a value to a pointer, we need a way of obtaining addresses. There are two common ways, and we define the unimportant one first. The operator &, applied to a variable, yields the address of that variable. The address can be stored as a pointer. The following code defines two pointers, `pk1` and `pk2`, both pointing to the integer `k`.

```
int k = 42;
int *pk1 = &k;
int *pk2 = &k;
```

If we have a pointer to an object, we can obtain the object itself by applying the *prefix* operator `*`. In the example above, `*pk1` is an integer variable (it is actually, of course, the integer `k`).

The declaration `int *pk1` is a kind of pun. We can read it as `(int *)pk1` (" `pk1` has type `int *`") or as `int (*pk1)` ("`*pk1` has type `int`"). The forms with parentheses are illegal, but the C++ compiler is not fussy about spaces: we can write any of

```
int *pk1 = &k;
int* pk2 = &k;
int * pk3 = &k;
```

Some programmers prefer the first form and some the second; few use the third, although it is legal. There is one syntactic trap that you should be aware of. The statement

```
int *p, q;
```

declares `p` to be a pointer to `int`, but `q` is just a plain `int`.

Using pointers can have strange results — although they are not all that strange if you think about them carefully. For example, the code

```
int k = 42;
int *pk1 = &k;
int *pk2 = &k;
++(*pk1);
cout << k << ' ' << *pk2 << endl;
```

displays `43 43`. Since both pointers point to the same integer, any change to its value will be seen by *all* pointers.

C++ provides arithmetic operations (+ and −) on pointers. However, the arithmetic is rather special. If `p1` and `p2` are pointers and `i` is an integer, then:

`p1 + i`	is a pointer
`p1 - i`	is a pointer
`p1 - p2`	is a signed integer (`p1` and `p2` must have the same type)

These operations would be utterly meaningless if it were not for the fact that C++ uses the size of the object pointed to when doing arithmetic. For example, suppose that an instance of type `T` occupies 20 bytes and that we write:

```
T t;
T *pt = &t;
```

Then `pt+1` is an address 20 bytes greater than `pt`.

This kind of arithmetic is esactly what we need for arrays. In fact, subscript operations in C++ are *defined* in terms of pointer arithmetic. The declaration

```
double a[6];
```

introduces `a` as an array of 6 `doubles`. C++ treats `a` as a `const` pointer. We have:

$$
\begin{aligned}
\texttt{a[0]} &\equiv *\texttt{a} \\
\texttt{a[1]} &\equiv *(\texttt{a}+1) \\
\texttt{a[2]} &\equiv *(\texttt{a}+2) \\
&\ldots \\
\texttt{a[}i\texttt{]} &\equiv *(\texttt{a}+i) \qquad \text{for any integer } i
\end{aligned}
$$

5.1.1 Stack Allocation

Most data in C++, and all the data we have seen so far in this course, is allocated on the *run-time stack*.

The run-time stack, “stack” for short, is initially empty. When execution starts, global data is pushed onto th stack. On entry to a function, the local data associated with the function (including its arguments) are pushed onto the stack. When the function returns, the local data is popped off the stack. Thus the stack varies in size as the program runs.

When a function has returned, its local data no longer exists.[11] Normally, this is not a problem, because the function's local variables are no longer accessible. Playing tricks with pointers, however, can cause problems. Consider the code shown in Figure 45.

```
int * f()
{
    int k = 42;
    int *pk = &k;
    return pk;
}

int main()
{
    int *p;
    p = f();
    return 0;
}
```

Figure 45: Dangerous tricks

This program compiles because there are no syntactic or semantic errors. The final assignment, `p = f()`, makes `p` a pointer to `k`. But `k` no longer exists, having been popped off the stack when `f` returns. Any attempt to use `p` will have unpredictable results.

```
char *read()
{
    char buffer[20];
    cin >> buffer;
    return buffer;
}

int main()
{
    char *pc = read();
    cout << pc << endl;
    return 0;
}
```

Figure 46: Subtler dangerous tricks

Figure 45 shows a more subtle version of the same problem. The function `read` has result type `char*` but actually returns an array of characters: this works because the compiler treats these types as the same. Similarly, the main function treats function `read` as having type `char*`.

The reason that this is a bad program is that the array `buffer` is allocated on the stack. It is destroyed when the function returns. The pointer `pc` is undefined and cannot be used safely.

[11] Actually, it's still there, on the stack, but will be overwritten when the next function is called. Consequently, we must *assume* that it no longer exists.

The compiler[12] actually recognizes the mistake and issues a warning message:

```
f:\Courses\COMP446\src\Pointers\pointers.cpp(24) :
    warning C4172: returning address of local variable or temporary
```

This provides another reason for paying attention to warnings from the compiler!

Problems like this tend to occur when we use "old-style" C++ code. No problems arise if we use the STL class `string` instead of an array of characters.

5.1.2 Heap Allocation

Stack allocation works because function calls and returns match the "last-in-first-out" (LIFO) discipline of a stack. Sometimes this is not good enough. For example, we might want to allocate data within a function, use that data for a while after the function has returned, and then deallocate the data. We can do this by allocating the data on the *heap*, an area of memory that does not obey any particular discipline such as LIFO or FIFO. The heap is used like this:

```
T *p = new T();
....
delete p;
```

The operator `new` requires a call to a constructor on its right. The value returned by `new` is a pointer to the constructed object. We can use this pointer to perform operations on the object. When we have finished with the object, we apply `delete` to the pointer to destroy the object and deallocate the heap memory it was using.

Suppose class `T` provides a public function `f`. To call `f` using the pointer `p`, we must first dereference `p` and the use the "dot" operator to call `f`. Thus we write `(*p).f()`. (The parentheses are necessary, because the compiler reads `*p.f()` as `*(p.f()`, which is wrong.) Since this construction occurs often, there is an abbreviation for it:

```
p->f()  ≡  (*p).f()
```

Figure 47 provides a simple example of heap allocation and deallocation. The function `maketest` constructs a new instance of class `Test` on the heap, calls its function `f`, and returns a pointer to it. The function `killTest` deletes the object. The program displays the following output:

```
makeTest
Constructor
Function
killTest
Destructor
```

[12] VC++ 7.1

```
class Test
{
public:
    Test() { cout << "Constructor\n"; }
    ~Test() { cout << "Destructor\n"; }
    void f() { cout << "Function\n"; }
};

Test *makeTest()
{
    cout << "makeTest\n";
    Test *pt = new Test();
    pt->f();
    return pt;
}

void killTest(Test *p)
{
    cout << "killTest\n";
    delete p;
}

int main()
{
    Test *p = makeTest();
    killTest(p);
    return 0;
}
```

Figure 47: Heap allocation and deallocation

5.1.3 A note on null pointers

The careful reader will have observed that null pointers are represented by 0, rather than NULL, in the code for the traversal program. This is because NULL presents a problem for C++ programmers.

In C, NULL was `#defined` by a preprocessor statement something like

```
#define NULL 0
```

This is compatible with the spirit of C's rather loose approach to types, but is not consistent with C++'s safer typing. Improvements such as

```
#define NULL (int) 0
#define NULL (void*) 0
```

do not help, because the first version does not work for pointers and the second version requires an ugly-looking cast whenever we use NULL for a pointer type other than `void*`.

In C++, `#defines` are deprecated, and we are supposed to use constant declarations instead. Unfortunately, any reasonable declaration of NULL has the same problems as the attempts to `#define` NULL:

```
const int NULL = 0;
const void * NULL = 0;
```

The simplest solution seems to be:

> ***Forget about NULL — just use 0.***

Problems with zero arise in overloading. We might declare

```
void f(int n);      // first overload
void f(char *p);    // second overload
```

and call `f(NULL)` thinking that the compiler would say to itself: "The programmer has used NULL, which suggests a pointer, and I will therefore pass `(char*)NULL` to the second overload".

However, the compiler does not reason like this. Instead, it says: "NULL is 0 and 0 is an `int`, and I will therefore pass 0 to the first overload".

As Meyers (1992, pages 87–89) explains, this is an unusual case because people tend to think that there is an ambiguity but the compiler does not. (Usually, people think their meaning is perfectly obvious and are annoyed when the compiler calls it ambiguous.) The morale is:

> ***Prefer not to overload a function with integer types and pointer types.***

5.2 Iterators

Iterators are one of the keys to the flexibility of the STL. We have seen that the STL provides containers and algorithms. Iterators provide the glue that allows us to attach one to the other:

- Each container specifies the iterators that it provides
- Each algorithm specifies the iterators that it needs

For example, `vector<T>` provides the kind of iterators that `sort` requires; it follows that we can sort vectors.

A pair of iterators specifies a range of container elements. The range typically defines a semi-closed interval: the first iterator of a range accesses the first element of the range, and the last iterator accesses the first element *not* in the range. In this typical loop

Property	Input	Output	Forward	Bidirectional	Random Access	Insert
Assign (=)	√	√	√	√	√	√
Compare (==, !=)	√	√	√	√	√	√
Read (`*it`)	√		√	√	√	
Write (`*it`)		√	√	√	√	√
Order (<, <=, >, >=)					√	
Increment (++)			√	√	√	
Decrement (--)				√	√	
Arithmetic ($\pm i$, +=, -=)					√	

Figure 48: Properties of iterators

```
for (⟨iterator⟩ it = first; it != last; ++it)
    .... *it ....
```

⟨*iterator*⟩ stands for some iterator type, and we see that the iterator must provide the operations:

- `first != last` (and therefore `first == last`): equality comparison
- `++it`: increment, or step to next element
- `*it`: deference to provide access to the container element

C++ programmers will recognize that all of these operations are provided by pointers. In fact, we can use raw pointers as iterators:

```
const int MAX = 20;
double values[MAX] = .... ;
sort(&values[0], &values[MAX]);
```

5.2.1 Kinds of iterator

There are several ways of classifying iterators. Below, we define individual properties; most iterators possess several of these properties. In each of the following cases, we use `it` to stand for an iterator with the given property. Figure 48 summarizes the properties that various kinds of iterator provide.

- All iterators implement the operators = (assignment), == and != (comparison).
- An *input iterator* can be used to read elements from a container but does not provide write access. That is, `*it` is an rvalue.

- An *output iterator* can be used to update elements in a container but may not provide read access. That is, `*it` is a lvalue.
- A *forward iterator* is an input and output iterator that can traverse the container in one direction. A forward iterator must implement ++.
- A *bidirectional iterator* is an input and output iterator that can traverse the container forwards and backwards. A bidirectional iterator must implement ++ and `--`.

  There are no "backwards only" iterators; an iterator is either forward or bidirectional.
- A *random access iterator* must allow "jumps" in access as well as traversal. The principal operation that a random access must provide is indexing: `it[n]`. Random access iterators also provide:
  - Addition and subtraction of integers: `it + n` and `it - n`
  - Assignment operators: `it += n` and `it -= n`
  - Subtraction: `it1 - it2` yields a signed integer
  - Comparisons: the operators <, <=, >=, and >
- An *insert iterator* is an output iterator that puts new values into a container rather than just updating the values that are already there.

All of the STL containers provide bidirectional iterators. It follows that they all provide input, output, and forward iterators. These categories are useful because we can construct special iterators that may not have all of the properties.

The only container classes that provide random access iterators are `deque`, `string`, and `vector`. This is because these containers are required to store elements in consecutive locations, which means that random access is a simple address calculation.[13]

We have often mentioned that a range is specified by an iterator accessing the first element of the range and another iterator accessing the element following the last element of the range — which, in most cases, does not even exist. Here are some reasons for this choice:

1. Two equal iterators specify an empty range.
2. We can use == and `!=` to test for an empty range and for end of range — we do not need < and friends.
3. We have an easy way to indicate "out of range", namely, the last iterator of the range.

   A function can return an iterator for a valid element to indicate success or an iterator for an invalid element to indicate failure. This avoids the need for special values, flags, etc.

An iterator with type `iterator` is allowed to change the contents of its associated container. An iterator with type `const_iterator` (called a *constant iterator*) is *not* allowed to change the contents of its associated container. It is best to use constant iterators whenever possible.

[13] The address of `c[i]` is $\&c + s \times i$, where $\&c$ is the address of the container and s is the size of an element.

5.2.2 Using iterators

Suppose that we want to divide the students of the grading program into two groups, according as to whether they passed or failed the course. The first thing we will need is a criterion for deciding whether a student has passed. We add the following member function to class `Student`:

```
bool passed() const { return total > 50; }
```

There are two things to note about this function:

- Its definition appears within the class declaration. This is allowed, and a consequence is that the compiler may inline calls to the function.
- The `const` indicates that the function does not change the state of the object. The significance of this will emerge shortly.

Let us first consider a straightforward approach, in which we create two empty vectors for passed and failed students, and iterate through the class assigning each student to one or the other vector: see Figure 49. In this code, note that `it->passed()` is an abbreviation for `(*it).passed()`.

```
vector<Student> passes;
vector<Student> failures;
for (  vector<Student>::const_iterator it = data.begin();
       it != data.end();
       ++it )
    if (it->passed())
        passes.push_back(*it);
    else
        failures.push_back(*it);
```

Figure 49: Separating passes and failures: first version

If we omit the `const` in the declaration of `passed`, this code will not compile: see Figure 50. This is because a `const_iterator` can only be used with a constant container. Although `passed` does not alter the state of a student, the compiler cannot tell this, and will reject Figure 49. By stating explicitly that `passed` is a `const` function, we inform the compiler that the container will *not* change and we can use a `const_iterator`.

The code in Figure 49 would also work if we omitted `const` from `passed` and from the iterator declaration. However, it is good practice to use `const` wherever it applies, and so Figure 49 is preferable to code without `const`s.

After executing the code in Figure 49, we now have *three* vectors and two copies of each student record, one in the original vector and the other in either `passes` or `failures`. It would be more efficient in terms of space to move the failed students into a new vector and remove them from the original vector. Vectors provide the function `erase` to remove elements from a container. The code in Figure 51 does this.

```
f:\....\grader.cpp(111):
   error C2662: 'Student::passed' :
   cannot convert 'this' pointer from
       'const std::allocator<_Ty>::value_type'
   to  'Student &'
   with
        [
            _Ty=Student
        ]
```

Figure 50: Compiler error caused by omitting `cosnt`

```
vector<Student> failures;
vector<Student>::iterator it = data.begin();
while (it != data.end())
{
    if (it->passed())
        ++it;
    else
    {
        failures.push_back(*it);
        it = data.erase(it);
    }
}
```

Figure 51: Separating passes and failures: second version

The first point to note is that we have to use a `while` loop rather than a `for` loop, because the loop step is not necessarily `++it`. However, for students that pass the course, `++it` is all we have to do.

When a student fails, the corresponding record is stored in `failures`. In order to remove the record from the class data vector, we write

```
it = data.erase(it);
```

The effect of `data.erase(it)` is to remove the element indicated by `it` from the vector `data`. After this has been done, the iterator is *invalid* because it accesses an element that no longer exists. The function `erase` returns an iterator that accesses the next element of the vector.

It is important to use the iterator returned by `erase` and not to assume that it is just `++it`. A iterator operation that invalidates an iterator may do other things as well, even including moving the underlying data. When an iterator operation invalidates an iterator, it potentially invalidates *all* iterators.

For example, calling `erase` in Figure 51 invalidates the iterator for the end of the vector, because removing one component forces the remaining components to move. It would therefore be a serious mistake to attempt to "optimize" Figure 51 like this:

```
vector<Student> failures;
vector<Student>::iterator it = data.begin();
vector<Student>::iterator last = data.end(); // Save final iterator
while (it != last)
    ....
```

> ***Check whether an operation invalidates iterators before using it.***

The solution we have developed works correctly, but is inefficient. The inefficiency is negligible for a class of sixty students but could be a problem if we used the same strategy for very large vectors. To understand the reason for the inefficiency, suppose that an entire class of 50 students fails. The program would execute as follows:

Remove first record, move remaining 49 records

Remove second record, move remaining 48 records

Remove third record, move remaining 47 records

....

The total number of operations is $\underbrace{49 + 48 + 47 + \cdots + 1}_{49 \text{ terms}}$ and is clearly $\mathcal{O}(N^2)$ for N students.

To improve the performance, we must change the data structure. A list can erase in constant time (i.e., $\mathcal{O}(1)$) and can perform the other operations that we require. It is a straightforward exercise to replace each `vector` declaration by a corresponding `list` declaration.

It is obviously important to know which operations invalidate iterators. Fortunately, good STL reference documents usually provide this information. If you are not sure, you can make a good guess by thinking about how the operation must work on a given data structure — but it's much safer to look up the correct answer.

5.2.3 Range Functions

Most containers have *range functions* — that is, functions with a pair of parameters representing a range of elements in the container. If a suitable range function exists, it is better to use it than to use a loop.

Suppose that you want to create a vector `v1` consisting of the back half of the vector `v2` (Meyers 2001, Item 5). You could do it with a loop:

```
v1.clear();
for (   vector<Widget>::const_iterator ci = v2.begin() + v2.size()/2;
        ci != v2.end();
        ++ci )
   v1.push_back(*ci);
```

but it is quicker to write any one of the following statements:

```
v1.assign(v2.begin() + v2.size()/2, v2.end());

v1.clear();
copy(v2.begin() + v2.size()/2, v2.end(), back_inserter(v1));

v1.insert(v1.end(), v2.begin() + v2.size()/2, v2.end());
```

In this case, `insert` is probably the best choice.

6 Template Programming

Templates enable us to write generic, or parameterized, code. The basic idea is simple but some of the implications — at least for C++— are subtle.

All of the material in this section is covered in (Koenig and Moo 2000). For a more detailed and complete description of templates, (Vandervoorde and Josuttis 2003) is recommended.

6.1 Template functions

Consider these three functions:[14]

```
void Swap(char & x, char & y)       // Function (1)
{
   char t = x; x = y; y = t;
}

void Swap(int & x, int & y)
{
   int t = x; x = y; y = t;
}

void Swap(double & x, double & y)
{
   double t = x; x = y; y = t;
}
```

These functions perform the same task for different types. Since C++ allows overloading, we could use all three in the same program, but there does appear to be some redundancy.

6.1.1 Type Parameters

We can use templates to avoid source code redundancy. We replace the previous three definitions by the following *template function declaration*:

```
template <typename T>
void Swap(T & x, T & y)
{
    T t = x;
    x = y;
    y = t;
}
```

The new function can be called in the same way as the previous versions. It is not necessary to specify the type of the arguments, because the compiler already has this information. The code

[14] The name `Swap`, rather than `swap`, was chosen to avoid confusion with the standard library function.

```
char c1 = 'a';
char c2 = 'b';
Swap(c1, c2);
cout << c1 << ' ' << c2 << endl;

int m = 3;
int n = 5;
Swap(m,n);
cout << m << ' ' << n << endl;
```

prints

```
b a
5 3
```

To compile the call `Swap(c1, c2)`, the compiler must perform the following steps:

1. Infer the types of the arguments `c1` and `c2` (in this case: `char` and `char`).
2. Look for functions named `Swap`.
3. Find (in this case) a template function `Swap`.
4. Check that the substitution T = `char` matches the call.
5. Generate source code for the function `Swap<char>` (which should be the same as function (1) above).
6. Compile the generated source code.
7. Generate a call to this function.

The compiler detects errors in the template code, if there are any, at step 6, when it compiles the code obtained by expanding the template. This means that errors in template code are reported *only if the template is instantiated*. You can write all kinds of rubbish in a template declaration and the compiler won't care if you never use the template.

The process of deriving an actual function from a template declaration is often called "instantiating" the template. This usage is confusing, because we also talk about objects obtained by instantiating a class. These notes use the expression *applying a template* or *the application of a template*, by analogy with "applying a function".

Although templates remove redundancy in the source code, they do not affect the object code. If the program calls `Swap` with N different argument types, there will be N versions of `Swap` in the object code.

The following template function returns the greater of its two arguments.[15]

[15] The name `Max` is used to avoid confusion with the library function `max`.

```
template <typename T>
T Max(const T & x, const T & y)
{
    return x > y ? x : y;
}
```

We would expect this function to work with various types, and indeed it does. Each of the following statements compile and execute correctly:

```
cout << Max(7, 3) << endl;
cout << Max('a', '5') << endl;
cout << Max(7.1, 3.3) << endl;
```

This statement *appears* to execute correctly:

```
cout << Max("COMP", "6441") << endl;
```

But further investigation reveals problems. For instance, executing

```
cout << Max("apple", "berry") << endl;
```

displays `berry` but executing

```
cout << Max("berry", "apple") << endl;
cout << Max("apple", "berry") << endl;
```

displays `apple` twice. Even worse, the statement

```
cout << Max("berry", "cherry") << endl;
```

gives a compiler error:

```
error C2782: 'T Max(const T &,const T &)' :
   template parameter 'T' is ambiguous
```

To find out what is going wrong, we add a line to `Max`:

```
template <typename T>
T Max(const T & x, const T & y)
{
    cout << "Max called with " << typeid(x).name() << endl;
    return x > y ? x : y;
}
```

In order to compile this, we must add the directive

```
#include <typeinfo>
```

Executing

```
cout << Max("apple", "berry") << endl;
cout << Max("cherry", "orange") << endl;
```

displays

```
Max called with char const [6]
apple
Max called with char const [7]
orange
```

and reveals the problem: we cannot compare strings of different lengths because they have different types. Also, `Max` is not comparing the strings; it is comparing the addresses of the strings (i.e., the pointers).

There are no good, general solutions to problems like this. For `Max`, the best thing to do is to define a non-template version for `strings`:

```
string Max(const string & x, const string & y)
{
    return x > y ? x : y;
}
```

When the compiler encounters `Max("apple", "berry")`, it will consider the `string` version a better match than the template version.

6.1.2 Missing functions

Suppose, however, that we define our own class `Widget` as follows:

```
class Widget
{
public:
    Widget(int w) : w(w) {}
private:
    int w;
};
```

Attempting to use `Max` with widgets

```
Max(Widget(1), Widget(2));
```

gives several error messages, including this one:

```
error C2676: binary '>' : 'const Widget' does not define
    this operator or a conversion to a type acceptable
    to the predefined operator
```

It is easy to see what has happened: the compiler has generated the function

```
Widget Max(const Widget & x, const Widget & y)
{
    return x > y ? x : y;
}
```

and has then discovered that `Widget` does not implement `operator>`. The correction is also straightforward: we just have to add the function

```
friend bool operator>(const Widget & left, const Widget & right)
{
    return left.w > right.w;
}
```

to the declaration of class `Widget`.

> ***Ensure that template arguments satisfy the requirements of the corresponding template parameter.***

6.1.3 Conversion failure

The use of templates prevents some of the conversions that we expect. If we declare

```
double Max(const double & x, const double & y)
{
    return x > y ? x : y;
}
```

then the following calls all compile and execute correctly:

```
Max(1.2, 3.4);
Max(1, 3.4);
Max(1, 3);
```

The first call works because the types match exactly, and the other two calls work because the compiler includes code to convert 1 and 3 from `int` to `double` before the function is called.

With the template version, however, the compiler does not allow the second call. It matches `Max(int,int)` and `Max(double,double)` to the template pattern, but `Max(int,double)` does not match and the compiler will not insert conversions to make it match.

