5. C++ elementary data types

We have seen how symbols like `C`, `F`, `number`, `maximum` and `minimum` have been used to represent values in calculations.

We mentioned earlier that computers can use memory to store information in the form of integers, reals and characters. These are the elementary data types available in C++ (and most other high-level languages). If the value that is being stored in this location can vary then the identifier used is called a **variable**. If it is not subject to change then what is stored there is called a **constant**.

But first let's see what sorts of **identifiers** (names for entities) can be used.

5.1 Identifiers

In algebra most variables are represented by a single letter. This is fine when we know what that symbol represents. When we choose names to represent variables in our programs, we would like to be able to identify what that name represents. Words other than our identifiers appear in programs. We want to distinguish our identifiers. To accomplish this, we must follow a prescribed **syntax** so that the name is acceptable by the compiler.

An identifier can consist of a sequence of letters (both upper and lower case), digits and the underscore character (_). It should start with a letter. (Although the standard allows _ as the first letter, it is usually reserved for system use.) Some words have special meaning to the C++ compiler. These are called **reserved words** and cannot be used for our variables. They are:

```
and            const          false       not             signed     typeid
and_eq         const_cast     float       not_eq          sizeof     typename
asm            continue       for         operator        static     union
auto           default        friend      or              static_cast     unsigned
bitand         delete         goto        or_eq           struct     using
bitor          do             if          private         switch     virtual
bool           double         inline      protected       template   void
break          dynamic_cast   int         public          this       volatile
case           else           long        register        throw      wchar_t
catch          enum           mutable     reinterpret_cast           true while
char           explicit       namespace   return          try        xor
class          export         new         short           typedef    xor_eq
compl          extern
```

Identifiers containing two consecutive underscores should also be avoided, as these may also have special meanings. Compilers usually warn of the misuse of reserved words.

Some valid identifiers:

```
age, Celsius, litresperkm, LitresPerKm, litres_per_km
```

Some invalid ones:

```
1value, float, max-so-far, Freds height, π
```

Can you see why each is invalid?

The length of an identifier can be of any length, but you shouldn't need (or use) that many.

C++ is **case-sensitive**. This means that identifiers `dollar` and `Dollar` are different. Capitals or underscores are often used in identifiers to indicate word breaks in complex names (where normally a space would be used).

Let's now look at how C++ handles each of the basic data types.

5.2 Integers

C++ defines three basic forms of integer storage: `short`, `int`, and `long`.

The number of bytes of memory used for each of the types is dependent upon the implementation (computer and/or compiler). Standard C++ does place some conditions on the *relative* size of the three forms.

`short` integers must be at least 16 bits

`long` integers must be at least 32 bits

`short` integers cannot be longer than the type `int`

`long` integers cannot be shorter than the type `int`

In CodeWarrior, `shorts` are 16 bits and `longs` are 32 bits. The int data type is 32 bits.

When you write a program using integer values, use `int` unless you need to store numbers of a particular magnitude. For example, use a `long` if numbers exceeding 32767 are to be stored. Use a `short` if space is at a premium.

C++ identifies integer constants as strings of digits with an optional leading sign. For example

```
12     -345         +5678
```

Should the leading digit be a zero, the compiler interprets the number as an octal constant. If it begins with 0x (or 0X) it is a hex constant. Here are examples:

```
077    0124          044           0xff          0xAe4
```

To use an integer variable in a program, the variables must be **declared** as integer in a **variable declaration**. The C++ syntax is

```
type variable-identifier-list;
```

For example

```
int       distance, height1, employee_number;
short     age, IQ;
long      phone_no, Salary;
```

Note that the word `int` can follow the reserved word `short` or `long` but is usually omitted. Thus, the last two declarations sometimes appear as

```
short int age, IQ;
long int  phone_no, Salary;
```

Note how all the identifiers in the list are declared to be the same type. The use of a tab instead of a space after the type is a matter of style, but you should be consistent.

5.3 Reals

C++ provides three sizes of floating-point types: `float`, `double` and `long double`. Again, these are implementation-dependent with the proviso that `double` is no less precise than `float`, and `long double` is no less precise than `double`. (The word `double` can't be left out here, or the compiler would assume `long int`.) Most implementations use 32 bits for a `float`, and 64 bits for a `double`. CodeWarrior uses 64, 80 and 96 for a `double` for different option settings. `long double` in CodeWarrior is usually the same as `double`. As for integer declarations, use the type which provides the required precision and magnitude.
Here are some examples of real variable declarations:

```
float Temperature, height_in_feet;
double theta, rad_per_degree;
```

C++ provides two ways of representing real constants: fixed point and floating point.
A **fixed point** real constant consists of an integer part, a decimal point and a fractional part, where either the integer or fractional part can be missing, but not both.
Examples of fixed point constants are

```
23.5          0.345  50.          .5
```

Scientific notation or **floating point** allows us to write very small or very large numbers using the e notation for a multiplier power of 10. A floating point constant contains an integer constant or fixed point real constant, followed by either the character e (or E) and an integer exponent (either positive or negative).

