INTRODUCTION

TO

C++

A PRACTICAL

APPROACH

1. Introducing C++

Programming is a core activity in the process of performing tasks or solving problems with the aid of a computer. An idealised picture is:

[problem or task specification] - **COMPUTER** - [solution or completed task]

Unfortunately things are not (yet) that simple. In particular, the "specification" cannot be given to the computer using natural language. Moreover, it cannot (yet) just be a description of the problem or task, but has to contain information about how the problem is to be solved or the task is to be executed. Hence we need programming languages.

There are many different programming languages, and many ways to classify them. For example, "high-level" programming languages are languages whose syntax is relatively close to natural language, whereas the syntax of "low-level" languages includes many technical references to the zeroes and ones (0's and 1's, etc.) of the computer. "Declarative" languages (as opposed to "imperative" or "procedural" languages) enable the programmer to minimize his or her account of *how* the computer is to solve a problem or produce a particular output. "Object-oriented languages" reflect a particular way of thinking about problems and tasks in terms of identifying and describing the behavior of the relevant "objects". *Smalltalk* is an example of a pure object-oriented language. C++ includes facilities for object-oriented programming, as well as for more conventional procedural programming.

The Origins of C++

C++ was developed by Bjarne Stroustrup of AT&T Bell Laboratories in the early 1980's, and is based on the C language. The name is a pun - "++" is a syntactic construct used in C (to increment a variable), and C++ is intended as an incremental improvement of C. Most of C is a subset of C++, so that most C programs can be compiled (i.e. converted into a series of low-level instructions that the computer can execute directly) using a C++ compiler.

C is in many ways hard to categories. Compared to assembly language it is high-level, but it nevertheless includes many low-level facilities to directly manipulate the computer's memory. It is therefore an excellent language for writing efficient "systems" programs. But for other types of programs, C code can be hard to understand, and C programs can therefore be particularly prone to certain types of error. The extra object-oriented facilities in C++ are partly included to overcome these shortcomings.

ANSI/ISO C++

The American National Standards Institution (ANSI) and the International Standards Organization (ISO) provide "official" and generally accepted standard definitions of many programming languages, including C and C++. Such standards are important. A program written only in ANSI/ISO C++ is guaranteed to run on any computer whose supporting software conforms to the standard. In other words, the standard guarantees that standard-compliant C++ programs are portable. In practice most versions of C++ include ANSI/ISO C++ as a core language, but also include extra machine-dependent features to allow smooth interaction with different computers' operating systems. These machine dependent features should be used sparingly. Moreover, when parts of a C++ program use non-compliant components of the language, these should be clearly marked, and as far a possible separated from the rest of the program, so as to make modification of the program for different machines and operating systems as easy as possible. The three most significant revisions of the C++ standard are C++98 (1998), C++03 (2003) and C++11 (2011). Of course, it can be a challenging task for software engineers, compiler writers and lecturers (!) to keep track of all the revisions that appear in each major version of the standard.

The C++ Programming Environment

The best way to learn a programming language is to try writing programs and test them on a computer! To do this, we need several pieces of software:

- An editor with which to write and modify the C++ program components or source code,
- A compiler with which to convert the source code into machine instructions which can be executed by the computer directly,
- A linking program with which to link the compiled program components with each other and with a selection of routines from existing libraries of computer code, in order to form the complete machine-executable object program,
- A debugger to help diagnose problems, either in compiling programs in the first place, or if the object program runs but gives unintended results.

An Example C++ Program

Here is an example of a complete C++ program:

```
// The C++ compiler ignores comments which start with
// double slashes like this, up to the end of the line.

/* Comments can also be written starting with a slash
 followed by a star, and ending with a star followed by
 a slash. As you can see, comments written in this way
 can span more than one line. */
```

```
/* Programs should ALWAYS include plenty of comments! */

/* Author: Nifty Global and Univtech Ghana
 Program last changed: September 2013 */

/* This program prompts the user for the current year, the user's
current age, and another year. It then calculates the age
that the user was or will be in the second year entered. */

#include <iostream>

using namespace std;

int main()
{
        int year_now, age_now, another_year, another_age;

        cout << "Enter the current year then press the key Enter.\n";
        cin >> year_now;

        cout << "Enter your current age in years.\n";
        cin >> age_now;

        cout << "Enter the year for which you wish to know your age.\n";
        cin >> another_year;

        another_age = another_year - (year_now - age_now);

        if (another_age >= 0) {
                cout << "Your age in " << another_year << ": ";
                cout << another_age << "\n";
        } else {
                cout << "You weren't even born in ";
                cout << another_year << "!\n";
        }

        return 0;
}
```

Program 1

This program illustrates several general features of all C++ programs. It begins (after the comment lines) with the statement

```
#include <iostream>
```

This statement is called an include directive. It tells the compiler and the linker that the program will need to be linked to a library of routines that handle input from the keyboard and output to the screen (specifically the **cin** and **cout** statements that appear later). The header file "iostream" contains basic information about this library. You will learn much more about libraries of code later in this course.

After the include directive is the line:

```
using namespace std;
```

This statement is called a *using* directive. The latest versions of the C++ standard divide names (e.g. **cin** and **cout**) into subcollections of names called *namespaces*. This particular *using* directive says the program will be using names that have a meaning defined for them in the std namespace (in this case the iostream header defines meanings for cout and cin in the std namespace).

Some C++ compilers do not yet support namespaces. In this case you can use the older form of the include directive (that does not require a *using* directive, and places all names in a single global namespace):

```
#include <iostream.h>
```

Much of the code you encounter in industry will probably be written using this older style for headers.

Because the program is short, it is easily packaged up into a single list of program statements and commands. After the include and using directives, the basic structure of the program is:

```
int main()
{
        First statement;
        ...
        ...
        Last statement;

        return 0;
}
```

All C++ programs have this basic "top-level" structure. Notice that each statement in the body of the program ends with a semicolon. In a well-designed large program, many of these statements will include references or calls to sub-programs, listed after the main program or in a separate file. These sub-programs have roughly the same outline structure as the program here, but there is always exactly one such structure called main. Again, you will learn more about sub-programs later in the course.

When at the end of the main program, the line

```
return 0;
```

means "return the value 0 to the computer's operating system to signal that the program has completed successfully". More generally, *return statements* signal that the particular sub-program has finished, and return a value, along with the flow of control, to the program level above.

Our example program uses four *variables*:

year_now, age_now, another_year and another_age

Program variables are not like variables in mathematics. They are more like symbolic names for "pockets of computer memory" which can be used to store different values at different times during the program execution. These variables are first introduced in our program in the variable declaration

```
int year_now, age_now, another_year, another_age;
```

which signals to the compiler that it should set aside enough memory to store four variables of type "int" (integer) during the rest of the program execution. Hence variables should always be declared before being used in a program. Indeed, it is considered good style and practice to declare all the variables to be used in a program or sub-program at the beginning. Variables can be one of several different types in C++, and we will discuss variables and types at some length later.

Simple Input, Output and Assignment

After we have compiled the program above, we can *run* it. The result will be something like

```
Enter current year then press RETURN.
1996
Enter your current age in years.
36
Enter the year for which you wish to know your age.
2001
Your age in 2001: 41
```

The first, third, fifth and seventh lines above are produced on the screen by the program. In general, the program statement

```
cout <<Expression1<<Expression2<< ... <<ExpressionN;
```

will produce the screen output

Expression1Expression2...ExpressionN

The series of statements

```
cout <<Expression1;
cout <<Expression2;
...
...
cout <<ExpressionN;
```

will produce an identical output. If spaces or new lines are needed between the output expressions, these have to be included explicitly, with a " " or a "\n" respectively. The expression endl can also be used to output a new line, and in many cases is preferable to using "\n" since it has the side-effect of flushing the output buffer (output is often stored internally and printed in chunks when sufficient output has been accumulated; using endl forces all output to appear on the screen immediately).

The numbers in **bold** in the example screen output above have been typed in by the user. In this particular program run, the program statement

```
cin >> year_now;
```

has resulted in the variable year_now being *assigned* the value 2001 at the point when the user pressed ENTER key from the key board after typing in "2001". Programs can also include assignment statements, a simple example of which is the statement

```
another_age = another_year - (year_now - age_now);
```

Hence the symbol = means "is assigned the value of". ("Equals" is represented in C++ as ==.)

Flow of Control

The last few lines of our example program (other than "return 0") are:

```
if (another_age >= 0) {
        cout << "Your age in " << another_year << ": ";
        cout << another_age << "\n";
}
else {
        cout << "You weren't even born in ";
        cout << another_year << "!\n";
}
```

The "if ... else ..." branching mechanism is a familiar construct in many procedural programming languages. In C++, it is simply called an *if statement*, and the general syntax is

```
if (condition) {
        Statement1;
        ...
        ...
        StatementN;
} else {
        StatementN+1;
        ...
        ...
        StatementN+M;
}
```

The "else" part of an "if statement" may be omitted, and furthermore, if there is just one *Statement* after the "if (condition)", it may be simply written as

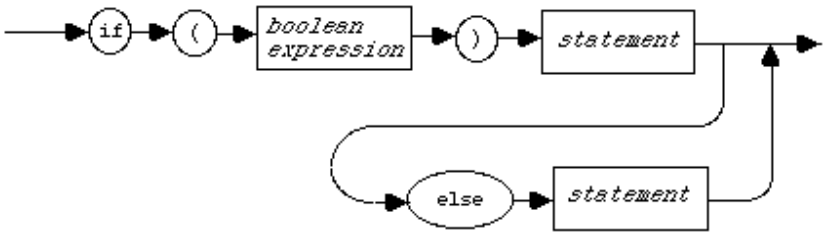
```
if (condition)
        Statement;
```

It is quite common to find "if statements" strung together in programs, as follows:

```
...
...
if (total_test_score < 50)
        cout << "You are a failure. You must study much harder.\n";
else if (total_test_score < 65)
        cout << "You have just scraped through the test.\n";
else if (total_test_score < 80)
        cout << "You have done quite well.\n";
else if (total_test_score < 95)
        cout << "Your score is excellent. Well done.\n";
else {
        cout << "You cheated!\n";
        total_test_score = 0;
}
...
...
```

This program fragment has quite a complicated logical structure, but we can confirm that it is legal in C++ by referring to the syntax diagram for "if statements". In such diagrams, the terms enclosed in ovals or circles refer to program components that literally appear in programs. Terms enclosed in boxes refer to program components that require further definition, perhaps with another syntax diagram. A collection of such diagrams can serve as a formal definition of a programming language's syntax (although they do not help distinguish between good and bad programming style!).

Below is the syntax diagram for an "if statement". It is best understood in conjunction with the syntax diagram for a "statement". In particular, notice that the diagram doesn't explicitly include the ";" or "{}" delimiters, since these are built into the definition (syntax diagram) of "statement".



Syntax diagram for an If Statement

The C++ compiler accepts the program fragment in our example by counting all of the **bold** text in

```
...
...
if (total_test_score < 50)
        cout << "You are a failure. You must study much harder.\n";
else if (total_test_score < 65)
        cout << "You have just scraped through the test.\n";
else if (total_test_score < 80)
        cout << "You have done quite well.\n";
else if (total_test_score < 95)
        cout << "Your score is excellent. Well done.\n";
else {
        cout << "You cheated!\n";
        total_test_score = 0;
}
...
...
```

as the single statement which must follow the first else.

Remarks about Program Style

As far as the C++ compiler is concerned, the following program is exactly the same as the program in above:

```
#include <iostream> using namespace std; int main()
   { int year_now, age_now, another_year, another_age; cout <<
   "Enter the current year then press RETURN.\n"; cin >> year_now; cout
<< "Enter your current age in years.\n"; cin >> age_now; cout <<
   "Enter the year for which you wish to know your age.\n"; cin >>
   another_year; another_age = another_year - (year_now - age_now); if
   (another_age >= 0) { cout << "Your age in " << another_year << ": ";
   cout << another_age << "\n"; } else { cout <<
   "You weren't even born in "; cout << another_year << "!\n"; } return
   0; }
```

However, the lack of *program comments*, *spaces*, *new lines* and *indentation* makes this program unacceptable. There is much more to developing a good programming style than learning to lay out programs properly, but it is a good start! Be consistent with your program layout, and make sure the indentation and spacing reflects the logical structure of your program. It is also a good idea to pick meaningful names for variables; "year_now", "age_now", "another_year " and "another__age " are better names than "y_n", "a_n", "a_y" and "a_a", and much better than "w", "x", "y" and "z". Remember that your programs might need modification by other programmers at a later date.

Variables, Types and Expressions

Identifiers

As we have seen, C++ programs can be written using many English words. It is useful to think of words found in a program as being one of three types:

1. Reserved Words. These are words such as if, int and else, which have a predefined meaning that cannot be changed.
2. Library Identifiers. These words are supplied default meanings by the programming environment, and should only have their meanings changed if the programmer has strong reasons for doing so. Examples are cin, cout and sqrt (square root).
3. Programmer-supplied Identifiers. These words are "created" by the programmer, and are typically variable names, such as year_now and another_age.

An identifier cannot be any sequence of symbols. A valid identifier must start with a letter of the alphabet or an underscore ("_") and must consist only of letters, digits, and underscores.

Data Types

Integers

C++ requires that all variables used in a program be given a data type. We have already seen the data type int. Variables of this type is used to represent integers (whole numbers). Declaring a variable to be of type int, signals to the compiler that it must associate enough memory with the variable's identifier to store an integer value or integer values as the program executes. But there is a (system dependent) limit on the largest and smallest integers that can be stored. Hence C++ also supports the data types short int and long int which represent, respectively, a smaller and a larger range of integer values than int. Adding the prefix unsigned to any of these types means that you wish to represent non-negative integers only. For example, the declaration

```
unsigned short int year_now, age_now, another_year, another_age;
```

reserves memory for representing four relatively small non-negative integers.

Some rules have to be observed when writing integer values in programs:

1. Decimal points cannot be used; although 26 and 26.0 have the same value, "26.0" is not of type "int".
2. Commas cannot be used in integers, so that (for example) 23,897 have to be written as "23897".Integers cannot be written with leading zeros. The compiler will, for example, interpret "011" as an octal (base 8) number, with value 9.

Real numbers

Variables of type "float" are used to store real numbers. Plus and minus signs for data of type "float" are treated exactly as with integers, and trailing zeros to the right of the decimal point are ignored. Hence "+523.5", "523.5" and "523.500" all represent the same value. The computer also accepts real numbers in *floating-point* form (or "scientific notation"). Hence 523.5 could be written as "5.235e+02" (i.e. 5.235 x 10 x 10), and -0.0034 as "-3.4e-03". In addition to "float", C++ supports the types "double" and "long double", which give increasingly precise representation of real numbers, but at the cost of more computer memory.