There are several ways to make the second call work properly:

- We can cast the argument that is causing the problem:

```
Max(static_cast<double>(1), 3.4);
```

- We can specify the template argument explicitly:

```
Max<double>(1, 3.4);
```

- We can avoid calls of the form `Max(1, 3.4)` with mixed-type arguments.

6.1.4 Non-type Parameters

Template parameters are not restricted to types. We can also use integral types (that is, `char`, `short`, `int`, and `long`) as parameters. This function has an integer template parameter[16]

```
template<int MAX>
int randInt()
{
    return rand() % MAX;
}
```

and is used like this:

```
// Throwing dice
for (int i = 0; i != 20; ++i)
    cout << setw(2) << randInt<6>() + 1;
cout << endl;
```

Of course, we could have written this function without using templates:

```
int randInt(const int MAX)
{
    return rand() % MAX;
}
```

Then we would call it in the usual way: `randInt(6)`.

The difference between the two versions of `randInt` is that the template version substitutes the integer *at compile time* whereas the conventional version substitutes the integer *at run time*. In this case, the difference is slight — the template version probably runs slightly faster than the conventional version but the difference will hardly be noticeable — but may be significant in more realistic situations.

The argument corresponding to a non-type template parameter can be a constant, but it cannot be a variable:

```
const int MAXRAND = 100;
.... randInt<MAXRAND>() ....     // OK

int MAXVAR = 100;
.... randInt<MAXVAR>() ....      // Compiler error
```

[16] This is a terrible way to generate random integers! We will consider better ways later.

6.2 Template classes

Classes can be parameterized with templates; the notation is similar to that of function templates. Here is a simple class for 2D coordinates, in which each coordinate consists of two `floats`.

```
class Coordinate
{
public:
    Coordinate(float x, float y) : x(x), y(y) {}
    void move(float dx, float dy);
private:
    float x;
    float y;
};
```

To parameterize this class, we:

1. Write `template<typename T>` in front of it;
2. replace each occurrence of `float` by `T`; and
3. — important! — within the declaration, replace occurrences of the class name *C* by *C*`<T>`.

Performing these steps for class `Coordinate` yields the following declaration:

```
template<typename T>
class Coordinate
{
public:
    Coordinate<T> (T x, T y) : x(x), y(y) {}
    void move(T dx, T dy);
private:
    T x;
    T y;
};
```

Unlike functions, the compiler cannot infer the argument type for classes. Whenever we create an application of a template class, we must provide a suitable argument:

```
Coordinate<int> p(1, 2);
Coordinate<float> q(3.4, 5);
```

The integer argument `5` is acceptable for the `Coordinate<float>` constructor because the template application is explicit: the compiler knows that `float` values are expected, and inserts the appropriate conversion.

Here is how `move` is defined for class `Coordinate`:

```
template<typename T>
void Coordinate<T>::move(T dx, T dy)
{
    x += dx;
    y += dy;
}
```

In general, if the definition of a member function for a template class uses the template parameter, we must:

1. Write `template<typename T>` before the function; and
2. write *C*<T> wherever the class name *C* is needed.

Like functions, template classes can have non-type template parameters, provided that the parameters have integral types. Instances of the class must be provided with constant arguments of appropriate types.

6.2.1 A template class for vectors

Figure 52 shows part of a template class for vectors. It is parameterized by `Type`, the type of a vector element, and `Dim`, the dimension of the vectors. A typical declaration would be

```
Vector<double, 3> v;
```

If the program uses many vectors of this type, we would probably define a special type for them, to reduce the amount of writing required and improve the clarity of the program:

```
typedef Vector<double, 3> vec;
```

Some points to note about the declaration of `Vector`:

- There is a default constructor that sets the elements of the vector to zero. The compiler's default constructor would leave the elements uninitialized.
- The function `operator[]` returns a reference to a vector element. This function allows a user to get or set any element.

  It is good practice (as we will discuss later) to provide two versions of this function, one returning an lvalue (as here) and the other returning an rvalue.
- The function `operator[]` performs a range check and aborts the program if the range check fails. It would probably be better to handle a range check error by throwing an exception.

```
template<typename Type, int Dim>
class Vector
{
public:
Vector<Type, Dim>()
{
    for (int d = 0; d != Dim; ++d)
       v[d] = 0;
}

Type & operator[](int i)
{
    assert(0 <= i && i < Dim);
    return v[i];
}

friend Vector<Type, Dim> operator+(
      const Vector<Type, Dim> & left,
      const Vector<Type, Dim> & right )
{
    Vector<Type, Dim> result;
    for (int d = 0; d != Dim; ++d)
        result.v[d] = left.v[d] + right.v[d];
    return result;
}

friend ostream & operator<<(ostream & os,
                            const Vector<Type, Dim> & vec)
{
    os << '(';
    for (int d = 0; d != Dim; ++d)
    {
    os << vec.v[d];
        if (d < Dim - 1)
            os << ", ";
    }
    return os << ')';
}

private:
    Type v[Dim];
};
```

Figure 52: Part of a template class for vectors

- Several of the functions have `for`-loops for the range `[0,Dim)`. A good compiler might optimize these away for small values of `Dim` (this optimization is a special case of *loop unrolling*). If we were worried about the overhead of a `for`-loop, we could rewrite the code using `if` or `switch` statements. Figure 53 shows how this might be done for `operator+`. The code looks long but, when the compiler expands `Vector<double,3>`, it will see something like this:

```
if (3 == 1)
    ....
else if (3 == 2)
    ....
else if (3 == 3)
    ....
else
```

  Any reasonable compiler should be smart enough to compile the code for the "`3 == 3`" case and generate no code for the conditional expressions or the other cases.

  `<aside>` People sometimes wonder why a compiler should bother optimizing code such as

```
if (1 == 0)
    ....
```

  on the grounds that no sane programmer would write code like this. Such optimizations are important, and very common, because a lot of code is *not* written explicitly by programmers, but is generated by template expansion, code generators, and in other ways. Unless the generator is very smart, generated code may be very stupid. `</aside>`

When we have obtained a type by applying a class template, we can do all of the usual things with it:

```
Vector<double, 3> v;              // 3D vector
Vector<double, 3> & rv;           // reference to a 3D vector
const Vector<double, 3> & rv;     // constant reference
                                  //    to a 3D vector
Vector<double, 3> * pv;           // pointer to a 3D vector
....
```

6.2.2 `class` or `typename`?

Early versions of C++ with templates used "`class`" where we have been using "`typename`". For example:

```
template<class T>
class Coordinate { ....
```

```
friend Vector<Type, Dim> operator+(
      const Vector<Type, Dim> & left,
      const Vector<Type, Dim> & right )
{
    Vector<Type, Dim> result;
    if (Dim == 1)
    {
        result.v[0] = left.v[0] + right.v[0];
    }
    else if (Dim == 2)
    {
        result.v[0] = left.v[0] + right.v[0];
        result.v[1] = left.v[1] + right.v[1];
    }
    else if (Dim == 3)
    {
        result.v[0] = left.v[0] + right.v[0];
        result.v[1] = left.v[1] + right.v[1];
        result.v[2] = left.v[2] + right.v[2];
    }
    else
    {
        for (int d = 0; d != Dim; ++d)
            result.v[d] = left.v[d] + right.v[d];
    }
    return result;
}
```

Figure 53: A more elaborate version of `operator+`

This usage suggested that the argument replacing `T` had to be a class and could not be a built-in type, such as `int`. The keyword `typename` suggests that *any* type can be used, including the built-in types.

Prefer `typename` to `class` for template parameters.

6.3 Complications

6.3.1 Compiling template code

We have seen that normal practice in C++ programming is to put declarations into header files and definitions into implementation files. This does not work for templates. The compiler cannot generate code for a function such as

```
template<typename T>
void Coordinate<T>::move(T dx, T dy)
{
    x += dx;
    y += dy;
}
```

without knowing that the argument that replaces `T` and, if this definition is in an implementation file, the compiler cannot access it when it compiles a call such as `v.move(2,3)`.

There are several solutions. Different platforms have different policies, but a solution that works on (almost?) all platforms is to treat any template code, even a function such as `move` above, as a *declaration*.

Put all template code into header files.

6.3.2 Template return types

It does not make sense to use a template type as a return type:

```
template<typename T>
T f()
{
    return ??
}
```

Even if we could write a sensible expression after `return`, the compiler could not deduce the template argument at the call site.

In general, if the return type of a function is a template parameter, then the function must have at least one parameter typed with the same template parameter, as in:

```
template<typename T>
T f(const T & param) { .... }
```

6.3.3 Template Specialization

It is sometimes necessary or desirable, for efficiency or other reasons, to provide a special implementation for some particular value of the template parameters. Suppose, for example, that we want to have both a general template class for all kinds of vectors, as above, but we also want to give special treatment to three-dimensional vectors with `double` elements. This is called *specialization*. Suppose the original class was

```
template<typename T>
class Widget
....
```

The specialized version will begin

```
template<>
class Widget<specType>
....
```

where `specType` is the value of `T` for which we are providing a specialized implementation. In the rest of the class, we must replace all occurrences of `<T>` with `specType`, including changing `Widget<T>` to `Widget<specType>`.

Figure 54 shows the result of specializing the generic vector class for 3D `double` vectors. The constructor has been modified with the addition of an output statement to demonstrate that the specialized version is actually used. The definition

```
Vector<double,3> v;
```

produces the message

```
3D Vector
```

Within the declaration of the specialized version, we have unrolled the `for`-loops and made other small changes. We could also add functions, such as cross product, that are useful for 3D vectors but not other vectors.

It is also possible to *partially specialize* a template class with two or more template parameters. For example, we could specialize `Vector` to 3D vectors with any element type. The class declaration would begin

```
template<typename Type>
class Vector<Type,3>
....
```

Within the class declaration, instances of `Dim` are replaced by 3, but instances of `Type` are left unchanged. References to the class all have the form `Vector<Type,3>`.

Template specialization is the key to *template metaprogramming*. The following class declaration is allowed:

```
template<int N>
class Fac
{
public:
    Fac()
    {
        Fac<N-1> f;
        val = N * f.getVal();
    }
    int getVal() { return val; }
private:
    int val;
};
```

```
template<>
class Vector<double,3>
{
public:
    Vector<double,3>()
    {
        v[0] = 0; v[1] = 0; v[2] = 0;
        cout << "3D Vector" << endl;
    }

    double & operator[](int i)
    {
         assert(0 <= i && i < 3);
         return v[i];
    }

    friend Vector<double,3> operator+(
          const Vector<double,3> & left,
          const Vector<double,3> & right )
    {
        Vector<double,3> result;
        result.v[0] = left.v[0] + right.v[0];
        result.v[1] = left.v[1] + right.v[1];
        result.v[2] = left.v[2] + right.v[2];
        return result;
    }

    friend ostream & operator<<(ostream & os,
                                const Vector<double,3> & vec)
    {
        return os << '(' <<
            vec.v[0] << ", " <<
            vec.v[1] << ", " <<
            vec.v[2] << ')';
    }

private:
    double v[3];
};
```

Figure 54: A specialized version of class `Vector`

However, there is a problem: instantiating `Fac<4>` requires instantiating `Fac<3>` requires instantiating . . . . We can terminate the recursion by providing a specialized class `Fac<0>`, as follows:

```
template<>
class Fac<0>
{
public:
    Fac() : val(1) {}
    int getVal() { return val; }
private:
    int val;
};
```

With these declarations, the following code prints 24:

```
Fac<4> f;
cout << f.getVal() << endl;
```

The compiler constructs class declarations for `Fac<4>`, `Fac<3>`, `Fac<2>`, `Fac<1>`, and `Fac<0>`. At run-time, executing the constructor for `Fac<4>` invokes the constructor for `Fac<3>`, and so on.

6.3.4 Default Arguments

Function declarations may have default arguments:

```
void foo(int n = 0) { .... }
```

The same is true of template class declarations. For example, if we changed the declaration of `Vector` in Figure 52 to

```
template<typename Type = double, int Dim = 3>
class Vector
....
```

and defined

```
Vector<int> u;
Vector<> v;
```

then `u` would be a 3D vector of `int`s and `v` would be a 3D vector of `double`s.

Note that the definition

```
Vector v;
```

is *not allowed*. The brackets <> are required even when we are using the default values of all parameters. (The same is true for functions, of course.)

7 Designing classes

Class design is a big topic in C++, with many aspects to consider. Since any one class does not illustrate all of the aspects of class design, we use two example classes to illustrate the issues. The first example class is `Rational`, which implements fractions and provides examples of the issues that arise in developing a numerical or algebraic class. The second example class is `Account`, which – with a bit of artifical manipulation — provides examples of memory management and other problems.

Rational Figure 65 on page 110 shows the declaration of class `Rational`. Comments that would normally be included in such a declaration have been omitted to save space. The accompanying text provides adequate explanation.

An instance of `Rational` is a rational number, or "fraction", represented by two long integers. The fraction $\frac{n}{d}$ is represented by the pair (n, d). The class provides arithmetic operations (`+`, `-`, `*`, `/`), and `^` (exponentiation)), the associated assignment operators (`+=`, `-=`, `*=`, `/=`), and comparisons (`==`, `!=`, `<`, `>`, `<=`, `>=`) for fractions. It also provides stream operators for reading and writing fractions.

Rational numbers are stored in *normalized form*. In the pair (n, d), we require $d > 0$ and $\gcd(n, d) = 1$. (We can consider $d > 0 \wedge \gcd(n, d) = 1$ to be a *class invariant*.) For example, the construction `Rational(4,-6)` yields the pair $(-2, 3)$. An attempt to create a fraction with zero denominator raises an exception.

Account The other example class, `Account`, is rather different. Some of the differences are due to the application and some are due to an intentional complication: `Account` has a data member that is a pointer.

An account is associated with a person who has a name. In class `Account`, the name is represented as a `char*` (pointer to array of characters). This introduces various problems, all of which could be avoided by using the standard class `string` instead of `char*`. We use `char*`, however, just to show what the problems are and how they can be solved.

Other differences between `Rational` and `Account` concern comparison, accessors, and mutators.

Figure 66 on page 111 shows the declaration of class `Account`. The data members of an account are: the `name` of the owner or client; an `id` number; and the `balance` in the account.

C++ does not have a type that is really suitable for financial work. Accountants do not like `double` because rounding prevents precise balancing of accounts. Long integers (type `long`) used to represent cents are quite good, provided that input and output functions make appropriate conversions between dollars and cents. However, the largest amount that can be represented with a long integer is $42,949,673 — not even enough for a bank president's annual income. To do good banking, we would need a special currency class. Curiously, the Boost library developers have never accepted a currency class, although examples exist[17] that could be raised to the standard of a Boost class fairly easily.

[17] See, for example, `http://www.colosseumbuilders.com/sourcecode.htm`

```
#ifndef RATIONAL_H
#define RATIONAL_H

#include <ostream>

class Rational
{
public:
    Rational(long num = 0, long den = 1);
    Rational & operator+= (const Rational & right);
    Rational & operator-= (const Rational & right);
    Rational & operator*= (const Rational & right);
    Rational & operator/= (const Rational & right);
    Rational operator-() const;
    Rational operator^ (int e) const;
    friend bool operator== (const Rational & left,
                            const Rational & right);
    friend bool operator!= (const Rational & left,
                            const Rational & right);
    friend bool operator<  (const Rational & left,
                            const Rational & right);
    friend bool operator>  (const Rational & left,
                            const Rational & right);
    friend bool operator<= (const Rational & left,
                            const Rational & right);
    friend bool operator>= (const Rational & left,
                            const Rational & right);
    friend std::istream & operator>> (std::istream & is,
                                      Rational & r);
    friend std::ostream & operator<< (std::ostream & os,
                                      const Rational & r);
    double toDouble() const;
    enum Exceptions { BAD_INPUT, ZERO_DENOMINATOR };

private:
    void normalize();
    long num;
    long den;
};

Rational operator+ (const Rational & left, const Rational & right);
Rational operator- (const Rational & left, const Rational & right);
Rational operator* (const Rational & left, const Rational & right);
Rational operator/ (const Rational & left, const Rational & right);

#endif
```

Figure 65: Declaration for class `Rational`

```
#ifndef ACCOUNT_H
#define ACCOUNT_H

#include <string>

class Account
{
public:
    Account();
    Account(char *name, long id, long balance = 0);
    Account(const Account & other);
    ~Account();
    Account & operator=(const Account & other);
    long getBalance() const;
    void deposit(long amount);
    void withdraw(long amount);
    void transfer(Account & other, long amount);
    friend bool operator==(const Account & left,
                           const Account & right);
    friend bool operator!=(const Account & left,
                           const Account & right);
private:
    void storeName(char *s);
    char *name;
    long id;
    long balance;
};

#endif
```

Figure 66: Declaration for class `Account`

We avoid the issue of a currency class in our example by using `long`.

There are other ways, too, in which the account class given here is unrealistic and nothing like a class that would be used in a banking application, for example. Nevertheless, the example provides some useful insights into class design.

7.1 Constructors

All classes have constructors. If you do not provide a constructor, the compiler generates a default constructor that allocates memory for the object but doesn't do anything else. Except in rare cases, for very simple classes, it is best to provide one or more constructors. If you provide any constructor at all, the compiler does *not* generate a default constructor.

A *default constructor* is a constructor without any parameters. There are a number of situations in which a default constructor is required; for example, an array declaration is allowed only for types

that have a default constructor.

Define a default constructor for every class.

Warning There is a small but important inconsistency in C++ notation. Suppose that C is a class with two constructors, one of which is a default constructor and consider these statements:

```
C c1(45);
C c2();
```

It is natural to read the statements like this:

- `c1` is an instance of C constructed from an integer;
- `c2` is an instance of C constructed using the default constructor.

Unfortunately, this interpretation is wrong! In fact, `c2` has been declared (but not defined) as a function that takes no arguments and returns a C. To invoke the default constructor, you must *leave out the parentheses*:

```
C c1(45);
C c2;
```

Rational Class `Rational` requires only one constructor. It is a general purpose constructor with two `long` parameters, corresponding to the numerator and denominator of the fraction, stored as `num` and `den`, respectively. Both parameters have default values (`num` = 0, `den` = 1), which implies that this constructor counts as a default constructor. The constructor calls `normalize` (see Figure 67 on page 113) to put the fraction in normal form.

```
Rational::Rational(long num, long den)
    : num(num), den(den)
{
    normalize();
}
```

The data members are set using initializers rather than assignments. This is the best way to initialize data members and you should use it whenever possible.

Use initializers in constructors.

```
long gcd(long i, long j)
{
    assert(i > 0 && j > 0 && "Error in gcd arguments");
    while (true)
    {
        long tmp = j % i;
        if (tmp == 0)
            return i;
        j = i;
        i = tmp;
    }
    assert(false && "Error in gcd");
}

void Rational::normalize()
{
    if (den == 0)
        throw Rational::ZERO_DENOMINATOR;
    if (num == 0)
    {
        den = 1;
        return;
    }
    if (den < 0)
    {
        num = -num;
        den = -den;
    }
    assert(num != 0 && den > 0);
    long g = gcd(abs(num), den);
    num /= g;
    den /= g;
}
```

Figure 67: Functions to normalize instances of class `Rational`

The private member function `normalize` is called whenever a new fraction is created, either by a constructor or by some other means. If `den` $= 0$, it throws the exception `ZERO_DENOMINATOR`, which is one of the values of the enumeration `Exceptions` declared in the class. If `num` $= 0$, it sets `den` $= 1$, to ensure that the fraction zero has the unique representation $(0,1)$. If `den` < 0, it changes the sign of both numerator and denominator. Finally, it divides both by their greatest common divisor to cancel common factors.

An important consequence of providing a default value for `den` in the constructor is that `Rational(n)` constructs the pair $(n,1)$ corresponding to the fraction $n/1$. This form of the constructor is used in a context where the compiler expects a `Rational` but finds a value of integral type. It allows

us to perform "mixed mode" arithmetic so that, for example `Rational(1,3) + 1` evaluates to `Rational(4,3)`.

Account The normal constructor for an `Account` is passed the name, identification code, and opening balance for the new account. If the opening balance is omitted, it defaults to zero.

The name is passed as a `char*`. Simply copying the pointer would be a serious mistake. As one of the many ways in which things could go wrong, consider this function:

```
Account * createAccount()
{
    char buffer[10000];
    cout << "Please enter your name: ";
    cin >> buffer;
    int id;
    cout << "Please enter your account ID: ";
    cin >> id;
    return new Account(buffer, id);
}
```

Since the bad guys use buffer overflow as the basis of many attacks, it is always a mistake to read characters into a character array. To make it hard for them, we can use a long buffer. The real problem with this function, however, is that the buffer is destroyed when the function returns. If the new account is not to be left with a pointer into outer space, it had better make a copy of the buffer.

This constructor does indeed make a copy of the name passed to it:

```
Account::Account(char *name, long id, long balance)
    : id(id), balance(balance)
{
    storeName(name);
}
```

It turns out that storing the name is something that we will have to do several times. Consequently, it makes sense to put it into a function. Space for the name is allocated dynamically (i.e., using `new`, with space taken from the heap) and we must remember to allocate one character position for the terminator, `'\0'`. (We are using the old-style functions `strlen` and `strcpy`. As mentioned above, these problems do not arise with the more modern class `string`.)

```
void Account::storeName(char *s)
{
    name = new char[strlen(s) + 1];
    strcpy(name, s);
}
```

The normal constructor is *not* a default constructor because the parameters do not have default values. We could give them default values or we could define another constructor for use as the default, like this:

```
Account::Account()
    : id(0), balance(0)
{
    storeName("");
}
```

The *copy constructor* is an important component of every C++ class. It is used whenever an object has to be copied. Common uses of the copy constructor include:

- Initialized declarations of the form

  ```
  Account anne = bill;
  ```

- Passing an object by value.
- Returning an object by value.

If we do not define a copy constructor, the compiler generates one for us. This *default copy constructor* simply copies the values of the data members of the object. The default copy constructor for `Rational` does exactly what we want and there was no need to define our own version. Class `Account` is different.

Class `Account` has the data member `name`, which is a pointer. The default copy constructor just copies the pointer, not the object it points to. This would be a disaster for accounts, because we would end up with more than one pointer pointing to the same name. If one account is deleted, the others would be left with *dangling pointers*. (Technically, the copy constructor performs a *shallow copy* but what we need is a *deep copy*.)

The copy constructor must therefore behave like the constructor, making a new copy of the name. Fortunately, we have a function `storeName` that does exactly the right thing.

The signature (or prototype) of the copy constructor for a general class `T` is `T::T(const T &)`. For our class, `Account`, the implementation looks like this:

```
Account::Account(const Account & other)
      : id(other.id), balance(other.balance)
{
    storeName(other.name);
}
```

Define a copy constructor for any class that has pointer members.

There is an alternative way of defining a copy constructor that is sometimes useful:

- Declare the copy constructor `T(const T &)` in the `private` part of the class declaration.
- Do *not* provide a definition of the copy constructor.

The effect is to prevent copying of the object. By declaring the copy constructor, we prevent the compiler from generating it. By making the copy constructor private, we prevent outsiders from calling it. By not defining the copy constructor, we prevent member functions from using it. The result is that any initialized declaration, passing by value, or returning by value, will be flagged as an error by the compiler.