Examples are

```
    0.12e5 .12E5        12e3          12.e3  120000e-1
```

C++ considers all real constants to be of type double.

5.4 Named constants

Often in programs we want to use special constants which are better referenced by a name. Universal constants such as π and e are prime examples. C++ allows us to specify identifiers which represent memory locations containing values which cannot change. Merely prefix the type by the keyword `const` and append an = and the constant's value. For example

```
const double pi = 3.14159265;   // geometric constant
const double e = 2.71828182864; // base of natural logs
```

In fact any value which is to be referenced within a program and is not to change can be specified as a `const`.

For example
```
const int Year = 1995;     // the current year
const short days_in_week = 7, minutes_in_hour = 60;
```

C++ interprets the keyword `const` to mean that the compiler must ensure that the value contained at the named memory location is not changed (even by accident) by the programmer's code.

5.5 Initialization

In the previous section, we saw how named constants are given a value. In fact any variable can be given a value when that variable is declared. For example

```
int i = 0;
double Altitude = 100.3;
```

Without the keyword `const`, however, these values can be altered by later operations.

5.6 Logical Values

Recently, the C++ standard has added a data type to enable the storage of logical values. The two possible values are the two possible results of any logical expression – **`true`** or **`false`**. The variable type is called **`bool`** and variables and named constants can be declared in the usual manner. For example

```
bool IsMale;
const bool use_debug = true;
```

The type `bool` is usually stored in one byte, with 0 representing `false` and 1 representing `true`, although ang non-zero valeu can be interpreted as `true`.

5.7 Characters

As was discussed earlier, computers can store letters and other characters using the ASCII code, each character occupying a byte.

Character constants in C++ consist of the character placed within a pair of single quotes, as in `'a'`, `'F'`, `'+'`, `'3'`. Note that the quotes are **not** part of the constant's value.

So how do you make a constant with the character `'` ?
Well, we use an **escape sequence** by preceding the `'` by a `\`, as in `'\''`.
If that's the case, how do we get a `\`?
Using `'\\'` of course.

It turns out that there is really only a small number of these escape sequences. We will see the use of many of these later.

new-line	`\n`	alert	`\a`
horizontal tab	`\t`	double quote	`\"`
vertical tab	`\v`	backslash	`\\`
backspace	`\b`	single quote	`\'`
carriage return	`\r`	question mark	`\?`
form feed	`\f`		

In fact any of the 256 (extended ASCII) characters can be generated by the escape sequence `\`*ooo* where *ooo* is an octal number (of one, two, or three octal digits), or by `\x`*hhh* where *hhh* is a similar hex number. Thus `'\101'` and `'\x41'` are the same as `'A'`. Any other escape sequence is not specified in the standard and hence could do anything.

A character variable is declared using the `char` type. Here are some examples.

```
const char dollar_sign = '$', backspace = '\b';
char first_initial;
```

Character constants and variables can also be used as an integer, with the value of the ASCII code for that character. Whether the top bit is treated as a sign bit, thereby giving character values in the range –128 to 127 instead of 0 to 255 is implementation dependent. Linux considers it to be the former. This is called a `signed char`. So there must be `unsigned chars`.

In fact, all integer types (`char`, `short`, `int`, and `long`) can be qualified with the word `unsigned` meaning that the sign bit is to be used as an extra bit of magnitude.
Thus

```
unsigned short I;
```

would mean that the variable `I` could take on any value in the range 0 to 65535. This means, however, that negative numbers cannot be stored in such unsigned variables.

Another form of character constant is called the **character string**. This consists of one or more characters such as `"The answer is "` or `"University of Wollongong"`. Each character in such a string occupies one byte. So that a program can recognise where a string ends in memory, the memory occupied by a string is terminated by a zero byte. Thus, `"The String"` occupies 11 bytes instead of the expected 10. The zero byte, with value `'\0'` is called the null character or NUL.

Remember how you get a `"` into a string? By using `\"`.

There is no basic C++ data type for the storage of strings as variables (or named constants). However, we will see later how such strings can be manipulated. We will also see later how C++ can define a string type, so we'll use the term **C-string** to refer to one of these null-terminated strings. (The term refers to the fact that the predecessor of C++, namely C, has this form of string only.)