Type Casting

Sometimes it is important to guarantee that a value is stored as a real number, even if it is in fact a whole number. A common example is where an arithmetic expression involves division. When applied to two values of type int, the division operator "/" signifies integer division, so that (for example) 7/2 evaluates to 3. In this case, if we want an answer of 3.5, we can simply add a decimal point and zero to one or both numbers - "7.0/2", "7/2.0" and "7.0/2.0" all give the desired result. However, if both the numerator and the divisor are variables, this trick is not possible. Instead, we have to use a type cast. For example, we can convert "7" to a value of type double using the expression "static_cast<double>(7)". Hence in the expression

```
answer = static_cast<double>(numerator) / denominator
```

the "/" will always be interpreted as real-number division, even when both "numerator" and "denominator" have integer values. Other type names can also be used for type casting. For example, "static_cast<int>(14.35)" has an integer value of 14.

Characters

Variables of type "char" are used to store character data. In standard C++, data of type "char" can only be a single character (which could be a blank space). These characters come from an available character set which can differ from computer to computer. However, it always includes upper and lower case letters of the alphabet, the digits 0, ... , 9, and some special symbols such as #, £, !, +, -, etc. Perhaps the most common collection of characters is the ASCII character set

Character constants of type "char" must be enclosed in single quotation marks when used in a program, otherwise they will be misinterpreted and may cause a compilation error or unexpected program behavior. For example, "'A'" is a character constant, but "A" will be interpreted as a program variable. Similarly, "'9'" is a character, but "9" is an integer.

There is, however, an important (and perhaps somewhat confusing) technical point concerning data of type "char". Characters are represented as integers inside the computer. Hence the data type "char" is simply a subset of the data type "int". We can even do arithmetic with characters.

For example, the following expression is evaluated as true on any computer using the ASCII character set:

'9' - '0' == 57 - 48 == 9

The ASCII code for the character '9' is decimal 57 (hexadecimal 39) and the ASCII code for the character '0' is decimal 48 (hexadecimal 30) so this equation is stating that

57(dec) - 48(dec) == 39(hex) - 30(hex) == 9

It is often regarded as better to use the ASCII codes in their hexadecimal form.

However, declaring a variable to be of type "char" rather than type "int" makes an important difference as regards the type of input the program expects, and the format of the output it produces. For example, the program

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

int main()
{
        int number;
        char character;

        cout << "Type in a character:\n";
        cin >> character;

        number = character;

        cout << "The character '" << character;
        cout << "' is represented as the number ";
        cout << number << " in the computer.\n";

        return 0;
}
```

Program 2

produces output such as

```
Type in a character:
9
The character '9' is represented as the number 57 in the computer.
```

We could modify the above program to print out the whole ASCII table of characters using a "for loop". The "for loop" is an example of a *repetition statement* - we will discuss these in more detail later. The general syntax is:

```
for (initialisation; repetition_condition ; update) {
        Statement1;
        ...
```

```
        ...
        StatementN;
}
```

C++ executes such statements as follows: (1) it executes the *initialisation* statement. (2) it checks to see if *repetition_condition* is true. If it isn't, it finishes with the "for loop" completely. But if it is, it executes each of the statements *Statement1* ... *StatementN* in turn, and then executes the expression *update*. After this, it goes back to the beginning of step (2) again.

We can also 'manipulate' the output to produce the hexadecimal code. Hence to print out the ASCII table, the program above can be modified to:

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

int main()
{
        int number;
        char character;

        for (number = 32 ; number <= 126 ; number = number + 1) {

                character = number;
                cout << "The character '" << character;
                cout << "' is represented as the number ";
                cout << dec << number << " decimal or "
                <<hex<<number<< " hex.\n";
        }

        return 0;
}
```

Program 3

which produces the output:

```
The character ' ' is represented as the number 32 decimal or 20 hex.
The character '!' is represented as the number 33 decimal or 21 hex.
...
...
The character '}' is represented as the number 125 decimal or 7D hex.
The character '~' is represented as the number 126 decimal or 7E hex.
```

Strings

Our example programs have made extensive use of the type "string" in their output. As we have seen, in C++ a string constant must be enclosed in double quotation marks. Hence we have seen output statements such as

```
cout << "' is represented as the number ";
```

in programs. In fact, "string" is not a fundamental data type such as "int", "float" or "char". Instead, strings are represented as arrays of characters, so we will return to subject of strings later, when we discuss arrays in general.

User Defined Data Types

Later in the course we will study the topic of data types in much more detail. We will see how the programmer may define his or her own data types. This facility provides a powerful programming tool when complex structures of data need to be represented and manipulated by a C++ program.

Tips on Formatting Real Number Output

When program output contains values of type "float", "double" or "long double", we may wish to restrict the precision with which these values are displayed on the screen, or specify whether the value should be displayed in fixed or floating point form. The following example program uses the library identifier "sqrt" to refer to the square root function, a standard definition of which is given in the header file cmath (or in the old header style math.h).

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

int main()
{
        float number;

        cout << "Type in a real number.\n";
        cin >> number;
        cout.setf(ios::fixed);    //          LINE 10
        cout.precision(2);
        cout << "The square root of " << number << " is approximately ";
        cout << sqrt(number) << ".\n";

        return 0;
}
```

This produces the output

```
Type in a real number.
200
The square root of 200.00 is approximately 14.14.
```

Whereas replacing line 10 with "cout.setf(ios::scientific)" produces the output:

```
Type in a real number.
200
The square root of 2.00e+02 is approximately 1.41e+01.
```

We can also include tabbing in the output using a statement such as "cout.width(20)". This specifies that the next item output will have a width of at least 20 characters (with blank space appropriately added if necessary). This is useful in generating tables. However the C++ compiler has a default setting for this member function which makes it right justified. In order to produce output left-justified in a field we need to use some fancy input and output manipulation. The functions and operators which do the manipulation are to be found in the library file iomanip (old header style iomanip.h) and to do left justification we need to set a flag to a different value (i.e. left) using the setiosflags operator:

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

int main()
{
        int number;

        cout << setiosflags ( ios :: left );
        cout.width(20);
        cout << "Number" << "Square Root\n\n";

        cout.setf(ios::fixed);
        cout.precision(2);
        for (number = 1 ; number <= 10 ; number = number + 1) {
                cout.width(20);
                cout << number << sqrt( (double) number) << "\n";
        }

        return 0;
}
```

This program produces the output

```
Number          Square Root

1               1.00
2               1.41
3               1.73
4               2.00
5               2.24
6               2.45
7               2.65
8               2.83
9               3.00
10               3.16
```

(In fact, the above programs work because "cout" is an identifier for an object belonging to the class "stream", and "setf(...)", "precision(...)" and "width(...)" are member functions of "stream".

Declarations, Constants and Enumerations

As we have already seen, variables have to be declared before they can be used in a program, using program statements such as

```
float number;
```

Between this statement and the first statement which assigns "number" an explicit value, the value contained in the variable "number" is arbitrary. But in C++ it is possible (and desirable) to initialize variables with a particular value at the same time as declaring them. Hence we can write

```
double PI = 3.1415926535;
```

Furthermore, we can specify that a variable's value cannot be altered during the execution of a program with the reserved word "const":

Enumerations

Constants of type "int" may also be declared with an enumeration statement. For example, the declaration

```
enum { MON, TUES, WED, THURS, FRI, SAT, SUN };
```

is shorthand for

```
const int MON = 0;
const int TUES = 1;
const int WED = 2;
const int THURS = 3;
```

```
const int FRI = 4;
const int SAT = 5;
const int SUN = 6;
```

By default, members of an "enum" list are given the values 0, 1, 2, etc., but when "enum" members are explicitly initialised, uninitialised members of the list have values that are one more than the previous value on the list:

```
enum { MON = 1, TUES, WED, THURS, FRI, SAT = -1, SUN };
```

In this case, the value of "FRI" is 5, and the value of "SUN" is 0.

Where to put Constant and Variable Declarations

Generally speaking, it is considered good practice to put constant declarations before the "main" program heading, and variable declarations afterwards, in the body of "main". For example, the following is part of a program to draw a circle of a given radius on the screen and then print out its circumference:

```
#include <iostream>

using namespace std;

const float PI = 3.1415926535;
const float SCREEN_WIDTH = 317.24;

int drawCircle(float diameter); /* this is a "function prototype" */

int main()
{
        float radius = 0;

        cout << "Type in the radius of the circle.\n";
        cin >> radius;

        drawCircle(radius * 2);

        cout.setf(ios::fixed);
        cout.precision(2);
        cout << "The circumference of a circle of radius " << radius;
        cout << " is approximately " << 2 * PI * radius << ".\n";
        return 0;
}

int drawCircle(float diameter)
{
        float radius = 0;

        if (diameter > SCREEN_WIDTH)
                radius = SCREEN_WIDTH / 2.0;
        else
```

```
                radius = diameter / 2.0;
        ...
        ...
}
```

After the definition of "main()", this program includes a definition of the function "drawCircle(...)", the details of which need not concern us here (we can simply think of "drawCircle(...)" as a function like "sqrt(...)"). But notice that although both "main()" and "drawCircle(...)" use the identifier "radius", this refers to a *different* variable in "main()" than in "drawCircle(...)". Had a variable "radius" been declared before the "main" program heading, it would have been a public or *global variable*. In this case, and assuming there was no other variable declaration inside the function "drawCircle(...)", if "drawCircle(...)" had assigned it the value "SCREEN_WIDTH / 2.0", "main()" would have subsequently printed out the wrong value for the circumference of the circle. We say that the (first) variable "radius" is *local to the main part of the program*, or *has the function* main *as its scope*. In contrast, it usually makes sense to make constants such as "PI" and "SCREEN_WIDTH" *global*, i.e. available to every function.

In any case, notice that the program above incorporates the safety measure of *echoing the input*. In other words, the given value of "radius" is printed on the screen again, just before the circumference of the circle is output.

Assignments and Expressions

Shorthand Arithmetic Assignment Statements

We have already seen how programs can include variable assignments such as

```
number = number + 1;
```

Since it is often the case that variables are assigned a new value in function of their old value, C++ provides a shorthand notation. Any of the operators "+" (addition), "-" (subtraction), "*" (multiplication), "/" (division) and "%" (modulus) can be prefixed to the assignment operator (=), as in the following examples

Example:	**Equivalent to:**
number += 1;	number = number + 1;
total -= discount;	total = total - discount;
bonus *= 2;	bonus = bonus * 2;
time /= rush_factor;	time = time / rush_factor;
change %= 100;	change = change % 100;

```
amount *= count1 + count2;      amount = amount * (count1 + count2);
```

The first of the above examples may be written in even shorter form. Using the increment operator "++", we may simply write

```
number++;
```

The operator "++" may also be used as a prefix operator:

```
++number;
```

but care must be taken, since in some contexts the prefix and postfix modes of use have different effects. For example, the program fragment

```
x = 4;
y = x++;
```

results in "x" having the value 5 and "y" having the value 4, whereas

```
x = 4;
y = ++x;
```

results in both variables having value 5. This is because "++x" increments the value of "x" before its value is used, whereas "x++" increments the value afterwards. There is also an operator "--", which decrements variables by 1, and which can also be used in prefix or postfix form.

In general, assignment statements have a value equal to the value of the left hand side after the assignment. Hence the following is a legal expression which can be included in a program and which might be either evaluated as true or as false:

```
(y = ++x) == 5
```

It can be read as the assertion: "after x is incremented and its new value assigned to y, y's value is equal to 5".

Boolean Expressions and Operators

Intuitively, we think of expressions such as "2 < 7", "1.2 != 3.7" and "6 >= 9" as evaluating to "true" or "false" ("!=" means "not equal to"). Such expressions can be combined using the logical operators "&&" ("and"), "||" ("or") and "!" ("not"), as in the following examples:

Expression:	**True or False:**
(6 <= 6) && (5 < 3)	false
(6 <= 6) || (5 < 3)	true
(5 != 6)	true
(5 < 3) && (6 <= 6) || (5 != 6)	true
(5 < 3) && ((6 <= 6) || (5 != 6))	false
!((5 < 3) && ((6 <= 6) || (5 != 6)))	true

The fourth of these expressions is true because the operator "&&" has a higher precedence than the operator "||". Compound Boolean expressions are typically used as the condition in "if statements" and "for loops". For example:

```
...
...
if (total_test_score >= 50 && total_test_score < 65)
        cout << "You have just scraped through the test.\n";
...
...
```

Once again, there is an important technical point concerning Boolean expressions. In C++, "true" is represented simply as any non-zero integer, and "false" is represented as the value 0. This can lead to errors. For example, it is quite easy to type "=" instead of "==". Unfortunately, the program fragment

```
...
...
if (number_of_people = 1)
        cout << "There is only one person.\n";
...
...
```

will always result in the message "There is only one person" being output to the screen, even if the previous value of the variable "number_of_people" was not 1.

Functions and Procedural Abstraction

The Need for Sub-programs

A natural way to solve large problems is to break them down into a series of sub-problems, which can be solved more-or-less independently and then combined to arrive at a complete solution. In programming, this methodology reflects itself in the use of *sub-programs*, and in C++ all sub-programs are called *functions* (corresponding to both "functions" and "procedures" in Pascal and some other programming languages).

We have already been using sub-programs. For example, in the program which generated a table of square roots, we used the following "for loop":

```
...
#include<cmath>
...
...
        for (number = 1 ; number <= 10 ; number = number + 1) {
                cout.width(20);
                cout << number << sqrt(number) << "\n";
        }
...
...
```

The function "sqrt(...)" is defined in a sub-program accessed via the library file cmath (old header style math.h). The sub-program takes "number", uses a particular algorithm to compute its square root, and then returns the computed value back to the program. We don't care what the algorithm is as long as it gives the correct result. It would be ridiculous to have to explicitly (and perhaps repeatedly) include this algorithm in the "main" program.

In this chapter we will discuss how the programmer can define his or her own functions. At first, we will put these functions in the same file as "main". Later we will see how to place different functions in different files.

User-defined Functions

Here's a trivial example of a program which includes a user defined function, in this case called "area(...)". The program computes the area of a rectangle of given length and width.