It is quite possible that a real-life banking application might choose to prevent copying of accounts. This is the mechanism that could be used.

Note that preventing copying does not make `Account` a Singleton; we can have as many accounts as we need, but we cannot make copies of them.

7.2 Destructor

A *destructor* is a member function that is called when an object is to be deallocated or "destroyed". The compiler provides a default destructor if you don't. There are two circumstances in which you *must* define a destructor:

1. The class has members that are pointers.
2. The class will be used as a base class.

Rational Class `Rational` does not have any pointer members and is not intended to be used as a base class. Consequently, we do not define a destructor for it.

Account We must define a destructor for class `Account` because it has a pointer member. Later, we will discuss using `Account` as a base class, which provides another reason for having a destructor.

Define a destructor for any class that has pointer members.

The destructor must destroy any data that was created dynamically (using `new`) by the constructor. Destruction is performed by `delete`, which has two forms:

- `delete x` for simple objects
- `delete [] x` for arrays

The distinction between the two kinds of destruction is very important, because the compiler cannot correct you if you are wrong. In `Account`, we must use `delete [] name`; if we wrote `delete name` instead, only the first character of the name would be deallocated.

Use `delete` for simple variables and `delete[]` for arrays.

Here is the destructor for class `Account`:

```
Account::~Account()
{
    delete [] name;
}
```

7.3 Operators

C++ allows most operators to be overloaded. This is an extremely useful feature of the language, but it should not be abused. If operators are defined, they should be defined consistently and they should behave in a reasonable way.

Rational Class `Rational` is an obvious candidate for overloaded operators, because fractions are numbers and users will expect to use arithmetic operations with fractions. To respect C++ conventions, if you provide +, you should also provide +=, and similarly for the other operators. It turns out to be easier, and more efficient, to define the assignment operators (+=, -=, *=, /=) as member functions and then to use them in the definitions of the simple operators (+, -, *, /).

Figure 68 on page 118 shows the implementation of the assignment operators. Each one normalizes its result and returns a reference to the new fraction. This is another convention that you should follow; it allows users to write statements such as

```
p += q *= r;
```

assuming that they can figure out what such expressions mean.

7.3.1 Assignment (`operator=`)

By default, the compiler will generate a default assignment operator (`operator=`). The effect of this operator will be to copy all of the fields of the object. If this is what you want, you do not need to define your own version of `operator=`.

Rational The default assignment operator is just what we need for `Rational` objects: the statement `r = s` will copy the numerator and denominator of `s` to `r`. Consequently, we do not define `operator=`.

```
Rational & Rational::operator+= (const Rational & right)
{
    num = num * right.den + den * right.num;
    den *= right.den;
    normalize();
    return *this;
}

Rational & Rational::operator-= (const Rational & right)
{
    num = num * right.den - den * right.num;
    den *= right.den;
    normalize();
    return *this;
}

Rational & Rational::operator*= (const Rational & right)
{
    num *= right.num;
    den *= right.den;
    normalize();
    return *this;
}

Rational & Rational::operator/= (const Rational & right)
{
    num *= right.den;
    den *= right.num;
    normalize();
    return *this;
}
```

Figure 68: Arithmetic assignment operators for class `Rational`

```
Account & Account::operator=(const Account & other)
{
    storeName(other.name);
    id = other.id;
    balance = other.balance;
    return *this;
}
```

Figure 69: An *incorrect* implementation of `operator=`

Account Since `Account` has a pointer member, we *must* define an assignment operator for it. Not doing so will lead to the same problems as not having a copy constructor: we will end up with many accounts all pointing to the same name. The assignment operator looks rather like the copy constructor, but there are two important differences. Consider the *incorrect* version of the assignment operator shown in Figure 69.

Note first that the type of the assignment operator is `Account&` and that the function returns `*this`. This is conventional for assignment operators and it allows statements such as

```
a1 = a2 = a3 = a4 = a5;
```

for programmers who want to write such statements.

Now suppose that the programmer writes

```
a = a;
```

and think about the effect in `Account::operator=` as defined above. `storeName` allocates space for the name, and copies the name into. The old name lost — a memory leak. If we insert the statement `delete [] name` to avoid the memory leak, things get even worse: `storeName` would attempt to copy the deleted name! The solution is to check for self-assignment. These considerations lead to the correct assignment operator shown in Figure 70.

```
Account & Account::operator= (const Account & other)
{
    if (this == &other)
        return *this;
    delete [] name;
    storeName(other.name);
    id = other.id;
    balance = other.balance;
    return *this;
}
```

Figure 70: Assignment operator for class `Account`

We might ask: "What programmer could be so stupid as to write `a = a`?" The answer is that a programmer might not write this assignment as such, but it could easily be generated by template expansion. Also, a programmer might quite reasonably write

```
a[i] = a[j];
```

and not feel it necessary to check $i \neq j$.

> ***Define an assignment operator for any class that has pointer members. The assignment operator should check for self-assignment and return a reference to `*this`.***

If you want to prevent assignment of instances of a class, declare `operator=` as a `private` member function and do not provide an implementation for it.

7.3.2 Arithmetic

If it makes sense to perform arithmetic operations on instances of the class, we can provide the appropriate operators. These are usually implemented in conjunction with the corresponding assignment operators, described above.

In general, you should implement the arithmetic functions so that they obey standard laws of algebra. For example, – is the inverse of +, / is the inverse of *, and so on.

There are occasional exceptions to this rule. For example, `string` uses + for concatenation, even though concatenation is neither commutative nor associative, and has no inverse. But people seem to accept + as a concatenation operator, so this is perhaps excusable.

Arithmetic operators, such as `operator+`, can be implemented as member functions with one parameter or as free functions with two parameters. If we implement them as member functions, then

```
x + y
```

is effectively translated as

```
x.operator+(y)
```

The compiler will use the static type of `x` to choose the appropriate overload of `operator+` and may convert `y` to match the type of `x`. Suppose that we implemented + for class `Rational` in this way and that `i` is an `int` and `r` is a `Rational`. Then `r+i` converts `i` to `Rational` and adds the resulting fractions but `i+r` does not compile because the integer version of `operator+` cannot accept a `Rational` right argument. In other words, the advantage of defining comparison operators as free functions is that the order of operands does not matter.

There is another issue to consider when we choose to implement free functions associated with a class: do we provide access functions for data members of the class, or do we declare the free functions as `friend`s? The choice depends very much on the particular application. For example, if the class already provides accessor functions for some reason, the free functions can make use of them. If security is important, and data members should not be exposed, then `friend` functions may be a better choice.

Rational Figure 71 on page 121 shows the standard arithmetic operators for class `Rational`, implemented as free functions that use the corresponding assignment operators. Since class `Rational` does not export the numerator and denominator of a fraction, they are declared as `friend`s of the class.

If you give users the arithmetic operators, they will expect unary minus as well. Unary minus is best implemented as a member function with no arguments. It does *not* negate the value of the fraction, but instead returns the negated value while remaining unchanged. Consequently, we can qualify it with `const`.

```
Rational Rational::operator- () const
{
    return Rational(-num, den);
}
```

```
Rational operator+ (const Rational & left, const Rational & right)
{
    Rational result = left;
    result += right;
    return result;
}

Rational operator- (const Rational & left, const Rational & right)
{
    Rational result = left;
    result -= right;
    return result;
}

Rational operator* (const Rational & left, const Rational & right)
{
    Rational result = left;
    result *= right;
    return result;
}

Rational operator/ (const Rational & left, const Rational & right)
{
    Rational result = left;
    result /= right;
    return result;
}
```

Figure 71: Arithmetic operators for class `Rational`, implemented using the arithmetic assignments of Figure 68 on page 118

C++ programmers expect to use `pow` for exponentiation. But `pow` takes arguments of many types, even for the exponent. For fractions, we take the slightly daring approach of providing `^` as an exponential operator. The exponent must be an integer, but the integer may be positive or negative: $\left(\frac{n}{d}\right)^{-e}$ is evaluated as $\left(\frac{d}{n}\right)^{e}$. If $r = 0$, then r^e will fail (throwing an exception) if $e < 0$. The function uses the identity $x^{2e} = (x^2)^e$ when e is even to achieve complexity $\mathcal{O}(\log e)$ rather than $\mathcal{O}(e)$. Since exponentiation is not commutative, and the left operand must be a `Rational`, we implement `operator^` as a member function, as shown in Figure 72 on page 122.

There is a minor problem with using `operator^` as an exponent operator. Although we can overload operators in C++, we cannot change their precedence. As it happens, `operator^`, which is normally used as exclusive-or on bit strings, has a *lower* precedence than the other arithmetic operators. Its precedence is even lower than that of `operator<<` and `operator>>`. So we had better warn our users to put exponential expressions in parentheses!

```
Rational Rational::operator^ (int e) const
{
    if (e < 0)
        return (1/ *this)^(-e);
    else if (e == 0)
        return 1;
    else if (e % 2 == 0)
        return (*this * *this)^(e/2);
    else return *this * (*this^(e - 1));
}
```

Figure 72: An exponent function for class `Rational`

7.3.3 Comparison

There are two important kinds of comparison: *equality* and *ordering*. Identity comparison corresponds to the operators == and != and is useful for many kinds of objects. Ordering corresponds to the operator < and its friends and is useful only for objects for which some kind of ordering makes sense.

By convention, the ordering operators return a Boolean value `true` or `false`. They can implement only a *total ordering* in which, for any objects x and y, one of the following must be true: $x < y$, $x = y$, or $x > y$. They cannot be used to implement a partial ordering, in which two objects may be unrelated.

Rational Fractions are totally ordered. We can define the ordering by

$$\frac{n_1}{d_1} > \frac{n_2}{d_2} \quad \Longleftrightarrow \quad n_1 \times d_2 > n_2 \times d_1.$$

Figure 73 on page 123 shows the corresponding functions for class `Rational`. Like the arithmetic operators, they are implemented as free, `friend` functions.

Account Considering the comparison operators for class `Account` raises interesting questions about equality.

- Does it make sense to say that two accounts are "equal"?
- If it does make sense, what does it mean to say "account A equals account B"?

Comparing balances probably is not very helpful. Comparing names is unsafe, because two people might have the same name.[18] There are two comparisons that might be reasonable:

- Two accounts are equal if they have the same ID (*extensional equality*)
- Two accounts are equal if they are the same object (*intensional equality* or *identity*)

[18] This is, in fact, the cause of many "mistaken identity" problems.

```
bool operator== (const Rational & left, const Rational & right)
{
    return left.num * right.den == right.num * left.den;
}

bool operator!= (const Rational & left, const Rational & right)
{
    return !(left == right);
}

bool operator<  (const Rational & left, const Rational & right)
{
    return left.num * right.den < right.num * left.den;
}

bool operator>  (const Rational & left, const Rational & right)
{
    return left.num * right.den > right.num * left.den;
}

bool operator<= (const Rational & left, const Rational & right)
{
    return left < right || left == right;
}

bool operator>= (const Rational & left, const Rational & right)
{
    return left > right || left == right;
}
```

Figure 73: Comparison operators for class `Rational`

The following comparison functions check for identity (accounts are equal only if they are the same object). Note that this is not really realistic because, in a practical application, account objects would spend most of their lives on disks and would only occasionally be brought into memory.

```
bool operator==(const Account & left, const Account & right)
{
    return &left == &right;
}

bool operator!=(const Account & left, const Account & right)
{
    return &left != &right;
}
```

It would be straightforward to compare IDs instead.

Accounts could be ordered by name or by ID; this would allow us to sort a `vector` of accounts, for example. These operators are easy to add if needed.

7.3.4 Input and output

For many classes, it useful to provide the insertion operator (<<), if only for debugging purposes. The extraction operator, >>, is less often needed, but can be provided as well. When these operators are provided, they are commonly implemented as `friend`s.

Rational For fractions, both input and output operators make sense, as we implement them as friends. Both present complications.

For input, the first decision we have to make is: what should the user enter? Let's say that the user must enter two integers separated by a slash:

```
Enter a rational:  2/3
```

The next decision is: how does the program respond if the user does *not* enter the data in the form that we expect? There are many possible solutions, ranging from accepting only input that is exactly correct to parsing what the user enters and figuring out what was meant.

In the following function, we follow a middle way: we read an integer, a character, and another integer, and we throw an exception if the character is not `'/'`. Although rationals can be constructed from integers, the user is required to enter a complete fraction, even if it is `3/1`.

```
istream & operator>> (istream & is, Rational & r)
{
    long m, n;
    char c;
    is >> m >> c >> n;
    if (c != '/')
        throw Rational::BAD_INPUT;
    r = Rational(m, n);
    return is;
}
```

Output is usually more straightforward than input, but there is one problem. Suppose that we implement the inserter in the "obvious" way

```
os << num << '/' << den;
```

and the user writes

```
cout << setw(12) << r;
```

in which `r` is a `Rational`. Our function will use 12 columns to write the numerator and then will write the slash and the numerator — probably not what the user expected!

Of the various ways to correct this error, the simplest is to format the entire fraction, in the obvious way, into a buffer, and then to insert the buffer into the output stream. This ensures that any width modifiers will be applied to the complete fraction, not just to the numerator.

The remaining issue is how to write whole numbers (fractions with denominator 1). To avoid sillinesses like `3/1`, we will not write the slash or the denominator for whole numbers. This reasoning leads to the following function.

```
ostream & operator<< (ostream & os, const Rational & r)
{
    ostringstream buffer;
    buffer << r.num;
    if (r.den != 1)
        buffer << '/' << r.den;
    return os << buffer.str();
}
```

String streams can be used for either output (`ostringstream`) or input (`istringstream`) and require the directive

```
#include <sstream>
```

- An output string stream behaves like any other output stream (e.g., `cout`).
- After writing things to it, you can extract the string of formatted data using the function `str`, which returns a `string`.
- After `str` has been invoked on an `ostringstream`, the stream is *frozen* and does not permit further write operations.
- When the string stream is deleted (usually at the end of the current scope), data associated with it is deleted.

An input string stream can be used to parse a string. In the following code, the input string stream `isstr` is initialized with `myString`. The `isstr` is used, like any other input stream (e.g., `cin`), with extract operators.

```
string myString = "3 4 5 words";
istringstream isstr(myString);
int a, b, c;
string w;
isstr >> a >> b >> c >> w;
```

7.4 Conversions

Conversions are a delicate issue in C++. Programmers don't like writing explicit conversions and want the compiler to do the dirty work for them but too many conversions can be a bad thing.

Rational It seems reasonable to convert rationals to floating-point numbers. Sometimes, for example, we might want 0.66667 rather than 2/3. C++ provides a powerful way of implementing such conversions, using operator notation. We can define a conversion from `Rational` to `double` by adding this member function to class `Rational`:

```
operator double () const
{
    return double(num) / double(den);
}
```

This function provides an *implicit conversion* from `Rational` to `double`: in any context where the compiler expects to find an expression of type `double`, but in fact finds an expression of type `Rational`, it will insert a call to this function to perform the conversion.

Unfortunately, this solution is *too* effective! If we write `1/r`, for example, the compiler complains about ambiguity. In `1/r`, it can either convert `1` to `Rational` and evaluate the reciprocal of `r` as a `Rational` *or* it can convert both `1` and `r` to `double` and evaluate the reciprocal of `r` as a `double`. Since we would like to keep the convenience of being able to write `1/r` for the `Rational` reciprocal of a `Rational`, we choose *not* to provide `operator double`.

It is not hard to find a better alternative: we simply provide the same function with a funny name to prevent the compiler from using it implicitly. The following function does what we need and will be invoked only when the user explictly requests it by writing `r.toDouble()`.

```
double Rational::toDouble() const
{
    return double(num) / double(den);
}
```

7.5 Accessors

An *accessor* or *inspector* is a member function that returns information about an object without changing the object. It follows immediately from this definition that accessors should always return a non-`void` value and should be declared `const`.

Rational The only candidates for accessors for class `Rational` are `getNum` and `getDen`, implemented in the obvious way. The decision *not* to provide these was that the representation of a rational number is a "secret" and providing these accessors would give away part of the secret.

If the class did provide `getNum` and `getDen`, the `friend` functions would no longer need to be declared as friends.

Account There are several candidates for accessors for class `Account`. We provide just an accessor for the account balance, to demonstrate the general idea.

```
long Account::getBalance() const
{
    return balance;
}
```

7.6 Mutators

A *mutator* is a member function that changes the state of the object. Usually, the return type of a mutator is `void` (otherwise the mutator would be a function with a side-effect).

Rational Any function that changes the state of a rational should do so in a way that makes semantic sense. Thus += is acceptable, because it changes the state by adding another rational. Arbitrary mutations of the numerator and denominator do not make semantic sense, and so we do not provide mutators for class `Rational`.

Account There are several ways in which the state of an account may reasonably be changed. We provide three "obvious" functions: `deposit`, `withdraw`, and `transfer`. Of these, `deposit` is straightforward.

```
void Account::deposit(long amount)
{
    balance += amount;
}
```

For withdrawals, we have to decide what to do when the balance would go negative. The solution adopted here is to throw an exception. Since we do not know who will be handling the exception, we must ensure that sufficient information is provided. To achieve this, we declare a special exception class (described below) and store the account ID and a helpful message in the exception object.

```
void Account::withdraw(long amount)
{
    if (amount > balance)
        throw AccountException(id,
              "Withdrawal: amount greater than balance");
    else
        balance -= amount;
}
```

Standard exception classes provide a function `what` that returns a description of the exception. Although we could inherit from one of the standard exception classes, we choose not to do so here, but we provide `what` anyway. Figure 74 on page 128 shows the declaration and implementation of the class `AccountException`.

The third mutator for class `Account` is `transfer`, which transfers money from one account to another. We allow an account to transfer funds *to* another account but not to transfer funds *from* another account. This function will throw an exception if the transfer would leave the giving account with a negative balance.

```
void Account::transfer(Account & other, long amount)
{
    withdraw(amount);
    other.deposit(amount);
}
```

```
class AccountException
{
public:
    AccountException(long id, std::string reason);
    std::string what();
private:
    long id;
    std::string reason;
};


AccountException::AccountException(long id, string reason)
    : id(id), reason(reason)
{}

string AccountException::what()
{
    ostringstream ostr;
    ostr << "Account " << id << ".  " << reason << '.';
    return ostr.str();
}
```

Figure 74: Declaration and implementation of class `AccountException`

The function `transfer` is implemented using the previously defined functions `withdraw` and `deposit`. It is always a good idea to use existing code when it applies, rather than introducing more code, with possible errors.

In real-life banking, an operation such as `transfer` must be implemented very carefully: either the transaction must succeed completely, or it must fail completely and have no effect. (Database gurus are familiar with this problem of *transaction integrity*.) Without pretending that `transfer` is really a secure function, we note that there is only one (obvious) way that it can fail — `withdraw` may throw an exception — and that this failure leaves both accounts unchanged.

7.7 Odds and ends

7.7.1 Indexing (`operator[]`)

C++ allows us to overload the "operator" `[]`. The overload has one parameter, which can be of any type, and returns a value, which can be of any type. The syntax for calling the function is `a[i]`, in which `a` is an object and `i` is the argument passed to the function. Typically, we would use `operator[]` for a class that represented an array, vector, or similar kind of object. Figure 75 on page 129 shows an inefficient and incomplete map class that associates names with telephone numbers, both represented as strings. The store is accessed by calling `operator[]` with a string argument. The following code illustrates the use of this store (note the final statement).

```
template<typename T>
class Store
{
public:
    void insert(string key, T value)
    {
        data.push_back(pair<string, T>(key, value));
    }
    T operator[](string key)
    {
        for (   vector<pair<string, T> >::const_iterator it =
                                                data.begin();
                it != data.end();
                ++it )
            if (it->first == key)
                return it->second;
        return T();
    }
private:
    vector<pair<string, T> > data;
};
```

Figure 75: A simple map class

```
Store<string> myPhoneBook;
myPhoneBook.insert("Abe", "486-2849");
myPhoneBook.insert("Bo", "982-3847");
cout << myPhoneBook["Abe"];
```

In this example, `operator[]` has one parameter. This is the only possibility: you cannot declare `operator[]` with no parameters or with more than one parameter.

7.7.2 Calling (`operator()`)

C++ allows us to overload the "operator" `()`. The overload has one parameter, which can be of any type, and returns a value, which can be of any type. The syntax is `f(x)`, in which `f` is an object and `x` is the argument passed to the function. Typically, we would use `operator()` in a situation where it makes sense to treat an instance as a function.

The Command pattern (Gamma, Helm, Johnson, and Vlissides 1995, page 233) "encapsulates a request as an object, thereby enabling you to parameterize clients with different requests, queue or log requests, and support undoable operations". At the appropriate time, a request is activated in some way. The Gang of Four use a function called `Execute` to activate a request, but `operator()` provides a slightly neater solution.

The `Command` shown in Figure 76 on page 130 class stores requests to print messages. The message to be printed is passed as an argument to the constructor. To activate a command `c`, the user

```
class Command
{
public:
    Command(string message) : message(message) {}
    void operator()() { cout << message << endl; }
private:
    string message;
};
```

Figure 76: A simple command class

```
vector<Command> commands;
commands.push_back(Command("First message"));
commands.push_back(Command("Second message"));
// ....
for (   vector<Command>::iterator it = commands.begin();
        it != commands.end();
        ++it )
    (*it)();
```

Figure 77: Using the command class

writes `c()`.

For example, commands can be stored in a vector and then activated sequentially, as shown in Figure 77 on page 130.

You can declare `operator()` with any number of parameters and the syntax is conventional: see Figure 78.

```
class Test
{
public:
    void operator()(int i, int j) { cout << i+j << endl; }
};

int main()
{
    Test t;
    t(5,6);
}
```

Figure 78: Defining `operator()` with two parameters

7.7.3 Explicit constructors

A constructor with a single parameter provides a form of type conversion. For example, suppose class `Widget` has a constructor with a parameter of type `int`:

```
class Widget
{
public:
    Widget(int n);
    ....
```

This constructor gives the compiler permission to convert an integer to a `Widget` whenever it has an opportunity to do so. This behaviour might be appropriate, but it might also be an error. For example, class `string` has a constructor with an integer argument so that

```
string s(N);
```

constructs a string with N characters. A possible consequence is that

```
string s = 'a';
```

constructs a string with 97 characters — probably not what the programmer intended.

In fact, this does *not* happen, because the string constructor does not allow implicit conversions. The conversion is prevented by qualifying the constructor with `explicit`. To prevent `Widgets` being constructed from integers, rewrite the example above as

```
class Widget
{
public:
    explicit Widget(int n);
    ....
```

The keyword `explicit` does not prevent the constructor from being used at all, of course. It says that, in order to use this constructor, the call `Widget(N)` must actually appear in the program; the compiler will never call the constructor implicitly.

The keyword `explicit` can be used with constructors only; it cannot be used with conversion operators such as `operator double` described in Section 7.4.