6. Expressions

Obviously, the purpose of storing values on a computer is to manipulate them to produce further values. To do this, we need operators. The first of these are the arithmetic operators.

6.1 Arithmetic expressions

C++ provides five basic arithmetic operators:

`+`	addition, unary plus
`–`	subtraction, unary minus
`*`	multiplication
`/`	division
`%`	modulus (remainder in integer division)

When these operators are combined with variables and constants we create **arithmetic expressions**.

The unary operator + means essentially nothing. That is

```
+ expr
```

is treated as the same as *expr*.

The unary operator reverses the sign of the expression following it. For example

```
-x
```

has the magnitude of `x` but the opposite sign.

The five binary operators perform the common mathematical processes on the pair of operands on either side. But care must be taken concerning the type of the two operands. If the operands are both of the same type, then the operation is easily explained by the usual mathematical result. The result of such operations is of the same type as the operands.

In particular, integer division results in an integer answer. So

```
5/2
```

results in the answer `2`, not `2.5` as expected from mathematics.
The fifth operator `%` yields the remainder (of an **integer** division it is an error to use `%` on reals), so

```
5%2
```

has the value 1. This is often called modulo arithmetic in mathematics and is written

```
5 mod 2
```

C++ defines that

```
(x/y)*y + x%y
```

must always equal `x`.

This may seem obvious, but consider if one of the two operands is negative. Suppose that `x` and `y` above have the integer values `-5` and `2` respectively. One of the usual definitions of modulus specifies that the remainder must be positive. So `x % y` or `(-5) % 2` has a value of `1` so the division must have a result of `-3` for the expression above to be `x`.

Another approach is to say that the result of a division involving different signs is found by ignoring the signs, performing the division, truncating the answer and adding the sign back on to the result. That means that the result of `(-5)/2` is `-2` so the remainder must be `-1`.

g++ uses the latter. That is, if only one of the two operands of `%` is negative, the result is negative. For example, if `x` and `y` above were `5` and `-2`, then `x/y` would still be `-2` and `x%y` would now be `1`.

If the right operand of either of the integer operators `/` or `%` is zero, the result is an **exception** called **division by zero** which causes the program's execution to fail.

There are problems if the result of such an operation is not a valid value for the particular types.
For example, what is the result of

```
x + y
```

where `x` and `y` are both (signed) `shorts` with values of `32767` and `2`?
The arithmetic result 32769 cannot be represented as a `short` integer. This is also an exception in this case called an **overflow**. This exception does not cause the program to crash however. Integer overflows merely drift into the sign bit. Thus, the answer is reported as -32767, which shares its (twos complement) bit pattern with 32769. If the result is even bigger than 65535, then only the remainder modulo 65536 is the stored result (interpreted as a negative value if integer is signed).

Arithmetic involving unsigned integers produces results modulo 2^n where n is the number of bits in the representation. There is no overflow on `unsigned` integer types.

When `floats` overflow, such as the result of

```
u * v
```

when `u` and `v` are both `1e30` (a result of `1e60`), we get an **overflow**. The IEEE standard says the answer is then `INF` (infinity).

If the above variables are both `1e-30`, then the product should be `1e-60`, a result not permissible for a `float`. This is an **underflow**, but here the result is set to 0.

If the denominator is 0 for a real division, and the numerator is non-zero, the result is INF with the sign of the numerator. If both the numerator and denominator are zero, the result is mathematically undefined, so the standard sets the result to `NAN`, not a number.

Be aware that results with the value `±INF` can be used in further expressions, but NAN will generate more `NAN` results if used in further expressions.

6.2 Mixed mode expressions

We have covered all the possible binary expressions where both operands are of the same type. Many teaching languages prohibit expressions where the operands differ in type. These so-called **mixed mode** expressions are allowed in C++. When encountered, such operands are converted to identical types by **propagation** of the operand of less range to that of greater range.

Expressions involving mixed integer types other than `long` are all converted to type `int`, unless one of the operands is `unsigned`, in which case the common type is `unsigned int`. If the `unsigned` type's maximum value could fit in an `int`, then an `int` is still used.

An expression involving a `float` and a `double` would convert the `float`'s value to the equivalent `double` value and perform the operation with a `double` result.

If an expression contains an integer type and a real type, the integer is converted to the real type with a possible loss in precision.

Confused? Then try to avoid mixed expressions. We'll see later how we can perform our own type conversions to avoid them.