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

int area(int length, int width);                /* function declaration */

/* MAIN PROGRAM: */
int main()
{
```

```
        int this_length, this_width;

        cout << "Enter the length: ";          /* <--- line 9 */
        cin >> this_length;
        cout << "Enter the width: ";
        cin >> this_width;
        cout << "\n";                          /* <--- line 13 */

        cout << "The area of a " << this_length << "x" << this_width;
        cout << " rectangle is " << area(this_length, this_width);

        return 0;
}
/* END OF MAIN PROGRAM */

/* FUNCTION TO CALCULATE AREA: */
int area(int length, int width)         /* start of function definition */
{
        int number;

        number = length * width;

        return number;
}                       /* end of function definition */
/* END OF FUNCTION */
```

Although this program is not written in the most succinct form possible, it serves to illustrate a number of features concerning functions:

- The structure of a function definition is like the structure of "main()", with its own list of variable declarations and program statements.
- A function can have a list of zero or more parameters inside its brackets, each of which has a separate type.
- A function has to be declared in a function declaration at the top of the program, just after any global constant declarations, and before it can be called by "main()" or in other function definitions.
- Function declarations are a bit like variable declarations - they specify which type the function will return.

A function may have more than one "return" statement, in which case the function definition will end execution as soon as the first "return" is reached. For example:

```
double absolute_value(double number)
{
        if (number >= 0)
                return number;
        else
                return 0 - number;
}
```

Value and Reference Parameters

The parameters in the functions above are all *value parameters*. When the function is called within the main program, it is passed the values currently contained in certain variables. For example, "area(...)" is passed the current values of the variables "this_length" and "this_width". The function "area(...)" then stores these values in its own private variables, and uses its own private copies in its subsequent computation.

Functions which use Value Parameters are Safe

The idea of value parameters makes the use of functions "safe", and leads to good programming style. It helps guarantee that a function will not have hidden *side effects*. Here is a simple example to show why this is important. Suppose we want a program which produces the following dialogue:

```
Enter a positive integer:
4
The factorial of 4 is 24, and the square root of 4 is 2.
```

It would make sense to use the predefined function "sqrt(...)" in our program, and write another function "factorial(...)" to compute the factorial n! = (1 x 2 x ... x n) of any given positive integer n. Here's the complete program:

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

int factorial(int number);

/* MAIN PROGRAM: */
int main()
{
        int whole_number;

        cout << "Enter a positive integer:\n";
        cin >> whole_number;
        cout << "The factorial of " << whole_number << " is ";
        cout << factorial(whole_number);
        cout << ", and the square root of " << whole_number << " is ";
        cout << sqrt(whole_number) << ".\n";

        return 0;
}
/* END OF MAIN PROGRAM */

/* FUNCTION TO CALCULATE FACTORIAL: */
int factorial(int number)
{
        int product = 1;

        for ( ; number > 0 ; number--)
```

```
                product *= number;

        return product;
}
/* END OF FUNCTION */
```

By the use of a value parameter, we have avoided the (correct but unwanted) output

```
Enter a positive integer:
4
The factorial of 4 is 24, and the square root of 0 is 0.
```

which would have resulted if the function "factorial(...)" had permanently changed the value of the variable "whole_number".

Reference Parameters

Under some circumstances, it is legitimate to require a function to modify the value of an actual parameter that it is passed. For example, going back to the program which inputs the dimensions of a rectangle and calculates the area, it would make good design sense to package up lines 9 to 13 of the main program into a "get-dimensions" sub-program (i.e. a C++ function). In this case, we require the function to alter the values of "this_length" and "this_width" (passed as parameters), according to the values input from the keyboard. We can achieve this as follows using reference parameters, whose types are post-fixed with an "&":

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

int area(int length, int width);

void get_dimensions(int& length, int& width);

/* MAIN PROGRAM: */
int main()
{
        int this_length, this_width;

        get_dimensions(this_length, this_width);
        cout << "The area of a " << this_length << "x" << this_width;
        cout << " rectangle is " << area(this_length, this_width);

        return 0;
}
/* END OF MAIN PROGRAM */

/* FUNCTION TO INPUT RECTANGLE DIMENSIONS: */
void get_dimensions(int& length, int& width)
{
        cout << "Enter the length: ";
        cin >> length;
        cout << "Enter the width: ";
```

```
	cin >> width;
	cout << "\n";
}
/* END OF FUNCTION */

/* FUNCTION TO CALCULATE AREA: */
int area(int length, int width)
{
	return length * width;
}
/* END OF FUNCTION */
```

Notice that although the function "get_dimensions" permanently alters the values of the parameters "this_length" and "this_width" it does not return any other value (i.e. is not a "function" in the mathematical sense). This is signified in both the function declaration and the function heading by the reserved word "void".

Polymorphism and Overloading

C++ allows *polymorphism*, i.e. it allows more than one function to have the same name, provided all functions are either distinguishable by the typing or the number of their parameters. Using a function name more than once is sometimes referred to as *overloading* the function name. Here's an example:

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

int average(int first_number, int second_number, int third_number);

int average(int first_number, int second_number);

/* MAIN PROGRAM: */
int main()
{
	int number_A = 5, number_B = 3, number_C = 10;

	cout << "The integer average of " << number_A << " and ";
	cout << number_B << " is ";
	cout << average(number_A, number_B) << ".\n\n";

	cout << "The integer average of " << number_A << ", ";
	cout << number_B << " and " << number_C << " is ";
	cout << average(number_A, number_B, number_C) << ".\n";

	return 0;
}
/* END OF MAIN PROGRAM */

/* FUNCTION TO COMPUTE INTEGER AVERAGE OF 3 INTEGERS: */
int average(int first_number, int second_number, int third_number)
{
	return ((first_number + second_number + third_number) / 3);
```

```
}
/* END OF FUNCTION */

/* FUNCTION TO COMPUTE INTEGER AVERAGE OF 2 INTEGERS: */
int average(int first_number, int second_number)
{
        return ((first_number + second_number) / 2);
}
/* END OF FUNCTION */
```

This program produces the output:

```
The integer average of 5 and 3 is 4.

The integer average of 5, 3 and 10 is 6.
```

Procedural Abstraction and Good Programming Style

One of the main purposes of using functions is to aid in the *top down* design of programs. During the design stage, as a problem is subdivided into tasks (and then into sub-tasks, sub-sub-tasks, etc.), the problem solver (programmer) should have to consider only what a function is to do and not be concerned about the details of the function. The function name and comments at the beginning of the function should be sufficient to inform the user as to what the function does. (Indeed, during the early stages of program development, experienced programmers often use simple "dummy" functions or *stubs*, which simply return an arbitrary value of the correct type, to test out the control flow of the main or higher level program component.)

Developing functions in this manner is referred to as *functional* or *procedural abstraction*. This process is aided by the use of value parameters and local variables declared within the body of a function. Functions written in this manner can be regarded as "black boxes". As users of the function, we neither know nor care why they work.

Splitting Programs into Different Files

As we have seen, C++ makes heavy use of predefined standard libraries of functions, such as "sqrt(...)". In fact, the C++ code for "sqrt(...)", as for most functions, is typically split into two files:

- The *header file* "cmath" contains the function declarations for "sqrt(...)" (and for many other mathematical functions). This is why in the example programs which call "sqrt(...)" we are able to write "#include<cmath>", instead of having to declare the function explicitly.

- The *implementation file* "math.cpp" contains the actual function definitions for "sqrt(...)" and other mathematical functions. (In practice, many C++ systems have one or a few big file(s) containing all the standard function definitions, perhaps called "ANSI.cpp" or similar.)

It is easy to extend this library structure to include files for user-defined functions, such as "area(...)", "factorial(...)" and "average(...)". As an example, Program 3.6.1 below is the same as

Program 3.4.1, but split into a main program file, a header file for the two average functions, and a corresponding implementation file.

The code in the main program file is as follows:

```
#include<iostream>
#include"averages.h"

using namespace std;

int main()
{
	int number_A = 5, number_B = 3, number_C = 10;

	cout << "The integer average of " << number_A << " and ";
	cout << number_B << " is ";
	cout << average(number_A, number_B) << ".\n\n";

	cout << "The integer average of " << number_A << ", ";
	cout << number_B << " and " << number_C << " is ";
	cout << average(number_A, number_B, number_C) << ".\n";

	return 0;
}
```

Program 3.6.1

Notice that whereas "include" statements for standard libraries such as "iostream" delimit the file name with angle ("<>") brackets, the usual convention is to delimit user-defined library file names with double quotation marks - hence the line " #include"averages.h" " in the listing above.

The code in the header file "averages.h" is listed below. Notice the use of the file identifier "AVERAGES_H", and the reserved words "ifndef" ("if not defined"), "define", and "endif". "AVERAGES_H" is a (global) symbolic name for the file. The first two lines and last line of code ensure that the compiler (in fact, the *preprocessor*) only works through the code in between once, even if the line "#include"averages.h"" is included in more than one other file.

Constant and type definitions are also often included in header files. You will learn more about this in the object-oriented part of the course.

```
#ifndef AVERAGES_H
#define AVERAGES_H

/* (constant and type definitions could go here) */

/* FUNCTION TO COMPUTE INTEGER AVERAGE OF 3 INTEGERS: */
int average(int first_number, int second_number, int third_number);

/* FUNCTION TO COMPUTE INTEGER AVERAGE OF 2 INTEGERS: */
int average(int first_number, int second_number);

#endif
```

averages.h

Finally, the code in the implementation file "averages.cpp" is as follows:

```
#include<iostream>
#include"averages.h"

using namespace std;

/* FUNCTION TO COMPUTE INTEGER AVERAGE OF 3 INTEGERS: */
int average(int first_number, int second_number, int third_number)
{
	return ((first_number + second_number + third_number) / 3);
}
/* END OF FUNCTION */

/* FUNCTION TO COMPUTE INTEGER AVERAGE OF 2 INTEGERS: */
int average(int first_number, int second_number)
{
	return ((first_number + second_number) / 2);
}
/* END OF FUNCTION */
```

averages.cpp

Note the modularity of this approach. We could change the details of the code in "averages.cpp" without making any changes to the code in "averages.h" or in the main program file.

Files and Streams

Why Use Files?

All the programs we have looked at so far use input only from the keyboard, and output only to the screen. If we were restricted to use only the keyboard and screen as input and output devices, it would be difficult to handle large amounts of input data, and output data would always be lost as soon as we turned the computer off. To avoid these problems, we can store data in some secondary storage device, usually magnetic tapes or discs. Data can be created by one program, stored on these devices, and then accessed or modified by other programs when necessary. To achieve this, the data is packaged up on the storage devices as data structures called *files*.

The easiest way to think about a file is as a linear sequence of characters. In a simplified picture (which ignores special characters for text formatting) these lecture notes might be stored in a file called "LectureNotes_4" as:

Figure 4.1.1

Streams

Before we can work with files in C++, we need to become acquainted with the notion of a *stream*. We can think of a stream as a channel or conduit on which data is passed from senders to receivers. As far as the programs we will use are concerned, streams allow travel in only one direction. Data can be sent out from the program on an *output stream*, or received into the program on an *input stream*. For example, at the start of a program, the standard input stream "cin" is connected to the keyboard and the standard output stream "cout" is connected to the screen.

In fact, input and output streams such as "cin" and "cout" are examples of (stream) *objects*. So learning about streams is a good way to introduce some of the syntax and ideas behind the object-oriented part of C++. The header file which lists the operations on streams both to and from files is called "fstream". We will therefore assume that the program fragments discussed below are embedded in programs containing the "include" statement

```
#include<fstream>
```

As we shall see, the essential characteristic of stream processing is that data elements must be sent to or received from a stream one at a time, i.e. in *serial* fashion.

Creating Streams

Before we can use an input or output stream in a program, we must "create" it. Statements to create streams look like variable declarations, and are usually placed at the top of programs or function implementations along with the variable declarations. So for example the statements

```
ifstream in_stream;
ofstream out_stream;
```

respectively create a stream called "in_stream" belonging to the *class* (like type) "ifstream" (input-file-stream), and a stream called "out_stream" belonging to the class "ofstream" (output-file-stream). However, the analogy between streams and ordinary variables (of type "int", "char", etc.) can't be taken too far. We cannot, for example, use simple assignment statements with streams (e.g. we can't just write "in_stream1 = in_stream2").

Connecting and Disconnecting Streams to Files

Having created a stream, we can connect it to a file using the *member function* "open(...)". (We have already come across some member functions for output streams, such as "precision(...)" and "width(...)") The function "open(...)" has a different effect for ifstreams than for ofstreams (i.e. the function is polymorphic).

To connect the ifstream "in_stream" to the file "LectureNotes_4", we use the following statement:

```
in_stream.open("LectureNotes_4");
```

This connects "in_stream" to the beginning of "LectureNotes_4". Diagramatically, we end up in the following situation:

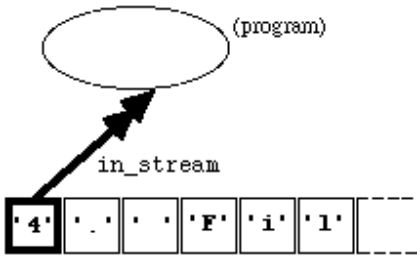


Figure 4.2.1

To connect the ofstream "out_stream" to the file "LectureNotes_4", we use an analogous statement:

```
out_stream.open("LectureNotes_4");
```

Although this connects "out_stream" to "LectureNotes_4", it also deletes the previous contents of the file, ready for new input. Diagramatically, we end up as follows:

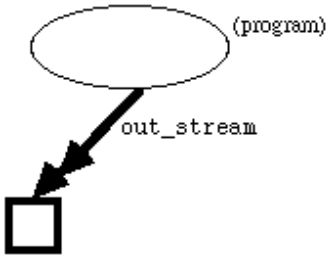


Figure 4.2.2

To disconnect connect the ifstream "in_stream" to whatever file it is connected to, we write:

```
in_stream.close();
```

Diagramatically, the situation changes from that of Figure 4.2.1 to:

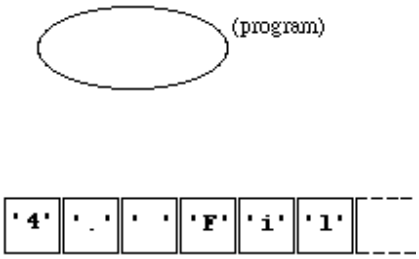


Figure 4.2.3

The statement:

```
out_stream.close();
```

has a similar effect, but in addition the system will "clean up" by adding an "end-of-file" marker at the end of the file. Thus, if no data has been output to "LectureNotes_4" since "out_stream" was connected to it, we change from the situation in Figure 4.2.2 to:

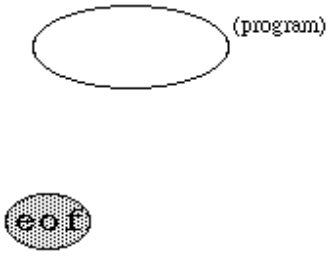


Figure 4.2.4

In this case, the file "LectureNotes_4" still exists, but is *empty*.