7.7.4 Friends

A function associated with a class can have three properties (Stroustrup 1997, page 278):

1. it can access private members of the class
2. it is in the scope of the class

3. it must be invoked on an object

Property 2 means that the function `f` associated with class `C` must be invoked in one of the following ways:

- `C::f()`
- `c.f()` where `c` is an instance of `C`
- `pc->f()` where `pc` is a pointer to an instance of `C`

Property 3 means that statements in the function can use the `this` pointer to the object for which the function is invoked.

The following table shows which kinds of function have these properties:

	Property		
Function kind	1	2	3
`friend`	√		
`static`	√	√	
member	√	√	√

The table shows that the three kinds of function form a hierarchy, with member functions having the most, and friend functions the least, access to the class.

Contrary to popular belief (especially amongst Java programmers), friends do not violate encapsulation in C++. The important point is that friends *must be declared within the class*. This means that a class has complete control over its friends — enemies cannot use friends to subvert the class's protection mechanisms. Here is Stroustrup's opinion of friends:

> *A friendship declaration was seen as a mechanism similar to that of one protection domain granting a read-write capability to another. It is an explicit and specific part of a class declaration. Consequently, I have never been able to see the recurring assertions that a* `friend` *declaration "violates encapsulation" as anything but a combination of ignorance and confusion with non-C++ terminology.* (Stroustrup 1994, page 53)

Here are some useful things to know about friends:

- A `friend` declaration can be placed anywhere in a class declaration. It is not affected by `public` or `private` attributes.
- A function or a class can be declared as a `friend`. In either case, friends of a class can access private members (data and functions) of the class.
- Two or more classes can declare the same class or function as a friend.

Used correctly, friends actually provide *better* protection than other techniques. Here is an example (Stroustrup 1997, page 278): we have classes `Vector` and `Matrix` and we want to define functions that, for example, multiply a matrix by a vector. We could define the multiply function as a free function, external to both classes, but then we would have to expose the representations of both `Vector` and `Matrix` for this function to use. Instead, we use friends:

```
class Vector
{
    friend Vector operator* (const Matrix & m,
                             const Vector & v);
    ....
}

class Matrix
{
    friend Vector operator* (const Matrix & m,
                             const Vector & v);
    ....
}

Vector operator* (const Matrix & m,
                  const Vector & v)
{
    ....
}
```

This provides a neat solution to the problem: the multiply function has access to private data in *both* classes but we have not "opened them up" to anyone else.

Nevertheless, if there is a choice between a member function and a friend, it is better to use a member function. The main reason for this preference is conversion: in the class `f(x,y)`, the compiler may perform conversions to match the types of the arguments `x` and `y` to the parameter types. The call `x.f(y)` invokes `X::f` where X is the class of X (although `y` may still be converted).

8 Inheritance and templates

Some of the topics in this section are discussed more fully in Scott Meyers's book (1992, Items 35 through 44), although you should note that this book is quite old (1992) and occasionally refers to problems that have been solved in more recent versions of C++.

We will use (mostly) C++ terminology for inheritance: a *derived class* inherits from a *base class*. Inheritance may be `public` (the most common case) or `private`, depending on the keyword used before the base class in the derived class declaration:

```
class Base { .... };
class D1 : public Base { .... };    // public inheritance
class D2 : private Base { .... };    // private inheritance
```

8.1 What is inheritance?

Perhaps the most confusing aspect of the word "inheritance" is that it is used with several quite different meanings:

1. Inheritance as an "is-a" relationship.

   This form of inheritance models the human urge to form hierarchies of classification ("taxonomies"). Each of the following sets contains its successors: living things, animals, mammals, dogs, terriers. Read backwards, these sets form an is-a hierarchy: a terrier *is a* dog, a dog *is a* mammal, a mammal *is an* animal, etc.

   The important feature of a taxonomy is that a smaller set *inherits* all of the behaviour of the larger sets that contain it. A dog has *all* of the properties of a mammal — otherwise it wouldn't be a mammal. (We discuss apparent exceptions to this rule below.)

   There are two kinds of is-a relationship, depending on what gets inherited. A base class declares various functions, and the derived class(es) may inherit:

   (a) the declarations of the function only, or

   (b) the declarations and definitions of the functions.

   We refer to the first case as *interface inheritance* and the second case as *implementation inheritance*.[19]

2. Inheritance as "implemented using". For example, we could implement a `Stack` by inheriting a `vector`, giving a "stack implemented using vector" class. The important point here is that `vector` and `Stack` provide different sets of functions, but the `Stack` functions are easily implemented using the `vector` functions. However, users of `Stack` must not be allowed to use the `vector` functions because that would violate the integrity of the stack.

The implementation mechanisms for these kinds of inheritance in C++ are, roughly:

[19] These correspond roughly to `implements` and `extends` in Java. Note, however, that Java `implements` corresponds to *interface inheritance* in C++ and Java `extends` corresponds to *implementation inheritance* in C++.

1. `public` inheritance
   (a) abstract base class with pure `virtual` functions
   (b) base class with `virtual` declarations and default definitions
2. `private` inheritance

The distinction between 1(a) and 1(b) can become blurred. A base class with one or more pure virtual functions is called an *abstract base class* (or "ABC") and defines, in effect, an interface: this is 1(a). A base class with implemented virtual functions is a complete, working class that can be specialized by derived classes that inherit some or most of its implementation and modify the rest: this is 1(b). In between, there are base classes with a mixture of pure and impure virtual functions, defining a sort of partially-implemented interface.

It is very important not to confuse "is-a" with "has-a". In some (older) books, you may see things like Figure 79 described as "inheritance hierarchies". But this is a "has-a" hierarchy: a car has an engine, an engine has pistons, etc. It is absurd to say "a valve is an engine" or "a handle is a door". Hierarchies of this kind are represented by *layering* (also called *composition*), in which an instance of class `Car` contains instances of classes `Door`, `Wheel`, and `Engine`, etc.

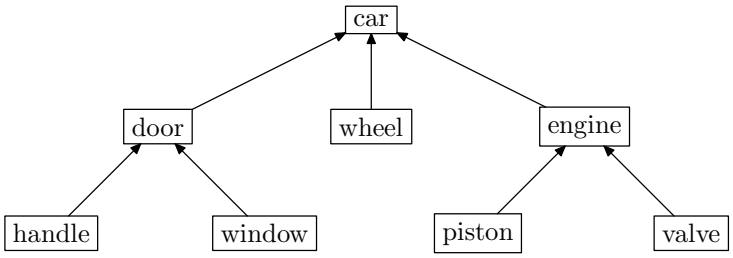


Figure 79: Not a class hierarchy

Figure 80 shows a popular example used to demonstrate the "difficulties" of inheritance. There are various solutions for this "problem". One solution is to redefine `Penguin::fly`:

```
class Penguin : public Bird
{
    void fly() { throw PENGUINS_CANNOT_FLY_EXCEPTION; }
};
```

This is not a good solution, because it violates the rule that a penguin is-a bird: a penguin is not a bird, because birds fly and penguins don't. Figure 81 shows a better solution obtained by realizing that the class hierarchy that we have defined is incomplete: there are birds that fly and birds that don't fly. The revised class hierarchy easily accommodates kiwis, ostriches, turkeys, and cassowaries.

```
class Bird
{
public:
    virtual void fly();
    ....
};

class Penguin : public Bird
{
    ....
};

int main()
{
    Bird pb = new Penguin();
    pb->fly();                  // oops - penguins can't fly!
}
```

Figure 80: The problem of the flightless penguin

```
class Bird
{
public:
    // no definition of fly()
    ....
};

class FlyingBird : public Bird
{
    virtual void fly();
};

class NonFlyingBird : public Bird
{
public:
    // no definition of fly()
    ....
};

class Penguin : public NonFlyingBird
{
    ....
};
```

Figure 81: Recognizing that some birds don't fly

8.1.1 Slicing

A derived class inherits data members from its base class and may define its own data members. Consequently, a derived class instance `d` is as large or larger than an instance `b` of the corresponding base class. There are several consequences:

- The assignment `b = d` is allowed. Since the data in `d` does not fit into `b`, it is not copied: we say that `d` is *sliced*. Slicing occurs not only during assignment but also when a derived object is passed or returned by value and the destination is a base class object.
- The assignment `d = b` is not allowed because it would leave the derived data in `d` undefined.
- It is usually a mistake to use the base class of a hierarchy as a template argument. Consider:

```
vector<Base> bases;
Derived der( .... );
bases.push_back(der);
```

  Since the argument of `push_back` is passed by value, `der` is sliced; data in `Derived` but not in `Base` is lost. To avoid this problem, use pointers, as in `vector<Base*>`.

Slicing does not occur when we address the object through references or pointers. Suppose that `pb` is a pointer to a base class object and `pd` is a pointer to an instance of a derived class. Then:

- The assignment `pb = pd` is allowed. After the assignment, `pb` will point to the complete object, containing base and derived data, but only base class functions and data will be accessible, because of the type of `pb`. Assignments of this kind are called *upcasts* because they go "up" the class hierarchy, from derived class to base class..
- The assignment `pd = pb` is not allowed. Allowing it would give the program apparent access to functions and data of the derived class, but those fields do not exist. Assignments of this kind are called *downcasts*, because they go "down" the class hierarchy.

References work in the same way as pointers: whenever we have the address of an object (that is, a reference or a pointer to it), slicing does not occur and dynamic binding works.

For now, the thing to remember *upcast good, downcast bad.* We will discuss these casts in more detail later.

8.1.2 Constructors and destructors

Constructors and destructors work in a special and well-defined way with derived classes.

- When a constructor of a derived class is called, the constructor of the base class is invoked first, then the constructor of the derived class.
- When the destructor of a derived class is called, the destructor for the derived class is invoked first, then the destructor for the base class.

```
class Parent
{
public:
    Parent() { cout << "Construct Parent" << endl; }
    ~Parent() { cout << "Destroy Parent" << endl; }
};

class Child : public Parent
{
public:
    Child() { cout << "Construct Child" << endl; }
    ~Child() { cout << "Destroy Child" << endl; }
};

class GrandChild : public Child
{
public:
    GrandChild() { cout << "Construct GrandChild" << endl; }
    ~GrandChild() { cout << "Destroy GrandChild" << endl; }
};


int main()
{
    GrandChild g;
}
```

Figure 82: Constructors and destructors in inheritance hierarchies

This behaviour is illustrated by the code in Figure 82. When this program is executed, it prints:

```
Construct Parent
Construct Child
Construct GrandChild
Destroy GrandChild
Destroy Child
Destroy Parent
```

8.2 Designing a base class

Designing a class is difficult because you have to think about all the ways in which the class might be used. Designing a base class is harder still, because you have to think about the people who want to inherit from your class as well as the people who want to use it. The differences between a simple class and a potential base class are:

- `virtual` functions

- `protected` attributes

Virtual functions open up the possibility of redefinition in derived classes, leading to polymorphic behaviour. Protected attributes can be accessed by derived classes but not by outsiders. These two features combine to make inheritance useful and manageable.

8.2.1 `virtual` functions

Inheritance polymorphism, which means simply class-dependent behaviour, requires a particular set of circumstances. All of the following conditions must hold for polymorphism to take place:

- There must be a base class `B` containing a virtual function `f`.
- There must be a class `D`, derived from `B`, that redefines `f`.
- There must be an instance, `d`, of `D`.
- There must be a reference, `rd`, or a pointer, `pd`, to the object `d`. These might be defined as:

```
B & rd = d;
B * pd = &d;
```

- The function `f` must be invoked using either a reference or a pointer, as in these statements:

```
rd.f();
pd->f();
```

  Each of these statements will call the redefined function `D::f()`, even though `rd` (`pd`) is a reference (pointer) to the base type. This is polymorphism.

A virtual function may be *pure virtual*, meaning that it has no implementation. The body of a pure virtual function is written using the special notation `=0`. A class that has one or more pure virtual functions cannot be instantiated. It may have constructors, but these constructors can be called only by derived classes that provide definitions for the virtual functions.

There are essentially three different kinds of member function in a base class: normal functions, virtual functions, and pure virtual functions. The following base class has on example of each kind.

```
class Animal
{
public:
    string getName() { .... }
    virtual void move() { /* how to walk */ };
    virtual void communicate() = 0; // no implementation
    ....
};
```

From this declaration, we can infer the following intentions of the designer of the `Animal` class hierarchy. Assume that `pa` is a pointer declared as `Animal*` but actually pointing to an instance of a class derived from `Animal`.

- `getName` is a non-virtual function with an implementation. The idea in this example is that the mechanism for accessing the name of an animal is the same for all derived classes. A non-virtual function should not be redefined in a derived class for reasons explained in Section 8.2.2.

> ***Use a non-virtual function for behaviour that is common to all classes in the hierarchy and should not be overridden in derived classes.***

- Function `move` is virtual and has an implementation. In this example, `Animal::move` defines walking, which is a way of moving shared by many animals, and is a kind of *default behaviour*. A derived class has the option of either inheriting this method of waling or of redefining `move` in a special way for some other kind of movement. Calling `pa->move()` invokes movement that depends on the particular kind of animal.

> ***Use a virtual function to define default behaviour that may be overridden in derived classes.***

- Function `communicate` is a pure virtual function. In this example, the use of a pure virtual function suggests that there is no default behaviour for communication. It corresponds to the fact that animals communicate in many different and unrelated ways (using sound, gesture, smell, touch, etc.). Since class `Animal` has a pure virtual function, it cannot be instantiated: there are no objects of type `Animal`. This corresponds to the real world, where if you see an "animal", it is always a particular kind of animal: aardvark, beaver, cat, dog, elephant, giraffe, hedgehog, etc. Any class for which instances are needed must provide an implementation of `communicate`.

> ***Use a pure virtual function to require behaviour for which there is no reasonable default.***

There is one function that should *always* be virtual in a base class: it is the destructor. If a class is intended as a base, declare:

```
class Base
{
public:
    virtual ~Base() { .... }
    ....
};
```

```
class Base
{
public:
    ~Base() { cout << "Deleting Base" << endl; }
private:
    int x;
};

class Derived : public Base
{
public:
    ~Derived() { cout << "Deleting Derived" << endl; }
private:
    int y;
};

int main()
{
    Base * pb = new Derived;
    delete pb;
}
```

Figure 83: The danger of a non-virtual destructor

even if the body of the destructor is empty.

To see the importance of this, consider the code in Figure 83. Running this program produces the following output:

```
Deleting Base
```

We have constructed an instance of `Derived`, containing data members `x` and `y`, but the run-time system has deleted an instance of `Base`, with data member `x` only. The memory occupied by `y` will almost certainly *not* be deallocated — a memory leak!

If we change the declaration of class `Base` to

```
class Base
{
public:
    virtual ~Base() { cout << "Deleting Base" << endl; }
    ....
```

and run the program again, it displays

```
Deleting Derived
Deleting Base
```

By making `Base::~Base` virtual, we have ensured that `delete pb` calls `Derived::~Derived`. This destructor first executes its own code, then invokes the destructor for the base class, and finally deallocates the memory for all of the variables of the derived object.

A base class must have a virtual destructor.

8.2.2 Don't redefine non-virtual functions

C++ allows you to redefine non-virtual functions, as in this example:

```
class Animal
{
public:
    Animal() : name("Freddie") {}
    string getName() { return name; }
private:
    string name;
};

class Bison : public Animal
{
public:
    string getName() { return "Grrrrr"; }
};
```

Although it is possible, redefining non-virtual functions is not a good idea, for several reasons (Meyers 1992, Item 37). Suppose that we define some variables:

```
Bison b;
Animal *pa = &b;
Bison *pb = &b;
```

These definitions give us two pointers, `pa` and `pb`, pointing to the same object, `b`. It is therefore something of a surprise when we execute

```
cout << pa->getName() << endl;
cout << pb->getName() << endl;
```

and obtain

```
Freddie
Grrrrr
```

What has happened here is that, since `Animal::getName` is not `virtual`, the compiler uses the type of the pointer to choose the function. Thus `pa` causes `Animal::getName` to be called and `pb` causes `Bison::getName` to be called.

If `Animal::getName` was a `virtual` function, then the function called would depend on the type of the object, not the pointer. This is more natural behaviour: we expect a bison to answer `"Grrrrr"` whether we refer to it as an animal or as a bison (at least in this example).

More formally, overriding a non-virtual function is a violation of the is-a relationship. The purpose of a non-virtual function in a base class is to define behaviour that is invariant over the whole class hierarchy. In the example, to be an `Animal` (that is, to satisfy the predicate "is-a `Animal`"), you must respond to `getName` by returning your name. If you don't do that, you are not an `Animal` and do not belong in the `Animal` hierarchy. There are two possibilities:

- It is essential that bisons have a way of saying `"Grrrrr"`. In this case, the function that does this should not be called `getName`.
- It is correct modelling to say that bisons reply `"Grrrrr"` when asked their name. In this case, `Animal::getName` should be a virtual function.

8.2.3 `protected` attributes

Attributes in the `public` part of a class declaration are accessible to anyone who owns an instance of the object. Attributes in the `private` part of the declaration are accessible only to members and friends of the object.

A class declaration may contain a third part, introduced by the keyword `protected`, for attributes that are accessible to members and friends of classes derived from the class.

According to Stroustrup (1994, page 301), the keyword `protected` was introduced into C++ at the request of Mark Linton, who was writing Interviews, an extensible X-windows toolkit. Five years later, Linton banned the use of `protected` data members in Interviews because they had become "a source of bugs: 'novice users poking where they shouldn't have in ways they ought to have known better than'". Stroustrup goes on to say (1994, page 302):

> In my experience, there have always been alternatives to placing significant amounts of information in a common base class for derived classes to use directly. In fact, one of my concerns about `protected` is exactly that it makes it too easy to use a common base the way one might sloppily have used global data.
>
> Fortunately, you don't have to use protected data in C++; `private` is the default in classes and is usually the better choice. Note that none of these objections are significant for protected member *functions*. I still consider `protected` a fine way of specifying operations for use in derived classes.

So there you have it from the master: use `protected` data if you have to, but prefer using `protected` functions to access `private` data members of the base class.

```
class Account
{
public:
    Account();
    Account(std::string name, long id, long balance = 0);
    Account(const Account & other);
    virtual ~Account();
    Account & operator=(const Account & other);
    const std::string getName() const;
    long getID() const;
    long getBalance() const;
    virtual void deposit(long amount);
    virtual void withdraw(long amount);
    virtual void transfer(Account & other, long amount);
    friend bool operator==(const Account & left,
                           const Account & right);
    friend bool operator!=(const Account & left,
                           const Account & right);
    friend std::ostream & operator<<(std::ostream & os,
                                     const Account & acc);

protected:
    void setID(long newID);
    void setBalance(long newBalance);

private:
    std::string name;
    long id;
    long balance;
};
```

Figure 84: Class `Account` as a base class

8.2.4 A base class for bankers

As an example of base class design, we will take class `Account` from Section 7 (see page 111) and redesign it as a base class. Class `Account` is quite appropriate as a base class, because banks provide many kinds of account, and the various kinds are often represented as a hierarchy of classes in banking software. A number of design decisions are mentioned or implied in the following notes; few of them are cast in stone, and most might be made differently in specific circumstances.

Figure 84 shows the new version of class `Account`. The type of `Account::name` has been changed from `char*` to `std::string`, to avoid the memory management problems that we encountered in Section 7. Some of the functions, and the class `AccountException`, are not changed, and we do not repeat their descriptions here. The redesign takes into account the following considerations.

```
const string Account::getName() const
{
    return name;
}

long Account::getID() const
{
    return id;
}

long Account::getBalance() const
{
    return balance;
}
```

Figure 85: Accessors for class `Account`

- Constructors cannot be virtual, so we leave them unchanged.
- The destructor must be virtual, as explained in Section 8.2.1.
- The assignment, `operator=`, cannot be virtual, so we leave it unchanged.
- We provide some accessors (`getName`, `getID`, and `getBalance`) that were not present in the original class but are likely to be useful. They are `public` and non-virtual, since they access private data members and there should never be a need to redefine them. As explained above, these functions express *invariant properties* of the account class hierarchy: the meaning of `getBalance` is independent of the type of account we are dealing with. Figure 85 shows their definitions.
- The definitions of `deposit`, `withdraw`, and `transfer` are not changed. However, it is quite possible that a derived class might need to modify their behaviour; consequently, we declare them to be `virtual`.
- There are some functions that should be available to derived classes but not to everybody. These functions are declared in the `protected` section of the class declaration. This group consists of the new functions `setID` and `setBalance`, which have obvious definitions:

```
void Account::setID(long newID)
{
    id = newID;
}

void Account::setBalance(long newBalance)
{
    balance = newBalance;
}
```

8.3 Designing a derived class

Designing a derived class is usually easier than designing a base class because there are fewer options. The base class designer has decided what gets inherited and what can be redefined; the derived class designer has to decide:

- which functions to inherit without change
- which functions to redefine
- new data to introduce
- functions to operate on the new data

If there are no new data or functions, there is probably no need for a derived class at all. In a banking application, for example, it would be pointless to introduce new classes for different rates of interest: the interest rate is just a data member of an appropriate base class.

Some functions *cannot be* inherited and must be defined if the derived class needs them. The functions that are not inherited are:

- constructors
- the assignment operator (`operator=`)
- friends

Derived class constructors can, and usually should, call the base class constructors explicitly, as in this example:

```
class Base
{
public:
    Base(int n) { .... }
};

class Derived : public Base
{
public:
    Derived(int k, char c) : Base(k) { .... }
};
```

The *Liskov Substitution Principle* provides a helpful guideline for designing derived classes:

> ***Subclasses must be usable through the base class interface without the need for the user to know the difference.***

```
class SavingsAccount : public Account
{
public:
    SavingsAccount();
    SavingsAccount(std::string name, long id, long balance = 0,
        double rate = 0, long minimumBalance = 0);
    SavingsAccount(const SavingsAccount & other);
    SavingsAccount & operator=(const SavingsAccount & other);
    void withdraw(long amount);
    void addInterest();
    friend std::ostream & operator<<(std::ostream & os,
        const SavingsAccount & sacc);
private:
    double rate;
    long minimumBalance;
    long lowestInPeriod;
};
```

Figure 86: Class `SavingsAccount` derived from `Account`

8.3.1 A derived class for bankers

In this section, we develop a class `SavingsAccount` derived from the base class `Account`.[20] The important features of a savings account are:

- A savings account *is an* account. `SavingsAccount` provides all of the functionality of `Account`.
- `SavingsAccount` has a default constructor and a full constructor that allows all data for the account to be initialized.
- `SavingsAccount` provides a copy constructor and an assignment operator.
- A savings account collects interest. `SavingsAccount` has a data member for the interest rate and a member function for computing interest and adding the interest to the balance.
- Interest is paid only if the balance exceeds a specified minimum. `SavingsAccount` updates the minimum balance when money is withdrawn from the account.
- The insertion `operator<<` shows the interest rate and minimum balance for the `SavingsAccount` in addition to the information shown for a `Account`.

Figure 86 shows the class declaration for savings accounts. The class declaration contains exactly those functions that we wish to define or redefine.

The data members of `SavingsAccount` have the following roles:

[20]The savings account is designed to demonstrate features of object-oriented programming. Any resemblance to actual banking practice is entirely coincidental.

- `rate` is the interest rate for the account.
- `minimumBalance` is the minimum balance that the client must maintain over each period in order to earn interest.
- `lowestInPeriod` is the actual minimum balance during the period.