Although constants have an assumed type (such as `double` for reals and integer type determined by size), you can specify that a constant is of a particular precision by appending the letter `f` (or `F`) to force a real to be a `float`, and `l` (or `L`) to force an integer to be a `long`, or a real to be a `long double`. That is, the integer constant 23 is considered an `int`, while `23L` is a long.

```
float mine, yours, his;
mine = yours + 7.1;  //this is a double precision operation
mine = yours + 7.1f; //this is a single precision operation
```

6.3 More complex expressions

We've discussed binary expressions of the form

```
expr1 op expr2
```

but most of the time expressions involve more than two operands and one operator. For example,

```
5.*(F-32.)/9.
```

involves three operators and four operands. How is this expression evaluated? One operation at a time. It's called **precedence**.

Given an expression of the form

```
expr1 op1 expr2 op2 expr3
```

the operations are performed in either the order

expr1 op1 expr2 producing *result*
followed by *result op2 expr3*

or

expr2 op2 expr3 with answer *result*
followed by *expr1 op1 result*.

The precedence is `*`, `/` and `%` before `+` and `-`. If both are of the same precedence, say a `*` and a `/`, the calculation is performed **left to right**. Note that unary operators take precedence over all these binary operators.

Thus, the expression

```
5.*F-32./9.
```

would be performed in the order

```
5.*F          resulting in answer r1
32./9. resulting in answer r2 (=3.5....)
r1-r2
```

where *r1* and *r2* are intermediate results (not variables) stored internally in the computer.

If you wish to ensure that certain operations are performed first, surround the operator and its operands by `(` and `)`, which ensures precedence. Thus

```
5.*(F-32.)/9.
```

is calculated in the order

```
F-32.  giving r1
5./r1  giving r2
r2/9.
```

The following examples illustrate traps for the unwary.

```
(i)  5/9*a_double
```

yields 0 no matter what the `double` variable `a_double` is, since `5/9` is an integer division with result of 0. `a_double*5/9` would calculate correctly.

```
(ii) x/y*z
```

is the same as `x*z/y` **not** `x/(y*z)`.

Note that given an expression such as

```
expr1 op expr2
```

there is **no** specification as to which of the two operand expressions is evaluated first. This may not seem important now, but its importance will be noted soon.

You may also note that one of the traditional arithmetic operations, namely exponentiation (taking values to a power), does not have an equivalent C++ operator. Of course we can perform low magnitude integer powers by repeated multiplication, such as the square of `x` being `x*x`. Later on, we will see how generalised exponentiation can be performed.

6.4 Assignment operator

We saw earlier that we can give a variable an initial value when it is declared by merely following the name with an equal sign and the value. This is in fact an example of the use of the assignment operator =. The general form of an expression involving the assignment operator is

```
variable_name = expression
```

The meaning of this expression is to calculate the value of the expression (the right operand of the =) and then store that value in the memory location given the name of the variable on the left.

For example

```
Area_of_circle = pi*radius*radius
```

means that the computer fetches the value of `pi` (probably a named constant), multiplies that value by the value stored in the variable `radius`, then multiplies that intermediate result by the contents of `radius`. This final value is then stored in the variable `Area_of_circle`.

We usually say 'becomes' instead of 'equals' to avoid the confusion with the mathematical concept of =. Thus

```
A = 23
```

is said as ' A becomes 23' or 'A is given the value 23'.

If the types on either side of the = operator differ, the resultant value of the right operand is converted to the type of the left. This may involve loss of precision or an exception (overflow or underflow) or rounding.

Note that C++ specifies this to be an assignment **expression** which means that it also has a value, a concept not easily understood at this early stage in the knowledge of the C++ language. The type of this value is the same as that of the left operand (the variable). This means that more than one = can appear in an expression, such as

```
max_val = min_val = expression
```

In this case operands are determined from **right to left**. That is, the *`expression`* is evaluated, assigned to `min_val`, whose new value is also assigned to `max_val`.

As we said earlier, initialization of named constants and variables is a form of assignment operation. Thus such initialization can incorporate arithmetic expressions. For example,

```
const double cm_in_inch = 2.54;
const double inch_in_cm = 1./cm_in_inch;
```

In fact, many advocates of C++ advise that any declaration should contain an initialization. If such an initialisation requires the result of calculations appearing in the body of the program, then declaration of such a variable or constant should appear mixed in with calculations. This is contrary to traditional C which requires all declarations to appear before any other statements. In this course, style will follow the C convention. Thus, all named entities must be declared before any other statement type in a program block.

It should be noted here that a **constant expression**, a term which appears in many syntax descriptions, is an expression containing only numeric constants or expressions involving constants and named constants initialised with constant expressions. (A circular definition!)