Checking for Failure with File Commands

File operations, such as opening and closing files, are a notorious source of errors. Robust commercial programs should always include some check to make sure that file operations have completed successfully, and error handling routines in case they haven't. A simple checking mechanism is provided by the member function "fail()". The function call

```
in_stream.fail();
```

returns True if the previous stream operation on "in_stream" was not successful (perhaps we tried to open a file which didn't exist). If a failure has occurred, "in_stream" may be in a corrupted state, and it is best not to attempt any more operations with it. The following example program fragment plays very safe by quitting the program entirely, using the "exit(1)" command from the library "cstdlib":

```
#include <iostream>
#include <fstream>
#include <cstdlib>

using namespace std;

int main()
{
        ifstream in_stream;

        in_stream.open("Lecture_4");
        if (in_stream.fail())
        {
                cout << "Sorry, the file couldn't be opened!\n";
                exit(1);
        }
        ...
```

Character Input and Output

Input using "get(...)"

Having opened an input file, we can extract or read single characters from it using the member function "get(...)". This function takes a single argument of type "char". If the program is in the state represented in Figure 4.2.1, the statement

```
in_stream.get(ch);
```

has two effects: (i) the variable "ch" is assigned the value "'4'", and (ii) the ifstream "in_stream" is re- positioned so as to be ready to input the next character in the file. Diagramatically, the new situation is:

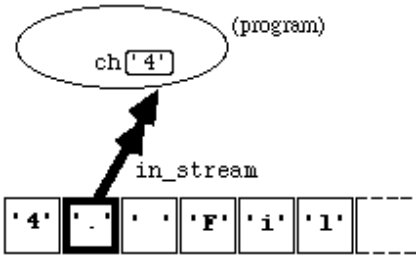


Figure 4.4.1

Output using "put(...)"

We can input or *write* single characters to a file opened via an ofstream using the member function "put(...)". Again, this function takes a single argument of type "char". If the program is in the state represented in Figure 4.2.2, the statement

```
out_stream.put('4');
```

changes the state to:

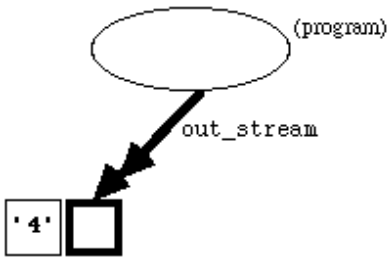


Figure 4.4.2

The "putback(...)" Function

C++ also includes a "putback(...)" function for ifstreams. This doesn't really "put the character back" (it doesn't alter the actual input file), but behaves as if it had. Diagramatically, if we started from the state in Figure 4.4.1, and executed the statement

```
in_stream.putback(ch);
```

we would end up in the state:

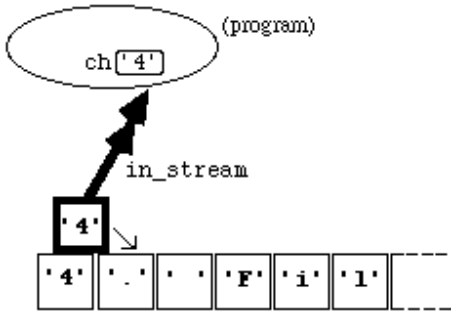


Figure 4.4.3

Indeed, we can "putback" any character we want to. The alternative statement

```
in_stream.putback('7');
```

would result in:

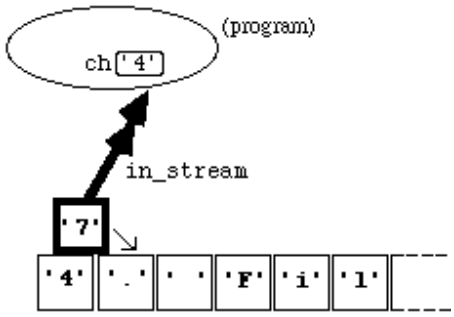


Figure 4.4.4

Branch and Loop Statements

Boolean Values, Expressions and Functions

In this topic we will look more closely at branch and loop statements such as "for" and "while" loops and "if ... else" statements. All these constructs involve the evaluation of one or more logical (or "Boolean") expressions, and so we begin by looking at different ways to write such expressions.

As we have seen, in reality C++ represents "True" as the integer 1, and "False" as 0. However, expressions such as

```
condition1 == 1
```

or

```
condition2 == 0
```

aren't particularly clear :-it would be better to be able to follow our intuition and write

```
condition1 == True
```

and

```
condition2 == False
```

Furthermore, it is desirable to have a separate type for variables such as "condition1", rather than having to declare them as of type "int". We can achieve all of this with a *named enumeration*:

```
enum Logical {False, True}
```

which is equivalent to

```
enum Logical {False = 0, True = 1}
```

This line acts a kind of *type definition* for a new data type "Logical", so that lower down the program we can add variable declarations such as:

```
Logical condition1, condition2;
```

Indeed, we can now use the identifier "Logical" in exactly the same way as we use the identifiers "int", "char", etc. In particular, we can write functions which return a value of type "Logical". The following example program takes a candidate's age and test score, and reports whether the

candidate has passed the test. It uses the following criteria: candidates between 0 and 14 years old have a pass mark of 50%, 15 and 16 year olds have a pass mark of 55%, over 16's have a pass mark of 60%:

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

enum Logical {False, True};

Logical acceptable(int age, int score);

/* START OF MAIN PROGRAM */
int main()
{
	int candidate_age, candidate_score;

	cout << "Enter the candidate's age: ";
	cin >> candidate_age;
	cout << "Enter the candidate's score: ";
	cin >> candidate_score;

	if (acceptable(candidate_age, candidate_score))
		cout << "This candidate passed the test.\n";
	else
		cout << "This candidate failed the test.\n";

	return 0;
}
/* END OF MAIN PROGRAM */

/* FUNCTION TO EVALUATE IF TEST SCORE IS ACCEPTABLE */
Logical acceptable(int age, int score)
{
	if (age <= 14 && score >= 50)
		return True;
	else if (age <= 16 && score >= 55)
		return True;
	else if (score >= 60)
		return True;
	else
		return False;
}
/*END OF FUNCTION */
```

Note that since "True" and "False" are constants, it makes sense to declare them outside the scope of the main program, so that the type "Logical" can be used by every function in the file. An alternative way to write the above function "acceptable(...)" would be:

```
/* FUNCTION TO EVALUATE IF TEST SCORE IS ACCEPTABLE */
Logical acceptable(int age, int score)
{
	Logical passed_test = False;
```

```
	if (age <= 14 && score >= 50)
		passed_test = True;
	else if (age <= 16 && score >= 55)
		passed_test = True;
	else if (score >= 60)
		passed_test = True;

	return passed_test;
}
/*END OF FUNCTION */
```

Defining our own data types (even if for the moment they're just sub-types of "int") brings us another step closer to object-oriented programming, in which complex types of data structure (or *classes* of *objects*) can be defined, each with their associated libraries of operations.

Note: The Identifiers "true" and "false" in C++

Note that C++ implicitly includes the named enumeration

```
enum bool {false, true};
```

So you can't (re)define the all-lower-case constant identifiers "true" and "false" for yourself. In addition, you can use the type bool in the same way as we used Logical in our example.

"For", "While" and "Do ... While" Loops

We have already been introduced to "for" loops and "while" loops in the previous discussions. Notice that any "for" loop can be re-written as a "while" loop. For example,

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

int main()
{
	int number;
	char character;

	for (number = 32 ; number <= 126 ; number = number + 1) {

		character = number;
		cout << "The character '" << character;
		cout << "' is represented as the number ";
		cout << number << " in the computer.\n";
	}

	return 0;
}
```

can be written equivalently as

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

int main()
{
	int number;
	char character;

	number = 32;
	while (number <= 126)
	{
		character = number;
		cout << "The character '" << character;
		cout << "' is represented as the number ";
		cout << number << " in the computer.\n";
		number++;
	}

	return 0;
}
```

Moreover, any "while" loop can be trivially re-written as a "for" loop: - we could for example replace the line

```
while (number <= 126)
```

with the line

```
for ( ; number <= 126 ; )
```

in the program above.

There is a third kind of "loop" statement in C++ called a *"do ... while" loop*. This differs from "for" and "while" loops in that the statement(s) inside the {} braces are always executed once, before the repetition condition is even checked. "Do ... while" loops are useful, for example, to ensure that the program user's keyboard input is of the correct format:

```
...
...
do
{
	cout << "Enter the candidate's score: ";
	cin >> candidate_score;
	if (candidate_score > 100 || candidate_score < 0)
		cout << "Score must be between 0 and 100.\n";
}
while (candidate_score > 100 || candidate_score < 0);
...
...
```

Program Fragment

This avoids the need to repeat the input prompt and statement, which would be necessary in the equivalent "while" loop:

```
...
...
cout << "Enter the candidate's score: ";
cin >> candidate_score;
while (candidate_score > 100 || candidate_score < 0)
{
        cout << "Score must be between 0 and 100.\n";
        cout << "Enter the candidate's score: ";
        cin >> candidate_score;
}
...
...
```

Program Fragment

Multiple Selection and Switch Statements

We have already seen in the beginning how "if" statements can be strung together to form a "multiway branch". Here's a simplified version of the previous example:

```
...
...
if (total_test_score >=0 && total_test_score < 50)
        cout << "You are a failure - you must study much harder.\n";
else if (total_test_score < 60)
        cout << "You have just scraped through the test.\n";
else if (total_test_score < 80)
        cout << "You have done quite well.\n";
else if (total_test_score <= 100)
        cout << "Your score is excellent - well done.\n";
else
        cout << "Incorrect score - must be between 0 and 100.\n";
...
...
```

Because multiple selections can sometimes be difficult to follow, C++ provides an alternative method of handling this concept, called the *switch* statement. "Switch" statements can be used when several options depend on the value of a single variable or expression. In the example above, the message printed depends on the value of "total_test_score". This can be any number between 0 and 100, but we can make things easier to handle by introducing an extra integer variable "score_out_of_ten", and adding the assignment:

```
score_out_of_ten = total_test_score / 10;
```

The programming task is now as follows: (i) if "score_out_of_ten" has value 0, 1, 2, 3 or 4, print "You are a failure - you must study much harder", (ii) if "score_out_of_ten" has value 5, print "You have just scraped through the test", (iii) if "score_out_of_ten" has value 6 or 7, print "You have

done quite well", and finally (iv) if "score_out_of_ten" has value 8, 9 or 10, print "Your score is excellent - well done". Here's how this is achieved with a "switch" statement:

```
...
...
score_out_of_ten = total_test_score / 10;

switch (score_out_of_ten)
{
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:    cout << "You are a failure - you ";
                   cout << "must study much harder.\n";
                   break;

        case 5:    cout << "You have just scraped through the test.\n";
                   break;

        case 6:
        case 7:    cout << "You have done quite well.\n";
                   break;

        case 8:
        case 9:
        case 10:   cout << "Your score is excellent - well done.\n";
                   break;

        default:   cout << "Incorrect score - must be between ";
                   cout << "0 and 100.\n";
}
...
...
```

In general, the syntax of a "switch" statement is (approximately):

```
switch (selector)
  {
    case label1:    <statements 1>
                    break;
    ...
    ...
    ...

    case labelN:    <statements N>
                    break;

    default:        <statements>
  }
```

There are several things to note about such "switch" statements:

- The statements which are executed are exactly those between the first label which matches the value of selector and the first "break" after this matching label.
- The "break" statements are optional, but they help in program efficiency and clarity and should ideally always be used to end each case. When a "break" is encountered within a case's statement, control is transferred immediately to the first program statement following the entire "switch" statement. Otherwise, execution continues.
- The selector can have a value of any ordinal type (e.g. "char" or "int" but not "float").
- The "default" is optional, but is a good safety measure.

Blocks and Scoping

We have already seen how *compound statements* in C++ are delimited by "{}" braces. These braces have a special effect on variable declarations. A compound statement that contains one or more variable declarations is called a *block*, and the variables declared within the block have the block as their *scope*. In other words, the variables are "created" each time the program enters the block, and "destroyed" upon exit. If the same identifier has been used both for a variable inside and a variable outside the block, the variables are unrelated. While in the block, the program will assume by default that the identifier refers to the inner variable - it only looks outside the block for the variable if it can't find a variable declaration inside. Hence the program

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

int integer1 = 1;
int integer2 = 2;
int integer3 = 3;

int main()
{
	int integer1 = -1;
	int integer2 = -2;
	{
		int integer1 = 10;
		cout << "integer1 == " << integer1 << "\n";
		cout << "integer2 == " << integer2 << "\n";
		cout << "integer3 == " << integer3 << "\n";
	}
	cout << "integer1 == " << integer1 << "\n";
	cout << "integer2 == " << integer2 << "\n";
	cout << "integer3 == " << integer3 << "\n";

	return 0;
}
```

produces the output

```
integer1 == 10
integer2 == -2
```

```
integer3 == 3
integer1 == -1
integer2 == -2
integer3 == 3
```

The use of variables local to a block can sometimes be justified because it saves on memory, or because it releases an identifier for re-use in another part of the program. The following program prints a series of "times tables" for integers from 1 to 10:

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

int main()
{
        int number;

        for (number = 1 ; number <= 10 ; number++)
        {
                int multiplier;

                for (multiplier = 1 ; multiplier <= 10 ; multiplier++)
                {
                        cout << number << " x " << multiplier << " = ";
                        cout << number * multiplier << "\n";
                }
                cout << "\n";
        }
        ...
        ...
```

However, we can achieve the same effect, and end up with a clearer program, by using a function:

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

void print_times_table(int value, int lower, int upper);

int main()
{
        int number;

        for (number = 1 ; number <= 10 ; number++)
        {
                print_times_table(number,1,10);
                cout << "\n";
        }
        ...
        ...
}

void print_times_table(int value, int lower, int upper)
{
        int multiplier;

        for (multiplier = lower ; multiplier <= upper ; multiplier++)
```

```
        {
                cout << value << " x " << multiplier << " = ";
                cout << value * multiplier << "\n";
        }
}
```

or eliminate all variable declarations from "main()" using two functions:

```
#include <iostream>

void print_tables(int smallest, int largest);

void print_times_table(int value, int lower, int upper);

int main()
{
        print_tables(1,10);
        ...
        ...
}

void print_tables(int smallest, int largest)
{
        int number;

        for (number = smallest ; number <= largest ; number++)
        {
                print_times_table(number,1,10);
                cout << "\n";
        }
}

void print_times_table(int value, int lower, int upper)
{
        int multiplier;

        for (multiplier = lower ; multiplier <= upper ; multiplier++)
        {
                cout << value << " x " << multiplier << " = ";
                cout << value * multiplier << "\n";
        }
}
```

Nested Loop Statements

The above "times table" programs illustrate how nested loop statements can be made more readable by the use of functional abstraction. By making the body of the loop into a function call, its design can be separated from the design of the rest of the program, and problems with scoping of variables and overloading of variable names can be avoided.