Since constructors are *not* inherited, we must define constructors for `SavingsAccount`. These constructors can, and should, invoke the base class constructors when necessary. The default constructor uses the default constructor of class `Account` and sets the new data members to zero.

```
SavingsAccount::SavingsAccount()
    : Account(), rate(0), minimumBalance(0), lowestInPeriod(0)
{ }
```

The full constructor allows the caller to specify all of the fields of an account with default values for the `balance`, `rate`, and `minimumBalance`. Like the default constructor, it calls the base class constructor and initializes the new data members separately.

```
SavingsAccount::SavingsAccount(string name, long id,
      long balance, double rate, long minimumBalance)
    : Account(name, id, balance), rate(rate),
      minimumBalance(minimumBalance), lowestInPeriod(balance)
{ }
```

The copy constructor must call the copy constructor for the base class and then initialize the new data members explicitly. Calling the copy constructor for `Account` with the `SavingsAccount other` is an example of upcasting.

```
SavingsAccount::SavingsAccount(const SavingsAccount & other)
:   Account(other),
    rate(other.rate),
    minimumBalance(other.minimumBalance),
    lowestInPeriod(other.lowestInPeriod)
{ }
```

The assignment operator is implemented by using `*this` (implicitly) to call the assignment operator of the base class, and then assigning the new data members explicitly.

```
SavingsAccount & SavingsAccount::operator=
                  (const SavingsAccount & other)
{
    Account::operator=(other);
    rate = other.rate;
    minimumBalance = other.minimumBalance;
    lowestInPeriod = other.lowestInPeriod;
    return *this;
}
```

We must update the value of `lowestInPeriod` whenever the balance might get smaller. This can happen only when money is withdrawn from the account. We override the definition of `Account::withdraw` with a function that updates `lowestInPeriod`.

```
void SavingsAccount::withdraw(long amount)
{
    Account::withdraw(amount);
    if (lowestInPeriod > getBalance())
        lowestInPeriod = getBalance();
}
```

The new function `addInterest`, which we assume is called periodically, checks that the client has the required minimum balance and, if so, adds interest. Since `balance` is private, `addInterest` must call the protected member function `Account::setBalance` to update the balance. This function also reinitializes `lowestInPeriod`.

```
void SavingsAccount::addInterest()
{
    if (lowestInPeriod >= minimumBalance)
    {
        long interest = static_cast<long>(rate * getBalance());
        setBalance(getBalance() + interest);
    }
    lowestInPeriod = getBalance();
}
```

The insertion operator calls `Account(sacc)` to upcast the `SavingsAccount` to an `Account` before inserting it into the stream. It then inserts the new data members into the stream.

```
std::ostream & operator<<(std::ostream & os,
                          const SavingsAccount & sacc)
{
    return os <<
        Account(sacc) <<
        fixed << setprecision(6) << setw(10) << sacc.rate <<
        setw(6) << sacc.minimumBalance;
}
```

If `sa1` and `sa2` are savings account objects, the expressions `sa1 == sa2` and `sa1 != sa2` compile and evaluate correctly. This is *not* because `operator==` and `operator!=` are inherited from `Account` but because the compiler can convert `sa1` and `sa2` to `Account`.

Figure 87 shows a short test program for accounts and savings accounts. When it is run, this program displays:

```
Account anne("Anne Bailey", 1234567, 1000);
cout << anne << endl;

Account & ar = anne;
cout << ar << endl;

SavingsAccount chas("Charles Daumier",
                    7654321, 1500, 0.01, 1000);
cout << "Before interest: " << chas << endl;
chas.addInterest();
cout << "After  interest: " << chas << endl;

Account *pa = & chas;
pa->transfer(anne, 1000);

cout << "Before interest: " << chas << endl;
chas.addInterest();
cout << "After  interest: " << chas << endl;
```

Figure 87: Testing the banker's hierarchy

```
Anne Bailey           1234567    1000
Anne Bailey           1234567    1000
Before interest: Charles Daumier    7654321   1500  0.010000  1000
After  interest: Charles Daumier    7654321   1515  0.010000  1000
Before interest: Charles Daumier    7654321    515  0.010000  1000
After  interest: Charles Daumier    7654321    515  0.010000  1000
```

The first two lines show that an account can be constructed and copied. The next two lines show that Charles earns $15 interest because his balance is greater than $1,000. After transferring $1,000 to Anne, his balance drops to $500 and does not earn interest, as shown in the last two lines.

Note that the transfer is performed using a reference to an `Account`, not a `SavingsAccount`. (This is realistic, because the object doing the transferring should not have to know the kind of accounts involved in the transfer.) Since `transfer` is not redefined in `SavingsAccount`, the function called is `Account::transfer`. When `transfer` called `withdraw`, it uses `this`, which is a pointer to a `SavingsAccount` object. Consequently, `SavingsAccount::withdraw` gets called, and updates the lowest balance.

8.3.2 Additional base class functions

After completing the base class `Account` and the derived class `SavingsAccount`, we notice that there is information that might be helpful to users: does a particular account pay interest? It is easy to add a function to `SavingsAccount`:

```
bool paysInterest() { return true; }
```

This is of little use, however, to the owner of a pointer to an `Account` that may or may not be a `SavingsAccount`. If `paysInterest` is to be useful, it must be declared `virtual` in the base class, `Account`:

```
virtual bool paysInterest() { return false; }
```

Thus, by default, an account class does not pay interest. Any class corresponding to an account that does pay interest must redefine `paysInterest` to return `true`.

This seems fairly innocuous: after all, the question 'does it pay interest?' can be asked of any account, and therefore the function `paysInterest` should be defined in the base class. The problem — trivial in this example — is that the concept of 'paying interest' is not otherwise mentioned in class `Account`.

In a hierarchy with many classes, there will be many functions of this kind. The base class will become cluttered with accessors that provide specialized information that is only relevant to particular derived classes. This can become a maintenance problem because, each time one of these functions is added, the base class is changed, requiring recompilation of the entire hierarchy. Unfortunately, there does not seem to be a clean solution to this problem.

8.4 Designing a template class

In this section, we discuss the development of a template class with features similar to an STL container. Koenig and Moo (2000, Chapter 11) provides a very similar presentation.

Our objective is a class `Vec` that behaves in more or less the same way as the STL class `vector`. We will provide memory management, but in a somewhat less sophisticated way than Koenig and Moo (2000). We will obtain the class declaration by filling in the blanks of this skeleton:

```
template <typename T>
class Vec
{
public:
    // interface
private:
    // hidden data
};
```

The data in the vector will be stored in a dynamic array of the template type, `T`. We need a pointer to the first element of the array and either: (a) the number of elements in the array, or (b) a pointer to one-past-the-last element of the array. Following STL conventions, we will adopt choice (b) and, of course, the user will see our pointers as iterators.

It would be possible to store exactly the number of elements that we need. This can be inefficient, however, if the user inserts elements into the array one at a time. Consequently, we provide a pointer to one-past-the-last element that is actually in use and another pointer to one-past-the-last element that is available for use. The three pointers are

1. `data` points to the first element
2. `avail` points to one-past-the-last element in use
3. `limit` points to one-past-the-last element of allocated memory

These three pointers define a *class invariant* for class `Vec`:

- `data` points to the first element
- `data` ≤ `avail` ≤ `limit`
- The range [`data`, `avail`) contains the allocated data
- The range [`avail`, `limit`) contains the allocated but uninitialized space

Figure 88 shows these pointers for an array in which 8 elements are in use and 13 elements are available altogether.

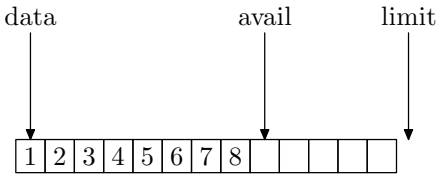


Figure 88: Pointers for class Vec

Following STL conventions, class `Vec` defines several types for its clients. These types hide, for example, the fact that `Vec` uses pointers to implement iterators. (We do not hide this fact because we are ashamed of using pointers, but rather to present a consistent abstraction to our clients.) The types we define are:

- `iterator` for the type of iterators
- `const_iterator` for the type of constant iterators
- `size_type` for the type of the size of a `Vec`
- `value_type` for the type of an element of a `Vec`

For `size_type`, we use `size_t` from namespace `std`.

We will provide three constructors: a default constructor; a constructor that specifies a (number of elements) size and an initial value for each element; and a copy constructor. The default constructor creates an empty vector. We declare the second constructor to be `explicit` to avoid accidental conversion. In the second constructor, the value of the element defaults to `T()`, which implies that the type `T` provided by the user must have a default constructor. Clearly, we will also need a destructor. Figure 89 shows the class declaration with the features we have incorporated so far.

```
template <typename T>
class Vec
{
public:
    typedef T* iterator;
    typedef const T* const_iterator;
    typedef size_t size_type;
    typedef T value_type;

    Vec();
    explicit Vec(size_t n, const T & val = T());
    Vec(const Vec & other);
    ~Vec();

    // rest of interface

private:
    iterator data;
    iterator avail;
    iterator limit;
};
```

Figure 89: Declaration for class `Vec`: version 1

Figure 90 shows implementations for the constructors and destructor. Because this is a template class, these declarations are in the header file `vec.h` after the class declaration. The second constructor is inefficient: for example, if the caller assumes the default value for the second parameter, the constructor `T()` will be called $2n$ times: n times for `new T[n]` and another n times in the `for` statement. Koenig and Moo (2000) show how to avoid this inefficiency by allocating uninitialized memory, but we will not bother with this improvement.

The copy constructor requires a function that returns the size (number of elements) of the `Vec` object:

```
size_type size() { return avail - data; }
```

Since the assignment operator is somewhat similar to the copy constructor, we consider it next. Although they are similar, the difference between assignment and initialization is significant and must not be overlooked. Before assigning to a variable, we must destroy its current value. Also, as we have seen previously, we must provide the correct behaviour for the self-assignment, `x = x`. The implementation of `operator=` shown in Figure 91 takes care of all this. The constructors and assignment operator introduce duplicated code into the implementation, but we can clean that up later.

It is straightforward to define the functions `begin` and `end` that provide clients with iterators pointing to the first and one-past-the-last elements. Two versions are required, one returning an `iterator` and the other returning a `const_iterator`.

```
template <typename T>
Vec<T>::Vec<T>() : data(0), avail(0), limit(0) {}

template <typename T>
Vec<T>::Vec<T>(size_t n, const T & val)
    : data(new T[n]), avail(data + n), limit(data + n)
{
    for (iterator p = data; p != avail; ++p)
        *p = val;
}

template <typename T>
Vec<T>::Vec<T>(const Vec<T> & other) :
    data(new T[other.size()]),
    avail(data + other.size()),
    limit(avail)
{
    const_iterator q = other.begin();
    for (iterator p = data; p != avail; ++p, ++q)
        *p = *q;
}

template <typename T>
Vec<T>::~Vec<T>()
{
    delete [] data;
}
```

Figure 90: Constructors and destructor for class `Vec`

```
iterator begin() { return data; }
const_iterator begin() const { return data; }

iterator end() { return avail; }
const_iterator end() const { return avail; }
```

These functions provide all that is needed for code such as this loop

```
for (Vec<int>::const_iterator it = v.begin(); it != v.end(); ++it)
    .... *it ....
```

because the operators `!=`, `++`, and `*` are all provided for pointers.

We enable the user to subscript `Vecs` by implementing `operator[]`. Two overloads of this operator are required:

```
T & operator[](size_type i) { return data[i]; }
const T & operator[](size_type i) const { return data[i]; }
```

```
template <typename T>
Vec<T> & Vec<T>::operator=(const Vec<T> & rhs)
{
    if (&rhs != this)
    {
        delete [] data;
        data = new T[rhs.size()];
        avail = data + rhs.size();
        limit = avail;
        const_iterator q = rhs.begin();
        for (iterator p = data; p != avail; ++p, ++q)
            *p = *q;
    }
    return *this;
}
```

Figure 91: Assignment operator for class `Vec`

The need for two versions arises as follows. We want a version of `operator[]` that allows us to use subscripted elements on the left of an assignment, as in `a[i] = e`. The first version of `operator[]` does this by returning a reference. But the compiler will not accept this function if it is applied to a `const Vec`, because the reference makes it possible to change the value of the object. Consequently, we need a second version that returns a `const T &` and promises not to change the object. Consider the following code:

```
Vec<int> v(5);
v[3] = 5;                    // (1)

const Vec<int> w(v);
n = w[2];                    // (2)
```

The statement labelled (1) fails if the first (non-`const`) version of `operator[]` is omitted, because it changes the value of `v`. The statement labelled (2) fails if the second (`const`) version of `operator[]` is omitted, because the compiler needs assurance that the call `w[2]` does not change the value of `w`.

Normally, we cannot provide two versions of a function with the same parameter list. In this case, the object (`*this`) is an implicit first parameter, and the overloads distinguish a `const Vec` and a non-`const Vec`.

As with similar STL functions, neither version of `operator[]` checks to see if the subscript is in range. Such a check could easily be added, perhaps as an assertion.

The next function that we provide is `push_back`, which appends a new value to the end of the array. There are two cases: if there is space already allocated, we store the new element at the position indicated by `avail` and then increment `avail`. In the other case, `avail = limit`, and we must allocate more space. We will introduce a private function, `grow`, to find more space. Figure 92 shows the definitions of both functions.

```
template<typename T>
void Vec<T>::push_back(const T & val)
{
    if (avail == limit)
        grow();
    *avail = val;
    ++avail;
}

template<typename T>
void Vec<T>::grow()
{
    size_type oldSize = avail - data;
    size_type newSpace = 2 * (limit - data);
    if (newSpace == 0)
        newSpace = 1;  // Ensure that we have at least one slot.
    iterator newData = new T[newSpace];
    iterator p = newData;
    for (const_iterator q = data; q != avail; ++q, ++p)
        *p = *q;
    delete [] data;
    data = newData;
    avail = data + oldSize;
    limit = data + newSpace;
}
```

Figure 92: Appending data to a `Vec`

The algorithm that we use for increasing the size of a `Vec` has important implications for efficiency. It is usually not possible simply to increase the size of a dynamic array, because the adjacent memory may already be allocated. Consequently, if we have and area of M bytes and we want to use N bytes, where $M < N$, we must: (1) allocate a new area of N bytes; (2) copy M bytes from the old area to the new area; and (3) delete the old area. This operation requires time $\mathcal{O}(M)$.

If we add one element, or in fact any *constant* number of elements each time a `Vec` grows, the performance of `push_back` will be quadratic. For example, if we add one element at each call of `grow`, we will copy 1, then 2, then 3 elements. To get 4 elements into the `Vec`, we will have to copy $1 + 2 + 3 = 6$ elements. To get N elements into the `Vec`, we will have to copy $\frac{1}{2}N(N-1)$ elements.

If, instead, we *multiply* the size of the `Vec` by a constant factor, the time spent copying falls to $\mathcal{O}(N \log N)$, which is much better. The implementation of `grow` in Figure 92 uses this technique with a constant factor of 2.

We can improve the clarity of the implementation of `Vec` by doing some simple refactoring. *Refactoring* means rearranging code to improve it maintainability or performance without changing its functionality. In this case, we introduce two overloaded versions of a private function called `create` to manage the creation of new `Vecs`. We can use this function to simplify the code for the constructors and the assignment operator. The two versions of `create` are declared in the

```
template<typename T>
void Vec<T>::create(size_type n = 0, const T & val = T())
{
    data = new T[n];
    avail = data + n;
    limit = avail;
    for (iterator p = data; p != avail; ++p)
        *p = val;
}

template<typename T>
void Vec<T>::create(const_iterator begin, const_iterator end)
{
    size_type size = end - begin;
    data = new T[size];
    avail = data + size;
    limit = avail;
    for (iterator p = data; p != avail; ++p, ++begin)
        *p = *begin;
}
```

Figure 93: Implementation of two overloads of `create`

`private` part of the class declaration:

```
void create(size_type n = 0, const T & val = T());
void create(const_iterator begin, const_iterator end);
```

The first version has two parameters, for the size of the array and the initial values, respectively. Both parameters have default values, so that calling `create()` yields an empty `Vec`. The second version also has two parameters that must be iterators defining a range of some other compatible `Vec`.

The function `create` is responsible for allocating memory and for initializing the pointers `data`, `avail`, and `limit`. Figure 93 shows the implementations of both versions. Figure 94 shows the revised definitions of functions that use `create`. Finally, Figure 95 shows the declaration of class `Vec` with all the changes that we have discussed.

8.5 Note on iterators

It might seem from the example of class `Vec` that an iterator is just a pointer and that we can use ++, ==, and so on, just because these functions are defined for pointers.

In fact, this is not always the case. The STL class `list`, for example, stores data in a linked list of nodes. An iterator value identifies a node. Incrementing the iterator means moving it to the next node. As Figure 96 shows, achieving this requires defining all of the iterator functions, including * and ->.

```
template <typename T>
Vec<T>::Vec<T>()
{
    create();
}

template <typename T>
Vec<T>::Vec<T>(size_t n, const T & val)
{
    create(n, val);
}

template <typename T>
Vec<T>::Vec<T>(const Vec<T> & other)
{
    create(other.begin(), other.end());
}

template <typename T>
Vec<T> & Vec<T>::operator=(const Vec<T> & rhs)
{
    if (&rhs != this)
    {
        delete [] data;
        create(rhs.begin(), rhs.end());
    }
    return *this;
}
```

Figure 94: Revised definitions of functions that use `create`

In particular, note that the prefix operators `--it` and `++it` are more efficient than the postfix operators `it--` and `it++`, because the postfix operators use the copy constructor to create the result.

```
template <typename T>
class Vec
{
public:
    typedef T* iterator;
    typedef const T* const_iterator;
    typedef size_t size_type;
    typedef T value_type;
    Vec();
    explicit Vec(size_t n, const T & val = T());
    Vec(const Vec & other);
    ~Vec();

    Vec & operator=(const Vec & rhs);

    T & operator[](size_type i) { return data[i]; }
    const T & operator[](size_type i) const { return data[i]; }

    size_type size() const { return avail - data; }

    iterator begin() { return data; }
    const_iterator begin() const { return data; }

    iterator end() { return avail; }
    const_iterator end() const { return avail; }

    void push_back(const T & val);

private:
    void create(size_type n = 0, const T & val = T());
    void create(const_iterator begin, const_iterator end);
    void grow();
    iterator data;
    iterator avail;
    iterator limit;
};
```

Figure 95: Declaration for class `Vec`: version 2

```
const_reference operator*() const
    {    // return designated value
    return (_Myval(_Ptr));
    }

_Ctptr operator->() const
    {    // return pointer to class object
    return (&**this);
    }

const_iterator& operator++()
    {    // preincrement
    _Ptr = _Nextnode(_Ptr);
    return (*this);
    }

const_iterator operator++(int)
    {    // postincrement
    const_iterator _Tmp = *this;
    ++*this;
    return (_Tmp);
    }

const_iterator& operator--()
    {    // predecrement
    _Ptr = _Prevnode(_Ptr);
    return (*this);
    }

const_iterator operator--(int)
    {    // postdecrement
    const_iterator _Tmp = *this;
    --*this;
    return (_Tmp);
    }

bool operator==(const const_iterator& _Right) const
    {    // test for iterator equality
    return (_Ptr == _Right._Ptr);
    }

bool operator!=(const const_iterator& _Right) const
    {    // test for iterator inequality
    return (!(*this == _Right));
    }
```

Figure 96: An extract from the STL class `list`

9 When things go wrong

In any non-trivial program, there will come a time when things go wrong. There are several ways of responding:

1. do nothing
2. report an error
3. set an error status flag
4. return a special value
5. trigger an assertion failure
6. throw an exception

The first four mechanisms are easy to implement and the fifth (triggering an assertion) was discussed in Section 2.7. The next section contrasts two approaches to coding, Section 9.2 discusses the mechanics of exception handling, and Section 9.3 evaluates the appropriateness of the mechanisms for various situations.

Error handling is an important topic. In large applications, as much as *half of all the code* may be devoted to checking for errors. Moreover, this code is hard to get right. There can be real errors, such as error reports getting lost or not being created in the first place, and inefficiencies caused by repeated, unnecessary checks.

9.1 Design by contract *versus* defensive coding

The function

```
Matrix rotation(const Vector & u, const Vector & v);
```

returns a `Matrix` corresponding to a rotation that takes vector `u` to vector `v`. In other words, the function returns a matrix M such that $M\mathbf{u} = \mathbf{v}$.

The matrix computed by this function is correct only if `u` and `v` are unit vectors: $|\mathbf{u}| = |\mathbf{v}| = 1$. The problem is what to do if `u` or `v` is not a unit vector.[21]

The first response is *do nothing*. We will assume that this is not an acceptable alternative because the consequences could be serious. (Perhaps the rotation will be used to rotate an aircraft into the correct orientation for landing.)

The second response is to specify that the vectors passed to the function must be unit vectors and that the behaviour of the function is undefined if they are not. The usual way to do this is to write an assertion or comment called a *precondition* in the code:

[21] One obvious possibility would be to normalize the vectors before using them. We will asssume that, for some reason, this is not a feasible solution for this application.

```
/**
 * Construct the matrix that rotates one vector to another.
 * \param u is a vector representing an initial orientation.
 * \param v is a vector representing the final orientation.
 * The matrix, applied to \a u, will yield \a v.
 * \pre The vectors \a u and \a v must be unit vectors.
 */
Matrix rotation(const Vector & u, const Vector & v);
```

The precondition passes the responsibility to the caller. Effectively, the comments define a contract between the caller and the function: if you give me unit vectors, I will give you a rotation matrix; if you give me anything else, I guarantee nothing. This form of coding is called *design by contract* or DBC; it was pioneered by Bertrand Meyer (1997) but has been recommended by other people as well (see, for example, (Hunt and Thomas 2000, pages 109–119)). The code for the function does not check the length of the vectors `u` and `v`.

The third response is to have the function inspect the vectors and take some action if they are not unit vectors:

```
Matrix rotation(const Vector & u, const Vector & v)
{
    if (fabs(u.length() - 1) > TOL || fabs(v.length() - 1) > TOL)
        // error handling
    else
        // compute rotation matrix
}
```

This approach is called *don't trust the user* or, more politely, *defensive coding*. We will discuss possible ways of handling errors later.

There are advocates and opposers of both DBC and defensive coding. There is perhaps a general trend away from defensive coding and towards DBC. For example, the home page for the Java Modeling Language[22] says that JML is a "design by contract" language for Java. Here are some arguments for and against each approach.

Overhead DBC has no run-time overhead. Defensive coding executes tests which almost always fail (e.g., most vectors passed to `rotation` are in fact unit vectors).

Safety DBC relies on programmers to respect contracts. Defensive coding ensures that malingerers will be caught.

Clarity DBC requires maintainers to read comments. With defensive coding, the checks are in the code.

Completeness There are situations that DBC cannot handle, such as the validity of user input. In these situations, we can — and must — code defensively.

DBC and defensive programming are not mutually exclusive: you can use both. The best policy is to decide which fits the needs of the particular application better and then use it.