6.5 Other Forms of assignment operator

Many times, in programs we want to modify the value stored in a particular variable by adding one to it, or multiplying it by something. For example

```
    sum = sum + x
```

or

```
    counter = counter + 1
```

or

```
    divisor = divisor * 2
```

These examples really show the difference between the assignment operator = and the mathematical concept of equality. C++ provides a shorthand for these assignments – in fact any assignment of the form

```
    variable = variable op expression
```

where *op* is any of the five arithmetic operators above (and others we will introduce later). The C++ shorthand is to define new assignment operators of the form *op*=, that is `+=`, `-=`, `*=`, `/=` and `%=`, leading to the form

```
    variable op= expression
```

So, the above three examples can be written as

```
    sum += x
```

and

```
    counter+= 1
```

and

```
    divisor *= 2
```

Note that, as the pair of characters is one operator, they must not be separated by whitespace. These new assignment operators act the same as the original operator in terms of order of operation when more than one assignment operator appears in an expression.

6.6 Incrementing and Decrementing

C++ provides two special unary operators for the express purpose of incrementing and decrementing the value of a variable by 1. These are the operators `++` for incrementing and `--` for decrementing. These two operators have a special property that the two unary operators + and do not have: they can appear before or after the operand! These are called **prefix** and **postfix** operators.

The expression

```
    ++variable
```

increments the *`variable`* by 1. Thus, it is essentially equivalent to

```
    variable += 1
```

and it has the same expression value, namely the **new** value of *`variable`*. Its advantage is the ability to perform two assignments without the need for two = operators.
For example

```
    sum += (++x)
```

would increment `x`, and then add the new value of `x` to `sum`, just as in

```
    sum = sum + (x = x + 1)
```

When used as a postfix operator, the incrementing is still performed on the variable but the expression has the **old** value of the variable.

For example

```
sum = 0.;
x = 5;
sum += (x++);
```

would result in `x` being 6, but `sum` would be 5.

Decrement operates analogously.

There is a danger here. When is the increment done? The standard says that the order of calculation of individual operands within a single operator expression is undefined. In particular, **side-effects** (such as increment and decrement) which occur separately from the calculation of the operands can only be guaranteed to have been performed upon reaching a semi-colon or the comma operator which has not yet been discussed).

The value generated by the expression

```
(x++) - (--x)
```

is indeterminate, as we cannot determine which of the expressions is evaluated first and when the postincrement is performed. It is well-advised not to use a variable in an expression involving other subexpressions with side-effects on that variable.

7. The C++ program

Now that we have some components of C++, let's try to put them together into a **program**. The simplest program consists of a **main** function (which will be explained in more detail later). Here is a skeleton of this basic program:

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

int main(int argc, char* argv[])
{

     return 0;
}
```

The line after the `{` is where we put the **statements** which make up the rest of our program. The lines up to the `{` and from `return` onwards will remain 'magic' for the moment.

We have already met some C++ statements. Most statements that appear in C++ programs are **expression statements**. These consist of an expression followed by a semi-colon. Examples are

```
     degF = 23.5;
     Initial = 'P';
     C = 5.*(F-32.)/9.;
```

To convert the temperature 100°F to Celsius, we could use the statement sequence

```
     degF = 100.;
     degC = 5.*(degF-32.)/9.;
```

There also exists the null or empty statement consisting of just the semi-colon.

Another form of statement is the **declaration statement** shown earlier. This is where we tell C++ the names of variables and constants we want to use in our program and indicate the type of each of these entities. Thus, we would have to proceed the above statements by

```
     float degC, degF;
```

otherwise, C++ would not know what type the two variables `degC` and `degF` are.

Our 'magic' program exhibits a further statement, called the **return statement**, which will be discussed later.

For purposes which will become obvious soon, statements can be collected into **blocks** called **compound statements** by the use of a pair of braces `{` and `}`. One level of indenting is usually performed inside the braces, as part of style. Our 'magic' program shows one such block which forms the **body** of the main program.

7.1 An introduction to Input/Output

One of the limitations of our coverage of C++ programs so far is that, althought we can calculate solutions to formulae, we can't get the computer to tell us the answer. For that we need to introduce some **output** statements.

C++ (and C before it) does not actually provide input/output as part of the language. In C, a special library called **standard I/O** (`stdio`) is provided. We could use that in C++ as well, but the most commonly used I/O in C++ is called **stream I/O**. At this point in the course, we'll introduce enough I/O to get by, leaving the detail until later.

Stream I/O provides access to the keyboard and the screen (and to files) via objects called **streams**. There are three standard streams automatically available. **`cin`** is an input stream, while **`cout`** is an output stream. **`cerr`**, another output stream, will be discussed later.

Let's look at output first.
A stream is a sequence of characters. The stream **`cout`** acts like a tap. It guides characters that we provide it to the screen of our computer. We can provide those characters to `cout` in the form of all the basic types we have discussed so far. For example, if we wanted to send the characters `Here is the answer:` to the screen, we could include