Arrays and Strings

The Basic Idea and Notation

Although we have already seen how to store large amounts of data in files, we have as yet no convenient way to manipulate such data from within programs. For example, we might want to write a program that inputs and then ranks or sorts a long list of numbers. C++ provides a *structured data type* called an *array* to facilitate this kind of task. The use of arrays permits us to set aside a group of memory locations (i.e. a group of variables) that we can then manipulate as a single entity, but that at the same time gives us direct access to any individual component. Arrays are simple examples of structured data types - they are effectively just lists of variables all of the same data type ("int", "char" or whatever). Later in the course you will learn how to construct more complicated compound data structures.

Declaring an array

The general syntax for an array declaration is:

```
<component type><variable identifier>[<integer value>];
```

For example, suppose we are writing a program to manipulate data concerning the number of hours a group of 6 employees have worked in a particular week. We might start the program with the array declaration:

```
int hours[6];
```

or better,

```
const int NO_OF_EMPLOYEES = 6;
int hours[NO_OF_EMPLOYEES];
```

Indeed, if we are going to use a number of such arrays in our program, we can even use a *type definition*:

```
const int NO_OF_EMPLOYEES = 6;
typedef int Hours_array[NO_OF_EMPLOYEES];
Hours_array hours;
Hours_array hours_week_two;
```

In each case, we end up with 6 variables of type "int" with identifiers

```
hours[0]   hours[1]   hours[2]   hours[3]   hours[4]   hours[5]
```

Each of these is referred to as an *element* or *component* of the array. The numbers 0, ..., 5 are the *indexes* or *subscripts* of the components. An important feature of these 6 variables is that they are allocated consecutive memory locations in the computer. We can picture this as:

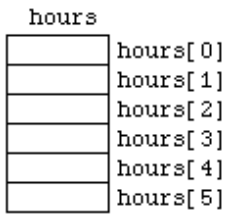


Figure 6.1.1

Assignment Statements and Expressions with Array Elements

Having declared our array, we can treat the individual elements just like ordinary variables (of type "int" in the particular example above). In particular, we can write assignment statements such as

```
hours[4] = 34;
hours[5] = hours[4]/2;
```

and use them in logical expressions, e.g.

```
if (number < 4 && hours[number] >= 40) { ...
```

A common way to assign values to an array is using a "for" or "while" loop. The following program prompts the user for the number of hours that each employee has worked. It is more natural to number employees from 1 to 6 than from 0 to 5, but it is important to remember that array indexes always start from 0. Hence the program subtracts 1 from each employee number to obtain the corresponding array index.

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

const int NO_OF_EMPLOYEES = 6;
typedef int Hours_array[NO_OF_EMPLOYEES];

int main()
{
        Hours_array hours;
        int count;

        for (count = 1 ; count <= NO_OF_EMPLOYEES ; count++)
        {
                cout << "Enter hours for employee number " << count << ": ";
                cin >> hours[count - 1];
        }

        return 0;
}
```

A typical run might produce the following input/output:

```
Enter hours for employee number 1: 38
Enter hours for employee number 2: 42
Enter hours for employee number 3: 29
Enter hours for employee number 4: 35
Enter hours for employee number 5: 38
Enter hours for employee number 6: 37
```

in which case our block of variables would then be in the state:

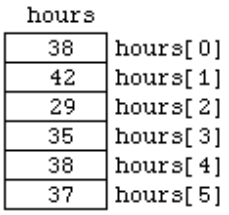


Figure 6.1.2

It is instructive to consider what would have happened had we forgotten to subtract 1 from the variable "count" in the "cin ..." statement (within the "for" loop) in Program 6.1.1. Unlike some languages, C++ does not do range bound error checking, so we would have simply ended up in the state:

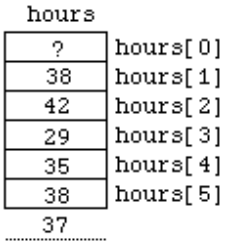


Figure 6.1.3 - A Range Bound Error

In other words, C++ would have simply put the value "37" into the next integer-sized chunk of memory located after the memory block set aside for the array "hours". This is a very undesirable situation - the compiler might have already reserved this chunk of memory for another variable (perhaps, for example, for the variable "count").

Array elements can be of data types other than "int". Here's a program that prints itself out backwards on the screen, using an array of type "char".

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

using namespace std;

const int MAX = 1000;
typedef char File_array[MAX];
```

```
int main()
{
	char character;
	File_array file;
	int count;
	ifstream in_stream;

	in_stream.open("6-1-2.cpp");
	in_stream.get(character);
	for (count = 0 ; ! in_stream.eof() && count < MAX ; count++)
	{
		file[count] = character;
		in_stream.get(character);
	}
	in_stream.close();

	while (count > 0)
		cout << file[--count];

	return 0;
}
```

Note the use of the condition "... && count < MAX ; ..." in the head of the "for" loop, to avoid the possibility of a range bound error.

Arrays as Parameters in Functions

Functions can be used with array parameters to maintain a structured design. Here is a definition of an example function which returns the average hours worked, given an array of type "Hours_array" from Program 6.1.1

```
float average(Hours_array hrs)
{
	float total = 0;
	int count;
	for (count = 0 ; count < NO_OF_EMPLOYEES ; count++)
		total += float(hrs[count]);
	return (total / NO_OF_EMPLOYEES);
}
```

Fragment of Program 6.2.1

We could make this function more general by including a second parameter for the length of the array:

```
float average(int list[], int length)
{
	float total = 0;
	int count;
```

```
        for (count = 0 ; count < length ; count++)
                total += float(list[count]);
        return (total / length);
}
```

It is quite common to pass the array length to a function along with an array parameter, since the syntax for an array parameter (such as "int list[]" above) doesn't include the array's length.

Although array parameters are not declared with an "&" character in function declarations and definitions, they are effectively reference parameters (rather than value parameters). In other words, when they execute, functions do not make private copies of the arrays they are passed (this would potentially be very expensive in terms of memory). Hence, like the reference parameters we have seen earlier, arrays can be permanently changed when passed as arguments to functions. For example, after a call to the following function, each element in the third array argument is equal to the sum of the corresponding two elements in the first and second arguments:

```
void add_lists(int first[], int second[], int total[], int length)
{
        int count;
        for (count = 0 ; count < length ; count++)
                total[count] = first[count] + second[count];
}
```

Fragment of Program 6.2.2

As a safety measure, we can add the modifier "const" in the function head:

```
void add_lists(const int fst[], const int snd[], int tot[], int len)
{
        int count;
        for (count = 0 ; count < len; count++)
                tot[count] = fst[count] + snd[count];
}
```

The compiler will then not accept any statements within the function's definition which potentially modify the elements of the arrays "fst" or "snd". Indeed, the restriction imposed by the "const" modifier when used in this context is stronger than really needed in some situations. For example, the following two function definitions will not be accepted by the compiler:

```
void no_effect(const int list[])
{
        do_nothing(list);
}

void do_nothing(int list[])
{
        ;
}
```

Fragment of Program 6.2.3

This is because, although we can see that "do_nothing(...)" does nothing, its head doesn't include the modifier "const", and the compiler only looks at the head of "do_nothing(...)" when checking to see if the call to this function from within "no_effect(...)" is legal.

Sorting Arrays

Arrays often need to be sorted in either ascending or descending order. There are many well known methods for doing this; the *quick sort* algorithm is among the most efficient. This section briefly describes one of the easiest sorting methods called the *selection sort*.

The basic idea of selection sort is:

For each index position I in turn:

1. Find the smallest data value in the array from positions I to (Length - 1), where "Length" is the number of data values stored.
2. Exchange the smallest value with the value at position I.

To see how selection works, consider an array of five integer values, declared as

```
int a[5];
```

and initially in the state:

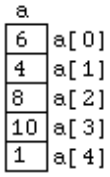


Figure 6.3.1

Selection sort takes the array through the following sequence of states:

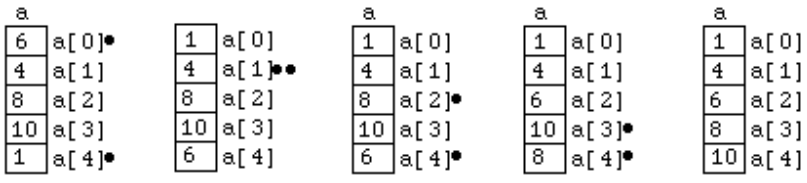


Figure 6.3.2

Each state is generated from the previous one by swapping the two elements of the array marked with a "bullet".

We can code this procedure in C++ with three functions. The top level function "selection_sort(...)" (which takes and array and an integer argument) sorts its first (array) argument by first calling the function "minimum_from(array,position,length)", which returns the index of the smallest element in "array" which is positioned at or after the index "position". It then swaps values according to the specification above, using the "swap(...)" function:

```
void selection_sort(int a[], int length)
{
	for (int count = 0 ; count < length - 1 ; count++)
		swap(a[count],a[minimum_from(a,count,length)]);
}

int minimum_from(int a[], int position, int length)
{
	int min_index = position;

	for (int count = position + 1 ; count < length ; count ++)
		if (a[count] < a[min_index])
			min_index = count;

	return min_index;
}

void swap(int& first, int& second)
{
	int temp = first;
	first = second;
	second = temp;
}
```

Two-dimensional Arrays

Arrays can have more than one dimension. In this section we briefly examine the use of two-dimensional arrays to represent two-dimensional structures such as screen bitmaps or nxm matrices of integers.

A bitmap consists of a grid of Boolean values representing the state of the dots or pixels on a screen. "True" means "on" or that the pixel is white; "False" means "off" or the pixel is black. Let's suppose the screen is 639 pixels wide and 449 pixels high. We can declare the corresponding array as follows:

```
enum Logical {False, True};
const int SCREEN_HEIGHT = 449;
const int SCREEN_WIDTH = 639;
Logical screen[SCREEN_HEIGHT][SCREEN_WIDTH];
```

References to individual data elements within the array "screen" simply use two index values. For example, the following statement assigns the value "True" to the cell (pixel) in row 4, column 2 of the array.

```
screen[3][1] = True;
```

All of the discussion in Section 6.2 about one-dimensional arrays as parameters in functions also applies to two-dimensional arrays, but with one additional peculiarity. In function declarations and in the heads of function definitions, the size of the first dimension of a multidimensional array parameter is not given (inside the "[]" brackets), but the sizes of all the other dimensions are given. Hence, for example, the following is a correct form for a function which sets all the screen pixels to black:

```
void clear_bitmap(Logical bitmap[][SCREEN_WIDTH], int screen_height)
{
        for (int row = 0 ; row < screen_height ; row++)
                for (int column = 0 ; column < SCREEN_WIDTH; column++)
                        bitmap[row][column] = False;
}
```

Strings

We have already been using string values, such as "'Enter age: '", in programs involving output to the screen. In C++ you can store and manipulate such values in *string variables*, which are really just arrays of characters, but used in a particular way.

The Sentinel String Character '\0'

The key point is that, to use the special functions associated with strings, string values can only be stored in string variables whose length is *at least 1 greater than* the length (in characters) of the value. This is because extra space must be left at the end to store the *sentinel string character* "'\0'" which marks the end of the string value. For example, the following two arrays both contain all the characters in the string value "'Enter age: '", but only the array on the left contains a proper string representation.

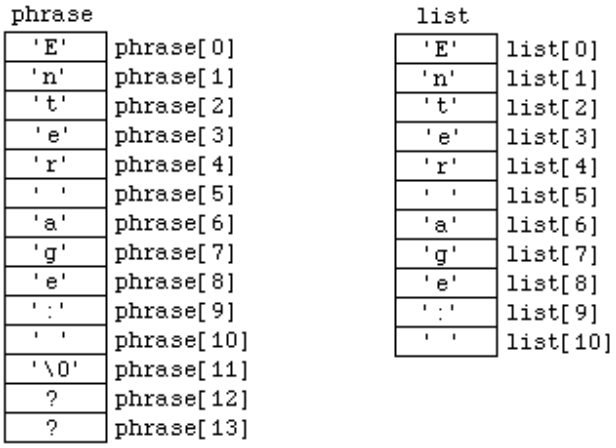


Figure 6.5.1

In other words, although both "phrase" and "list" are arrays of characters, only "phrase" is big enough to contain the string value ""Enter age: "". We don't care what characters are stored in the variables "phrase[12]" and "phrase[13]", because all the string functions introduced below ignore characters after the "'\0'".

String Variable Declarations and Assignments

String variables can be declared just like other arrays:

```
char phrase[14];
```

String arrays can be initialized or partially initialized at the same time as being declared, using a list of values enclosed in "{}" braces (the same is true of arrays of other data types). For example, the statement

```
char phrase[14] = {'E','n','t','e','r',' ','a','g','e',':',' ','\0'};
```

both declares the array "phrase" and initializes it to the state in Figure 6.5.1. The statement

```
char phrase[14] = "Enter age: ";
```

is equivalent. If the "14" is omitted, an array will be created just large enough to contain both the value ""Enter age: "" and the sentinel character "'\0'", so that the two statements

```
char phrase[] = {'E','n','t','e','r',' ','a','g','e',':',' ','\0'};
char phrase[] = "Enter age: ";
```

are equivalent both to each other and to the statement

```
char phrase[12] = "Enter age: ";
```

However, it is important to remember that string variables are arrays, so we cannot just make assignments and comparisons using the operators "=" and "==". We cannot, for example, simply write

```
phrase = "You typed: ";
```

Instead, we can use a special set of functions for string assignment and comparison.

Some Predefined String Functions

The library cstring (old style header string.h) contains a number of useful functions for string operations. We will assume that the program fragments discussed below are embedded in programs containing the "include" statement

```
#include<cstring>
```

Given the string variable "a_string", we can copy a specific string value or the contents of another string to it using the two argument function "strcpy(...)". Hence the statement

```
strcpy(a_string, "You typed: ");
```

assigns the first 11 elements of "a_string" to the respective characters in ""You typed: "", and assigns the sentinel character "'\0'" to the 12th element. The call

```
strcpy(a_string, another_string);
```

copies the string value stored in "another_string" to "a_string". But care has to be taken with this function. If "a_string" is less than (1 + L), where L is the length of the string value currently stored in "another_string", the call to the function will cause a range bound error which will not be detected by the compiler.

We can, however, check the length of the value stored in "another_string" using the function "strlen(...)". The call "strlen(another_string)" returns the length of the current string stored in "another_string" (the character "'\0'" is not counted).