[22] `http://www.cs.iastate.edu/~leavens/JML/`

9.2 Exceptions

We have already seen a simple example of the use of exceptions in Section 7.6. The exception handling mechanism of C++ provides one way of managing errors. It has two parts:

- The statement

  ```
  throw ⟨expression⟩
  ```

  raises, or throws, an exception. The exception is the object obtained by evaluating ⟨*expression*⟩.

- The statement

  ```
  try
  {
       ⟨try-sequence⟩
  }
  catch ( ⟨exc-type⟩ ⟨exc-name⟩ )
  {
       ⟨catch-sequence⟩
  }
  ```

  handles exceptions thrown in ⟨*try-sequence*⟩. More precisely:

  > If a statement in ⟨*try-sequence*⟩ throws an exception `e` whose class is ⟨*exc-type*⟩ or a class derived from ⟨*exc-type*⟩, then the statements ⟨*catch-sequence*⟩ are executed with ⟨*exc-name*⟩ bound to `e`.

There are several things to note about `try/catch` statements:

- The `throw` statement may not be visible in ⟨*try-sequence*⟩: it may be (and usually is) nested inside several levels of function call.
- The pair ⟨*exc-type*⟩ ⟨*exc-name*⟩ is analogous to the formal parameter of a function. It is as if the `catch` clause is "called" with the exception object as an argument.
- Although we mentioned the "class" of the exception object, basic objects such as `int`s and `char`s can be thrown as exceptions.
- There may be more than one `catch` clause. The run-time systems matches the exception against each `catch` clause in turn, executing the first one that fits.

  For example, suppose there is a class hierarchy `Dog` inherits `Carnivore` (meat-eater) inherits `Animal` and a `try` statement

  ```
  try { dangerous_stuff }
  catch (Dog d)  { .... }
  catch (Carnivore c) { .... }
  catch (Animal a) { .... }
  ```

then if `dangerous_stuff` throws a `Dog`, a `Carnivore`, or an `Animal`, the appropriate handler will catch it. But writing

```
try { dangerous_stuff }
catch (Animal a) { .... }
catch (Dog d)  { .... }
catch (Carnivore c) { .... }
```

is pointless because `Dog`s and `Carnivore`s will never be caught.

- The classes of multiple `catch` clauses do not have to be related by inheritance.
- The clause `catch(...)` (that is, you actually write three dots between parentheses) catches all exceptions. If you use this in a multiple `catch` sequence, it should be the last `catch` clause.
- Any C++ program behaves *as if* it was written like this:

```
try
{
    main();
}
catch (...)
{
    fail();
}
```

so that unhandled exceptions terminate the program, usually with some more or less helpful error message.

- If a `catch`-block discovers that it cannot handle the exception, it can execute `throw` (with no argument) to pass the same exception to the next level.
- You can throw exceptions by value, or you can create an exception dynamically (using `new`) and throw a pointer to it. If a pointer is thrown, the handler must delete the exception.
- A function can declare the exceptions it throws, like this:

```
void f1() throw();       // doesn't throw anything
void f2() throw(T1);     // may through a T1
void f3() throw(T2, T3);// may throw a T2 or a T3
void f4();               // may throw anything
```

It is useful to have an approximate understanding of exceptions. They work roughly as follows:

- When the run-time system reaches `try`, it pushes an *exception frame* onto the stack.[23] This frame contains descriptors for each `catch` clause.

[23] An implementation may use a separate stack for exceptions. This stack will contain pointers to the main stack.

- If the `try`-block executes normally, the exception frame is popped from the stack.
- If code in the `try`-block throws an exception, there will generally be a number of stack frames on top of the exception frame. The run-time system moves down the stack, popping these frames and performing any clean-up actions required (e.g., calling destructors), until it reaches the exception frame. This is called *unwinding the stack*.
- The run-time system inspects the `catch` descriptors until it finds a match. It then passes the exception to the `catch` block and gives control to the `catch` block.
- When the `catch` block terminates, it passes control to the statement following the entire `try-catch` sequence.

9.3 Managing failure

We have now discussed all of the possible responses (page 161) to a run-time problem. The next step is to decide which one to use in a particular situation. We begin by characterizing the kind of problems we are considering.

- If the problem is caused by a *logical inconsistency in the program*, the best way to deal with it is to use DBC and/or assertions (or DBC with contracts expressed as assertions). It is not appropriate to use `if` statements, error message, and the like to deal with simple incorrectness.

  Using the example above, suppose that the programmer(s) ensure that the function `rotation` is always given unit vectors. If `rotation` receives a vector of the wrong length, there is a logical error in the program that should trigger an assertion failure.
- If the problem is something that can reasonable be expected to happen, the best solution is normal code. For example, an `if` statement to detect the condition followed by an appropriate response.

  Users cannot be relied upon to enter valid data. Any system component that is reading data from the keyboard should validate the input and complain immediately if anything is wrong. This does not require fancy coding techniques.
- The remaining are *serious*, *rare*, and *unavoidable*. The following subsections discuss suitable responses to problems of this kind.

9.3.1 Error messages

The easiest response to a problem is an error message, sent to `cout`, or (better) `cerr`, or, in a windowing environment, a dialog or message box.

The streams `cout` and `cerr` behave in essentially the same way. Both send messages to a stream that C++ calls *standard output*. he difference is that `cout` is buffered and `cerr` is not. When the programs crashes, it is more likely to have displayed `cerr` output than `cout` output.

If this method works for your application, use it. However, it is inappropriate for many applications:

- The software may be *embedded* in a car, pacemaker, or electric toaster. There is nowhere to send messages to and perhaps no one to respond to them anyway.
- The software is part of a system that must do its job as well as possible without giving up or complaining. For example, an internet router.
- The user is not someone who can handle the problem. For example, a web browser should not display error messages — although, of course, they sometimes do — because there isn't much typical users can do.

9.3.2 Error codes

An effective system used by many software packages, such as UNIX™ and OpenGL, is to set an error flag and continue in the best feasible way.

- A UNIX™ library function may set the global variable `errno`. Most library functions also do something else to indicate failure, such as returning `-1` instead of a `char`.
- Many OpenGL functions perform according to the following specification: either the function behaves as expected, or an error number is set and *there is no other effect*.

This system requires some discipline from the programmers using it. They should look at the error status from time to time to make sure that nothing has gone wrong. A common strategy is to ignore the error status except just after doing something that is likely to fail.

Error codes are ideal for an application in which the consequences of failures are usually not serious, such as a graphics library. They are not secure enough for safety-critical applications.

9.3.3 Special return values

A function can return a funny value to indicate that it has not completed its task successfully. We will call these values *special return values*. The classic example of this behaviour is the `C` function `getchar`: you would expect its type to be `char` or even `unsigned char`, but in fact it returns an `int` (thus causing much grief to `C` beginners). It does this so that it can report end-of-file by returning `-1`.

Special return values are easy to code and require no special language features. They have strong advocates.[24] They also have disadvantages:

- There may be no suitable value to return. This is not a problem for `exp` and `log`, whose values are always positive, but how does `tan` report an error (its range is all of the reals from $-\infty$ to ∞)?
- In some cases, it is not feasible to pack all of the information about errors and returned data into a single value. Then we have to add reference parameters to the function to enable it to return all of the information.

[24] Graduate student Ravi Rao drew my attention to an interesting article by Joel Spolsky that argues for special return values rather than exceptions: `http://joelonsoftware.com/items/2003/10/13.html`.

```
int ch;
while ( ch = getchar() >= 0 )
{
    // process characters
}
// end of file
```

Figure 97: The consequence of ambiguous return values

```
double t1 = h(x);
if (t1 == h_error_value)
    // handle h_error
else
{
    double t2 = g(t1);
    if (t2 == g_error_value)
        // handle g_error
    else
    {
        double y = f(t2);
        if (y == f_error_value)
            // handle f_error
        else
        {
            // carry on
        }
    }
}
```

Figure 98: Catching errors in `y = f(g(h(x)))`

- Using the return value for multiple purposes leads to awkward code. In particular, conditions with side-effects are often needed, as in the C idiom of Figure 97.
- Code can become verbose. Instead of writing `y = f(g(h(x)))`, we have to write something like the code in Figure 98.
- Programmers have a tendency not to check for special return values. This problem is notorious in UNIX™.
- The caller may not know how to handle the problem. There will be multiple levels of functions, all specially coded to return funny results until someone can deal with the situation appropriately.

9.3.4 Exceptions

Exceptions provide a form of communication between distant parts of a program. The problem that exceptions solve is: something has gone wrong but, at this level or in this context, there is no way to fix it. Examples:

- In the middle of complex fluid-flow calculations, the control system software of a nuclear reactor detects that a pressure is sending improbable data.
- A library function that has a simple, natural interface (e.g., `Matrix::invert`) but may nevertheless fail for some inputs (e.g., singular matrix).

The main advantage of exceptions is that they can transfer control over many levels without cluttering the code. Unfortunately, this is also their main disadvantage. The sequence

```
make_a_mess();
clean_up_mess();
```

looks perfectly fine until you realize that `make_a_mess` might throw an exception.

In more detail, the advantages of exceptions include:

- The thrower does not have to know where the catcher is.
- The catcher does not have to know where the thrower is.
- A catcher can be selective, by only catching exceptions of particular types.
- If no exceptions are thrown, the overhead of the exception handling system is essentially zero.
- The `try` and `throw` statements make it obvious in the code that a problem is being detected and handled.

But exceptions also have some disadvantages:

- A programmer who calls a function that might throw an exception, either directly or indirectly (i.e., at some deeply nested place) cannot rely on that function to return. (This is sometimes called “the invisible `goto` problem”.)
- Exception handling in a complex system can quickly become unmanageable.
- Unhandled exceptions cause the program to terminate unexpectedly.
- Exceptions violate the “one flow in, one flow out” rule.
- It may be difficult for maintenance programmers to match corresponding `try` and `throw` statements. (The first problem is “who threw this?” and the second problem is “where is this going to end up?”.)

The cure for the disadvantages of exceptions is *disciplined use*.

- Use exceptions only when there is no alternative that is better.
- Document exceptions wherever they might affect behaviour in significant ways.
- Learn how to write "exception-safe code".
- For a large project:
  - plan an exception strategy during the initial design phase
  - define and use conventions for throwing and handling exceptions
  - make use of *exception domains*, which are regions of the program from which no exception can escape

> ***An application should execute correctly in all normal situations with the exception handlers removed.***

9.4 Exception-safe Coding

Exception-safe coding is an important topic for C++ programmers, but a rather large one for this course. Herb Sutter (2005) devotes 44 pages to it. The basic rules are:

- Write each function to be *exception safe*, meaning that it works properly when the functions it calls raise exceptions.
- Write each function to be *exception neutral*, meaning that any exceptions that the function cannot safely handle are propagated to the caller.

As an example, the constructor of the class `Vec`, discussed in Section 8.4, is exception-safe. Here is the code again:

```
template <typename T>
Vec<T>::Vec<T>(size_t n, const T & val)
    : data(new T[n]), avail(data + n), limit(data + n)
{
    for (iterator p = data; p != avail; ++p)
        *p = val;
}
```

The only action that may cause an exception to be thrown is `new`, because the default allocator throws an exception when there is no memory available. The constructor does not handle this exception but, by default, passes it on to the caller. Therefore the constructor is *exception neutral*.

The remaining danger is that the constructor does not work properly when the exception is raised. For example, it might happen that the pointers are set even though no data has been allocated. This doesn't matter, because *a constructor that raises an exception is assumed to have failed completely*. There is no object, and so values of the attributes of the object are irrelevant.

```
template<class T>
T Stack<T>::pop()
{
    if (vused_ == 0)
    {
        throw "Popping empty stack";
    }
    else
    {
        T result = v_[used_ - 1];
        --vused_;
        return result;
    }
}
```

Figure 99: Popping a stack: 1

```
template<class T>
void Stack<T>::pop(T & result)
{
    if (vused_ == 0)
    {
        throw "Popping empty stack";
    }
    else
    {
        result = v_[used_ - 1];
        --vused_;
    }
}
```

Figure 100: Popping a stack: 2

So far so good. Now consider a case where there *is* a problem. The code in Figure 99 is supposed to implement "popping" a stack (Sutter 2005, page 34). The subtle problem with this code is that `return result` requires a copy operation that may fail and throw an exception. If it does, the state of the stack has been changed (`--vused_`) but the popped value has been lost. The code in Figure 100 avoids this problem:

The STL is exception-safe. In order to avoid problems like these, it provides *two* functions for popping a stack:

`void stack::pop()` removes an element from the stack

`const T & stack::top() const` returns the top element of the stack

Another rule that is provided and assumed by the STL is: *destructors do not throw exceptions.*

10 System Design

For large systems, design must take into account not only the performance of the finished product, but also its development and maintenance. Three anecdotes (the first two from John Lakos's (1996) experiences) illustrate the problems of large-scale development.

1. A maintenance programmer needed to find the meaning of the undocumented declaration

   ```
   TargetID id;
   ```

   After a quick search failed, he tried using the UNIX™ utility `grep` to search for the string `"TargetID"`. There were several thousand header files involved, and `grep` complained "too many files". The programmer wrote a script to search through `a*.h`, then `b*.h`, and so on. This search eventually revealed

   ```
   typedef int TargetID;
   ```

2. Lakos worked at Mentor Graphics, a company that developed one of the first general-purpose graphics libraries for UNIX™ systems. Programmers were in the habit of `#include`ing all of the headers that they thought their component might need. The result was that compiling the library required more than a week — using a network of workstations.

3. A major software product was completed, the source code and executables were written to CDs, and the product was ready to be shipped to the customer. A couple of days later, a programmer made a small change to a member function, something like this:

   ```
   class Widget
   {
   public:
      double get()
      {
          ++accessCount;              // this line added
          return size;
      }
   private:
      static double size;
      unsigned long accessCount;
   };
   ```

   After this change, the program crashed during most runs.

   The problem was solved after several days of debugging. The following code was the culprit:

   ```
   Widget *pw;
   // .... several pages of code
   s = pw->get();
   ```

10.1 Logical and physical design

Object oriented *logical design* concerns the organization of classes from the point of view of functionality, inheritance, layering, and so on. It is discussed in many books on C++ programming and software engineering, often with the use of UML diagrams, patterns, and other aids.

Physical design concerns the distribution of the various classes and other entities into files, directories, and libraries. Physical design is important for large projects but is less often discussed in books (Lakos (1996) being a significant exception).

Logical and physical design are closely related. If the logical design is bad, it will not usually be possible to obtain a good physical design. But a good logical design can be spoilt by a poor physical design.

Logically, a system consists of classes (including enumerations) and free functions, with classes providing the main structure.

Physically, a system consists of *components*. There are various ways of defining components but, fortunately, there is one precise definition which is suitable for systems written mostly or entirely in C++.

> ***A component consist of a header (`.h`) and an implementation (`.cpp`) file.***

The header and implementation files may both use `#include` to read other files. However, these other files are not considered to be part of the component.

The compiler turns each component into an object file (`.obj`) or sometimes a library file (`.lib` or `.dll`). The linker takes all the object and library files and creates an `executable`. Components are also called *compilation units* (see Section 1.2.1) and *translation units*.

Since C++ does not allow a class declaration to be split over several files, the definition of component implies that *every class belongs to exactly one component*. However, the definition does allow a component to contain more than one class and, in fact, it is often useful to define components with several classes:

- Several closely-related classes may be declared in a single header file.
- A class, or classes, needed by a single component can be declared in the implementation file of that component.

In general, the *logical structure* of a design is determined by its classes; the *physical structure* of a design is determined by its components.

There are various kinds of logical association between two classes: a class can contain an instance of another class or a pointer to such an instance; inherit another class; have a value of another class passed to it by value, reference, or as a pointer; and so on. Physical associations are, in general, simpler: one component either uses another component or it doesn't.

A *client* of component *C* is another component of the system that uses *C*. With a few exceptions (see Figure 10.4.2), clients can be recognized because they `#include` header files: component `C1` is a client of `C2` if either `C1.h` or `C1.cpp` needs to read `C2.h`. Note that:

- "Reading" `C2.h` is not the same thing as `#include`ing `C2.h` because a client might `#include X.h` which `#includes C2.h`. In Figure 101, compiling `C1.cpp` requires reading `C2.h`:
- The client relationship is transitive: if `C1` is a client of `C2`, and `C2` is a client of `C3`, then `C1` is a client of `C3`.

```
#ifndef C1_H                    #ifndef X_H
#define C1_H                    #define X_H

#include "X.h"                  #include "C2.h"
....                            ....
#endif                          #endif
```

Figure 101: Physical dependency: `c1.h` "reads" `c2.h`

10.2 Linkage

We discussed linkage issues previously (see Section 4.3). To get good physical structure, we want to minimize dependencies between components. Doing this requires knowing how dependencies arise.

Declarations in a header or implementation file have no effect outside the file (except, of course, that declarations in a header file are read in the corresponding implementation file).

Definitions in a header or implementation file cause information to be written to the object file, and therefore create dependencies. As we have seen, definitions should be avoided altogether in header files.

General rules:

- Put only constant, enumeration, and class declarations into header files.
- Put all data and function definitions into implementation files.
- Put data inside classes wherever possible.
- If you have to put data at file scope, declare it as `static`.

The compiler considers inlined functions to be declarations, not definitions, even if they have bodies. It is therefore acceptable to put, for example,

```
inline double sqr(double x) { return x * x; }
```

in a header file. However, if you remove the `inline` qualifier, the program may not compile! This is because the header file may be read more than once, putting two copies of the function `sqr` into the program, which causes problems for the linker.

10.3 Cohesion

Every book on Software Engineering includes the slogan *low coupling, high cohesion*. This mantra is useful only if we know:

- what is coupling?
- what is cohesion?
- how do we reduce coupling?
- how do we increase cohesion?

Coupling is probably the more important of these two aspects of a system, and we discuss it in Section 10.4.

Cohesion is the less important partner of the "low coupling, high cohesion" slogan. Roughly speaking, a component is *cohesive* if it is independent of other components (as far as possible) and performs a single, well-defined role, or perhaps a small number of closely related tasks. A component is not cohesive if its role is hard to define or if it performs a variety of loosely related tasks.

One way to define cohesion is by saying that the capabilities of a class should be necessary and sufficient (like a condition in logic). The capabilities of a component are *necessary* if the system cannot manage without it. The capabilities of a class are *sufficient* if they jointly perform all the services that the client requires.

Necessary = every capability provided is in fact required by the system.
Sufficient = every capability required by the system is in fact provided.

The "necessary and sufficient" criterion is not the whole story. For example, we could write the entire system as one gigantic class that provided the required functionality and had no superfluous functions; clearly, this class would be both necessary and sufficient, but it would also be unmanageable and probably useless.

Sufficiency is a precise concept, but there is some leeway. For example, if it turns out that the sequence

```
p->f();
p->g();
```

occurs often in system code — perhaps, in fact, *every* invocation of `f` is followed by a call to `g` — then it probably makes sense to add a function `h` that combines `f` and `g` to the class. The old class was technically sufficient, but the new function improves clarity, simplifies maintenance, and may even make the program run a little faster.

Clearly there is no point in taking the time to code and test functions that will never be used. Nevertheless, there are programmers who like to spend large amounts of time writing code that "might be useful one day" rather than focusing on writing or improving code that was actually needed yesterday.

Consequently, cohesion also implies a division of the system into components of manageable size. A good guideline is that it should be possible to describe the role of a component with a single sentence. If you ask what a component does and the answer is either a mumble or a 10 minute peroration, there is probably something wrong with the design of the component.

Cohesion is related to coupling in the following way: a component that tries to do too much (that is, plays several roles) is likely to have many clients (perhaps one or more for each role). It may also need to be a client of several other components in order to perform its roles. Consequently, its coupling is likely to be high. On the other hand, a cohesive component is likely to have few clients and few dependencies, and therefore lower coupling.

There are always exceptions.

- A highly-cohesive component might provide an essential service to many parts of a system; in this case it would be contribute to heavy coupling.
- Low cohesion in the design can sometimes be corrected by careful physical design. For example, a group of related classes could be put into a single component, exposing on some class interfaces to the rest of the system.

10.4 Coupling

Coupling is any form of dependency between components.

Coupling is not an all-or-nothing phenomenon: there are *degrees* of coupling. At one end of the scale, if a component has no coupling with the rest of the system, it cannot be doing anything useful and should be thrown away. It follows that *some* coupling is essential and therefore that the issue is *reducing* coupling, not eliminating it.

The other end of the scale is very high coupling: every component of the system is strongly coupled to every other component. Such a system will be very hard to maintain, because a change to one component often requires changes to other components.

10.4.1 Encapsulation

The first step towards low coupling is encapsulation, which means hiding the implementation details of an object and exposing only a well-defined public interface. Figure 102 shows is a first (rather feeble) attempt at defining a class for points with two coordinates.

Class `Point1` provides very poor encapsulation: anyone can get and set its coordinates, and it has no control over its state at all. Obviously, we should make the coordinate data `private`, but then we would have to provide some access functions, as shown in Figure 103.

Access functions that provide no checking, like these, are not much better than `public` data. But access functions to at least provide the *possibility* of controlling state and maintaining class invariants. For example, `setX` might be redefined as

```
class Point1
{
public:
    double x;
    double y;
};
```

Figure 102: A class for points: version 1

```
class Point2
{
public:
    double getX() { return x; }
    double getY() { return y; }
    void setX(double nx) { x = nx; }
    void setY(double ny) { y = ny; }
private:
    double x;
    double y;
};
```

Figure 103: A class for points: version 2

```
void Point2::setX(double nx)
{
    if (nx < X_MIN) nx = X_MIN;
    if (nx > X_MAX) nx = X_MAX;
    x = nx;
}
```

Access functions provide a way of hiding the representation of an object. For example, Figure 104 shows how we could define points using complex numbers without affecting users in any way (except, perhaps, efficiency).

Even with access functions, however, `Point` hardly qualifies as an object. Why do users want points? What are the operations that points should provide? A class for points should look more like Figure 105, in which function declarations appear in the class declaration, but function definitions are in a separate implementation file.

`Point4` has less coupling than `Point3` in another respect. If any of the inline functions of `Point3` are changed, its clients will have to be recompiled. Since the functions of `Point4` are defined in `Point4.cpp` rather than `Point4.h`, they can be changed without affecting clients (although the system will have to be re-linked, of course).

A client of one of the `Point`*n* classes will be coupled to the `Point`*n* component. The degree of coupling depends on the `Point` class: it is highest for `Point1` (the user has full access to the coordinates) and lowest for `Point4` (the user can manipulate points only through functions such as `move` and `draw`).

```
class Point3
{
public:
    double getX() { return z.re; }
    double getY() { return z.im; }
    void setX(double nx) { z.re = nx; }
    void setY(double ny) { z.im = ny; }
private:
    complex<double> z;
};
```

Figure 104: A class for points: version 3

```
class Point4
{
public:
    void draw();
    void move(double dx, double dy);
    ....
private:
    // Hidden representation
};
```

Figure 105: A class for points: version 4

From the point of view of the owner of the `Point` class, lower coupling means greater freedom. If the owner of `Point1` makes almost any change at all, all clients will be affected. The owner of `Point4`, on the other hand, can change the representation of a point, or the definitions of the member functions, without clients needing to know, provided only that the class continues to meet its specification. This is one advantage of low coupling:

Low coupling makes maintenance easier.