```
     cout << "Here is the answer:";
```

in our program. The operator << is used to insert characters into the stream – hence it is called the **insertion operator**. We can insert more than one entity into the stream in the one statement:

```
     cout << degF << " in degrees Fahrenheit is "
                << degC << " in Celsius.";
```

(Whoops. We've done something new. Note how this statement extends over two lines. C++ considers all whitespace – unless it is within a string – to be superfluous. This includes the return character at the end of a line. Thus, the above two lines make up one statement – terminated by the semi-colon.)

The stream library converts any non-character types, such as the `floats` `degF` and `degC`, to a string describing the value stored at the memory location referred to by the variable name, and then places that string in the stream.

Note the use of blanks at the beginning and end of the constant strings in the example above. This is because the conversion of numbers to strings does not generate any leading or trailing spaces. Hence, without the blanks, the output would run together.

Obviously, when we output strings to the screen, we sometimes want to place the strings on separate lines. We can use the newline character `\n` inside a string constant to generate a new line, or we can add the keyword `endl` to the stream. An example of the use of both methods is

```
      cout << "The first line.\n" << "The second line."
           << endl << endl << "The fourth line."  << endl;
```

Note that it takes two newlines to leave a blank line. It is common practice to place a newline at the end of each line of output. `endl` performs other tasks (apart from generating a new line) but is beyond the course at this time.

In an earlier section, we listed the keywords that C++ prohibits a programmer from using as identifiers in their programs. So how does C++ know about `cin`, `cout`, and `endl`? We have to tell it.

7.2 Header files

Stream I/O is an example of a **library**. This is a collection of pre-written programs and pre-defined variables and constants which we can use in our programs. So that C++ will know about these items and also so that we can reference the variables and constants, we provide C++ with a list of all the pre-defined identifiers. These are contained in a file, separate from our program, called a **header file**. Up until recently, we could always identify header files as they had the suffix ".h". However, recent changes to the C++ standard have introduced system header files without the extension. Later we will see how we can write our own header files, which we will **always** affix the .h to. Stream I/O supplies the header file `iostream` for us. To incorporate the definitions into our program, we include at the top of our file the line

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

as in our 'magic' program.
(The second line provides access to constants, types and classes that would normally require the explicit use of the prefix `std::` to access. Namespaces will not be covered in this course, other than to say that we will always incorporate the `using...` statement in our program files.)

In fact the compiler doesn't even see the line with the # on it. A program called the **C++ preprocessor** looks for instructions such as `#include` to replace the line by the file itself. We'll see other preprocessor commands later. For now, just realise that the < and > tell the preprocessor where to look for the file `iostream`. There must be no spaces after the < or before the > as they would become part of the filename. (Think of the angle brackets as similar to quotes surrounding a character string.)

One of the effects of the inclusion of a header file is the definition of many constants which we may never use or even know the existence of – until we try to declare the same identifier for our own variable or constant. We just have to hope that we don't encounter such problems.

If we do not include the header file, then the C++ compiler will not know the meaning of the streams, or the syntax of the stream I/O statements.

7.3 Back to I/O

Now that C++ knows about streams, we can incorporate output in our programs to tell us the answer to computations. For example

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

int main(int argc, char* argv[])
{
      float degF, degC;

      degF = 40.;
      degC = 5.*(degF-32.)/9.;
```

```
     cout << degF << " degrees F is " << degC << " degrees C\n";
     return 0;
}
```

will now convert 40ºF to its Celsius equivalent. Not a particularly useful program. What if we wanted to know what 50ºF is in Celsius? We could edit the program, changing the 4 to a 5, recompile it and then execute it again.

Or we could get the value we want to convert from the keyboard using the input stream `cin`. The input stream provides access to the characters we type on the keyboard. Obviously, we want such characters interpreted as various types of entities – numbers and characters and strings. Just like the reverse of `cout`, `cin` converts sequences of characters into equivalent integer or real values. It determines how it should convert characters by looking at what we want to get from the stream. For example

```
     cin >> degF;
```

would tell stream I/O that we want it to convert the characters to a `float` (the type of `degF`). Note the use of >> to mean 'extract from' the stream, hence the term **extraction operator**. Thus, the example means extract from the stream `cin` a `float` value and put that value in the storage location indicated by `degF`. This operates in much the same way as an assignment expression except for the origin of the value to be assigned. We could just replace the assignment statement in our program. But here is a more complete solution.

