APPENDIX A

APPENDIX

One can program in the object-oriented style using any programming language, even assembly language. However, such programming is much more difficult in a non-object-oriented language than it is in an object-oriented language, for in a non-object-oriented language, the programmer must code most examples of polymorphism or inheritance by hand, and encapsulation must generally be implemented by programmer convention; whereas in an object-oriented language the compiler supports polymorphism, inheritance, and encapsulation directly.

C++ is not an entirely object-oriented language; a C++ program *may* have classes and instances of those classes, but C++ programs usually have globally defined functions (certainly the main function), in addition to objects. It is possible to write a C++ program, which lacks any trace of object orientation, although, of course, this text does not advocate such a programming style. The C++ language is based on the C language, and many C programs may be recompiled in C++ without much modification. However, the object-oriented superstructure in C++ makes it a distinctly different language from C.

C++ is a compiled language, where source code is compiled into an executable, stand-alone program. There are numerous C++ compilers on many different platforms. Unfortunately, these compilers are not entirely consistent among themselves. This text will advocate a "conservative" approach to C++, which should produce good results with nearly any C++ compiler. The C++ compiler will compile pure C language code as the C++ language is an extension of the C language. Consequently, C++ is not a pure object-oriented language. It has three categories of data types: primitive types, classes, and pointers.

- Pointers contain the memory addresses of variables. It is possible to perform some arithmetic on pointers, but in general they do not participate in most operations possible on, for instance, primitive types. It would be meaningless, for instance, to multiply two pointers together (what could the product of two street addresses possibly mean? The product of two memory addresses is equally meaningless).

- Primitive types and classes are quite different. Although, unlike some languages, C++ does not place all classes into a single hierarchy, classes can be related in one or more hierarchies, whereas the primitive types do not belong to any hierarchy. The programmer can create new classes, but cannot create new primitive types.

An overview of C++ is given below. Anyone with a basic background in language C can skip most of this material. The material presented here is only an overview and not a complete exposition, but there are many good introductory texts which give fuller details. C++ is a full-featured programming language and, as such, it is difficult to present just one feature at a time. In the brief presentation below, therefore, some aspects will be mentioned before being fully explained. Some of the advanced features of C++ are omitted as they are not used in this text.

The canonical introductory program in every language is one which prints "Hello, world." Here is the C++ version:

```
#include <iostream>
using namespace std;

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

This code is saved in a plain text file named "Hello.cpp" and compiled according to the instructions for the C++ compiler. When the executable program is run, it prints "Hello, world" on the monitor. Below, we shall examine what this code means. For now, we will simply treat the line "#include <iostream>" as a required magic incantation, and state that if the user wishes to output a variable, "cout << variable << endl;" will output the variable on its own line.

It should be noted that the above is almost the minimum for an executable program; any C++ program must include a function called *main*(), with return type either void (as above) or int (to be discussed), and either no parameters (as above) or certain specified parameters (to be discussed).

A.1 Identifiers

Identifiers are the names given by programmers to objects in a program. In C++, identifiers can be of any length. They must start with a letter or underscore; after the first character, digits are also permitted.

C++ is case-sensitive, so Stack and stack are distinct identifiers.

In the introductory program above, *main* is an identifier.

A.2 Primitive Data Types

The basic primitive types of C++ are similar to the primitive types of other programming languages. They are described below; the value ranges given are examples from one specific compiler.

char	One byte (may or may not be signed)
unsigned char	One byte, unsigned; range 0 to 255
signed char	One byte, signed two's complement integer; range –128 to 127
short int	16-bit signed two's complement integer; range –32768 to 32767
unsigned int	32-bit unsigned integer; range 0 to 4294967295
int	32-bit signed two's complement integer; range –2147483648 to 2147483647
long	same as int
float	32-bit floating-point number; range 3.4×10^{-38} to 3.4×10^{38}
double	64- bit floating-point number; range 1.7×10^{-308} to 1.7×10^{308}
long double	80-bit floating-point number; range 3.4×10^{-4932} to 1.1×10^{4932}

Figure A.1: Basic primitive types with their range of values used in C++.

The char, short, int, and long types are all integer types, which hold integer values within the range allowed by the storage space. The char and short types require less storage than an int, but they are promoted to int if necessary before use in any operations. Integer literals may be written in decimal, octal, or hexadecimal. The C++ compiler identifies the base by examining the beginning of the integer literal: a leading 0x or 0X denotes a hexadecimal (base 16) number, a leading 0 otherwise denotes an octal (base 8) number, and any other leading digit denotes a decimal (base 10) number. An integer literal followed by L is considered a long integer.

It is important to note that the sizes of these types are *not* consistent across all platforms. That is, an int on a machine with a word size of two bytes will usually be two bytes long, but an int on a machine with a word size of four bytes will usually be four bytes long (depending on the compiler). If the exact size of a primitive data type is important in a program, the programmer must check it and it will be necessary to take precautions to ensure that the program will compile correctly on different platforms. In this text, the exact sizes of the primitive data types are unimportant.

There may be two kinds of pointers, near and far. They may or may not differ in size, depending on compiler and platform. In this text, we will firmly ignore the issue and assume that the reader will use the default pointers for the reader's particular compiler, and will consult the compiler's documentation if further information is desired.

A C++ compiler may or may not have a *bool* type which holds *true* or *false* values. The C language did lack this type, and so do some early C++ compilers. Therefore, programmers often use int variables to represent Boolean variables, with 0 representing false and anything else representing true. If the compiler has the bool type, then true and false are reserved words in the language representing the logical values.

The following logical operations are permitted on bool (or integer) variables (listed in order of precedence from highest to lowest):

!	Unary logical negation
&&	Logical AND
||	Logical OR

Figure A.2: Logical operators.

It usually makes little sense to use these logical operations on integer variables unless they are being treated as Boolean variables.

A char variable can hold any of the ASCII characters. Character literals can be written between single quotes: ′A′. Special characters may be represented by an escape sequence:

′\b′	backspace
′\f′	form feed
′\n′	new line
′\r′	return
′\t′	tab
′\\′	one backslash
′\′′	single quote
′\"′	double quote
′\ddd′	a character by octal value, where each *d* is one of 0-7

Figure A.3: Various special characters useful in C++.

Operators applicable to the integers are those applicable to floating point numbers (below), along with the bitwise operators (listed in order of precedence from highest to lowest):

~	Unary bitwise complement
<<	Shift left, filling with zero bits at right
>>	Shift right, extending the sign bit at left (for signed integers)
&	Bitwise AND
^	Bitwise exclusive OR
|	Bitwise inclusive OR

Figure A.4: Various bitwise operators.

Unlike integers, floating-point literals are always expressed in decimal notation, with an optional decimal point and an optional exponent. These are all valid floating-point numbers:

```
52 52. 52.0 5.2E1 0.52E2 .52E2 5E2
```

Operators applicable to floating point numbers, as well as to integers, are as follows (listed in order of precedence from highest to lowest):

var++, var--	Post-increment, Post-decrement
++var, --var, +var, -var	Pre-increment, Pre-decrement, unary sign
*, /, %	Multiplication, division, remainder
+, -	Addition, subtraction

Figure A.5: Operators used in C++ in order of precedence.

It must be noted, however, that the effect of / is different when used with two integer operands than when used with one or two floating point operands. When used with one or two floating point operands, it returns a floating point answer accurate within the limits of the floating point type. When used with two integer operands, however, it performs integer division, and returns an integer value, *not* a floating point value. Thus, 5/2 will return 2, not 2.5. This will prove quite useful in the implementations of data structures, since it generally produces the effect of the floor function $\lfloor x \rfloor$ without need of a method call. However, this is not the case if one of the integer operands is negative. In that case, we must call a library function to obtain the effect of the floor function $\lfloor x \rfloor$.

A.3 Variable Declaration

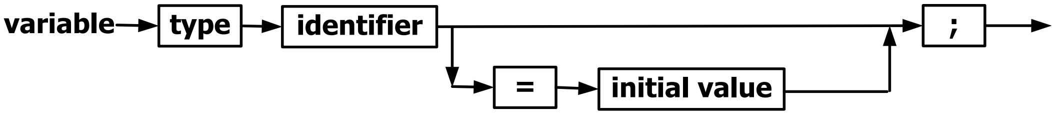

Figure A.6: Syntax for variable declaration.

In the above diagram for declaration of a variable, the type of the variable is declared, then the variable is named by an identifier, and finally the field is given an initial value, if desired.

Looking at the introductory program above, we see no variables…or do we? The mysterious *cout*, so far unexplained, is in fact a global variable. However, for the time being, we shall just use *cout* without explanation, and illustrate the use of variables by declaring our own:

```
#include <iostream>
using namespace std;

void main ()
{
     int i = 5;
     cout << i << endl;
     i = 10;
     cout << i << endl;
}
```

Note that, in the small program above, *i* is assigned a new value with the statement “i = 10”. C++ uses a single = for assignment and == for testing for equality.

A.4 Function Definition

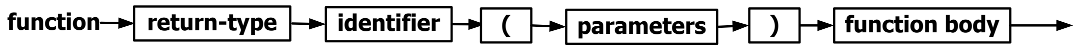

Figure A.7: Syntax for **function** definition.

Although we shall not have much occasion to write functions (as opposed to methods), other than *main*() functions, nevertheless we shall need an understanding of C++ functions. A function has a name, set out as an identifier; it has a return type, indicating what sort of value it will return to the caller; it may have parameters, which are listed as part of its declaration; and it has a function body.

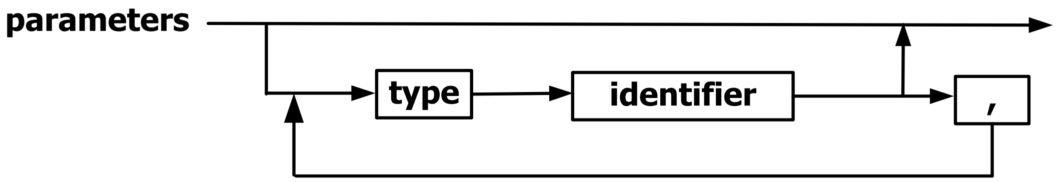

Figure A.8: Syntax diagram representing parameter list.

Each parameter is listed with its type. There may be any number of parameters, which must have distinct identifiers. In this book, the number of parameters is fixed for any given function; C++ does, in fact, allow a programmer to write functions with varying numbers of parameters.

A.4.1 Function Return Values

A function can return a value (those we have seen so far have had void return types). The word return causes an immediate jump out of the method. If the function is declared as returning a non-void type, then return must be followed by a value of the proper type; otherwise it must not. Further, if a function is declared as returning a non-void type, it *must* terminate through a return statement, that is, it should not just end as our main() functions have. Violation of these rules should cause a compiler error or, at least, a compiler warning. The programs below illustrate both void and non-void functions:

```
#include <iostream>
using namespace std;

void voidFunc1 ()
{
     cout << "voidFunc1()" << endl;
     // it is okay to simply end
}
// -------------------------------------------------------------
void voidFunc2 ()
{
     cout << "voidFunc2()" << endl;
     return;
     // it is also okay to return with no value
}
// -------------------------------------------------------------
void voidFunc3 ()
{
     cout << "voidFunc3()" << endl;
     return 0;
     // it is emphatically not okay to return a value;
     // this function is WRONG.
}
// -------------------------------------------------------------
int intFunc1 ()
{
     cout << "intFunc1()" << endl;
     return 0;
     // this is the correct way to end the function
}
// -------------------------------------------------------------
int intFunc2 ()
{
     cout << "intFunc2()" << endl;
     return;
     // this is emphatically WRONG;
     // the return must be followed by a value
}
// -------------------------------------------------------------
```