A complicated object is entitled to have a few access functions but, in general, a class should *keep a secret.*[25] If your class seems to need a lot of `get` and `set` functions, you should seriously consider redesigning it.

Here are formal definitions (Lakos 1996, pages 105 and 138):

1. The *logical interface* of a component is that which programmatically accessible or detectable by a client.

2. An implementation entity (type, variable, function, etc.) that is not accessible or detectable programmatically through the logical interface of a component is *encapsulated* by that component.

[25]The useful metaphor "keeping a secret" was introduced by David Parnas in a very influential paper (1978).

In set theoretic notation, for any component C

$$I_C \cup E_C = U \qquad \text{and} \qquad I_C \cap E_C = \emptyset$$

where I_C is the set of implementation entities in the logical interface of C, E_C is the set of entities encapsulated by C, and U is the "universe" of all entities in C.

10.4.2 Hidden coupling

C++ provides various ways in which coupling between components can be hidden: that is, there is a dependency between two components even though neither `#includes` the other's header file. For example, the following components `A` and `B` are coupled:

```
// file A.cpp                    // file B.cpp

int numWidgets;                  extern int numWidgets;

....                             ....
```

Using the definitions of the previous section, we note that `numWidgets` is in the logical interface of component `A`, because it can be accessed by component `B` by using `extern`.

The object file obtained when `A.cpp` is compiled will allocate memory for `numWidgets`. The object file obtained when `B.cpp` is compiled will not allocate memory for `numWidgets` but will expect the linker to provide an address for `numWidgets`. If the definition in `A.cpp` is changed or removed, both components will compile, but the system won't link.

Don't use global variables.

Don't use `extern` declarations.

10.4.3 Compilation dependencies

Coupling can affect the time needed to rebuild a system. Build times are significant for large projects, and it is useful to know how to keep them low. If we define class `Point5` as in Figure 106, `sizeof(Point5)` gives 16 bytes, showing that the compiler allocates two 8-byte `double` values to store a `Point5`. If we decide to add a new data member, `d`, to store the distance of the point from the origin, we obtain `Point6`, shown in Figure 107. Then `sizeof(Point6)` gives 24 bytes, showing that the compiler allocates three 8-byte `double` values to store a `Point6`.

At this point, we might decide that our class needs a function, as shown in Figure 108. Adding a function has *no effect* on the size of the class: `sizeof(Point7)` is 24 bytes, just like `Point6`.

But if we make the function `virtual`, as in Figure 109, then `sizeof(Point8)` gives 32 bytes, because a virtual function requires a *virtual function table* or *vtable* for short.[26]

[26]The pointer to the vtable needs 4 bytes on the Intel architecture. The additional 4 bytes may be added for alignment purposes.

```
class Point5
{
private:
    double x;
    double y;
};
```

Figure 106: A class for points: version 5

```
class Point6
{
private:
    double x;
    double y;
    double d;
};
```

Figure 107: A class for points: version 6

```
class Point7
{
public:
    void move(double dx, double dy) { x += dx; y += dy; }
private:
    double x;
    double y;
    double d;
};
```

Figure 108: A class for points: version 7

```
class Point8
{
public:
    virtual void move(double dx, double dy) { x += dx; y += dy; }
private:
    double x;
    double y;
    double d;
};
```

Figure 109: A class for points: version 8

```
#include "point.h"

int main()
{
    Point p;
    ....
}
```

Figure 110: Forward declarations: 1

```
#include "point.h"

int main()
{
    Point* pp;
    ....
    void f(Point & p);
    ....
}
```

Figure 111: Forward declarations: 2

In summary, adding or removing data members changes the size of the objects. Adding a virtual function also changes the size. (Adding virtual functions after the first makes no difference, because there is only one vtable and it contains the addresses of all virtual functions.)

The changes we have been discussing should not affect users of the class. Private data members cannot be accessed anyway, and whether a function is virtual affects only derived classes. Nevertheless, if we change the size of a `point`, *every component that includes* `point.h` *will be recompiled during the next build!* Since building a large project can take hours or even days, changing a class declaration, even the `private` part, is something best avoided.

To see why this happens, consider the first program in Figure 110. To allocate stack space for `p`, the compiler must be able to compute `sizeof(Point)`. To compute this, it must see the declaration of class `Point`. Finally, to see the declaration, it must read `"point.h"`. If `"point.h"` changes, the program must be recompiled.

The program in Figure 111 is slightly different. The compiler does *not* need to know `sizeof(Point)` to compile this program, because all pointers and references have the same size (the size of an address: usually 4 bytes). Consequently, reading the declaration of class `Point` is a waste of time. However, the compiler *does* need to know that `Point` is a class and, for this purpose, the forward declaration of Figure 112 is sufficient.

Don't include a header file when a forward declaration is all you need.

In early versions of the standard libraries, names such as `ostream` referred to actual classes. Consequently, the code shown in Figure 113 worked well in a header file. Times have changed, and

```
class Point; // forward declaration

int main()
{
    Point* pp;
    ....
    void f(Point & p);
    ....
}
```

Figure 112: Forward declarations: 3

```
class ostream;
....
friend ostream & operator<<(ostream & os, const Widget & w);
```

Figure 113: Forward declaration for `ostream`: 1

`ostream` is now defined by `typedef`. However, the standard library defines a header file that contains forward declarations for all stream classes: see Figure 114.

Suppose that points have colours. It is tempting to put the enumeration declaration outside the class declaration:

```
enum Colour { RED, GREEN, BLUE };

class Point
{
    ....
```

This is convenient, because we can use the identifiers RED, GREEN, and BLUE wherever we like. But this convenience is also a serious drawback: the global namespace is *polluted* by the presence of these new names. It is much better to put the enumeration *inside* the class declaration, like this:

```
class Point
{
public:
    enum Colour { RED, GEEN, BLUE };
    ....
```

```
#include <iosfwd>
....
friend ostream & operator<<(ostream & os, const Widget & w);
```

Figure 114: Forward declaration for `ostream`: 2

We now have to write `Point::RED` instead of RED, but there is no longer any danger of the colour names clashing with colour names introduced by someone else.

In situations where class declarations alone are inadequate, for example in the development of a library, we can use namespaces instead.

10.4.4 Cyclic Dependencies

The worst kind of dependencies are *cyclic dependencies*. When a system has cyclic dependencies, "nothing works until everything works".[27] Some cyclic dependencies are unavoidable, but many can be eliminated by careful design.

Suppose we have classes `Foo` and `Bar` that are similar enough to be compared. We might write

```
class Foo
{
public:
    operator==(const Bar * b);
    ....
};

class Bar
{
public:
    operator==(const Foo & f);
    ....
};
```

This creates a cyclic dependency between `Foo` and `Bar`. The dependency can be eliminated by using `friend` functions:

```
class Foo
{
    friend operator==(const Foo & f, const Bar & b);
    friend operator==(const Bar & b, const Foo & f);
    ....
};

class Bar
{
    friend operator==(const Foo & f, const Bar & b);
    friend operator==(const Bar & b, const Foo & f);
    ....
};

operator==(const Foo & f, const Bar & b) { .... }
operator==(const Bar & b, const Foo & f) { .... }
```

[27] I'm not sure who said this first, but it might have been David Parnas.

10.4.5 Pimpl

A complicated class might have quite a large `private` part. Every time the owner of the class changes the class declaration, every client gets recompiled. A "pimpl" is a simple way of avoiding this dependence: it is a mnemonic for *pointer to implementation*. Thus

```
class Widget
{
public:
    ....
private:
    int something;
    char somethingElse;
    double yetAnotherThing;
    ....
};
```

becomes

```
class Widget
{
public:
    ....
private:
    class WidgetImplementation;
    WidgetImplementation * pimpl;
};
```

There is naturally a price to pay for the increased indirection: the functions of `Widget` are reduced to messengers that just forward their requests to `WidgetImplementation`:

```
class Widget
{
public:
    Widget() { pimpl = new WidgetImplementation(); }
    ~Widget() { delete pimpl; }
    int getSomething() { return pimpl->getSomething(); }
    void doSomething(char what) { pimpl->doSomething(what); }
    ....
private:
    WidgetImplementation * pimpl;
};
```

The overhead of the extra calls will be small, possibly even zero. For a large system, however, the gains may be considerable. The owner of `WidgetImplementation` can change data and functions freely without forcing recompilation of `Widget` clients. In fact, `Widget` clients will be recompiled only if the declaration of class `Widget` itself has to be changed, perhaps by adding a new function. Herb Sutter (2000, page 110) says:

> I have worked on projects in which converting just a few widely-used classes to us Pimples has halved the system's build time.

Another name for pimpl is "handle/body class". In the example, above, `Widget` is the handle and `WidgetImplementation` is the body. Koenig and Moo (2000) use this term: Section 13.4 (pp. 243–245) gives a definition of class `Student_info` in which the only data member is `Core *cp`. The pointer `cp` points an instance of either the base class `Core` or the derived class `Grad`, illustrating a further advantage of pimpl: a class may provide behaviour that depends on the way in which it is initialized.

10.4.6 Inlining

Normally, a function is called by evaluating its arguments, placing them in registers or on the stack, storing a return link in the stack, and passing control to the function's code. When the function returns, it uses the return link to transfer control back to the call site. There is clearly a fair amount of overhead, especially if the function is doing something trivial, such as returning the value of a data member of a class.

If the compiler has access to the definition of a function, it can compile the body of the function directly, without the call and return. This is called *inlining* the function.

An inline function will usually be faster than a called function, although the difference will be significant only for small functions. (For large functions, especially if they have loops or recursion, the time taken by calling and returning will be negligible compared to the time taken to execute the function.)

Heavy use of inlined functions increases the size of the code, because the code of the function is compiled many times rather than once only. Thus inlining is an example of a *time/space tradeoff*.

Inlining is relevant to this section because *inlining affects coupling.* The compiler can inline a function only if its definition is visible. Within an implementation file, a definition of the form

```
inline void f() { .... }
```

permits the compiler to inline `f`, but only in that particular implementation file. To inline a function throughout the system, the `inline` qualifier, and the body of the function, must appear in a header file. As we have seen, this implies that any change to the function body will force recompilation of all clients.

Whenever a function definition appears inside a class declaration, the compiler is allowed to compile it inline. In Figure 115, the compiler may inline `f1` and `f3` and it cannot inline `f2` and `f4`.

The qualifier `inline` is a *compiler hint*, not a directive. The compiler is not obliged to inline functions that are defined inside the class declaration or declared `inline`, and it may inline functions of its own accord.

Do not place too much dependence on inlining: the rewards are not great and there may be drawbacks. Herb Sutter (2002, pages 83–86) argues that good compilers know when and when not to inline and programmers should leave the choice to them.

```
#ifndef WIDGET_H
#define WIDGET_H

class Widget
{
public:
    int f1() { return counter; }
    int f2();
private:
    int counter;
};

inline double f3()
{
    ....
}

char f4();
```

Figure 115: Inlining

10.5 Improving system structure

There are several techniques for developing well-structured systems or improving systems with structural weaknesses.

10.5.1 CRC Cards

CRC cards where introduced by Kent Beck and Ward Cunningham at OOPSLA (1989) for teaching programmers the object-oriented paradigm. A CRC card is an index card that is use to represent the responsibilities of classes and the interaction between the classes. The cards are created through scenarios, based on the system requirements, that model the behavior of the system. The name CRC stands for *Class*, *Responsibilities*, and *Collaborators*.

Although CRC cards are old and have largely been replaced by heavy-weight methods such as UML, the underlying ideas are still useful in the early stages of system design. The focus on responsibility and collaboration is appropriate and helps to improve the coupling/cohesion ratio.

10.5.2 Refactoring

When we change a system to improve its maintainability, performance, or other characteristics *without changing its functionality*, we are *refactoring* it. A common reason for refactoring is to increase cohesion and decrease coupling. Refactoring is a large topic and entire books have been written about it.

10.5.3 DRY

DRY stands for *don't repeat yourself*. As systems grow, it is easy for them to accumulate multiple implementations of a single function. These functions might vary in small details, such as the number and order of parameters, but they do essentially the same thing. Weed out such repetition by converting all the variations into a single function whenever you can. As Hunt and Thomas (2000, page 27) say:[28]

> ***Every piece of knowledge must have a unique, unambiguous, authoritative representation within a system.***

10.5.4 YAGNI

YAGNI stands for *you aren't going to need it* and it is a slogan associated with agile methods. Some programmers are tempted to provide all kinds of features that might come in useful one day. When writing a class, for instance, they will include a lot of methods but they don't actually need right now for the application, but look nice. This is usually a waste of time: there is a good chance that the functions will not in fact be needed and, if they are, it does not take long to write them.

10.6 Using Patterns

Patterns (Gamma, Helm, Johnson, and Vlissides 1995)[29] belong more to high-level design than to physical organization. However, interesting issues arise in mapping patterns to concrete C++ code, and this section describes some of these issues.

10.6.1 Singleton

GoF summarizes the Singleton pattern as: *ensure a class has only one instance, and provide a global point of access to it.*Figure 116 shows a declaration for class `Singleton` based on (Alexandrescu 2001, pages 129–133). Here are some points of interest in this declaration.

- The function `fun` and the data member `uid` are included just to illustrate that the singleton can do something besides merely exist.
- The only way that a user can access the singleton is through the static reference function `instance`.
- Access to the unique instance of the singleton is through `private`, `static` pointer, `pInstance`.
- The constructor is `private`.
- The copy constructor and assignment operator are `private` and unimplemented. This ensures that the singleton cannot be passed by value, assigned, or otherwise copied.

[28] I changed their word "single" to "unique" for greater emphasis.

[29] The four authors of this book are often referred to as "the Gang of Four". Like other authors, we will refer to the book as "GoF".

- The destructor is also `private` and unimplemented. This ensures that the singleton cannot be accidentally deleted. (This could be considered as a memory leak, but that is harmless because there is only one instance anyway.)

```
class Singleton
{
public:
    static Singleton & instance();
    void fun();
private:
    int uid;
    static Singleton * pInstance;
    Singleton();
    Singleton(const Singleton &);
    Singleton & operator=(const Singleton &);
    ~Singleton();
};
```

Figure 116: Declaration for class `Singleton`

Figure 117 shows the implementation of class `Singleton`. The constructor is conventional; the body given here merely demonstrates that it works. The function `instance` constructs the singleton when it is called for the first time, and subsequently just returns a reference to it.

```
Singleton::Singleton()
{
    uid = 77;
    cout << "Constructed Singleton " << uid << endl;
}

Singleton & Singleton::instance()
{
    if (!pInstance)
        pInstance = new Singleton;
    return *pInstance;
}

void Singleton::fun()
{
    cout << "I am Singleton " << uid << endl;
}

Singleton * Singleton::pInstance = 0;
```

Figure 117: Definitions for class `Singleton`

The following code demonstrates the singleton in action. A user must access the singleton as `Singleton::instance()`. Any attempt to copy it or pass it by value causes a compile-time error. For the particular singleton defined here, all the user can do is apply the function `fun` to the singleton.

```
int main()
{
    Singleton::instance().fun();
    Singleton::instance().fun();
    Singleton::instance().fun();
    return 0;
}
```

This is in fact a rather simple version of the Singleton pattern and it is not robust enough for all applications. For details about its improvement, see (Alexandrescu 2001, pages 133–156)

10.6.2 Composite

The composite pattern is used with part-whole hierarchies. A simple hierarchy allows the user to work with one part at a time. The composite pattern allows the user to work also with groups of parts as if they were single parts.

The example is a highly simplified typesetting application. The abstract base class `Text` has a single, pure virtual function `set` which "sets" type to an output stream provided as a parameter. Initially, there are three derived classes, `Blank`, `Character`, and `Word`, as shown in Figure 118.

Suppose that we want to typeset paragraphs. Paragraphs are related to the classes we have in that they should be able to implement the method `set`, but they are also different in that a paragraph has many words and is typically typeset between margins. Nevertheless, we can add a class `Paragraph` to the hierarchy, as shown in Figure 119.

Class `Paragraph` has a parameter in the constructor giving the width in which to set the paragraph. (A more realistic example would also allow this value to be set after construction.) We provide a function `addWord` that allows us to build paragraphs, and the words are stored in a STL `vector`. The important points are that class `Paragraph` inherits from class `Text` and implements `set`, making it a part of the hierarchy.

Figure 120 shows the implementation of the `Paragraph` functions `set` and `addWord`. Function `set` uses a `stringstream` to store the words in a line. When there is not enough space on the line for the next word (and the blank that precedes it), the string stream buffer is written to the output stream.

10.6.3 Visitor

Each time we add a function to a class hierarchy such as `Text`, we have to add one function to each class in the hierarchy. Furthermore, each of these functions has essentially the same structure. For example, every function we add to class `Paragraph` will iterate over the words in the paragraph.

```
class Text
{
public:
    virtual void set(std::ostream & os) = 0;
};

class Blank : public Text
{
public:
    Blank() {}
    void set(std::ostream & os) { os << ' '; }
};

class Character : public Text
{
public:
    Character(char c) : c(c) {}
    void set(std::ostream & os) { os << c; }
private:
    char c;
};

class Word : public Text
{
public:
    Word(const std::string & w) : w(w) {}
    void set(std::ostream & os) { os << w; }
    std::size_t size() const { return w.size(); }
private:
    std::string w;
};
```

Figure 118: Declarations for the text class hierarchy

```
class Paragraph : public Text
{
public:
    Paragraph(int width) : width(width) {}
    void set(std::ostream & os);
    void addWord(const Word & w);
private:
    std::vector<Word> words;
    size_t width;
};
```

Figure 119: A class for paragraphs

```
void Paragraph::set(std::ostream & os)
{
    Blank b;
    ostringstream oss;
    for (  vector<Word>::iterator it = words.begin();
           it != words.end(); ++it)
    {
        if (size_t(oss.tellp()) + 1 + it->size() > width)
        {
            while (size_t(oss.tellp()) < width)
                b.set(oss);
            os << oss.str() << endl;
            oss.seekp(0);
        }
        if (size_t(oss.tellp()) > 0)
            b.set(oss);
        it->set(oss);
    }
}

void Paragraph::addWord(const Word & w)
{
    words.push_back(w);
}
```

Figure 120: Definitions for class `Paragraph`

The purpose of the visitor pattern is to avoid both the need to add functions to every class and to repeat the iteration patterns over and over again. The idea is to define a "visitor" class that knows how to process each kind of object in the hierarchy. The visitor class is an abstract base class that cannot actually do anything useful, but it has derived classes for performing specific operations.

The following steps are required to support visitors in the `text` hierarchy.

1. Create a base class `Visitor` with a visiting function corresponding to each derived class in the `Text` hierarchy: see Figure 121. Each visiting function is passed a pointer to an object of the corresponding type in the `Text` hierarchy.

2. Add functions to "accept" visitors in the `Text` hierarchy, starting with a pure virtual function in the base class:

```
class Text
{
public:
    virtual void accept(Visitor & vis) = 0;
    ....
};
```

```
class Visitor
{
public:
    virtual void visitBlank(Blank*) = 0;
    virtual void visitCharacter(Character*) = 0;
    virtual void visitWord(Word*) = 0;
    virtual void visitParagraph(Paragraph*) = 0;
};
```

Figure 121: The base class of the `Visitor` hierarchy

Most of these functions are fairly trivial. For example:

```
void Blank::accept(Visitor & vis) { vis.visitBlank(this); }
void Character::accept(Visitor & vis) { vis.visitCharacter(this); }
```

For composite classes, function `accept` passes the visitor to each component individually:

```
void Paragraph::accept(Visitor & vis)
{
    for (  vector<Word>::iterator it = words.begin();
           it != words.end(); ++it)
       it->accept(vis);
}
```

This completes the framework for visiting.

3. The next step is to construct an actual visitor. We will reimplement the typesetting function, `set` as a visitor. For this, we need a class `setVisitor` derived from `Visitor`: see Figure 122.

```
class setVisitor : public Visitor
{
public:
    void visitBlank(Blank*);
    void visitCharacter(Character * pc);
    void visitWord(Word*);
    void visitParagraph(Paragraph*);
};
```

Figure 122: A class for typesetting derived from `Visitor`

4. Finally, we implement the member functions of `setVisitor`, as in Figure 123. It is not necessary to provide a body for `setVisitor::visitParagraph` because it is never called: when a `Paragraph` object accepts a visitor, it simply sends the visitor to each of its words (see `Paragraph::accept` above). However, we do have to provide a trivial implementation in order to make `Paragraph` non-abstract. (Alternatively, we could have defined a trivial default function in the base class.)

```
void setVisitor::visitBlank(Blank*)
{
    cout << ' ';
}

void setVisitor::visitCharacter(Character * pc)
{
    cout << pc->getChar();
}

void setVisitor::visitWord(Word * pw)
{
    cout << ' ' << pw->getWord();
}

void setVisitor::visitParagraph(Paragraph * pp) {}
```

Figure 123: Implementation of class `setVisitor`

5. To typeset a paragraph `p`, we call just:

```
p.accept(setVisitor());
```

For a single task (typesetting), setting up the visitor classes seems rather elaborate. Suppose, however, that there were many operations to be performed on the `Text` hierarchy. Once we have defined the `accept` functions, we do not need to make any further changes to that hierarchy. Instead, for each new operation that we need, we derive a class from `Visitor` and define its "visit" functions for each kind of `Text`.

The visitor pattern has some disadvantages:

- If we use a separate function for each operation (like `set` in the original example), we can pass information around simply by adding parameters. Since the operations are independent, each can have its own set of parameters.

  The operations in the visitor version, however, must all use the protocol imposed by `accept`. This means that they all get exactly the same parameters (none, in our example).

- The structure of composite objects is hard-wired into their `accept` functions. For example, `Paragraph::accept` iterates through its vector of `Words`. This makes it difficult to modify the traversal in any way.

  In this example, `Paragraph::set` inserted line breaks whenever the length of a line would otherwise have exceeded `width`. It is difficult to provide this behaviour with the visitor pattern, because `Paragraph::accept` does a simple traversal over the words and does not provide for any additional actions.

11 Using Design Patterns

Design patterns encode the knowledge of experienced designers. Patterns are design components that can be mapped into code for a particular application. A discussion of patterns in general is beyond the scope of this course; in this chapter, we look at some of the simpler patters and show how they can be mapped into C++ code.

The "bible" of design patterns is the book *Design Patterns* (Gamma, Helm, Johnson, and Vlissides 1995)[40] The patterns discussed in this chapter are all taken from the "GoF book". We have already seen some design patterns in these notes, including Command (Section 7.7.2) and Façade (Section 10.5.2).

11.1 Singleton

Problem Ensure that a class has only one instance at all times and provide a global point of access to it.

Solution The problem has two parts: the first part is to ensure that only one instance of a class can exist; the second is to provide users with access to the unique instance. 394

Figure 133 shows a declaration for class `Singleton` based on Alexandrescu's example (2001, pages 129–133). Here are some points of interest in this declaration.