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

int main(int argc, char* argv[])
{
     float degF, degC;

     cout << "Please enter a Fahrenheit temperature: ";
     cin >> degF;
     degC = 5.*(degF-32.)/9.;
     cout << degF << " degrees F is " << degC << " degrees C\n";
     return 0;
}
```

The output of a message prior to the input of the needed value **prompts** the person running the program for the information required. Otherwise, the execution of the program would be held up, waiting for the value to be entered. Note also that we output the value of the input variable degF, even though we know what it is because just typed it in. We will see later that often programs are not run at a keyboard and screen and input is actually from a file.

What happens when we request values from `cin`? Characters entered from the keyboard are interpreted as follows. Any whitespace (blanks, newlines and tabs) are generally ignored, but used to separate values. If we are looking for a numeric value, all leading whitespace is ignored until any non-whitespace character is encountered. Next, all characters that are legitimate components of the numeric value are processed until an illegal character is encountered. At this point the number processed is passed to the variable waiting for that value. For example, if we are reading an integer, and the stream contains

```
     -212 degrees F
```

then the variable would receive the value –212. The stream would be positioned at the space following the second 2. The same value would be received if the stream were

```
     -212degrees F
```

or

```
     -212. degrees F
```

If however the stream were positioned after the reading of the integer and we requested another integer, all these streams would encounter an illegal character (for an integer) before any legal character. In this case, the value which is stored in the variable is undefined.

Consider this program.

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

int main(int argc, char* argv[])
{
     float degF, degC;

     cout << "Please enter a Fahrenheit temperature: ";
     cin >> degF;
     degC = 5.*(degF-32.)/9.;
     cout << degF << " degrees F is " << degC << " degrees C\n";
```

```
      cout << "Enter another Fahrenheit temperature:";
      cin >> degF;
      degC = 5.*(degF-32.)/9.;
      cout << degF << " degrees F is " << degC << " degrees C\n";
      return 0;
}
```

This time, we've duplicated the calculation statements (and the I/O) so that the program can convert two temperatures both stored (sequentially) in the variable `degF`. When execution reaches the first `cin` statement, stream I/O looks at the input stream coming from the keyboard and converts a sequence of characters into a `float` which is stored in `degF`. When the input is requested, nothing happens until we enter a return. We could have entered

```
      -212.<ret>
```
(<ret> means pressing the return key.)

Stream I/O would convert the sequence of characters to the value -212. and store that in `degF`. When we get to the next `cin` statement, we could then type in the next value.

BUT, we could have entered

```
      -212.  43.<ret>
```

in response to the first prompt. After the conversion of the `-212.`, the input stream still contains characters, which would be interpreted by the second `cin` request (as 43.).
Note that integer values in the input stream can be interpreted as real values, but real values will not be read as integers. In fact, the reading of an integer value will terminate at the decimal point, leaving the decimal point as the next character to be interpreted.

Although input can create errors due to the incorrect types being entered, we will assume for now that we can enter the correct values for our programs. We will see later how to handle the situation of errors in input.

How about character input?
Using the standard input stream, we cannot enter whitespace into a character variable, as `cin` ignores any whitespace. Thus, if a program contained

```
      cout << "Please enter initial: ";
      cin >> Initial:
```

and we typed

```
^^^^^F<ret>
```

where the ^ indicates a space, then the variable `Initial` would receive the value `'F'`.

We haven't indicated that we can ask for more than one variable to be read from the input stream, but you could guess that it's similar to outputting more than one. For example,

```
      cout << "Enter your height and age: ";
      cin >> Height >> Age;
```

would convert the input stream into two values.

If we ask for two-character values as in

```
      cout << "Enter two letters: ";
      cin >> a_letter >> another_letter;
```

then, although numerical values have to be separated by whitespace, we do not need to separate characters, so

```
      ab<ret>
```
or
```
      a  b<ret>
```

would yield the same input values for the two-character variables.

7.4 Logical expressions

In the next section we will introduce control structures which enable the programmer to manipulate the order in which statements are executed. One of the mechanisms underlying such control is the **logical expression**. This is an expression involving variables and constants which has one of the two possible `bool` values `true` or `false`.

There are nine operators available for use in logical expressions (although arithmetic operators can also appear).

They can be grouped into two types:

(i) **relational operators** are binary operators whose two operands are both of the same type, although one may be a real type while the other is an integer type.

The six relational operators are

<	is less than
<=	is less than or equal to
==	is equal to
!=	is not equal to
>=	is greater than or equal to
>	is greater than

Note the difference between the equality operator and the assignment operator. The expression

```
a = b
```

means that the variable `a` is assigned the value that `b` currently has, and the whole expression has the value of the new value of `a`, while

```
a == b
```

has the value `true` if `a` and `b` both have the same value and `false` otherwise.
Confusing these two can cause major problems. Many compilers warn the programmer of possible confusions between the two operators.