The comparison function "strcmp(...)" returns "False" (i.e. 0) if its two string arguments are the same, and the two argument function "strcat(...)" concatenates its second argument onto the end of its first argument. Program 6.5.1 illustrates the use of these functions. Again, care must be taken with "strcat(...)". C++ does not check that the first variable argument is big enough to contain the two concatenated strings, so that once again there is a danger of undetected range bound errors.

String Input using "getline(...)"

Although the operator ">>" can be used to input strings (e.g. from the keyboard), its use is limited because of the way it deals with space characters. Supposing a program which includes the statements

```
...
...
cout << "Enter name: ";
cin >> a_string;
...
...
```

results in the input/output session

```
...
...
Enter name: Rob Miller
...
...
```

The string variable will then contain the string value "Rob", because the operator ">>" assumes that the space character signals the end of input. It is therefore often better to use the two argument function "getline(...)". For example, the statement

```
cin.getline(a_string,80);
```

allows the user to type in a string of up to 79 characters long, including spaces. (The extra element is for the sentinel character.) The following program illustrates the use of "getline(...)", "strcmp(...)", "strcpy(...)" and "strcat(...)":

```
#include <iostream>
#include <cstring>

using namespace std;

const int MAXIMUM_LENGTH = 80;

int main()
{
        char first_string[MAXIMUM_LENGTH];
        char second_string[MAXIMUM_LENGTH];

        cout << "Enter first string: ";
        cin.getline(first_string,MAXIMUM_LENGTH);
        cout << "Enter second string: ";
        cin.getline(second_string,MAXIMUM_LENGTH);

        cout << "Before copying the strings were ";
        if (strcmp(first_string,second_string))
                cout << "not ";
        cout << "the same.\n";

        strcpy(first_string,second_string);

        cout << "After copying the strings were ";
        if (strcmp(first_string,second_string))
                cout << "not ";
        cout << "the same.\n";
```

```
        strcat(first_string,second_string);

        cout << "After concatenating, the first string is: ";
        cout << first_string;

        return 0;
}
```

Program 6.5.1

An example input/output session is:

Enter first string: **Hello class.**
Enter second string: **Hello Rob.**
Before copying the strings were not the same.
After copying the strings were the same.
After concatenating, the first string is: Hello Rob.Hello Rob.

Pointers

Introducing Pointers

In the previous lectures, we have not given many methods to control the amount of memory used in a program. In particular, in all of the programs we have looked at so far, a certain amount of memory is reserved for each declared variable at compilation time, and this memory is retained for the variable as long as the program or block in which the variable is defined is active. In this lecture we introduce the notion of a *pointer*, which gives the programmer a greater level of control over the way the program allocates and de-allocates memory during its execution.

Declaring Pointers

A pointer is just the memory address of a variable, so that a *pointer variable* is just a variable in which we can store different memory addresses. Pointer variables are declared using a "*", and have data types like the other variables we have seen. For example, the declaration

```
int *number_ptr;
```

states that "number_ptr" is a pointer variable that can store addresses of variables of data type "int". A useful alternative way to declare pointers is using a "typedef" construct. For example, if we include the statement:

```
typedef int *IntPtrType;
```

we can then go on to declare several pointer variables in one line, without the need to prefix each with a "*":

```
IntPtrType number_ptr1, number_ptr2, number_ptr3;
```

Assignments with Pointers Using the Operators "*" and "&"

Given a particular data type, such as "int", we can write assignment statements involving both ordinary variables and pointer variables of this data type using the *dereference operator* "*" and the (complementary) *address-of operator* "&". Roughly speaking, "*" means "the variable located at the address", and "&" means "the address of the variable". We can illustrate the uses of these operators with a simple example program:

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

typedef int *IntPtrType;

int main()
```

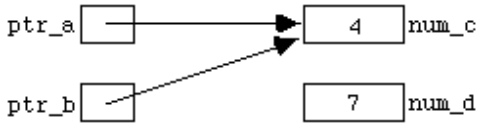
```
{
        IntPtrType ptr_a, ptr_b;
        int num_c = 4, num_d = 7;

        ptr_a = &num_c;          /* LINE 10 */
        ptr_b = ptr_a;           /* LINE 11 */

        cout << *ptr_a << " " << *ptr_b << "\n";

        ptr_b = &num_d;          /* LINE 15 */

        cout << *ptr_a << " " << *ptr_b << "\n";

        *ptr_a = *ptr_b;         /* LINE 19 */

        cout << *ptr_a << " " << *ptr_b << "\n";

        cout << num_c << " " << *&*&*&num_c << "\n";

        return 0;
}
```

Program 7.1.1

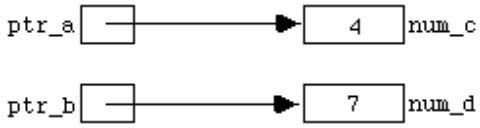
The output of this program is:

```
4 4
4 7
7 7
7 7
```

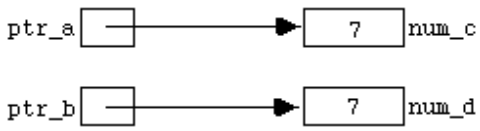
Diagramatically, the state of the program after the assignments at lines 10 and 11 is:



after the assignment at line 15 this changes to:



and after the assignment at line 19 it becomes:



Note that "*" and "&" are in a certain sense complementary operations; "*&*&*&num_c" is simply "num_c".

The "new" and "delete" operators, and the constant "NULL".

In Program 7.1.1, the assignment statement

```
ptr_a = &num_c;
```

(in line 10) effectively gives an alternative name to the variable "num_c", which can now also be referred to as "*ptr_a". As we shall see, it is often convenient (in terms of memory management) to use dynamic variables in programs. These variables have no independent identifiers, and so can only be referred to by dereferenced pointer variables such as "*ptr_a" and "*ptr_b".

Dynamic variables are "created" using the reserved word "new", and "destroyed" (thus freeing-up memory for other uses) using the reserved word "delete". Below is a program analogous to Program 7.1.1, which illustrates the use of these operations:

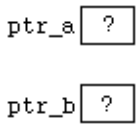
```
#include <iostream>
using namespace std;

typedef int *IntPtrType;

int main()
{
        IntPtrType ptr_a, ptr_b;    /* LINE 7 */

        ptr_a = new int;           /* LINE 9 */
        *ptr_a = 4;
        ptr_b = ptr_a;             /* LINE 11 */

        cout << *ptr_a << " " << *ptr_b << "\n";

        ptr_b = new int;           /* LINE 15 */
        *ptr_b = 7;                /* LINE 16 */

        cout << *ptr_a << " " << *ptr_b << "\n";

        delete ptr_a;
        ptr_a = ptr_b;             /* LINE 21 */

        cout << *ptr_a << " " << *ptr_b << "\n";

        delete ptr_a;              /* LINE 25 */

        return 0;
}
```

Program 7.1.2

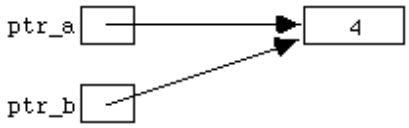
The output of this program is:

```
4 4
4 7
7 7
```

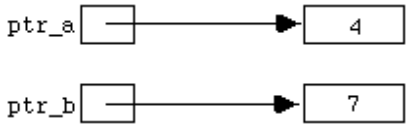
The state of the program after the declarations in line 7 is:



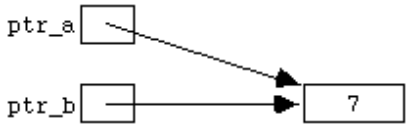
after the assignments in lines 9, 10 and 11 this changes to:



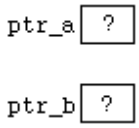
after the assignments at lines 15 and 16 the state is:



and after the assignment at line 21 it becomes:



Finally, after the "delete" statement in lines 25, the program state returns to:



In the first and last diagrams above, the pointers "ptr_a" and "ptr_b" are said to be dangling. Note that "ptr_b" is dangling at the end of the program even though it has not been explicitly included in a "delete" statement.

If "ptr" is a dangling pointer, use of the corresponding dereferenced expression "*ptr" produces unpredictable (and sometimes disastrous) results. Unfortunately, C++ does not provide any inbuilt mechanisms to check for dangling pointers. However, safeguards can be added to a program using the special symbolic memory address "NULL". Any pointer of any data type can

be set to "NULL". For example, if we planned to extend Program 7.1.2 and wanted to safeguard against inappropriate use of the dereferenced pointer identifiers "*ptr_a" and "*ptr_b", we could add code as follows:

```
#include <iostream>
...
...
        delete ptr_a;
        ptr_a = NULL;
        ptr_b = NULL;
        ...
        ...
        if (ptr_a != NULL)
        {
                *ptr_a = ...
                ...
                ...
```

Fragment of Program 7.1.3

In the case that there is not sufficient memory to create a dynamic variable of the appropriate data type after a call to "new", C++ automatically sets the corresponding pointer to "NULL". Hence the following code typifies the kind of safety measure that might be included in a program using dynamic variables:

```
#include <iostream>
#include <cstdlib>   /* ("exit()" is defined in <cstdlib>) */
...
...
        ptr_a = new int;
        if (ptr_a == NULL)
        {
                cout << "Sorry, ran out of memory";
                exit(1);
        }
        ...
        ...
```

Fragment of Program 7.1.4

Pointers can be used in the standard way as function parameters, so it would be even better to package up this code in a function:

```
void assign_new_int(IntPtrType &ptr)
{
        ptr = new int;
        if (ptr == NULL)
        {
                cout << "Sorry, ran out of memory";
                exit(1);
        }
}
```

Array Variables and Pointer Arithmetic

In the last topic we saw how to declare groups of variables called arrays. By adding the statement

```
int hours[6];
```

we could then use the identifiers

```
hours[0]   hours[1]   hours[2]   hours[3]   hours[4]   hours[5]
```

as though each referred to a separate variable. In fact, C++ implements arrays simply by regarding array identifiers such as "hours" as pointers. Thus if we add the integer pointer declaration

```
int *ptr;
```

to the same program, it is now perfectly legal to follow this by the assignment

```
ptr = hours;
```

After the execution of this statement, both "ptr" and "hours" point to the integer variable referred to as "hours[0]". Thus "hours[0]", "*hours", and "*ptr" are now all different names for the same variable. The variables "hours[1]", "hours[2]", etc. now also have alternative names. We can refer to them either as

```
*(hours + 1)   *(hours + 2)   ...
```

or as

```
*(ptr + 1)   *(ptr + 2)   ...
```

In this case, the "+ 2" is shorthand for "plus enough memory to store 2 integer values". We refer to the addition and subtraction of numerical values to and from pointer variables in this manner as pointer arithmetic. Multiplication and division cannot be used in pointer arithmetic, but the increment and decrement operators "++" and "--" can be used, and one pointer can be subtracted from another of the same type.

Pointer arithmetic gives an alternative and sometimes more succinct method of manipulating arrays. The following is a function to convert a string to upper case letters:

```
void ChangeToUpperCase(char phrase[])
{
        int index = 0;
        while (phrase[index] != '\0')
        {
                if (LowerCase(phrase[index]))
                        ChangeToUpperCase(phrase[index]);
                index++;
```

```
		}
}

int LowerCase(char character)
{
		return (character >= 'a' && character <= 'z');
}

void ChangeToUpperCase(char &character)
{
		character += 'A' - 'a';
}
```

Fragment of Program 7.2.1

Note the use of polymorphism with the function "ChangeToUpperCase(...)" - the compiler can distinguish the two versions because one takes an argument of type "char", whereas the other takes an array argument. Since "phrase" is really a pointer variable, the array argument version can be re-written using pointer arithmetic:

```
void ChangeToUpperCase(char *phrase)
{
		while (*phrase != '\0')
		{
				if (LowerCase(*phrase))
						ChangeToUpperCase(*phrase);
				phrase++;
		}
}
```

Fragment of Program 7.2.2

This re-writing is transparent as far as the rest of the program is concerned - either version can be called in the normal manner using a string argument:

```
char a_string[] = "Hello World";
...
...
ChangeToUpperCase(a_string);
```

Dynamic Arrays

The mechanisms described above to create and destroy dynamic variables of type "int", "char", "float", etc. can also be applied to create and destroy dynamic arrays. This can be especially useful since arrays sometimes require large amounts of memory. A dynamic array of 10 integers can be declared as follows:

```
int *number_ptr;
number_ptr = new int[10];
```

As we have seen, array variables are really pointer variables, so we can now refer to the 10 integer variables in the array either as

```
number_ptr[0]   number_ptr[1]     ...                    number_ptr[9]
```

or as

```
*number_ptr   *(number_ptr + 1)   ...          *(number_ptr + 9)
```

To destroy the dynamic array, we write

```
delete [] number_ptr;
```

The "[]" brackets are important. They signal the program to destroy all 10 variables, not just the first. To illustrate the use of dynamic arrays, here is a program fragment that prompts the user for a list of integers, and then prints the average on the screen:

```
...
...
int no_of_integers, *number_ptr;

cout << "Enter number of integers in the list: ";
cin >> no_of_integers;

number_ptr = new int[no_of_integers];
if (number_ptr == NULL)
{
        cout << "Sorry, ran out of memory.\n";
        exit(1);
}

cout << "type in " << no_of_integers;
cout << " integers separated by spaces:\n";
for (int count = 0 ; count < no_of_integers ; count++)
        cin >> number_ptr[count];
cout << "Average: " << average(number_ptr,no_of_integers);

delete [] number_ptr;
...
...
```

Dynamic arrays can be passed as function parameters just like ordinary arrays, so we can simply use the definition of the function "average()" from the previous lecture with this program.

Automatic and Dynamic Variables

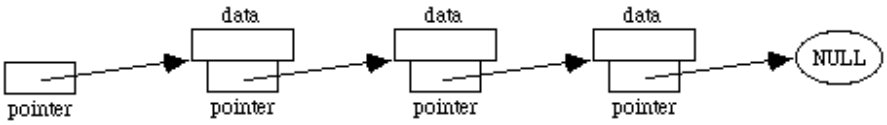
Although dynamic variables can sometimes be a useful device, the need to use them can often be minimized by designing a well structured program, and by the use of functional abstraction. Most of the variables we have been using in the previous lectures have been *automatic* variables. That is to say, they are automatically created in the block or function in which they are declared, and automatically destroyed at the end of the block, or when the call to the function terminates. So, for a well structured program, much of the time we don't even have to think about adding code to create and destroy variables.

(N.B. It is also possible to declare variables as being *static*, i.e. remaining in existence throughout the subsequent execution of the program, but in a well designed, non-object based program it should not be necessary to use any static variables other than the constants declared at the beginning.)

Linked Lists

In this section a brief description is given of an *abstract data type* (ADT) called a *linked list*, which is of interest here because it is implemented using pointers. You will learn much more about abstract data types in general later in the course.

In the implementation given below, a linked list consists of a series of *nodes*, each containing some data. Each node also contains a pointer pointing to the next node in the list. There is an additional separate pointer which points to the first node, and the pointer in the last node simply points to "NULL". The advantage of linked lists over (for example) arrays is that individual nodes can be added or deleted dynamically, at the beginning, at the end, or in the middle of the list.



In our example, we will describe how to implement a linked list in which the data at each node is a single word (i.e. string of characters). The first task is to define a node. To do this, we can associate a string with a pointer using a *structure definition*:

```
struct Node
{
        char word[MAX_WORD_LENGTH];
        Node *ptr_to_next_node;
};
```

or alternatively

```
struct Node;
typedef Node *Node_ptr;
```

```
struct Node
{
        char word[MAX_WORD_LENGTH];
        Node_ptr ptr_to_next_node;
};
```

(Note the semicolon after the "}".) The word "struct" is a reserved word in C++ (analogous to the notion of a *record* in Pascal). In the first line of the alternative (second) definition of a node above, "Node" is given an empty definition. This is a bit like a function declaration - it signals an intention to define "Node" in detail later, and in the mean time allows the identifier "Node" to be used in the second "typedef" statement.

The "." and "->" Operators

Having defined the structure "Node", we can declare variables of this new type in the usual way:

```
Node my_node, my_next_node;
```

The values of the (two) individual components of "my_node" can be accessed and assigned using the dot "." operator:

```
cin >> my_node.word;
my_node.ptr_to_next_node = &my_next_node;
```

In the case that pointers to nodes have been declared and assigned to nodes as follows:

```
Node_ptr my_node_ptr, another_node_ptr;
my_node_ptr = new Node;
another_node_ptr = new Node;
```

we can either use dot notation for these types of statement:

```
cin >> (*my_node_ptr).word;
(*my_node_ptr).ptr_to_next_node = another_node_ptr;
```

or write equivalent statements using the "->" operator:

```
cin >> my_node_ptr->word;
my_node_ptr->ptr_to_next_node = &my_next_node;
```

In other words, "my_node_ptr->word" simply means "(*my_node_ptr).word".

Creating a Linked List

Below is a function which allows a linked list to be typed in at the keyboard one string at a time, and which sets the node pointer "a_list" to point to the head (i.e. first node) of the new list. Typing a full-stop signals that the previous string was the end of the list.

```
void assign_list(Node_ptr &a_list)
{
        Node_ptr current_node, last_node;

        assign_new_node(a_list);
        cout << "Enter first word (or '.' to end list): ";
        cin >> a_list->word;
        if (!strcmp(".",a_list->word))
        {
                delete a_list;
                a_list = NULL;
        }
        current_node = a_list;                          /* LINE 13 */

        while (current_node != NULL)
        {
                assign_new_node(last_node);
                cout << "Enter next word (or '.' to end list): ";
                cin >> last_node->word;
                if (!strcmp(".",last_node->word))
                {
                        delete last_node;
                        last_node = NULL;
                }
                current_node->ptr_to_next_node = last_node;
                current_node = last_node;
        }
}
```

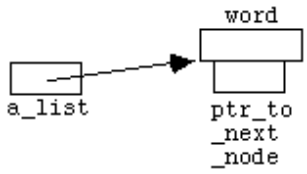
Fragment of Program 7.5.1

We will assume that the function "assign_new_node(...)" used in the above definition is exactly analogous to the function "assign_new_int(...)" in Program 7.1.5.

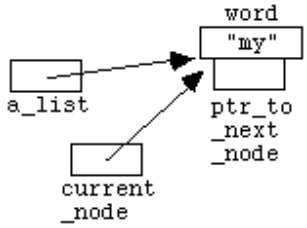
Here's how the function "assign_list(...)" works in words and diagrams. After the line

```
assign_new_node(a_list);
```

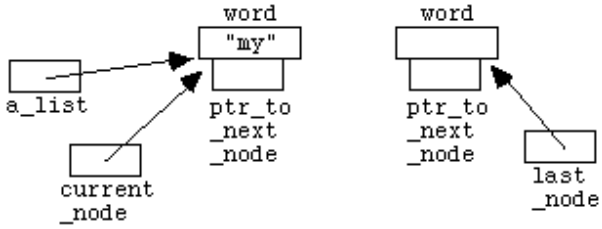
The state of the program is:



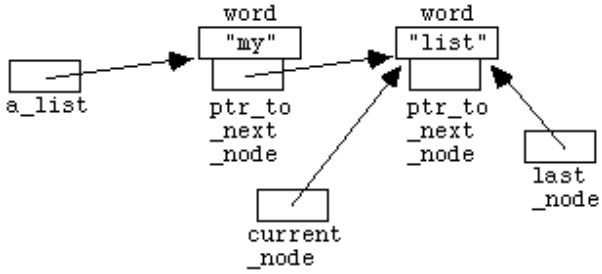
Assuming the user now types in "my" at the keyboard, after line 13 the program state is:



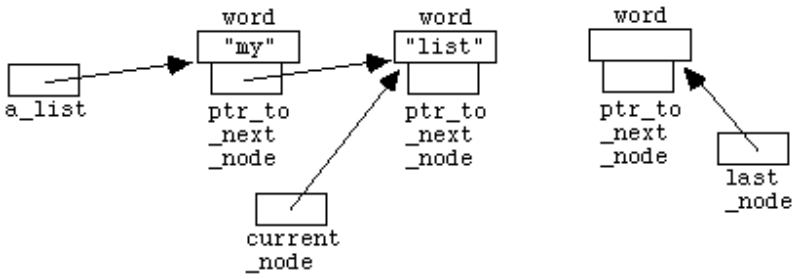
After the first line inside the "while" loop, the program state is:



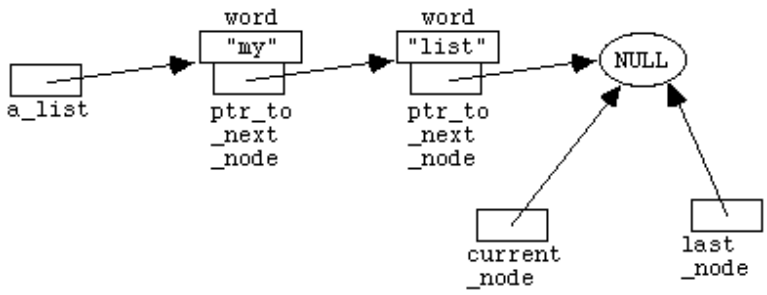
Assuming the user now types in "list" at the keyboard, after the "while" loop has finished executing for the first time the situation is:



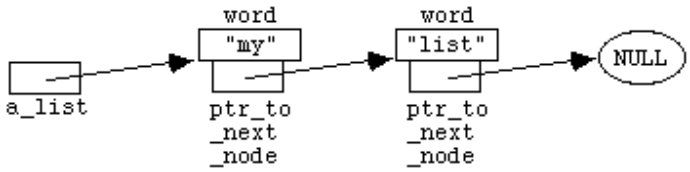
After the first line in the second time around the "while" loop, we have:



Assuming the user now types in "." at the keyboard, after the "while" loop has finished executing for the second time the situation is:



Since the condition for entering the "while" loop no longer holds, the function exits, the temporary pointer variables "current_node" and "last_node" (which were declared inside the function body) are automatically deleted, and we are left with:



Printing a Linked List

Printing our linked lists is straightforward. The following function displays the strings in the list one after another, separated by blank spaces:

```
void print_list(Node_ptr a_list)
{
        while (a_list != NULL)
        {
                cout << a_list->word << " ";
                a_list = a_list->ptr_to_next_node;
        }
}
```

Recursion

The Basic Idea

We have already seen how, in a well designed C++ program, many function definitions include calls to other functions (for example, in the last lecture the definition of "assign_list(...)" included a call to "assign_new_node(...)"). A function is *recursive* (or has a *recursive definition*) if the definition includes a call to itself.

Recursion is a familiar idea in mathematics and logic. For example, the natural numbers themselves are usually defined recursively. Very roughly speaking, the definition is:

- 0 is a natural number.
- if n is a natural number then s(n) (i.e. n+1) is a natural number, where s is the "successor function".

In this context, the notion of recursion is clearly related to the notion of *mathematical induction*. Notice also that the above definition includes a non-recursive part or *base case* (the statement that 0 is a natural number).

Another familiar mathematical example of a recursive function is the factorial function "!". Its definition is:

- 0! = 1
- for all n > 0, n! = nx(n-1)!

Thus, by repeatedly using the definition, we can work out that

6! = 6x5! = 6x5x4! = 6x5x4x3! = 6x5x4x3x2! = 6x5x4x3x2x1! = 6x5x4x3x2x1x1 = 720

Again, notice that the definition of "!" includes both a base case (the definition of 0!) and a recursive part.

Example

The following program includes a call to the recursively defined function "print_backwards()", which inputs a series of characters from the keyboard, terminated with a full-stop character, and then prints them backwards on the screen.

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

void print_backwards();
```

```
int main()
{
	print_backwards();
	cout << "\n";

	return 0;
}

void print_backwards()
{
	char character;

	cout << "Enter a character ('.' to end program): ";
	cin >> character;
	if (character != '.')
	{
		print_backwards();
		cout << character;
	}
}
```

Program 8.2.1

A typical input/output session is:

```
Enter a character ('.' to end program): H
Enter a character ('.' to end program): i
Enter a character ('.' to end program): .iH
```

We will examine how this function works in more detail in the next section. But notice that the recursive call to "print_backwards()" (within its own definition) is embedded in an "if" statement. In general, recursive definitions must always use some sort of branch statement with at least one non-recursive branch, which acts as the base case of the definition. Otherwise they will "loop forever". In Program 8.2.1 the base case is in the implicit "else" part of the "if" statement. We could have written the function as follows:

```
void print_backwards()
{
	char character;

	cout << "Enter a character ('.' to end program): ";
	cin >> character;
	if (character != '.')
	{
		print_backwards();
		cout << character;
	}
	else
	{
		;
	}
}
```

The Mechanics of a Recursive Call

It is easy to see why Program 8.2.1 works with the aid of a few diagrams. When the main program executes, it begins with a call to "print_backwards()". At this point space is set aside in the computer's memory to execute this call (and in other cases in which to make copies of the value parameters). This space is represented as a box in Figure 8.3.1a:

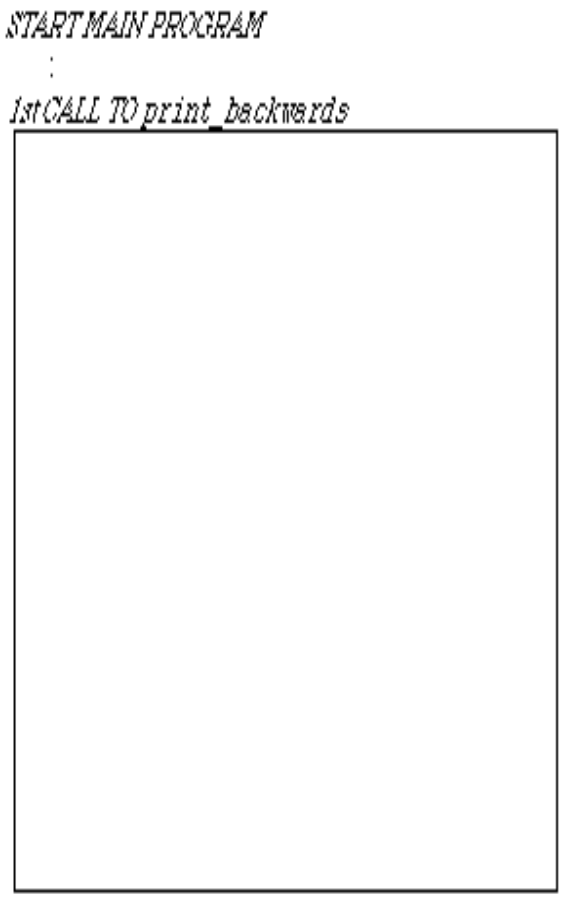


Figure 8.3.1a

The internal execution of this call begins with a character input, and then a second call to "print_backwards()" (at this point, nothing has been output to the screen). Again, space is set aside for this second call:

Figure 8.3.1b

The process repeats, but inside the third call to "print_backwards()" a full-stop character is input, thus allowing the third call to terminate with no further function calls:

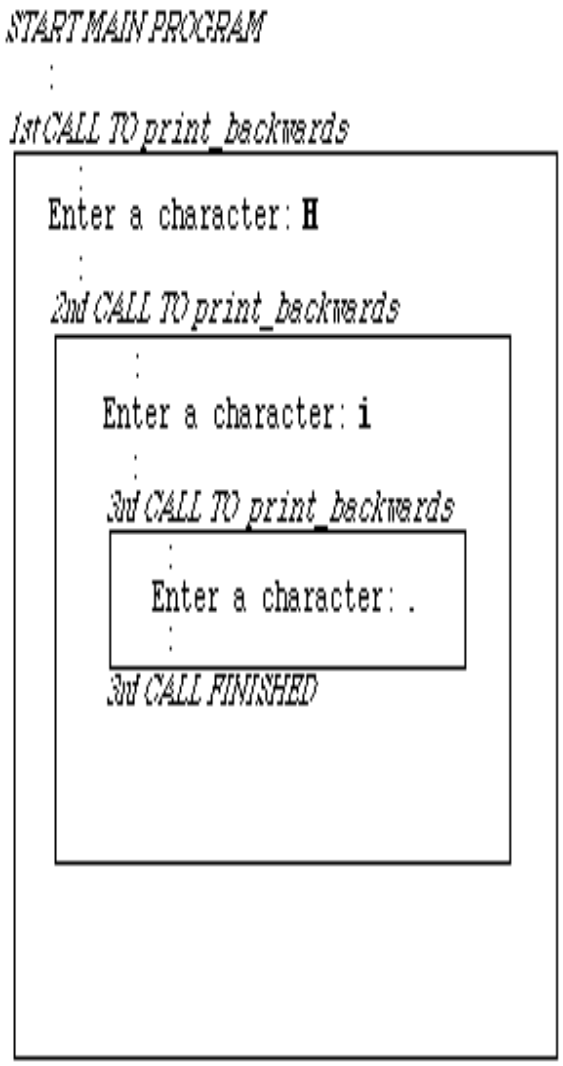


Figure 8.3.1c

This allows the second call to "print_backwards()" to terminate by outputting an "i" character, which in turn allows the first call to terminate by outputting an "H" character:

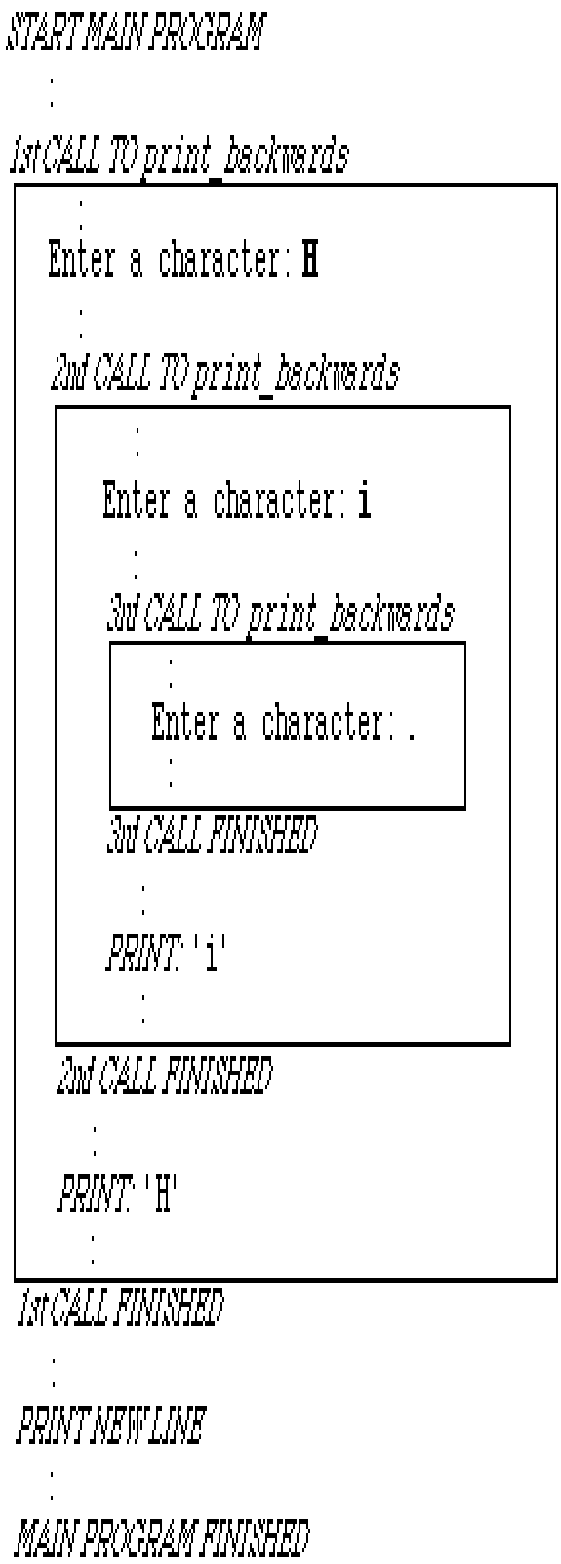

START MAIN PROGRAM
1st CALL TO print_backwards
Enter a character: H
2nd CALL TO print_backwards
Enter a character: i
3rd CALL TO print_backwards
Enter a character: .
3rd CALL FINISHED
PRINT: 'i'
2nd CALL FINISHED
PRINT: 'H'
1st CALL FINISHED
PRINT NEW LINE
MAIN PROGRAM FINISHED

Figure 8.3.1d

Technically speaking, C++ arranges the memory spaces needed for each function call in a *stack*. The memory area for each new call is placed on the top of the stack, and then taken off again when the execution of the call is completed. In the example above, the stack goes through the following sequence:

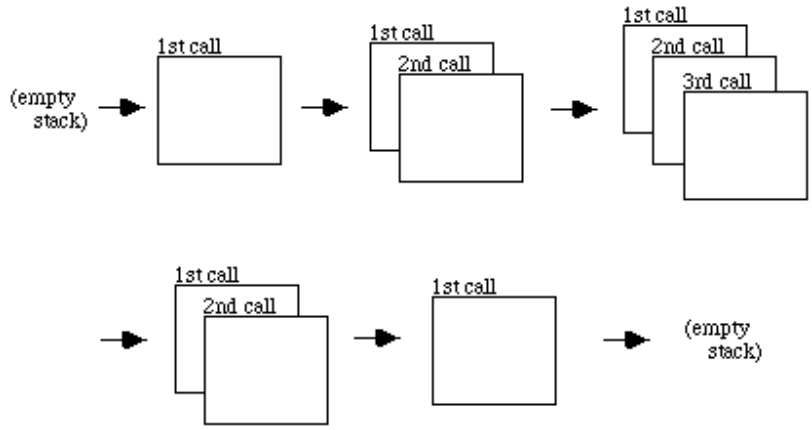


Figure 8.3.2

C++ uses this stacking principle for all nested function calls - not just for recursively defined functions. A stack is an example of a "last in/first out" structure (as opposed to, for example, a *queue*, which is a "first in/first out" structure).

Three More Examples

Below are three more examples of recursive functions. We have already seen a function to calculate the factorial of a positive integer. Here's an alternative, recursive definition:

```
int factorial(int number)
{
        if (number < 0)
        {
                cout << "\nError - negative argument to factorial\n";
                exit(1);
        }
        else if (number == 0)
                return 1;
        else
                return (number * factorial(number - 1));
}
```

Fragment of Program 8.4.1

As a second example, here's a function which raises its first argument (of type "float") to the power of its second (non-negative integer) argument:

```
float raised_to_power(float number, int power)
{
        if (power < 0)
        {
                cout << "\nError - can't raise to a negative power\n";
                exit(1);
        }
        else if (power == 0)
                return (1.0);
        else
                return (number * raised_to_power(number, power - 1));
}
```

Fragment of Program 8.4.2

In both cases, care has been taken that a call to the function will not cause an "infinite loop", - i.e. that the arguments to the functions will either cause the program to exit with an error message or are such that the series of recursive calls will eventually terminate with a base case.

The third example recursive function sums the first n elements of an integer array "a[]".

```
int sum_of(int a[], int n)
{
        if (n < 1 || n > MAXIMUM_NO_OF_ELEMENTS)
        {
                cout << "\nError - can only sum 1 to ";
                cout << MAXIMUM_NO_OF_ELEMENTS << " elements\n";
                exit(1);
        }
        else if (n == 1)
                return a[0];
        else
                return (a[n-1] + sum_of(a,n-1));
}
```

Fragment of Program 8.4.3

Recursion and Iteration

From a purely mechanical point of view, recursion is not absolutely necessary, since any function that can be defined recursively can also be defined iteratively, i.e. defined using "for", "while" and "do...while" loops. So whether a function is defined recursively or iteratively is to some extent a matter of taste. Here are two of the recursive functions above, re-defined iteratively:

```
void print_backwards()
{
        char chars[MAX_ARRAY_LENGTH];
        int no_of_chars_input = 0;

        do {
                cout << "Enter a character ('.' to end program): ";
                cin >> chars[no_of_chars_input];
                no_of_chars_input++;
        }
        while (chars[no_of_chars_input - 1] != '.'
                        && no_of_chars_input < MAX_ARRAY_LENGTH);

for (int count = no_of_chars_input - 2 ; count >=0 ; count--)
                cout << chars[count];
}
```

Fragment of Program 8.2.1b

```
int factorial(int number)
{
        int product = 1;

        if (number < 0)
        {
                cout << "\nError - negative argument to factorial\n";
                exit(1);
        }
        else if (number == 0)
                return 1;
        else
        {
                for ( ; number > 0 ; number--)
                product *= number;

                return product;
        }
}
```

Fragment of Program 8.4.1b

It's a matter of debate whether, for a particular function, a recursive definition is clearer than an iterative one. Usually, an iterative definition will include more local variable declarations - for example, the array "chars[MAX_ARRAY_LENGTH]" in the first example above, and the integer

variable "product" in the second example. In other words, temporary memory allocation is made explicit in the iterative versions of the functions by declaring variables, whereas it is implicit in the recursive definitions (C++ is implicitly asked to manipulate the stack by use of recursive calls).

Because of extra stack manipulation, recursive versions of functions often run slower and use more memory than their iterative counterparts. But this is not always the case, and recursion can sometimes make code easier to understand.

Recursive Data Structures

Recursive function definitions are often particularly useful when the program is manipulating *recursive data structures*. We have already seen one definition of a recursive data structure - the definition of a node in a linked list is given in terms of itself:

```
struct Node
{
        char word[MAX_WORD_LENGTH];
        Node *ptr_to_next_node;
};
```

Later in the course you will study other recursive data structures in more detail, and see how associated recursive function definitions behave in these contexts.

Quick Sort - A Recursive Procedure for Sorting

We will end this lecture by briefly looking at a standard recursive procedure for sorting. *Quick sort* is a recursively defined procedure for rearranging the values stored in an array in ascending or descending order.

Suppose we start with the following array of 11 integers:

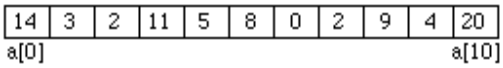


Figure 8.7.1

The idea is to use a process which separates the list into two parts, using a distinguished value in the list called a *pivot*. At the end of the process, one part will contain only values less than or equal to the pivot, and the other will contain only values greater than or equal to the pivot. So, if we pick 8 as the pivot, at the end of the process we will end up with something like:

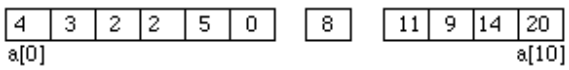


Figure 8.7.2

We can then reapply exactly the same process to the left-hand and right-hand parts separately. This re- application of the same procedure leads to a recursive definition.

The detail of the rearranging procedure is as follows. The index of the pivot value is chosen simply by evaluating

(first + last) / 2

where "first" and "last" are the indices of the initial and final elements in the array representing the list. We then identify a "left_arrow" and a "right_arrow" on the far left and the far right respectively. This can be envisioned as:

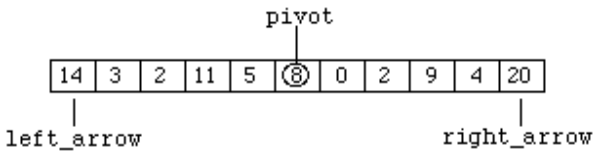


Figure 8.7.3

so that "left_arrow" and "right_arrow" initially represent the lowest and highest indices of the array components. Starting on the right, the "right_arrow" is moved left until a value less than or equal to the pivot is encountered. This produces:

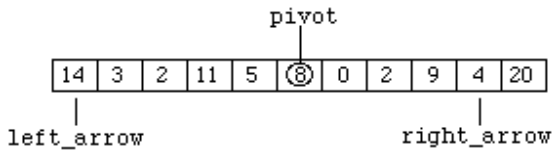


Figure 8.7.4

In a similar manner, "left_arrow" is moved right until a value greater than or equal to the pivot is encountered. This is already the situation in our example. Now the contents of the two array components are swapped to produce:

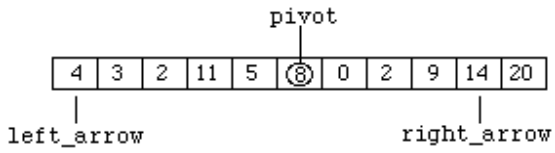


Figure 8.7.5

We continue by moving "right_arrow" left to produce:

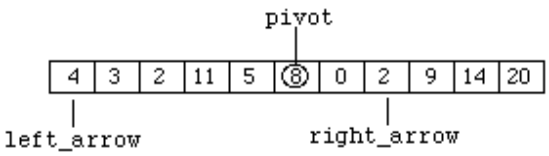


Figure 8.7.6

and then "left_arrow" right to produce:

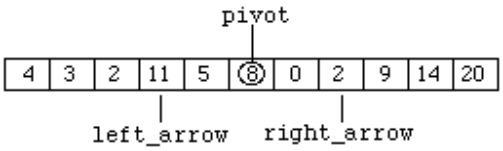


Figure 8.7.7

These values are exchanged to produce:

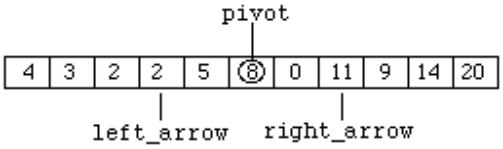


Figure 8.7.8

This part of the process only stops when the condition "left_arrow > right_arrow" becomes True. Since in Figure 8.7.8 this condition is still False, we move "right_arrow" left again to produce:

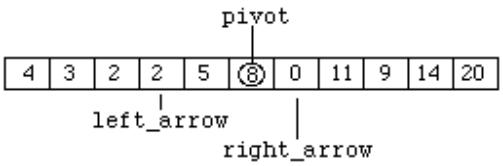


Figure 8.7.9

and "left_arrow" right again to produce:

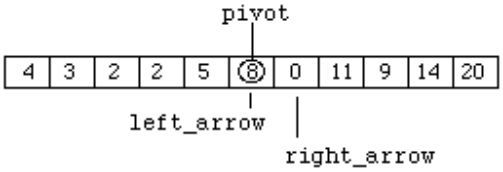


Figure 8.7.10

Because we are looking for a value greater than or equal to "pivot" when moving right, "left_arrow" stops moving and an exchange is made (this time involving the pivot) to produce:

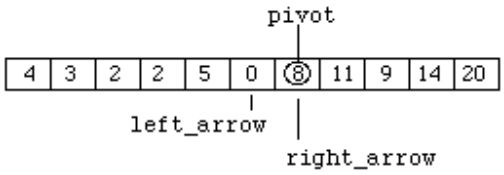


Figure 8.7.11

It is acceptable to exchange the pivot because "pivot" is the value itself, not the index. As before, "right_arrow" is moved left and "left_arrow" is moved right to produce:

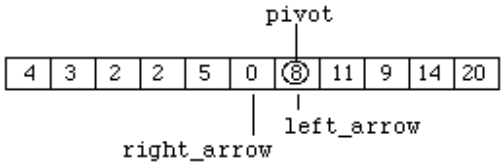


Figure 8.7.12

The procedure's terminating condition "left_arrow > right_arrow" is now True, and the first sub-division of the list (i.e. array) is now complete.

Here is the procedure Quick Sort coded up as a C++ function:

```
void quick_sort(int list[], int left, int right)
{
        int pivot, left_arrow, right_arrow;

        left_arrow = left;
        right_arrow = right;
        pivot = list[(left + right)/2];

        do
        {
                while (list[right_arrow] > pivot)
                        right_arrow--;
                while (list[left_arrow] < pivot)
                        left_arrow++;
                if (left_arrow <= right_arrow)
                {
                        swap(list[left_arrow], list[right_arrow]);
                        left_arrow++;
                        right_arrow--;
                }
        }
        while (right_arrow >= left_arrow);

        if (left < right_arrow)
```

```
                quick_sort(list, left, right_arrow);
        if (left_arrow < right)
                quick_sort(list, left_arrow, right);
}
```