- The function `fun` and the data member `uid` are included just to illustrate that the singleton can do something besides merely exist.
- The only way that a user can access the singleton is through the static reference function `instance`.
- Access to the unique instance of the singleton is through the `private`, `static` pointer, `pInstance`.
- The constructor is `private`. The copy constructor and assignment operator are `private` and unimplemented. Thus the singleton cannot be constructed (except for the first time), passed by value, assigned, or otherwise copied.
- The destructor is also `private` and unimplemented. This ensures that the singleton cannot be accidentally deleted. (This could be considered as a memory leak, but that is harmless because there is only one instance anyway.)

395

Figure 134 shows the implementation of class `Singleton`. The constructor is conventional; the body given here merely demonstrates that it works. The function `instance` constructs the singleton when it is called for the first time, and subsequently just returns a reference to it. A user must access the singleton object as `Singleton::instance()`. Any attempt to copy it or pass it by value causes a compile-time error. For the particular singleton defined here, all the user can do is apply the function `fun` to the singleton. Obviously, other functions would be added in a practical application. This is in fact a rather simple version of the Singleton pattern and it is not robust enough for all applications. For details about its improvement, see (Alexandrescu 2001, pages 133–156). 396

397

[40] The four authors are often referred to as "The Gang of Four" or "GoF".

```
class Singleton
{
public:
    static Singleton & instance();
    void fun();
private:
    int uid;
    static Singleton * pInstance;
    Singleton();
    Singleton(const Singleton &);
    Singleton & operator=(const Singleton &);
    ~Singleton();
};
```

Figure 133: Declaration for class `Singleton`

```
int globalUID = 77;

// Inaccessible pointer to the unique instance.
Singleton * Singleton::pInstance = 0;

// Private constructor.
Singleton::Singleton()
{
    uid = globalUID++;
}

// Public member function allows user to create the unique instance.
Singleton & Singleton::instance()
{
    if (!pInstance)
        pInstance = new Singleton;
    return *pInstance;
}

// Simple test function.
void Singleton::fun()
{
    cout << "I am Singleton " << uid << endl;
}
```

Figure 134: Definitions for class `Singleton`

Figure 135 demonstrates the singleton in action. The first three lines of output show that there is only one instance; if more instances had been created, `globalUID` would have been incremented. The fourth line shows that the singleton has been passed as a reference.

```
void f(Singleton & s)
{
   cout << "Inside f: ";
   s.fun();
}

int main()
{
   Singleton::instance().fun();
   Singleton::instance().fun();
   Singleton::instance().fun();

   // A singleton can be passed by reference.
   f(Singleton::instance());

   // The following lines are all illegal.
   // Singleton s1;
   // Singleton s2 = Singleton::instance();

   return 0;
}
```

Output:

```
I am Singleton 77
I am Singleton 77
I am Singleton 77
Inside f: I am Singleton 77
```

Figure 135: A program that tests class `Singleton` and its output

11.2 Composite

Problem Suppose we have a hierarchy of classes. We may also have collections of instances of these classes. A collection might be homogeneous (members all of the same class) or heterogeneous (members of different classes). The problem is to make these collections behave in the same way as elements of the class hierarchy. The collections are called *composite classes* and the pattern that solve the problem is called Composite.

Solution To implement the Composite pattern, all we have to do is incorporate the composite classes into the class hierarchy. Since putting them into the hierarchy forces them to implement the base class interface, the effect will be to make their behaviour similar to that of other classes in the hierarchy.

The following example is a highly simplified typesetting application. The abstract base class `Text` has a single, pure virtual function `set` which "sets" type to an output stream provided as a parameter. Initially, there are three derived classes, `Blank`, `Character`, and `Word`, as shown in Figure 136.

398
399

```
class Text
{
public:
   virtual void set(std::ostream & os) = 0;
   virtual void accept(Visitor & v) = 0;
};

class Blank : public Text
{
public:
   Blank() {}
   void set(std::ostream & os);
};

class Character : public Text
{
public:
   Character(char c) : c(c) {}
   void set(std::ostream & os);
   char getChar() const { return c; }
private:
   char c;
};

class Word : public Text
{
public:
   Word(const std::string & w) : w(w) {}
   void set(std::ostream & os);
   std::string getWord() const { return w; }
   std::size_t size() const { return w.size(); }
private:
   std::string w;
};
```

Figure 136: Declarations for the text class hierarchy

Suppose that we want to typeset paragraphs. Paragraphs are related to the classes we have in that they should be able to implement the method `set`, but they are also different in that a paragraph has many words and is typically typeset between margins. Nevertheless, we can add a class `Paragraph` to the hierarchy, as shown in Figure 137.

Class `Paragraph` has a parameter in the constructor giving the width in which to set the paragraph. 400
(A more realistic example would also allow this value to be set after construction.) We provide a function `addWord` that allows us to build paragraphs, and the words are stored in a STL `vector`. The important points are that class `Paragraph` inherits from class `Text` and implements `set`, making it a part of the hierarchy. 401

Figure 138 shows the implementation of the `Paragraph` functions `set` and `addWord`. Function `set` 402

```
class Paragraph : public Text
{
public:
   Paragraph(int width) : width(width) {}
   void set(std::ostream & os);
   void addWord(const Word & w);
private:
   std::vector<Word> words;
   size_t width;
};
```

Figure 137: A class for paragraphs

```
void Paragraph::set(std::ostream & os)
{
   Blank b;
   ostringstream oss;
   for (vector<Word>::iterator it = words.begin(); it != words.end(); ++it)
   {
      if (size_t(oss.tellp()) + 1 + it->size() > width)
      {
         while (size_t(oss.tellp()) < width)
            b.set(oss);
         os << oss.str() << endl;
         oss.seekp(0);
      }
      if (size_t(oss.tellp()) > 0)
         b.set(oss);
      it->set(oss);
   }
}

void Paragraph::addWord(const Word & w)
{
   words.push_back(w);
}
```

Figure 138: Definitions for class `Paragraph`

uses a `stringstream` to store the words in a line. When there is not enough space on the line for the next word (and the blank that precedes it), the string stream buffer is written to the output stream.

11.3 Visitor

403

Problem Figure 139 shows a grid of classes and operations that extend the `Text` hierarchy of the previous section. Note that a column corresponds to a class (that implements each operation)

	Blank	Char	Word	Paragraph	Chapter
set	⋆	⋆	⋆	⋆	⋆
cap	⋆	⋆	⋆	⋆	⋆
print	⋆	⋆	⋆	⋆	⋆
count	⋆	⋆	⋆	⋆	⋆

Figure 139: A grid of classes and operations

and a row corresponds to an operation (that must be implemented in each class). Each ⋆ indicates an action that must be implemented. Here are two ways of organizing the code that implements the operations:

1. In pre-object-oriented days, each operation was implemented as a single function with a big `switch` statement:

404

```
void set(....)
{
   switch(kind)
   {
   case BLANK:
      ....
      break;
   case CHAR:
      ....
      break;
   case WORD:
      ....
      break;
   ....
   }
}
```

   This made it easy to add a function, because all of the code would be in one place, but hard to add a class, because this would involve adding a clause to many different functions.

2. In an object-oriented language, we put the classes into a hierarchy; the root class of the hierarchy defines default or null versions of each function; and each class implements the functions in its own way. This design has two consequences:

   (a) The code becomes fragmented: a function implements one operation in one class.

   (b) It is easy to add a class, with an implementation for each operation, but hard to add an operation, because each class has to be modified.

   (c) If the objects are stored in a data structure, it is likely that each function will be responsible for traversing its part of the data structure.

```
class Visitor
{
public:
   virtual void visitBlank(Blank*) = 0;
   virtual void visitCharacter(Character*) = 0;
   virtual void visitWord(Word*) = 0;
   virtual void visitParagraph(Paragraph*) = 0;
};
```

Figure 140: The base class of the `Visitor` hierarchy

For example, each time we add a operation to a class hierarchy such as `Text`, we have to add one function to each class in the hierarchy. Furthermore, every function we add to class `Paragraph` will iterate over the words in the paragraph.

The problem is combine these two modes: we would like to retain the fragmented code, which is easier for maintenance, but we would like to organize it by operation rather than by class. We would also like to keep the organization of the data structure out of the individual functions.

Solution The idea of the Visitor pattern is to define a "visitor" class that knows how to process each kind of object in the hierarchy. The visitor class is an abstract base class that cannot actually do anything useful, but it has derived classes for performing specific operations.

The following steps are required to support visitors in the `Text` hierarchy.

1. Create a base class `Visitor`: see Figure 140. This class has a virtual "visiting" function corresponding to each derived class in the `Text` hierarchy. Each visiting function is passed a pointer to an object of the corresponding type in the `Text` hierarchy. 405

2. Add functions to "accept" visitors in the `Text` hierarchy, starting with a pure virtual function in the base class, as shown in Figure 141. Most of these functions are fairly trivial. For example: 406 407

   ```
   void Blank::accept(Visitor & vis) { vis.visitBlank(this); }
   void Character::accept(Visitor & vis) { vis.visitCharacter(this); }
   ```

   For composite classes, function `accept` passes the visitor to each component individually: 408

   ```
   void Paragraph::accept(Visitor & vis)
   {
       for (  vector<Word>::iterator it = words.begin();
              it != words.end(); ++it)
          it->accept(vis);
   }
   ```

   This completes the framework for visiting.

3. The next step is to construct an actual visitor. We will reimplement the typesetting function, `set` as a visitor. For this, we need a class `setVisitor` derived from `Visitor`: see Figure 142. 409

```
class Text
{
public:
   virtual void accept(Visitor & v) = 0;
   ....
};
```

Figure 141: Adding `accept` to the base class of the `Text` hierarchy

```
class setVisitor : public Visitor
{
public:
    void visitBlank(Blank*);
    void visitCharacter(Character * pc);
    void visitWord(Word*);
    void visitParagraph(Paragraph*);
};
```

Figure 142: A class for typesetting derived from `Visitor`

4. Finally, we implement the member functions of `setVisitor`, as in Figure 143. It is not necessary to provide a body for `setVisitor::visitParagraph` because it is never called: when a `Paragraph` object accepts a visitor, it simply sends the visitor to each of its words (see `Paragraph::accept` above). However, we do have to provide a trivial implementation in order to make `Paragraph` non-abstract. (Alternatively, we could have defined a trivial default function in the base class.) 410

5. To typeset a paragraph `p`, we call just: 411

```
p.accept(setVisitor());
```

To demonstrate the flexibility of the Visitor pattern, we can define another visitor that sets the same text in capital letters. The new class is called `capVisitor`: 412

```
class capVisitor : public Visitor
{
public:
   void visitBlank(Blank*);
   void visitCharacter(Character * pc);
   void visitWord(Word*);
   void visitParagraph(Paragraph*);
};
```

Its member functions are the same as those of `setVisitor` except for `visitCharacter` and `visitWord`: 413

```
void capVisitor::visitCharacter(Character * pc)
{
```

```
void setVisitor::visitBlank(Blank*)
{
    cout << ' ';
}

void setVisitor::visitCharacter(Character * pc)
{
    cout << pc->getChar();
}

void setVisitor::visitWord(Word * pw)
{
    cout << ' ' << pw->getWord();
}

void setVisitor::visitParagraph(Paragraph * pp)
{
   cout << "I should never be called";
}
```

Figure 143: Implementation of class `setVisitor`

```
    char ch = pc->getChar();
    cout << toupper(ch);
}

void capVisitor::visitWord(Word * pw)
{
    string word = pw->getWord();
    for (string::iterator it = word.begin(); it != word.end(); ++it)
       *it = toupper(*it);
    cout << ' ' << word;
}
```

To invoke `capVisitor`, we simply construct an instance:

```
p.accept(capVisitor());
```

For a single task (typesetting), setting up the visitor classes seems rather elaborate. Suppose, however, that there were many operations to be performed on the `Text` hierarchy. Once we have defined the `accept` functions, we do not need to make any further changes to that hierarchy. Instead, for each new operation that we need, we derive a class from `Visitor` and define its "visit" functions for each kind of `Text`.

The visitor pattern has some disadvantages:

- If we use a separate function for each operation (like `set` in the original example), we can pass information around simply by adding parameters. Since the operations are independent, each can have its own set of parameters.

The operations in the visitor version, however, must all use the protocol imposed by `accept`. This means that they all get exactly the same parameters (none, in our example).

- The structure of composite objects is hard-wired into their `accept` functions. For example, `Paragraph::accept` iterates through its vector of `Word`s. This makes it difficult to modify the traversal in any way.

  In the example above, `Paragraph::set` inserted line breaks whenever the length of a line would otherwise have exceeded `width`. It is difficult to provide this behaviour with the visitor pattern, because `Paragraph::accept` does a simple traversal over the words and does not provide for any additional actions. Nor is it possible to pass an argument to `Paragraph::accept` giving the position on the current line.

11.4 State

Problem An object has various states and behaves in different ways depending on its state. We could write a `switch` statement to capture this property, as shown in Figure 144. This solution is unsatisfactory because any changes — modification of a state's behaviour, adding or removing states, etc. — requires changes to this statement.

414 415

```
class StateChanger
{
public:
   void act()
   {
      switch (state)
      {
      case RUNNING:
         // ....
         break;
      case STUCK:
         // ....
         break;
      case BROKEN:
         // ....
         break;
      }
   }
private:
   enum { RUNNING, STUCK, BROKEN } state;
};
```

Figure 144: A class with various states

Solution The State pattern is often implemented by inheritance and dynamic binding. The base class, which may be abstract, defines the action corresponding to a default state or no action at all. Each derived class defines the action for a particular state. The following example demonstrates the State pattern with a scanner whose state determines the text that it recognizes.

The abstract base class `Scanner` shown in Figure 145 specifies that any scanner object must accept a character pointer and return a `string`. Note that the pointer is passed by reference, because a scanner will advance the pointer over the buffer and must return its new value to the caller. The result of scanning is the a string representing the token scanned. For example, a number scanner might return the string `"123.456"`. Figure 145 also shows derived classes that scan white space, numbers, and identifiers. 416 417 418

The scanners may be selected in various ways. Figure 146 shows a simple managing class. An instance of `ScanManager` has a pointer to each kind of scanner; its the constructor creates the corresponding objects dynamically and its destructor deletes them. The function `ScanManager::scan` is given a pointer to a character array and it uses the character to choose a scanner object. Figure 147 shows a simple example of the scanner in use. 419 420 421 422

Remarks:

- The nice thing about the State pattern is that it localizes the behaviour corresponding to a given state. There are many simple classes rather than a possibly huge `switch` statement or equivalent. It is easy to modify a state without looking at, or even knowing about, other states. Adding a state is also easy and does not require interfering with existing code.
- In this example, there is a single controlling object, the `ScanManager`, that handles all the state changes. This makes the State pattern look a bit pointless. However, there are other ways of changing state:
  - Each derived class might be responsible for choosing its successor state. A drawback of this approach is that each derived class has to know about one or more other derived classes, which goes against the usual practice of keeping derived classes independent of each other.
  - The state could be chosen externally. In the example above, `ScanManager` could provide another member function that was told what kind of token to expect and could set the state accordingly.
- It would be nice to put some useful data into the base class. In C++, we cannot do this efficiently, because the scanning is performed by separate objects that cannot access data in the base object. There are several workarounds for this problem.
  - We could construct state objects dynamically when needed, pass them the information they need, and delete them at the next state transition. This is clearly less efficient than the solution above, but the inefficiency might be acceptable if state transitions were infrequent.
  - We could provide additional functions for updating the `Scanner` objects before executing them. But it is not obvious when to call such functions.
  - Some object oriented languages (e.g., Self) provide *dynamic inheritance*, which allows an object to move around the class hierarchy, adopting the behaviour of the class it belongs to at any given time.

11.5 Bridge

Problem Tight coupling between an interface and an implementation make it difficult to change either part without affecting the other.

```
class Scanner
{
public:
   virtual string scan(char * & p) = 0;
};

class ScanBlanks : public Scanner
{
public:
   string scan(char * & p)
   {
      while (isspace(*p))
         p++;
      return string();
   }
};

class ScanNumber : public Scanner
{
public:
   string scan(char * & p)
   {
      string num;
      while (isdigit(*p))
         num += *p++;
      if (*p == '.')
      {
         num += *p++;
         while (isdigit(*p))
            num += *p++;
      }
      return num;
   }
};

class ScanIdentifier : public Scanner
{
public:
   string scan(char * & p)
   {
      string id(1, *p++);
      while (isalpha(*p) || isdigit(*p))
         id += *p++;
      return id;
   }
};
```

Figure 145: State pattern: scanners for white space, numbers, and identifiers

```
class ScanManager
{
public:
   ScanManager();
   ~ScanManager();
   string scan(char *buffer);
private:
   Scanner *ps;
   Scanner *psBlanks;
   Scanner *psNumber;
   Scanner *psIdentifier;
};

ScanManager::ScanManager() :
   ps(0),
   psBlanks(new ScanBlanks()),
   psNumber(new ScanNumber()),
   psIdentifier(new ScanIdentifier())
{ }

ScanManager::~ScanManager()
{
   delete psBlanks;
   delete psNumber;
   delete psIdentifier;
}

string ScanManager::scan(char *buffer)
{
   string result;
   char *p = buffer;
   while (*p != '\0')
   {
      if (isspace(*p))
         ps = psBlanks;
      else if (isdigit(*p))
         ps = psNumber;
      else if (isalpha(*p))
         ps = psIdentifier;
      else
         throw "illegal character.";
      string token = ps->scan(p);
      if (token.length() > 0)
         result += "<" + token + "> ";
   }
   return result;
}
```

Figure 146: State pattern: the scanner controller class

```
int main()
{
   ScanManager sm;
   char *test = "123 456 Pirate456\t789 anotherID ";
   try
   {
      cout << "Scan succeeded: " << sm.scan(test).c_str() << endl;
   }
   catch (const char *error)
   {
      cerr << "Scan failed: " << error << endl;
   }
   return 0;
}
```

Figure 147: State pattern: using the scanner

Solution The Bridge pattern separates an abstraction from its implementation so that the two can vary independently. It is also known as the ***Handle/Body*** pattern.

There is an example of the Bridge pattern (called Handle/Body) in Koenig and Moo (2000). In Section 13.4 (pages 243–245), there is a declaration of class `Student_info` in which the only data member is `Core *cp`. The pointer `cp` points an instance of either the base class `Core` or the derived class `Grad`: see Figure 148. An instance of class `Student_info` is constructed by reading data from a file. The class of `*cp` is determined by a character in the file: see Figure 149.

11.6 The Curiously Recurring Template Pattern

The Curiously Recurring Template Pattern (CRTP) was named by James "Cope" Coplien (1995). Here it is:

```
class X : public Base<X>
{
    ....
};
```

At first sight, it is rather mysterious. An example may help.

Consider this problem: we would like to provide a way of extending any class that provides `operator<` so that it also provides `operator>`.

We might try to apply inheritance in the usual way. We define a base class:

```
class Ordered
{
   virtual bool operator<(const Ordered & other) = 0;
   bool operator>(const Ordered & other)
   {
      return other < *this;
```

```
#ifndef GUARD_Student_info_h
#define GUARD_Student_info_h

#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>

#include "Core.h"

class Student_info {
public:
    // constructors and copy control
    Student_info(): cp(0) { }
    Student_info(std::istream& is): cp(0) { read(is); }
    Student_info(const Student_info&);
    Student_info& operator=(const Student_info&);
    ~Student_info() { delete cp; }

    // operations
    std::istream& read(std::istream&);

    std::string name() const {
        if (cp) return cp->name();
        else throw std::runtime_error("uninitialized Student");
    }
    double grade() const {
        if (cp) return cp->grade();
        else throw std::runtime_error("uninitialized Student");
    }

    static bool compare(const Student_info& s1,
                        const Student_info& s2) {
        return s1.name() < s2.name();
    }

private:
    Core* cp;
};

#endif
```

Figure 148: Separating interface and implementation (Koenig & Moo, pages 243–244)

```
istream& Student_info::read(istream& is)
{
    delete cp;          // delete previous object, if any

    char ch;
    is >> ch;           // get record type

    if (ch == 'U') {
        cp = new Core(is);
    } else {
        cp = new Grad(is);
    }

    return is;
}
```

Figure 149: Setting the pointer (Koenig & Moo, page 245)

```
   }
}
```

Then we can inherit this base class to give the desired functionality:

```
class Widget : public Ordered
{
    bool operator<(const Widget & other) { .... }
    ....
};
```

Unfortunately, this doesn't work. The redefinition of `operator<` in class `Widget` is incorrect because the parameter type is different. Moreover, `Ordered::operator>` provides an instance of `Ordered` for comparison, but we want to compare another `Widget`.

```
template <class T>
class Ordered
{
public:
    bool operator>(const T & rhs) const
    {
        const T & self = static_cast<const T &>(*this);
        return rhs < self;
    }
};
```

Figure 150: A class that provides `operator>` using `operator<`

CRTP comes to the rescue! We define the base class `Ordered` as a template class, as shown in Figure 150. We have to cast from `Ordered` to `T`, but we will see that this doesn't matter in practice. 428

The revised version of class `Widget` inherits from a version of class `Ordered` that is customized for `Widget`s, using the "recursive" template argument:

```
class Widget : public Ordered<Widget>
{
    bool operator<(const Widget & other) { .... }
    ....
};
```

After templates have been expanded, we have a class that looks something like this (or rather, would look like this if we could see it): 429

```
class OrderedWidget
{
public:
    bool operator>(const Widget & rhs) const
    {
        const Widget & self = static_cast<const Widget &>(*this);
        return rhs < self;
    }
};
```

We can now see that the cast is perfectly harmless, because it is just casting a `const Widget &` to itself!

As another example of CRTP, consider the problem of keeping track of the number of instances of a class. This is easily done by providing a `static` counter in the class. But suppose we want to encapsulate this behaviour, providing a base class whose children can all keep track of their instances.

The obvious way of doing this doesn't work. If we put a `static` counter in the base class, we will count *all* instances of child classes. We need a way of providing a separate counter for *each* child class. Once again, CRTP shows the way. Here is the base class: 430

```
template<class T>
class Counted
{
public:
   Counted() { ++counter; }
   ~Counted() { --counter; }
   static long num() { return counter; }
private:
   static long counter;
};
```

To make class `T` a "counted" class, it must inherit from `Counted<T>`. Whenever we create such as class, we must also provide an initialized definition for its counter:

```
class Widget : public Counted<Widget> { .... };
long Counted<Widget>::counter = 0;
```

We can create as many counted classes as we need:

```
class Goblet : public Counted<Goblet> { .... };
long Counted<Goblet>::counter = 0;
```

Here is a program that tests these ideas. The output shows that the counts are maintained correctly as objects are created and destroyed. 431

```
void report(string title)
{
   cout << title << endl <<
      "Widgets: " << Widget::num() << endl <<
      "Goblets: " << Goblet::num() << endl << endl;
}
```

432

```
int main()
{
   Widget w1;
   Goblet g1;
   Goblet g2;
   {
      Widget w2;
      Widget w3;
      Goblet g3;
      report("Inner:");
   }
   report("Outer:");
   return 0;
}
```

This program writes:

```
Inner:
Widgets: 3
Goblets: 3

Outer:
Widgets: 1
Goblets: 2
```