```
int intFunc3 ()
{
     cout << "intFunc3()" << endl;
     // this is also emphatically WRONG;
     // a non-void function MUST have a return followed by a value
}
```

As we have seen, the function body consists of variable declarations and executable statements. The executable statements are described in more detail below.

A.4.2 Local Variable Declarations

Let us repeat the small program given above:

```
#include <iostream>
using namespace std;

void main ()
{
     int i = 5;
     cout << i << endl;
     i = 10;
     cout << i << endl;
}
```

The variable *i* in the program above is a *local* variable in *main*(). This means that it has the following properties: (1) it does not exist until *main*() begins operation; (2) it is automatically destroyed when *main*() terminates; and (3) it is known only within *main*(). As we shall see when we begin to discuss classes and pointers, the first two properties of a local variable are very important with regard to instances of classes.

A function may have as many local variables as it needs (subject to the limitations of the stack, as we shall see), and they may be of any type: primitive, pointer, or instance of a class.

A.4.3 Function Parameters

We have seen that a function may have zero or more parameters passed to it, and that it may make use of them, for instance, by printing them out. The burning question is, however, what happens if the function *changes* one of its parameters? Consider the program below:

```
#include <iostream>
using namespace std;
```

```
void changeFunc (int val)
{
     cout << "val = " << val << endl;
     val = val * 2;
     cout << "now val = " << val << endl;
}
// ----------------------------------------------------------
void main ()
{
     int myVal = 10;
     cout << "myVal = " << myVal << endl;
     changeFunc (myVal);
     cout << "now myVal = " << myVal << endl;
}
```

The output from the program above shows us that *myVal* was not changed by *changeFunc*(), even though it was passed to *changeFunc*() by *main*():

```
myVal = 10
val = 10
now val = 20
now myVal = 10
```

What has happened here? The answer is that we have seen an example of pass-by-copy parameter passing.

Pass By Copy

The *main*() function declared a local variable, *myVal*. Therefore, when *main*() was called (as the program started up) a certain amount of memory was set aside for that variable, and a value was assigned to it: 10. The *changeFunc*() function declared a parameter, *val*. When the *changeFunc*() function was called, memory was set aside for that parameter in essentially the same way it would be set aside for a local variable. The value that was assigned to it was the value which appeared in the function call in *main*(): the value of *myVal*, which was 10.

Note particularly that *val* and *myVal* are distinct entities with separate memory set aside for them. The only connection between them is that *val* got its value from *myVal*, rather than from a literal value (as *myVal* did) or from a computation. Thus, a change in *val* would have no effect whatever on *myVal* and, indeed, we have seen that it did not. This would be true even if these variables had identical names.

A diagram may perhaps make this explanation clearer. In the diagram below, we see two blocks of memory (the addresses to the left are, of course, for illustration only), as they appear just before execution of the statement *val = val * 2*. The *myVal* variable declared in the *main*() function was placed in memory location 10000008. When the call to *changeFunc*() was made, the *val* variable was placed in an entirely different location in memory, 20000014. The value from location 10000008 was copied to location 20000014.

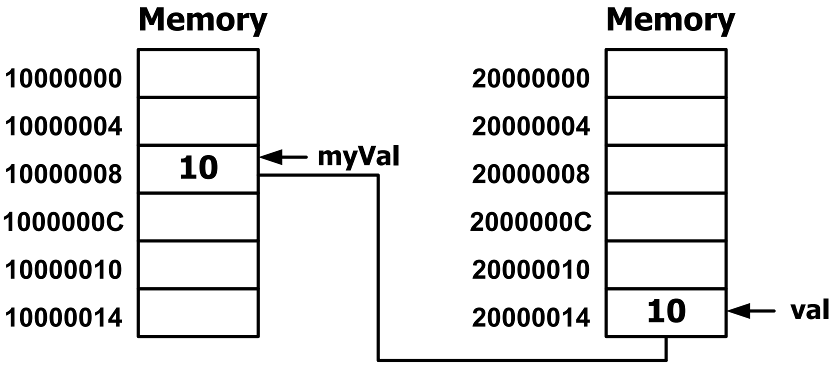

Figure A.9: Example for pass-by-copy. Note that myVal and val are referring to two different memory locations.

Now, after execution of the statement `val = val * 2`, the two blocks of memory appear as follows:

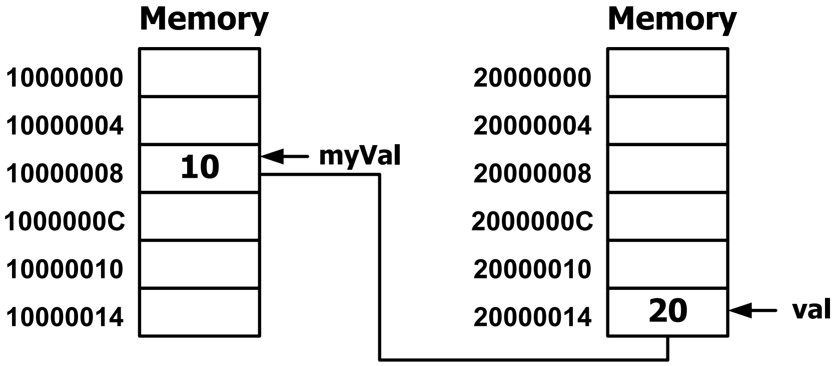

Figure A.10: Example of pass-by-copy and the memory boxes show the contents after execution of val=val*2 statement.

The value in the location assigned to `val` has changed, but that in the location assigned to `myVal` has not. That explains the output above, showing that `myVal` is unchanged on return from `changeFunc()`.

This form of parameter passing, where the function called obtains memory for its parameter and its parameter is initialized to the value given by the caller, is called "pass by copy" or "pass by value". It is the only form of parameter passing available in C[1]. It has the advantage of ensuring that there are no unintended side effects: the function cannot accidentally alter the value of the caller's variable, since it has only a copy of that variable. On the other hand, it has the disadvantage of inefficiency and can sometimes make programming more difficult.

In this case, the parameter was just an int: two or four bytes, depending on the compiler. But what if the parameter had been an instance of a large and complex class? What if it had required four hundred bytes, or four thousand? At some point, efficiency requires that we not copy the whole structure.

Furthermore, sometimes we would really like to allow the function to alter the variables of the caller. For instance, consider the problem of swapping the values of two variables, *x* and *y*. This is not difficult, but it requires three statements and a temporary variable:

```
int temp = x;
x = y;
y = temp;
```

It would be nice if we could write a function to perform this task, so that the three lines above could collapse into one function call:

```
swap (x, y);
```

It should be obvious, however, that pass-by-copy parameter passing rules out this simple function.

Pass By Reference

Both of the problems of pass-by-copy parameter passing can be avoided if we pass by reference instead. In this case, we want the function that is called to be able to refer to the same memory as the caller. In C++, we accomplish pass-by-reference parameter passing by appending an & to the type of the parameter. We illustrate this by changing just the signature of the

[1] C programmers may object that you can avoid pass-by-copy parameter passing by using pointers. Yes, but the pointers themselves are variables, and they are, indeed, passed by copy. In the program below, for instance, *pInt* contains only a copy of the value passed to *changeFunc*(), which was the address of *iMain*. We can change *pInt* (by pointing it at a different variable) without changing any variable in *main*(). We can certainly alter *iMain* using *pInt*, but *iMain* itself was never passed to *changeFunc*() in the first place.

```
void changeFunc (int* pInt) //This method is expecting a pointer to a integer.
{
     *pInt = (*pInt) * 2; //*pInt is same as iMain.
}
void main ()
{
     int iMain = 10;
     changeFunc (&iMain);
}
```

changeFunc() function given above, without any change to *main*() or the body of *changeFunc*():

```
void changeFunc (int& val)
{
     cout << "val = " << val << endl;
     val = val * 2;
     cout << "now val = " << val << endl;
}
```

Note that this tiny, almost unnoticeable change completely changes the effect of *changeFunc*(). The output from the program shows us that *myVal* most definitely was changed by *changeFunc*() this time:

```
myVal = 10
val = 10
now val = 20
now myVal = 20
```

What happened this time? The *main*() function still has a local variable which it calls *myVal*. This variable has a certain amount of memory allotted to it, and the compiler translates each use of *myVal* to a reference to the address of that memory. The *changeFunc*() function has a reference variable called *val*. When *changeFunc*() is called, that reference variable may be thought of as being "filled in" with the address of the memory set aside for *myVal*. Now, every use of *val* is translated to a reference to the address of that memory. Thus, an assignment to *val* is exactly the same (from the programmer's point of view) as an assignment to *myVal*: the specific area in memory to which they both refer gets a new value. It is now clear why *myVal* had a new value on return from *changeFunc*(): the memory to which *myVal* refers has been changed.

A diagram again may perhaps help with this concept. In the diagram below, we see a single chunk of memory as it appears just before execution of the statement val = val * 2. The *myVal* variable declared in the *main*() function was placed in memory location 10000008. When the call to *changeFunc*() was made, the *val* reference variable was pointed *to the same location*:

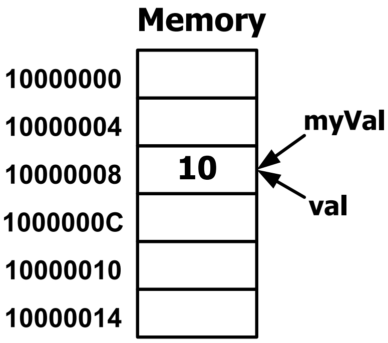

Figure A.11: Example of pass-by-reference. Both myval and val point to the same memory location.

Now, when the statement `val = val * 2` is executed, the value in the location pointed to by both *`myVal`* and `val` is changed.

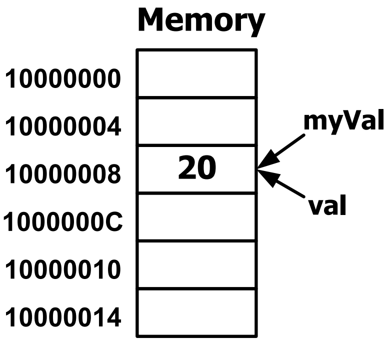

Figure A.12: Pass-by-reference example wherein the execution of the val=val*2 statement changes the value referenced by myval and val.

How does pass-by-reference parameter passing address the two disadvantages of pass-by-copy parameter passing? First, the efficiency problem: since the called function has only a reference to the instance of a large and complex class, there is no need to copy anything and no problem with efficiency. Second, the programming problem: it is simple to alter the variables of the caller, as we have seen. Thus, the *swap*() function which we desired is readily implemented with pass-by-reference parameter passing:

```
void swap (int& x, int& y)
{
     int temp = x;
     x = y;
     y= temp;
}
```

The reader should experiment with this function to verify that it works correctly, and that it would *not* work correctly if the & were omitted from one of the parameters.

Unfortunately, while pass-by-reference parameter passing takes care of the two disadvantages of pass-by-copy, at the same time it loses the advantage of pass-by-copy. Now it is entirely possible to have an unintended side effect of a function call as the called function accidentally changes the caller's variables. Fortunately, this problem was anticipated in the design of C++. If we want to use pass-by-reference for purposes of efficiency, but do not want to risk changes to the caller's variable, we can tag the reference as constant, alerting the compiler to reject any attempts to change it:

```
void noChange (const int& x)
{
     x = x * 2;
     // the line above will be rejected by the compiler
     // because x is a constant reference variable
}
```

A.5 Flow of Control Statements

We have seen that C++ has statements for assigning values to variables, performing calculations, and calling functions. However, unless we want strictly straight-line code, such as has been presented so far, where each statement follows the previous statement in exactly the order given, there must be a way of affecting the flow of control of the program. C++ has a flexible set of flow of control statements.

A.5.1 Conditional Statements

C++ has the following set of relational and equality operators, all of which return "Boolean" values (i.e., int values which can be treated as Boolean):

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to

!=	not equal to

Figure A.13: Various relational and equality operators.

Using Boolean variables and expressions including the above relational and equality operators, we can write conditional statements to choose between two or more execution paths in a C++ program. C++ has two forms of conditional statements and the conditional operator.

The simpler conditional statement is the if statement:

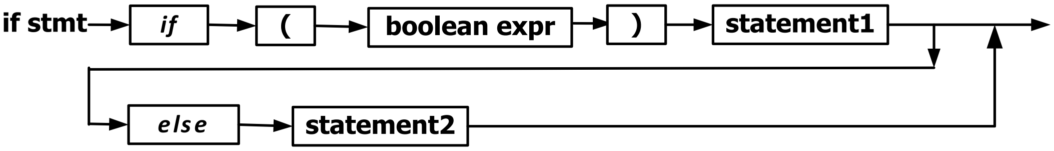

Figure A.14: Syntax for **if** statement.

An if statement allows two way branching: if the Boolean expression evaluates to true, statement1 will be executed; otherwise, statement1 will not be executed, but statement2 will be executed, assuming the optional else is present. If statement1 is single statement, then the if statement ends at the semicolon following statement1; if several statements are to be executed, they must be enclosed in braces.

```
if (a)
   doSomething();
   doSomethingElse();
```

Although the indentation indicates that *doSomethingElse*() is called only if *a* is true, in fact it will be called every time, regardless of whether *a* is true. To produce the result indicated by the indentation above, we must put the statements in braces:

```
if (a)
{
   doSomething();
   doSomethingElse();
}
```

The nested if statements must be handled with care. The following code snippet illustrates a possible "dangling else error":

```
if (a)
   if (b)
      doSomething();
else
   doSomethingElse();
```

It may appear to a reader that, if *a* is false, *doSomethingElse*() will be executed, but in fact, *doSomethingElse*() will be executed only if *a* is true and *b* is false. That is, despite the indentation (which the compiler ignores), the else belongs to the second if, not the first. What this illustrates is that an else always belongs to the last if which does not already have an else. To produce the result indicated by the indentation above, we must isolate the second if within braces:

```
if (a)
{
     if (b)
           doSomething();
}
else
     doSomethingElse();
```

The switch statement is the more complicated conditional statement, which allows multi-way branching:

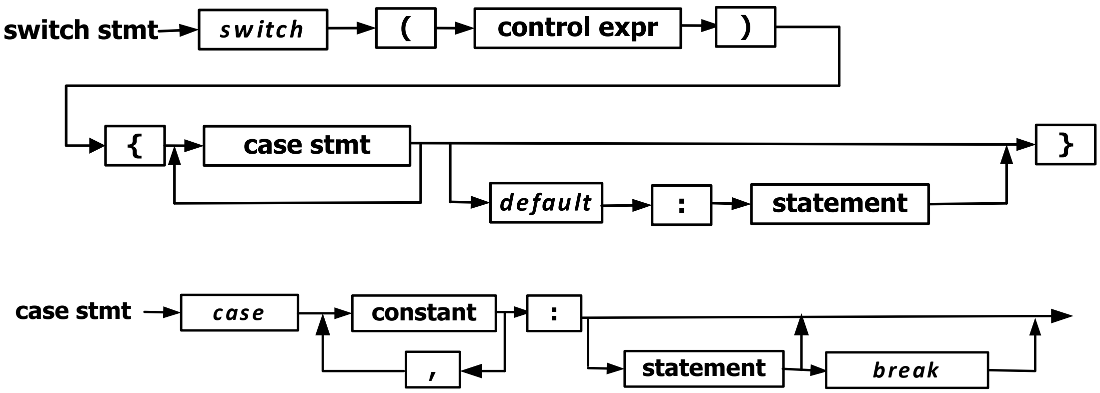

Figure A.15: Syntax for **switch** and **case** statement.

In the switch statement, the control expression is an integer or character type (not a floating point or an object). The control expression is evaluated and matched against the constants in the case statements, one by one. When a matching constant is found, the execution of program jumps to the statement immediately following it. If no matching constant is found, execution jumps to the statement following the word default or, if there is no default statement, execution resumes after the closing brace of the switch statement.

The switch statement exhibits "fall-through" behavior. That is, unless a case statement is terminated by the reserved word break, the flow of control will "fall through" to the statement following. In the following example, if the first break is omitted, and the value of UserInput is 'Y', then the program will print "Yes" then "No", rather than "Yes" alone, which would be the intent.

```
#include <iostream>
using namespace std;

void main ()
{
     char UserInput = 'Y';
     switch (UserInput)
     {
         case 'Y':
         case 'y': cout << "Yes";
                   break;  // if this is missing, print "No" too.
         case 'N':
         case 'n': cout << "No";
                   break;
         case 'M':
         case 'm': cout << "Maybe";
                   break;
         default:  cout << "Who knows?";
     }
}
```

The conditional operator ?: simplifies the following common section of code:

```
if (boolVal)
   a = b;
else
   a = c;
```

This can be written as:

```
a = boolVal ? b : c
```

C++ evaluates this by examining the value of the expression to the left of the question mark; if it is true, then the value between the question mark and the colon is evaluated and a is assigned that value, otherwise the value after the colon is evaluated and a is assigned that value instead.

Use of the conditional operator ?: is a matter of taste since it can be precisely translated into an if statement, as shown above.

A.5.2 Looping Structures

C++ has three looping structures: for, while, and do-while. Looping structures cause a statement or collection of statements to be executed zero, one, or more times.

The simplest looping structure is the while loop:

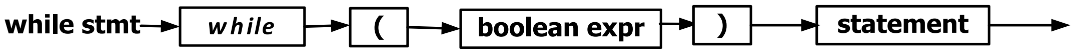

Figure A.16: Syntax for **while** statement.

On entry to a while statement, the Boolean expression is evaluated. If it is true, the statements are executed and, after execution, control returns to the entry of the while statement for another check of the expression; otherwise, the statements are skipped and control passes to the statements following the while statement. If more than one statement is to be executed (which is the usual case), they must be placed in braces.

```
#include <iostream>
using namespace std;

void main ()
{
     int i = 0;
     while (i < 10)
     {
          cout << i << " ";
          i++;
     }
}
```

This program prints the numbers from 0 to 9. Since control passes to the top of the loop after *i*++ is executed, the while loop ends as soon as *i* reaches 10, and so 10 is not printed.

An important point to note about the while loop is that the statements in the loop will not be executed at all if the Boolean expression is false when the while loop is entered. This is called a pre-test loop, since the test is done before the statements in the loop are executed.

If it is desirable that the statements in the loop be executed at least once, we need a post-test loop, and we have one in the do-while loop:

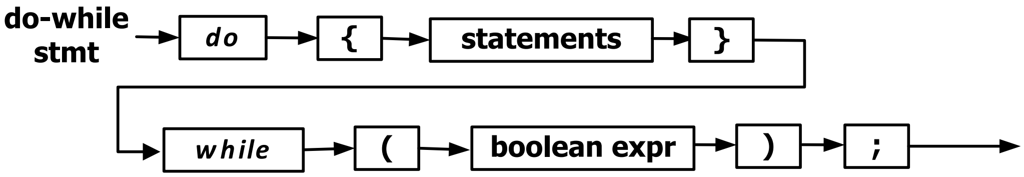

Figure A.17: Syntax for **do-while** statement.

In this loop, the statements are executed once, and then the boolean expression is evaluated. If it is false, control passes to the next statement after the loop; otherwise, the loop is reentered and the statements are executed again.

```
#include <iostream>
using namespace std;
```

```
void main ()
{
     int i = 10;
     do
     {
          cout << i << " ";
          i++;
     }
     while (i < 10);
}
```

The program above prints the number 10. Even though i is not, in fact, less than 10, the loop will be entered once.

The most complicated and powerful of the C++ looping structures is the for-loop.

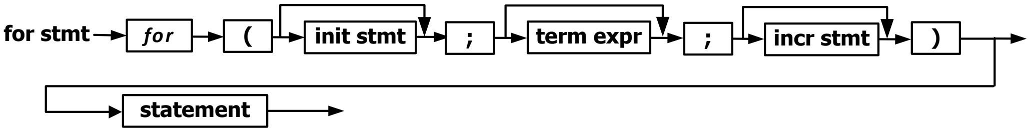

Figure A.18: Syntax for **for** statement.

The for-loop is conceptually equivalent to the following code:

```
initstmt;
while (termexpr)
{
     statements;
     incrstmt;
}
```

The initstmt is the initialization for the loop and usually, therefore, consists of setting the initial value of a loop variable, while the incrstmt increments (or somehow changes) the value of variables such that, eventually, the boolean expression termexpr will become false. There is a peculiarity in the syntax of the for statement: although initstmt and incrstmt may contain multiple statements, such statements must be separated by commas, not by semicolons. Any of the three (initstmt, termexpr, or incrstmt) may be empty, although this would require some other way of terminating the loop, such as break.

```
#include <iostream>
using namespace std;
void main ()
{
     for (int i = 1; i < 11; i++)
     {
          cout << i << " ";
     }
}
```

This program prints the numbers 1 to 10. Notice that we can declare a variable in the initialization of the loop and use it in that loop. The variable *i* declared here will, however, not continue to exist when the loop is exited; it does exist only within the body of the loop.

A.6 Comments

C++ has two forms of comments: /* comments and // comments.

// comments do not extend over multiple lines and have no end marker. The compiler will ignore anything following // on a line as in the following code snippet:

```
void main () // comment out Main
// void Main ()
```

/* comments may extend over multiple lines. They begin with /* and end with */, like this:

```
/* This is a lengthy comment
documenting the algorithm
used in a method. */
```

/* comments cannot be nested; the first */ following a /* ends the comment, regardless of how many times /* appears in the comment.

A.7 Separate Compilation, Modules, and Header Files

C++, like many other languages, allows for separate compilation of "modules": files containing sections of code, which may be only parts of programs. For example, a compiler package will include pre-compiled library modules – chunks of code useful in many different programs. For another example, a team of programmers may write one application by putting each programmer to work on his or her own module, with each module separately compiled into "object file" form (binary form but not executable) and then the whole "linked" together to form one application. Thus, one programmer might work on a module focused entirely on interaction with a database, while another works on a module focused entirely on interaction with the user, and a third works on a module focused on printer control.

One of the advantages of such separate compilation is that object code may be distributed in a usable form, ready to be linked with other code to form an executable program, without the necessity of distributing the source (human-readable) code. However, a disadvantage of such separate compilation is that the programmer intending to use a subroutine which exists in another module may make an error in calling that subroutine: passing the wrong parameters, expecting the wrong return type, or even just misspelling the subroutine name. How does the compiler detect such an error? Conversely, how does the compiler confirm that the call has no error, and thus write correct code to make the call?

Different compilers have different answers to these questions. A language may simply not check. In the absence of other instructions, C will make assumptions about the parameters and return type, and assume that the subroutine name is correct; if the subroutine name is wrong, then the resulting program cannot be linked, but errors in parameters and return type might not be detected in linking and might simply produce erroneous results. Other languages may examine the object code in which the subroutine appears and make sure that call is correct – though this obviously requires that the separately compiled object code of one module must be somehow available during compilation of another module which uses it (some implementations of Pascal behave in this way). Finally, a language may require that information about the subroutine be available, but place the onus on the programmer to provide that information. This is the C++ approach.

A.7.1 Function Prototypes

In order to compile correctly, a call to a function in another module, C++ must know the name of the function, its return type, and its parameters. We can supply the required information, without knowing the function body or even the module in which the function appears, by giving the signature of the function followed by a semicolon rather than a function body. The signature followed by a semicolon is called the prototype of the function. We can use the same technique to allow us to use a function which we have not yet declared – recall that we had previously stated that the *main*() function must come last; this is not true if we use function prototypes. Thus, the following program should not compile[2]:

```
#include <iostream>
using namespace std;

void main ()
{
     func();
}
// ------------------------------------------------------------
void func()
{
     cout << "func()" << endl;
}
```

[2] We say "should not compile" instead of "will not compile" since compilers do differ in rather annoying ways, and one compiler might allow this to pass, whereas others will not. Compilers often have settings governing the strictness with which the syntax is checked, and it may be possible to require strict compliance with standard C++.

However, the program can be repaired by simply adding the prototype of *func*() immediately before *main*():

```
#include <iostream>
using namespace std;

void func ();
void main ()
{
     func();
}
// -------------------------------------------------------------
void func()
{
     cout << "func()" << endl;
}
```

The use of prototypes allows us to solve any ordering problems we might have, as when function *A*() calls function *B*() and function *B*() calls function *A*(). The reader should confirm that no ordering will solve this problem, but the use of prototypes will. Also, although it is not required, it is permissible to place "void" between the empty parentheses in the prototype of a method that has no parameters.

A.7.2 Header Files

At long last, we are able to explain the "#include <iostream>" line at the beginning of each of our sample programs. Prototypes for functions and, as we shall see, declarations of classes in a particular module are conventionally collected in a single file with the same name as the module. This file is called a header file. Instead of laboriously typing each prototype into a program that uses that module, we can simply include the header file in the program.

The line "#include <iostream>" is a command to the compiler to include the file iostream in the current file, exactly as if the contents of the file iostream had been typed in the current file[3]. Without going into the gory details, the angle brackets <> tell the compiler to look for the file iostream in the directories set aside for header files which came with the compiler. If the header file name had been in double quotes instead, the compiler would have looked for the file among directories containing user header files.

In this text, we will expect user written header files to be in the same directory as the program, and we will include those using double quotes. For instance:

```
#include "myHeader.h"
```

[3] Or, as the lawyers would put it, this line tells the compiler that the contents of iostream "are made a part hereof as if fully set out herein."

The reader should note that the preprocessor commands are case-sensitive. Thus, "#include <iostream>" iis different from "#include <iostream>". If the underlying operating system has context-sensitive file names, like Unix, these two commands will have different effects. If the underlying operating system does not have context-sensitive file names, the commands will have the same effect.

The file iostream itself contains classes, such as *outstream*, variables, such as *cout* which is an instance of *outstream*, and functions, all for performing input and output.

A.8 Class Definition

Since an object-oriented program consists of interacting objects, obviously a key part of such programming is the definition of classes. The syntax of a C++ class declaration is fairly simple and straightforward:

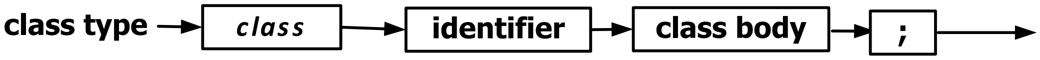

Figure A.19: Syntax for **class** definition.

In the above syntax diagram Figure A.19, the identifier is the name of the class. The class body consists of a number of sections with differing amounts of accessibility to the "outside world" (i.e., other classes and functions outside of classes).

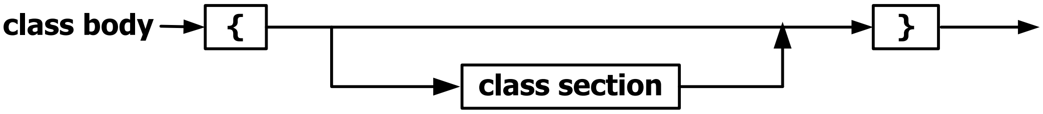

Figure A.20: Syntax for **class** body.

Each section consists of an access modifier and declarations of fields (data contained in an object) and methods (behavior of an object). It may have any number of either fields or methods, and they may be intermixed as is necessary for clarity:

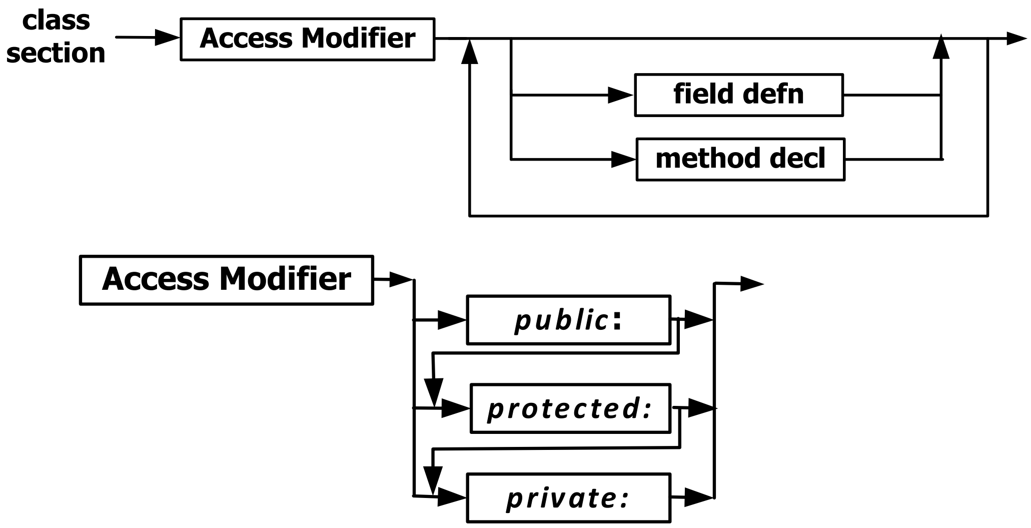

Figure A.21: Syntax for **class** declaration.

The access modifier assists the compiler in enforcing the encapsulation, which is part of object-oriented programming. The access modifiers can be in any order even though in Figure A.21 imposes a particular order (public, protected, and private). It should be mentioned an access modifier can be used only used. If a field or method is in a **public** section of the class, it may be freely accessed by other objects, even those which do not belong to its class. Clearly, this completely defeats the concept of encapsulation, and it is discouraged with regard to fields except in special circumstances.

If a field or method is in a **protected** section of the class, it may be accessed by methods declared in the class in which it is declared and by methods declared in subclasses of that class, but not by methods of any other class. This partially defeats the concept of encapsulation, and is also discouraged with regard to fields to some extent.

If a field or method is in a **private** section of the class, it may be accessed *only* by methods declared in the exact class in which it is declared (i.e., not by methods declared in any other class, not even subclasses of that class). This is usually the best choice for the access modifier for a field, as a private field is entirely encapsulated in the object.

Our First Class

Examining the syntax diagrams for a class definition, we can see that the only required parts of a class definition are the word "class", the identifier, the braces, and the final semicolon. So the simplest possible class definition is:

```
class Tester
{
};
```

We type this into a file called "Tester.cpp" and it compiles (it doesn't run, of course, because we have no *main*() function). We have created our first class! Of course, it is a fairly nondescript class, but the longest journey begins with a single step.

A.8.2 Field Declaration

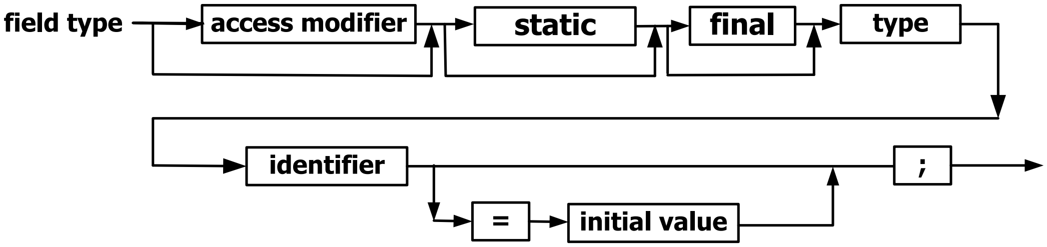

Figure A.22: Syntax for **field** declaration.

The above syntax for declaration of a field resembles the declaration of a variable. The word "static" is present for a class field (discussed below) and absent for an instance field. Following this word (if it is present), the type of the field is declared, then the field is named by an identifier, and finally the field is given an initial value, if desired.

Each instance (object) of a class has its own copy of the instance fields declared in the class. The instance fields of one instance are entirely separate and distinct from those of any other instance. They may have the same values, but that is mere chance, just as several integer variables may have same value without, thereby, having any special relationship.

It may be desirable to have a variable which applies to a class as a whole, for example, a count of instances created so far. Clearly, an instance field cannot fill this need, and so we need a class field, which is declared by including the word "static" in the declaration of the field. The word "static", preceding a field declaration indicates that the field belongs to the class, not to any particular instance of the class. The word "static" was chosen for historical reasons, and perhaps it would have been better to use a word like "classlevel", but we are stuck with the terminology we have. A static field is rather like a global variable, but access to it is strictly controlled by the class to which it belongs.

Note that the diagram above shows no way of giving the variables an initial value. This is no accident; an instance field is initialized in the constructor, while a class field must be defined and initialized *outside* of the class declaration which we are currently discussing. The definition and initialization of a class field are as follows:

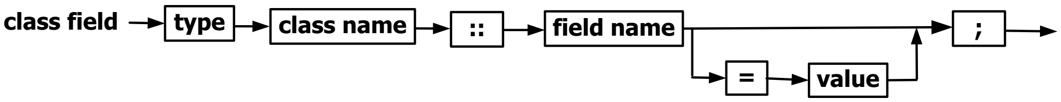

Figure A.23: Syntax for definition and initialization of class field.

Accessing a Field

To access a field of an instance of a class, we write the name of the instance, followed by a period (dot), and followed by the name of the field. We might therefore write *a.b* to refer to the *b* field of an object referred to by *a*. The static fields of a class may be accessed either exactly like an instance field or by writing the name of the class, followed by two colons, followed by the name of the field. Of course, we cannot access a field at all unless its access modifier is such as to allow such access. Putting all that we have said about classes together, we have the following example:

```
#include <iostream>
using namespace std;

class Tester
{
     public:
         static int classVar;
         int instanceVar;
};
// ------------------------------------------------------------
int Tester::classVar = 10;
// ------------------------------------------------------------
void main()
{
     Tester tester1;
     Tester tester2;
     tester1.instanceVar = 20;
     tester2.instanceVar = 40;
     cout << "Instance1: " << tester1.instanceVar << endl;
     cout << "Instance2: " << tester2.instanceVar << endl;
     cout << "Class through instance: " << tester1.classVar << endl;
     cout << "Class: " << Tester::classVar << endl;
}
```

Note that we have referred to *classVar* in both possible ways. Note also that we have made all of our fields public – this is normally deplored, but at this time we have not described how to write methods or constructors, which makes it rather difficult to give the fields values or to print them unless they are public. Finally, note that once we have declared the Tester class, we can declare a variable of type Tester as readily as we could declare a variable of type int.

The output from this program shows us that the two instances have separate instanceVar fields:

```
Instance1: 20
Instance2: 40
Class through instance: 10
Class: 10
```

A.8.3 Method Definition

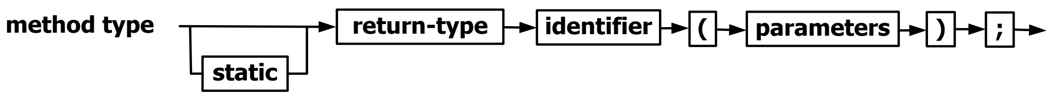

Figure A.24: Syntax for method definition.

Just as field declarations resemble variable declarations, method declarations resemble function declarations. Methods can be also defined as being static (i.e., class level). A method may also be virtual, but for now we will omit that detail.

Although public access to fields is normally deplored, public access is entirely permissible for a method; public methods constitute the interface between the object and other objects. Such methods should not depend too much on the underlying implementation, however, since one of the goals of object-oriented programming is to allow changes to the underlying implementation without forcing changes to the interface. "Helper" methods, which assist the public methods but directly reflect the implementation, should generally be either private or protected. When a public method is called on a object it is sometimes referred in the object oriented community as "sending a message" to that object.

A method may be declared to be static, in which case it can be called without reference to any particular instance of the class; indeed, it can be called even if no instance of the class exists at all. A static method can access static fields of the class (since static fields exist even if no instance of the class exists) but cannot access non-static fields, because there might be no instances of the class, and therefore no non-static fields, when the static method is invoked.

The parameters list of a method is identical to that of a function, with one invisible exception: every instance method of a class has, as one of its parameters, a hidden parameter: *this*. The *this* parameter always refers to the instance which will carry out the method. Non-static methods are allowed to refer to non-static fields without prefixing the name of the instance, but they can also use *this* as the name of the instance. If a method needs to pass to some other method a reference to the instance, it can pass *this*.

The Method Body

Of course, it is all very well to speak of calling methods, but since the above diagram does not offer any position for the method body, it is not clear what we would be calling, or why we would want to do so.

The body of the method appears *outside* of the class declaration we have been describing, just as did the class variable definition[4]. The syntax for the method definition is what you might expect based on the syntax for the class variable definition:

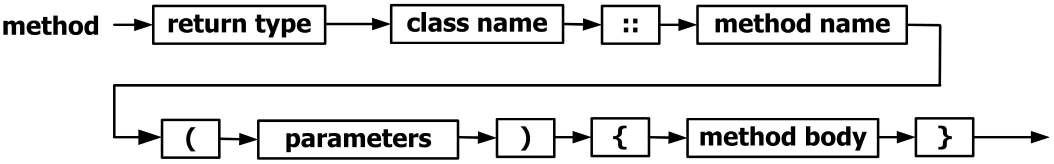

Figure A.25: Syntax for method definition.

The method body, like a function body, may contain local variable definitions, assignment statements, and flow of control statements, function and method calls, and all the panoply of procedural programming. However, as noted, a method body has an invisible *this* parameter and a method may access protected or private members of its class.

Methods are invoked in a manner very similar to accessing the fields of a class. Some in the object-oriented community would equate method invocation to "sending a message" to the class object. Demonstrating the writing and use of methods, we have our old friend the Tester class:

```
#include <iostream>
using namespace std;
```

[4] The method body can, in fact, appear inside of the class declaration. If it does, it is (probably) compiled as "in-line code" for reasons of efficiency. We are firmly resisting the impulse to burden the reader with the intricacies of C++, and for our purposes in-line code is not necessary.

```
class Tester
{
     protected:
        static int classVar;
        int instanceVar;
     public:
        void setInstance (int v);
        void printInstance ();
        static void printClass ();
};
// ==============================================================
int Tester::classVar = 10;
// --------------------------------------------------------------
void Tester::setInstance (int v)
{
     instanceVar = v;
}
// --------------------------------------------------------------
void Tester::printInstance ()
{
     cout << instanceVar << endl;
}
// --------------------------------------------------------------
void Tester::printClass ()
{
     cout << classVar << endl;
}
// ==============================================================
void main()
{
     Tester tester1;
     Tester tester2;
     tester1.setInstance (20);
     tester2.setInstance (40);
     tester1.printInstance();
     tester2.printInstance();
     tester1.printClass();
     Tester::printClass();
}
```

The output from this program, though somewhat less informative than that of the previous version, shows that the calls to instance methods did indeed affect the individual instances:

```
20
40
10
10
```

A.8.4 Constructors

As noted before, we cannot initialize the fields of an object as we did local variables. If we do not want to initialize them through methods such as *setInstance*(), above, we need a constructor method. A constructor method is similar to other methods except that it has no return type and its identifier is identical to the identifier of the class. Furthermore, constructors can be, and usually are, overloaded so that instances of a class can be customized to varying degrees. A constructor initializes the fields and does any other setup required (for example, it might allocate memory or open files). An instance of a class is constructed by writing the variable declaration followed by parentheses and the constructor's parameters. For example, let us abandon our trusty Tester class and work on a Point class, which will hold x and y coordinates:

```
#include <iostream>
using namespace std;

class Point
{
protected:
     double _x;
     double _y;
public:
     Point ();
     Point (double x, double y);
     void display ();
};
// =============================================================
Point::Point ()
{
     _x = 0;
     _y = 0;
}
// -------------------------------------------------------------
Point::Point (double x, double y)
{
     _x = x;
     _y = y;
}
// -------------------------------------------------------------
void Point::display ()
{
     cout << "x = " << _x << "  y = " << _y << endl;
}
// =============================================================
void main ()
{
     Point p1;  // constructor with no arguments
     Point p2 (5, 10);
     p1.display ();
     p2.display ();
}
```

When we compile and run this class, we get the following output:

```
x = 0  y = 0
x = 5  y = 10
```

This output demonstrates that the fields of the two instances were indeed initialized. If we do not design a constructor for a class, C++ includes a default constructor which does nothing to initialize any of fields, so they have random values. We can see this with our very first Tester class if we display *instanceVar* before assigning it a value.

Above, we have assigned values to the instance variables within the body of the constructor. It is also possible to call the constructors of the instance variables and construct them with the correct variables. An example follows:

```
Point::Point () : _x(0), _y(0)
{
}
// ---------------------------------------------------------
Point::Point (double x, double y) : _x(x), _y(y)
{
}
```

In the above example, the syntax _x(0) as part of the empty constructor is initializing the class variable _x to a value of 0. The syntax _x(x) in the non-empty constructor assigns a value of x (that is passed when the Point object is created) to the class variable _x.

Additionally, although a double does not belong to a class type in the same sense as a Point instance belongs to the Point class, the compiler is aware of the "constructor" for a double. Normally it constructs a double by doing nothing to the memory location that represents the double. The special syntax here instructs the compiler to construct the double by copying the given value into the memory location that represents the double. We can call the constructors for any instance variables in this way. This becomes important with more complex structures.

A.8.5 Instances As Parameters

We have previously discussed the difference between pass-by-copy parameter passing and pass-by-reference parameter passing. We can readily have an instance parameter using pass-by-reference:

```
void pointFunc (Point& p)
{
     p.display();
}
```

Here, *p* is just another name for an area of memory already set aside for an instance of the class Point, and is handled like any other name for an instance of class Point.

But consider this function:

```
void pointFunc2 (Point p)
{
     p.display();
}
```

This time, as previously discussed, *p* is a full-fledged variable; memory is set aside for it and the field values from another Point are copied into it. That is, *p* is an instance of Point, and is created with a copy constructor (to be discussed). The Point instance *p* is destroyed just like a local variable when *pointFunc*2() terminates. We shall see, in the section below on pointers, that pass-by-copy is considerably more of a problem than pass-by-reference.

A.8.6 The Class and Its Header File

Although we shall see an exception later, when we discuss templates, normally a class is divided into two files, the header file and the class file. The header file contains the declaration of the class, and the class file contains the implementation of the class. The class file includes the header file, and is separately compiled. Any program that uses the class includes the header file; the class file need not be recompiled, and will be linked with the program.

For example, we can divide the Point class into two files as follows:

```
// /////////////////////////////////////////////////////////////
// Point.h
// /////////////////////////////////////////////////////////////
#include <iostream>
using namespace std;

class Point
{
friend ostream& operator << (ostream& s, Point& p);
protected:
    double _x;
    double _y;
public:
    Point ();
    Point (double x, double y);
    void display ();
    Point operator+ (Point b);
};
```

```
// /////////////////////////////////////////////////////////////
// Point.cpp
// /////////////////////////////////////////////////////////////
#include "Point.h"
// =============================================================
Point::Point ()
{
    _x = 0;
    _y = 0;
}
// ------------------------------------------------------------
Point::Point (double x, double y)
{
    _x = x;
    _y = y;
}
// ------------------------------------------------------------
void Point::display ()
{
    cout << "x = " << _x << "  y = " << _y << endl;
}
// ------------------------------------------------------------
Point Point::operator+ (const Point& b)
{
    return Point (_x + b._x, _y + b._y);
}
 // ============================================================
ostream& operator << (ostream& s, Point& p)
{
    s << "(" << p._x << "," << p._y << ")";
    return s;
}
```

Now the program to test the *Point* class becomes extremely short:

```
// ////////////////////////////////////////////////////////////
// Main.cpp
// ////////////////////////////////////////////////////////////

#include <iostream>
using namespace std;

#include "Point.h"

void main ()
{
     Point p (1, 2);
     cout << "The point is " << p << "." << endl;
}
```

The reader should try the separate compilation of `Point.cpp` and `Main.cpp`, linking them together in accordance with the requirements of the reader's compiler.

A.9 C++ Libraries

C++ has quite extensive libraries, which offer a great deal of functionality. It has classes for file access, for input and output, and so on. Some of the functions, variables, and classes from the libraries will be discussed and used in this text but, since a detailed explanation of all of the libraries runs for hundreds of pages, no such explanation will be attempted here. The reader is advised to obtain access to a C++ library reference.

A.9.1 The *ostream* and *istream* classes

We provide a necessarily brief introduction to the *`istream`* and *`ostream`* classes, which will be used in the text for input and output.

ostream

We have seen the use of *`cout`* to output information for the user. The variable *`cout`* is a predefined variable of class *`ostream`*, which is declared in *`istream`*. It allows the program to access the *standard output*, which is normally the screen, but may be redirected to a file or another program using I/O indirection under DOS or Unix. Any predefined type (such as int, char, float, etc.) may be output by simply placing it to the right of the overloaded << operator used with *`cout`*. A user-defined type, such as *`Point`*, above, can be output only if an overloaded << operator is defined for it. Depending on your compiler, it may or may not be possible to use an overloaded << operator which works with an object of a parent type.

We have already seen that the return type of the overloaded << operator with *`ostream`* is itself *`ostream`*, which allows us to chain uses of << together like this:

```
cout << "The point is " << p << "." << endl;
```

Output with class *`ostream`* can be modified with various manipulators, of which the only one used in this text is *`endl`*. The *`endl`* manipulator, as the reader may have deduced, ends a line by sending a carriage return or carriage return/line feed pair. Other manipulators which may be of interest to the reader are *`setw`*`()`, which sets the width of the following field, and *`setprecision`*`()`, which sets the precision of the floating-point conversion to the specified number of digits (these manipulators, and others, are contained in *`iomanip`*). Thus, we can format output like this:

```
#include <iostream>
#include <iomanip>
using namespace std;

void main ()
{
     cout << setw(8) << 200 << setw(8) << 10000
          << setw(8) << 5 << endl;
     cout << setw(8) << 1200 << setw(8) << 3 << setw(8)
          << 50000 << endl;
}
```

This produces the following output, aligned in columns:

```
     200   10000       5
    1200       3   50000
```

ofstream

Output to the standard output is adequate for most purposes of this text, but output to a file is sometimes desirable. For that purpose, we need to use an *ofstream* instance. The *ofstream* class is a subclass of *ostream* which allows sequential access to a file, and is declared in fstream. We can create an *ofstream* instance, connect it to a file, and specify the mode of access. The mode of access is specified using one or more of these flags (there are others):

ios::app	Appends data to the file.
ios::binary	Opens the file in binary mode (as opposed to text mode).
ios::in	Opens the file for reading.
ios::nocreate	Opens the file only if it exists, not creating it.
ios::out	Opens the file for writing.

Figure A.26: Various mode of access in ofstream.

Thus, to open a file, creating it if necessary, and appending to it if it already exists, we might use the following code:

```
#include <fstream>
#include <iomanip>

void main ()
{
     ofstream outs ("outfile.txt", ios::app);
     outs << setw(8) << 200 << setw(8) << 10000
          << setw(8) << 5 << endl;
     outs << setw(8) << 1200 << setw(8) << 3 << setw(8)
          << 50000 << endl;
     outs.close();
}
```

If we had specified the following flag, any existing file would have been destroyed before the data was written:

```
  ofstream outs ("outfile.txt", ios::out);
```

If we had specified the following flags, data would have been appended to an existing file, but no new one would have been created:

```
  ofstream outs ("outfile.txt", ios::app | ios::nocreate);
```

Note that the *close*() method closes the file so that the data written to it is duly saved when the program ends. If the *close*() method is not called, the *ostream* object automatically closes the file when it goes out of scope that is when it exits the method in which it is declared and opened.

If we had not wished to open the file immediately, we could have declared the stream instance and opened it later, as follows:

```
#include <fstream>
#include <iomanip>

void main ()
{
     ofstream outs;
     outs.open ("outfile.txt", ios::app);
     outs << setw(8) << 200 << setw(8) << 10000
         << setw(8) << 5 << endl;
     outs << setw(8) << 1200 << setw(8) << 3 << setw(8)
         << 50000 << endl;
     outs.close();
}
```

We can test for an error on opening by testing whether (*outs*) is true, modifying the above function as follows:

```
#include <fstream>
#include <iomanip>

void main ()
{
     ofstream outs;
     outs.open ("outfile.txt", ios::app);
     if (outs)
     {
        outs << setw(8) << 200 << setw(8) << 10000
             << setw(8) << 5 << endl;
        outs << setw(8) << 1200 << setw(8) << 3 << setw(8)
             << 50000 << endl;
        outs.close();
     }
     else cout << "Error on opening << endl";
}
```

istream

Just as we can output all the predefined types to standard output using *cout* and <<, we can input all the predefined types from standard input using *cin* and >>. Thus, to read three integers from standard in and output them again, we would do the following:

```
#include <iostream>
#include <iomanip>
using namespace std;
```

```
void main ()
{
     int i1, i2, i3;

     cin >> i1 >> i2 >> i3;
     cout << setw(8) << i1 << setw(8) << i2
          << setw(8) << i3 << endl;
}
```

There are several points to notice about the use of *cin* and >>. One is that any "whitespace" separator between the integers is acceptable, whether spaces, tabs, or carriage returns. Thus any of these inputs is equally acceptable for this program:

```
1 2 3
```

```
1
2
3
```

```
1
2          3
```

Another point is that non-numeric input causes the input statement to fail to set the value of the variables.

Now, what if we don't want to extract one item at a time; suppose we want an entire string? For that, we can use *get*:

```
#include <iostream>
#include <iomanip>
using namespace std;

void main ()
{
char buff [50];
char c;

     cin.get (buff,50);
     cout << buff;
}
```

Here, the call to *get*() inputs characters from standard input to the string represented by *buff* until 49 characters are read, or a carriage return is encountered, whichever comes first.

If we wish to continue to obtain input until all input is exhausted, we can use the *eof*() function:

```
#include <iostream>
```

```
#include <iomanip>
using namespace std;

void main ()
{
     int i;
     int total = 0;

     do
     {
        cin >> i;
        total += i;
     } while (!cin.eof());
     cout << total << endl;
}
```

ifstream

Just as *ofstream* corresponds to *ostream* and provides file output, *ifstream* corresponds to *istream* and provides file input. The *ifstream* class has *open*() and *close*() methods, and uses the same flags for mode of access. An example of using the *ifstream* class is as follows:

```
#include <fstream>
void main ()
{
     int i1 = 1, i2 = 2, i3 = 3;
     ifstream infile ("infile.txt");

     infile >> i1;
     infile >> i2;
     infile >> i3;
     cout << i1 << ", " << i2 << ", " << i3 << endl;
     infile.close();
}
```

A.9.2 The Standard Template Library

The *istream* and *ostream* classes are in fact part of the C++ Standard Template Library. The Standard Template Library (abbreviated STL) is not part of the C++ language itself, but should be available with any compiler. The STL implementation for a given compiler provides high-quality, well-tested implementations of the streams previously described, the string class which is missing from the C++ language definition, many data structures including some of those described in this book, and more. The STL provides much larger "building blocks" than does C++, and you can construct a program more quickly using such building blocks than you can by constructing each block (such as the Stack class described herein) individually.

The STL makes extensive use of "templates" as should be obvious from its name. As templates are a very important part of the C++ language for purposes of this book, the discussion of templates was not relegated to this Appendix, but instead appears in Section 1.4.4.

Using the STL is fairly simple. Suppose you need a vector of integers, for instance. You particularly want to be able to resize your vector as you find you need more data, without losing anything already entered. You could use the C++ array type, but that is not resizable. To resize, you would be forced to create a new array and copy over your old data. It would be preferable to have a data structure that would handle that automatically, behind the scenes. The STL has implemented such a data structure; the type is *vector*. The STL *vector* type, like all STL container types, can report its size and has methods to set elements. It can also resize itself to hold more values, and when it does so, it automatically copies over all the old data as expected. The index operator (denoted operator[]) is overloaded to provide natural access to the elements. Below is an example program that uses an STL *vector*. The line "using namespace std;" is required since STL types exist in their own namespace. The "namespace" is a C++ concept to address the problem of name-space collision (suppose I have a type named *vector* and the STL has a type named *vector*; how can I use them both in the same program?). Since this Appendix does not attempt to teach the entire C++ language, we will not go further into these details.

```
#include <iostream> // for i/o functions
#include <vector>      // for vector
using namespace std;

void main ()
{
     int i = 0;
     vector<int> vectorInts;

     // create vector with 10 values and set values
     vectorInts.resize (10);
     for (i = 0; i < vectorInts.size(); ++i)
                       vectorInts[i] = i;
     cout << "Vector with 10 values" << endl;
     for (i = 0; i < vectorInts.size(); ++i)
                       cout << vectorInts[i] << ", ";
     cout << endl;
     // expand vector to 20 values; additional 10 values auto-initialize
     vectorInts.resize (20);
     cout << "Vector with 20 values but only first 10 set" << endl;
     for (i = 0; i < vectorInts.size(); ++i)
                       cout << vectorInts[i] << ", ";
     cout << endl;
     // overwrite all values
     for (i = 0; i < vectorInts.size(); ++i)
                       vectorInts[i] = i * i;
     cout << "Vector with 20 values" << endl;
     for (i = 0; i < vectorInts.size(); ++i)
                       cout << vectorInts[i] << ", ";
     cout << endl;
}
```

The output of the above program is given below.

```
Vector with 10 values
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Vector with 20 values but only first 10 set
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
Vector with 20 values
0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225,
256, 289, 324, 361,
```

All STL containers have the ability to return an iterator (the corresponding concept is generally called an "enumerator" in this book). An iterator either points to an element or is a reserved value indicating the end of the iteration. In certain STL containers, we can obtain an iterator by searching for a particular value; in the *vector*, we can obtain an iterator pointing to the first element or indicating the end of the iterator. The increment operator (operator++) is always overloaded for iterators, so we can step through a vector by starting at the beginning and going to the end. We can substitute the following code that uses an iterator for the code above resulting in an identical output:

```
// cout << "Vector with 20 values" << endl;
// for (i = 0; i < vectorInts.size(); ++i)
//                         cout << vectorInts[i] << ", ";
// cout << endl;

   cout << "Vector using iterator" << endl;
   vector<int>::iterator pVInt = vectorInts.begin();
   const vector<int>::iterator pVIntEnd = vectorInts.end();
   for (  ; pVIntEnd != pVInt; ++pVInt )
   {
       cout << *pVInt << ", ";
   }
   cout << endl;
```

The STL *list* type is also available, implementing a doubly-linked list. In this case we cannot access elements using the index operator, nor do we have a resize method (it would be meaningless), but we can add elements to a list, get its length, and display its values using an iterator. The code below to display the values is identical to that for a *vector*, but for the substitution of names, even though the underlying data types are quite different.

```
#include <iostream> // for i/o functions
#include <list>
using namespace std;

void main ()
{
    list<int> listInts;

    for (int i = 0; i < 10; ++i) listInts.push_back(i);
    cout << "List with 10 values using iterator" << endl;
    list<int>::iterator pLInt = listInts.begin();
    const list<int>::iterator pLIntEnd = listInts.end();
    for (  ; pLIntEnd != pLInt; ++pLInt )
    {
        cout << *pLInt << ", ";
    }
    cout << endl;
}
```

The containers in the standard template library such as vector and list discussed above have the following operations in common.

Operation	Description
=	Replaces the contents of the container in the left hand side of the assignment statement with the one from the right hand side.
==	Returns true if the container in the left hand side of the assignment statement contains the same data in the same order as the container in the right hand side.
empty()	Returns true if the container has no entries.
size()	Returns the number of items the container contains.
max_size()	Returns the maximum number of items the container can store.

A.10 Pointers and Memory Management

The variables that you declare in your program have to be stored somewhere in the memory of the computer. Pointers give direct access to the memory itself, and make C++ a very powerful language – but at the same time a somewhat dangerous language. This brief introduction gives you an idea of where variables are stored, how they are accessed using pointers, and some of the pitfalls that you may fall into with respect to pointers.

In C++, there are two kinds of variables: *static* and *dynamic*. Static variables are those that are created when the program begins and exist for the entire life of the program. Note that variables of type *static* defined earlier are different from the *static* variable concept described here. We will deal very little with static variables in this text, so we will just briefly state that memory for such variables is set aside for them at the same time that memory is set aside for the executable code (that is, your program) itself.

Dynamic variables include all other variables in your program, in particular the local variables that appear in your functions and methods and the actual parameters in a function or method call. Dynamic variables will be stored either in the *stack* or in the *free store*.

A.10.1 The Stack

The stack is a *limited* amount of memory set aside for the specific purpose of managing function calls. It is limited because increasing the amount of memory allocated to your program will not increase the size of the stack. In languages of higher level than assembly language, such as C++, the stack is not under the direct control of the programmer, and is changed only by function or method calls, as described here. Although we are not normally conscious of the stack, it is *critical* to the operations of the computer. The very last thing you want to do is to trash your stack!

The stack has, in addition to the memory itself, a stack pointer. This pointer holds the address of the next location into which values will be placed. Depending on the system, the stack pointer could start with the maximum value, decrease as each value is placed on the stack, and increase as each value is removed, or it could start with the minimum value, increase as each value is placed on the stack, and decrease as each value is removed. Your program will change this pointer when function (or method) calls occur. It is, in fact, the stack which allows the use of recursive functions.

When a function (or method) call occurs, three types of information must be placed on the stack: the actual arguments, the return address, and the local variables for the function.

A.10.2 Pointers

Now, the discussion of the stack is all very interesting, and perhaps gives a new perspective on function calls, but, as has been pointed out, we cannot change the workings of the stack; it works whether we understand it or not.

As we know, every variable name is really a symbolic reference for the address of a certain amount of memory. Sometimes it is useful to have a way of referring to a memory address without giving it a specific name. A **pointer** is a variable that contains a memory address. It is called a pointer because we can think of it as an arrow pointing to the variable that actually occupies that address.

Each pointer variable “points” to a particular type of variable. For instance, we may have a “**pointer to int**”, which holds the address of memory which is occupied by an int variable. Such a pointer is declared as follows:

```
int* pInt;
```

We can read “int*” as “pointer to int.” Now, it is all very well to have a variable that will hold a memory address, but how do we get such an address? We can acquire the address of memory occupied by a variable by prefixing the variable name with an & sign. Conversely, we can access the memory pointed to by a pointer variable by preceding pointer variable name with a *. This process is called *dereferencing*. In the Figure A.27 the values in various memory locations after the execution of the assignment statements below the boxes is shown. After the creation of the pointer variable pInt, the box pInt contains a value of 100000AC which a memory location initialized randomly at execution time.

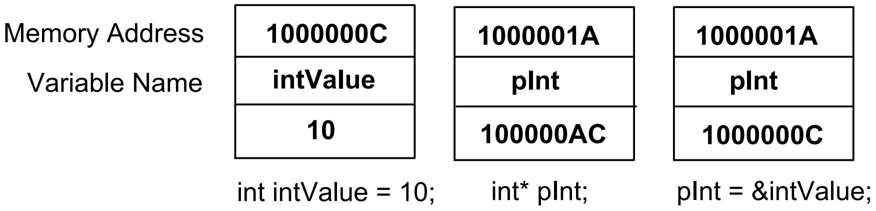

Figure A.27: Pointer variable and initialization

We pointer variables, initialization, and dereferencing with a simple example:

```
#include <iostream>
using namespace std;

void main ()
{
     int iVal1 = 10;       // a normal int var
     int iVal2 = 20;       // another normal int var
     int* pInt;            // a pointer to int - right
                           // now it doesn’t point to anything
     cout << iVal1 << endl;    // outputs 10
     cout << iVal2 << endl;    // outputs 20
     pInt = &iVal1;        // now pInt contains the address of the memory
                           // referred to by iVal1
     *pInt = 50;           // *pInt  is a reference to an int, just like
                           // iVal1; since *pInt and iVal1 refer to the
                           // same memory address, changing the value
                           // of *pInt will also change the value of iVal1
     cout << iVal1 << endl;
                           // This outputs 50 rather than 10
     pInt = &iVal2;        // now pInt contains the address of the memory
                           // referred to by iVal2
     *pInt = -10;          // now *pInt and iVal2 refer to the same
                           // memory address, so changing the value
                           // of *pInt will also change the value of iVal2,
                           // but not the value of iVal1
     cout << iVal2 << endl;
                           // This outputs -10 rather than 20
     cout << *pInt << endl;
                           // This outputs -10 also
}
```

A.10.3 NULL Pointers

Every pointer variable always contains some value. If the variable is not initialized, that value is random, based on whatever bytes happened to be in memory at the location where the variable is placed. If we know what value we need for the variable, we can initialize it to that value. If we do *not* know the value we need when the variable is created, we can initialize it to the special value *NULL*. The C++ language recognizes that a NULL pointer is invalid, and if you attempt to use it, as described below, the program will halt with an error. Additionally, you yourself can test a pointer variable to see if it is NULL and thus does not point to a valid, allocated memory location; there is no other “special” pointer value that can be identified as not pointing to a valid location. Thus, the NULL value is extremely useful, and we shall see many uses of it in this text.

Now that we know how to store the address of a variable, we shall expand our horizons and consider the free store.

A.10.4 The Free Store

The free store consists of all memory allocated to your program but not set aside for purposes such as code, static data, and stack. Thus, the more memory is allocated to your program, the more free store you have, so the free store is not limited in the same sense as the stack. You get memory from the free store for your objects by making calls to *new*. There are other ways of getting memory from the free store, but this is the only one we will deal with.

A call to *new* instructs the computer to set aside some memory from the free store and make a note that that memory is allocated, that is, in use. Different programming languages, and even different compilers for the same programming language, will have different ways of setting aside the memory and recording that it is allocated, so we will not go into any details about this operation. The *new* call returns the address of the first byte of the allocated memory, and you normally assign that address to a pointer variable. Storing addresses is, after all, the function of pointer variables.

As we have mentioned repeatedly, C++ does not automatically free memory that you have allocated with *new*. Thus, if you allocate over and over, but never deallocate, you will in time run out of free store and your program will not function correctly. You deallocate memory using *delete*. If you are deallocating an array (to be discussed in the next chapter), you deallocate its memory using *delete*[]. The following example shows the need for *delete*, assuming that we had declared a class MyObject:

```
void example ()
{
     MyObject* m = new MyObject;
     // the new call allocates some memory from the
     // free store, enough to hold an instance of
     // MyObject; then it returns the address of
     // that memory.  That address is stored in m.
// . . . code
}
```

It is very important to distinguish between *m*, which is the pointer variable, and the memory to which it points, which is in fact nameless. When the function terminates, *m* will be removed from the stack because *m* is a local variable. However, the memory allocated in the free store by the *new* call, with an address stored in *m*, will **not** be automatically deallocated[5].

It is the programmer's responsibility to take care of that memory, and if the programmer does not do so (as we did not), then the program has a memory leak. It is not conceptually difficult to deallocate this memory; we need only include this statement before the end of the function:

```
delete m;
```

This statement says "find the memory pointed to by *m* and deallocate it." The problem with which we must deal throughout this text is ensuring that we delete when we should and do not delete when we should not. We shall see that this can be a difficult problem.

A.10.5 Pitfalls of Memory Allocation

There are a number of possible pitfalls in memory allocation, which any C++ programmer must be conscious of and try to avoid.

[5] The compiler can handle *m* automatically because it knows that the programmer does not want to continue to use an area of the stack that is below the stack pointer; it does not know whether the programmer intends to use the memory allocated on the free store somewhere else.

Memory Leaks

A call to *new* should always be matched somewhere with a call to *delete*. This is not to say that there must be one reference to *delete* in your code for every reference to *new*. For instance, memory might be allocated in a switch statement that contains multiple *new* statements of which only one will actually be executed. In such cases only one *delete* would be required to deallocate the memory that was allocated.

If you don't match each *new* call with a *delete* call, you will have memory leaks, or in other words, memory that is allocated but never deallocated. One nice thing about object-oriented programming is that a lot of your memory allocations can be hidden inside of objects, with allocations done in constructors and deallocations done in destructors (described below). This makes it a little harder to produce memory leaks.

Double Deletion

Never delete memory that is already deleted. That is, do not do what is illustrated below:

```
void badPointer1 ()
{
     Point* p = new Point (1, 2);
     Point* q = p;
// . . . code
     delete p;
     delete q;
}
```

Normally you would not make this error as blatantly as above, but in a more complicated function you might have several different circumstances in which you deallocate memory (in a *switch* statement, for instance) and you must be careful not to deallocate twice.

Using Deallocated Memory

Never use memory that has been deallocated. That is, never do what is illustrated below:

```
void badPointer2 ()
{
     Point* p = new Point;
// . . . code
     delete p;
     (*p).y = 5;
}
```

Again, you would probably not make so blatant an error, but it might occur in a more complicated function, or you might even have a slip of the fingers (or mind) and switch the two statements. Curiously enough, whether it produces a problem or not will depend on the compiler. This illustrates the fact that you have to check your code carefully yourself; even a program that runs perfectly on one compiler might still have serious errors.

Using Memory That Was Never Allocated

It is not difficult to use inadvertently a pointer variable that was never initialized to point to any allocated memory. This may not produce a compiler error, and it may not produce a runtime error on a given run – depending on what random value may happen to be in the memory which is set aside for the pointer variable. On the other hand, it may produce dramatic results, like a crashed program, reformatted hard drive, or other disasters. Here is an example:

```
void badPointer3 () //Never do this
{
     int* pInt;
     *pInt = 20;
}
```

Deallocating Memory That Was Never Allocated

Deallocating memory that was never allocated in the first place can produce disasters as well, though normally not as dramatic as actually trying to use the memory. This error can be produced easily too:

```
void badPointer4 () //Never do this
{
     int* pInt;
     delete pInt;
}
```

This list of pitfalls is by no means complete. In this text, we will try to show how to anticipate and avoid such errors in programming.

A.10.6 Function and Method Return Types Revisited

So far, we have used functions and methods with void return types and, occasionally, the int return type. But it is in fact possible to return all sorts of types, even classes, references to instances of classes or primitive types, and pointers.

A.10.6.1 Pointers as Return Types

You can allocate memory in a function with *new* and return a pointer to that memory. Since the memory is allocated on the free store, it will not be affected by the changing stack and stack pointer. It is important, however, to deallocate that memory properly in the calling function. An example of allocating memory in a function appears below:

```
#include <iostream>
using namespace std;
Point* getAPoint ()
{
     Point* p = new Point(1,1);
     return p;
}
// -------------------------------------------------------------
void main ()
{
     Point* point = getAPoint();
     (*point}.display();
     delete point;
}
```

Pitfall: Pointers to Local Variables

As we have seen, the stack pointer moves up and down and values on the stack change with every function call. This means that you never, ever, want to return a pointer to a local variable as the return value of a function. For instance:

```
int* badFunction ()
{
     int iVal = 10;
     return &iVal;
}
```

There are two problems with this function, one bad and one worse. The bad problem is that, after we return from the function, the memory referred to by *iVal* will change with every function call, since *iVal* references some of the memory on the stack. The worse problem is that, if the value of *iVal* is changed, then the value of the corresponding stack bytes are changed too. This will not produce a problem *if* the memory which was once set aside for *iVal* is below the current stack pointer. However, if *iVal* happens to be passed to a function which changes it, then the changes to the stack may occur above the stack pointer, trashing the stack. This can produce truly remarkable results, the least damaging of which is a crashed program.

A.10.6.2 Classes as Return Types

In discussing overloaded operators in Section 1.4.1, we see a case in which a method had a class (Point) as its return type. In that case, the return value could be assigned to another instance of Point or used as the instance in a Point method call, just as a return value of a primitive type could be assigned to a variable of that type, printed out, or used in an expression.

The return type of the overloaded + operator was a Point: not a pointer to a Point or a reference to a Point, but a Point in itself. That Point had to be created by a constructor, as we saw, and it had to be destroyed after the method call terminated. It was not destroyed immediately as the method call terminated, but remained just long enough for its value to be assigned to *p*3. Thus the Point (the return value), was destroyed before any output.

What can you do with a function or method return value that is an instance of a class? Exactly the same things you can do with a function or method return value which is of a primitive type: save its value, as the Point return value was saved in *p*3, or use it immediately in an expression or as a parameter in a function or method call. It can even be used as the instance in a method call.

Consider this *main*() function, which is proper:

```
void main ()
{
     Point p1 (1,2);
     Point p2 (3,4);
     (p1 + p2).display();
}
```

A function or a method that returns an instance of a class cannot be placed on the left hand side of any assignment statement. The following statement is therefore illegal:

```
p1 + p2 = p3;
```

A.10.6.3 References as Return Types

Finally, we could return a reference to an instance of a class (or, for that matter, a primitive type). This necessarily implies that the instance is already in existence somewhere in memory. It also gives almost unlimited access to that instance, since it gives the caller a symbolic name for the address where the instance appears. We could add the following method to Point, for instance:

```
double& Point::getX ()
{
     return _x;
}
```

Note that we do not have to do anything special to the field *_x* in order to return a reference to it. If the *main*() function calls this method, it can access the protected field of the Point instance without the knowledge of the Point instance. In some circumstances, that is acceptable. In others, it is not.

A very important characteristic of a method or function return value that is a reference to an instance is that it can appear on the left hand side of an assignment, so the following code is legal:

```
p1.getX() = 1.5;
```

Pitfall: References to Local Variables:

It has been previously noted that pointers to local variables should never be returned. For the same reasons, references to local variables should never be returned either. The compiler may well catch such errors, and a function such as this may not compile (depending on the compiler):

```
int& badFunction ()
{
     int iVal = 10;
     return iVal;
}
```

A.10.7 Destructors And Memory Management

It was mentioned in section A.4.2 that a local variable has the following properties: (1) it does not exist until the function (or method) which declared it begins operation; (2) it is automatically destroyed when that function (or method) terminates. We have seen local variables that are instances of a class in the example above and the output demonstrates that they were correctly created. But what happens when these variables are automatically destroyed?

As already mentioned, C++ does *not* feature automatic garbage collection. Thus, when an object becomes completely inaccessible, the memory it consumes *cannot* be recovered, and neither can any resources it uses other than memory, such as file handles. We must ensure that such resources are properly released when the object is destroyed, either by using delete or by ensuring that the method or function in which the variable is declared terminates thereby releasing the allocated resources.

For that purpose, we can declare a destructor. A destructor is a method which has a name consisting of "~" plus the class name, takes no parameters, and has no return type. A destructor cannot have parameters or a return type since it is called automatically by the system when an object is destroyed (ruling out parameters), and the system wouldn't know what to do with a return value if it had one. A destructor for a class that participates in a hierarchy must *always* be flagged as "virtual" (this statement will be explained under Inheritance and Polymorphism). We modify the class Point above to show the declaration of a destructor. The definitions of the two constructors and other methods are omitted, along with *main*(), as they are all unchanged.

```
#include <iostream>
using namespace std;

class Point
{
protected:
     double _x;
     double _y;
friend ostream& operator << (ostream& s, Point& p);
public:
     Point ();
     Point (double x, double y);
     virtual ~Point ();
     void display ();
     Point operator+ (const Point& b);
};
// ==============================================================
Point::~Point ()
{
     cout << "Destructor called on Point" << endl;
}
// ==============================================================
void main()
{
     Point p;
            Point q (5,10);
            cout << "p = " << p << ", q = " << q << endl;
}
```

When the program is recompiled and run, we get the following output:

```
p = (0,0), q = (5,10)
Destructor called on Point
Destructor called on Point
```

We can see that both Points were destroyed automatically when the *main*() function terminated, and that the destructor was called on each of them. The same also happens with copies of instances that are created when a copy of an instance is passed as an actual parameter or a copy is returned from a function. The important point to note here is that the class destructor is *always* called whenever an instance is destroyed, whether automatically or through the use of delete.

In this case, the destructor has no real duty, as the program would (and did) run equally well without it. However, when a class includes a pointer field and memory is allocated on the free store with the address stored in the pointer field, the destructor is responsible for deallocating that memory so that the opportunity to use it will not be lost.

A.10.7.1 Destructors and Pointer Fields

For a specific example, suppose we create a class which contains (perhaps among other fields) a pointer to a Point object:

```
class PointPointer
{
friend ostream& operator<< (ostream& s, const PointPointer& pp);
protected:
     Point* p;
public:
     PointPointer (double x, double y);
     virtual ~PointPointer ();
};
```

We will write our constructor to create a Point instance using *new*:

```
PointPointer::PointPointer (double x, double y)
{
     p = new Point (x,y);
}
```

Further, we will implement the ostream << function by outputting the Point:

```
ostream& operator<< (ostream& s, const PointPointer& pp)
{
     s << *(pp.p);
     return s;
}
```

Note that just as *pInt* was a reference to an int variable, *pp.p* is a reference to a Point variable, and we can treat it as such. We can now use the * operator (to dereference) the references and get the values. For example, **pInt* is the integer value stored at address *pInt*. The pointer *pp.p* contains a pointer to a Point object and *(*pp.p*) is the Point object. We need the parentheses to ensure that the * operator is evaluated after the dot operator.

Here, the actual object pointed to by *p* in a PointPointer instance is allocated on the free store by a *new* call in the constructor. Since this object is allocated on the free store, we must be careful to deallocate it as well, at the same time that the PointPointer object is deallocated. So the destructor for the PointPointer class must contain the code to deallocate the object pointed to by *p*. The destructor code will therefore be something like this:

```
PointPointer::~PointPointer ()
{
     delete p;
}
```

Now, let's suppose that we have a pointer, *pp*, to an instance of class PointPointer allocated on the free store, and we dispose of it with *delete*, like this:

```
delete pp;
```

What happens? First, the computer determines that *pp* does, in fact, point to an instance of class PointPointer, so it calls ~*PointPointer*() on that instance. Then ~*PointPointer*() deletes *p*, so the computer determines that *p* is a pointer to an instance of Point and calls ~*Point*() on that instance. Now the memory held by the instance of Point pointed to by *p* can be freed, and then the memory held by the instance of PointPointer pointed to by *pp* can be freed, so all the memory held directly or indirectly has been freed and we have no memory leak.

One delete call or automatic destruction can produce a long chain of alternating delete calls and destructor calls, if the data structure is deeply nested.

It should be obvious that your constructors and destructors should allocate and deallocate, respectively, memory from the free store that your object might need.

A.10.8 Copy Constructors and Overloaded Assignment Operators

We have previously discussed constructors, but there is a particular type of constructor with which we must be concerned because of the pass-by-copy habits of C++, and that is the copy constructor. To motivate the development of the copy constructor, consider the following function, which has as a parameter an instance of the Point class previously presented:

```
void pointFunc (Point a)
{
     a.display();
}
void main ()
{
     Point b (1,1);
     pointFunc (b);
}
```

Given our previous discussion of the pass-by-copy method of parameter passing, it is obvious that *a* must be a copy of *b*, with its own memory set aside for it. How did C++ make the copy? It made a bitwise copy of each byte of *b*, which is fine in this case.

But now suppose we use the PointPointer class instead:

```
void pointPointerFunc (PointPointer a)
{
     a.display();
}
```

```
void main ()
{
     PointPointer b (1,1);
     pointPointerFunc (b);
}
```

Once again *a* is a copy of *b*, and the copy was made bitwise. Therefore, *a.p* and *b.p* point to exactly the same Point instance: the one created by the constructor for *b*. When *pointPointerFunc*() terminates, *a* will be destroyed and the destructor will be called on it. Therefore, *a.p* will be deleted. But *b* still exists and still has a pointer to the now-deallocated memory. It can access and change this memory, and it will try to deallocate it again when *main*() terminates. Our seemingly innocent little function has dropped us into two pitfalls: using deallocated memory and deallocating the same memory twice. The basic problem is *crosslinking* betweeen *a* and *b*: their pointer fields should point to different areas of memory, but instead point to the same area.

In this case, since we don't plan to change *a*, we can avoid the whole problem by declaring the parameter differently: *const PointPointer& a*. But this only avoids the problem in this one case. We would like a more permanent solution, one that will not produce an error even if we absent-mindedly passed a PointPointer instance by copy instead of by reference. To achieve this, we must somehow avoid the bitwise copying of the instance.

A.10.8.1 Copy Constructor

Fortunately, we can avoid bitwise copying by providing a copy constructor. A copy constructor is simply a constructor which has exactly one parameter: a reference to an instance of the class itself. This parameter *must* be a reference; pass-by-copy in a copy constructor will lead to infinite recursion. This parameter is often a constant reference, though it need not be. Generally we have no desire to change the instance being copied in the course of copying, and a constant reference ensures that there are no accidental changes.

The duty of the copy constructor is to make a copy of the instance passed as a parameter, ensuring that crosslinking problems such as we had with *pointPointerFunc*() will not arise. Let us see how we can solve PointPointer's problem with a copy constructor:

```
class PointPointer
{
protected:
    Point* p;
public:
    PointPointer (double x, double y);
    PointPointer (const PointPointer& pp);
    virtual ~PointPointer ();
    void display ();
};
```

```
// ============================================================
// previously defined methods unchanged
// ------------------------------------------------------------
PointPointer::PointPointer (const PointPointer& pp)
{
    p = new Point (pp.p->_x, pp.p->_y);
}
```

Of course, we have a little problem: the fields of Point are protected and the PointPointer class cannot access them. Although there are other ways to tackle this problem, we can kill two birds with one stone and declare a copy constructor for Point also (the modification to the class declaration is omitted):

```
Point::Point (const Point& p)
{
    _x = p._x;
    _y = p._y;
}
```

Now we can just use this copy constructor in PointPointer's copy constructor:

```
PointPointer::PointPointer (const PointPointer& pp)
{
    p = new Point (*(pp.p));
}
```

(Note that we need to use **(pp.p)*, not *pp.p*, because *pp.p* is a pointer to Point and Point's copy constructor expects a Point instead.) The *pointPointerFunc*() above, which started this whole discussion, will now operate correctly. The parameter *a* will have memory set aside for it, and it will be constructed using the copy constructor. It will have its own Point instance, which is a copy of *b*'s Point instance, and it can be safely destroyed.

A good rule of thumb for copy constructors is the following: if your class deallocates any memory in its destructor, it needs a copy constructor.

Copy Constructor in Initialization

The copy constructor can also be used to create an instance that is a copy of another instance simply by passing the correct parameter for the copy constructor:

```
void main ()
{
    Point p (3, 4);
    Point q (p);
    cout << "p = " << p << ", q = " << q << endl;
}
```

The reader can test that *p* and *q* do in fact have the same values. But this code is not quite identical to the combined variable declaration and assignment statement which we have used with primitive types:

```
int a = 5;
int b = a;
```

Fortunately, since the designers of the C++ language recognized our natural desire to use the same syntax for initializing both class instances and primitive types, we can initialize one instance of a class as a copy of another using very similar code[6]:

```
void main ()
{
    Point p (3, 4);
    Point q = p;
    cout << "p = " << p << ", q = " << q << endl;
}
```

Note that the assignment operator = above is not really acting as the assignment operator; it is just part of the alternate syntax for the constructor.[7] If we really want to make an assignment of the value of one instance to another (existing) instance, we need to consider the assignment operator itself.

A.10.8.2 Overloaded Assignment Operator

We saw, in discussing overloaded operators, that there was no difficulty with this assignment statement:

```
p3 = p1 + p2;
```

All three of the variables and the return value of the method (*operator*+) are of type Point. How did the compiler do the assignment? Once again, as in the case of pass-by-copy, it created a bitwise copy, as that is its default behavior. That is fine for class Point, but it is emphatically not fine for class PointPointer. The following *main*() function will fall into the double deallocation

[6] To complete the parallelism, we can initialize primitive types the same way we first initialized the class instances:

```
int a = 5;
int b (a);
```

[7] Furthermore, the use of the assignment operator in a class instance declaration and initialization is not limited to copy constructors; we can always use = followed by a value when the constructor we wish to use takes only a single parameter.

pitfall previously described, as *pp*1 and *pp*2 will be crosslinked because they have pointers to the same Point instance:

```
void main ()
{
     PointPointer pp1 (1,1);
     PointPointer pp2 (2,2);
     pp1 = pp2;
     pp1.display();
     pp2.display();
}
```

What is the solution? It is essentially the same as the solution for pass-by-copy: avoid the default behavior of the compiler by writing a method. In this case, the method is an overloaded = operator, where the sole (visible) parameter is a reference to an instance of the class. This method takes whatever action is necessary to avoid the cross linking problem above. For PointPointer, we could do the following:

```
void PointPointer::operator= (const PointPointer& pp){
     delete p;                 // get rid of the old Point instance
     p = new Point (*(pp.p)); // and make a new one which is a copy
}
```

Now the above *main*() function works properly.

When do we need an overloaded assignment operator? Whenever we need a copy constructor. If one is necessary, so is the other.