(ii) **logical operators** are used to combine logical expressions into more complex logical expressions. There is one unary operator ! (meaning not) which is used to negate the value of a logical expression. Thus

```
age > 60
```

is `true` if the variable `age` exceeds the value 60 and is `false` if `age` is less than or equal to 60. Then

```
!(age > 60)
```

is `false` if the expression inside the brackets is `true`, and `true` if the expression is `false`.

The other two logical operators are the binary operators

`&&`	logical AND
`||`	logical OR

The value of a logical expression such as

`expr1` `&&` *`expr2`*

depends on the combination of the values of the two expressions, as shown in the **truth table** on the right.

That is, the expression is `true` only if both the operands are true.

For example, if the two integer variables `num1` and `num2` are `1` and `2` respectively then

```
(num1 == 1) && (num2 == 2) is true
(num1 == 1) && (num2 < 2)  is false
!(num1 < 1) && !(num2 > 2) is true
```

		expr2	
		true	false
expr1	true	true	false
	false	false	false

The value of the expression

```
expr1 || expr2
```

is summarized in this truth table.

		expr2	
		true	false
expr1	true	true	true
	false	true	false

That is, the expression is `false` only if both operands are `false`.

For example, using the previous two variables `num1` and `num2`

```
(num1 == 1) || (num2 == 2)        is true
(num1 == 1) || (num2 < 2)         is true
(!(num1 < 1)) || (!(num2 > 2))    is true
```

The latter expression is equivalent to

```
!((num1 < 1) && (num2 > 2))
```

That is,

`!(expr1 && expr2)` is equivalent to `(!expr1) || (!expr2)`

Similarly

`!(expr1 || expr2)` is equivalent to `(!expr1) && (!expr2)`

Note the use of parentheses in the above.
When we start mixing arithmetic and logical operators, we need to ensure that the total expression is evaluated in the order we expect.

For example, the expression

```
b * b - 4. * a * c > 0.
```

would (logically) be expected to be calculated in the order

```
    b * b                        result1
    4. * a                       result2
    result2 * c                  result3
    result1 - result3            result4
    result4 > 0.
```

In fact that is how it is done. Here is a list of operator precedence, where parentheses override (and should be used if you are concerned about order).

highest

`!`	`+`	`-`		logical not, unary +, unary -
`*`	`/`	`%`		multiplication, division, modulus
`+`	`-`			addition, subtraction
`<`	`<=`	`>=`	`>`	relational inequality
`==`	`!=`			equal, not equal
`&&`				logical and
`||`				logical or
`=`				assignment

lowest

A Warning
You should be cautious about testing for exact equality between real values. Just because you may know that

```
(1./3.)*3. == 1.
```

should be `true`, remind yourself that fractions like `1./3.` are not exact and hence the expression on the left of the equality operator == may not be exactly 1. We will show later a better approach.

7.5 Collecting statements and expressions

In the next section we will introduce structures which control the order of execution of statements. Many times in programs we will need to manipulate a sequence of statements as if they were one statement. C++ provides two mechanisms: one for collecting statements together, and another for making many assignment expressions into one expression.

A **block** is a sequence of statements (of any type, including blocks) collected together by an opening `{` and a closing `}`.

For example

```
{
      a = 1;
      c = 'b';
}
```

could then be considered one statement. Note the use of indenting another bit of style. We've already seen such a construct in our 'magic container'.

The **comma** operator is a binary operator linking assignment expressions as in

```
a =1, c = 'b'
```

thereby producing one expression whose value is the value of the variable on the left of the rightmost assignment expression.

8. Control Structures

All the examples of programs described so far are executed **in sequence**. That is, the program's statements are executed one after the other from top to bottom. This is the normal sequence of execution.

There are, however, many situations in which the programmer may wish to vary this sequential process. For example, we might want to convert an arbitrary number of temperatures by repeating the same sequence of statements over and over. Or we may only want to execute certain statements depending upon certain conditions.

There are three main categories of statement execution:

(a) consecutive (one after the other);

(b) choice (or selection);

(c) repetition (or iteration).

Before getting down to the programming language level, it may be easier to see these groups at a more abstract level. In fact, before attempting to code any solution, we should remember that it is essential that the proposed solution is expressed in human-understandable form.

This very high level expression of the solution is called an **algorithm**, named after the Latinisation of a book title published by an early Arab mathematician *Abu Mohammad ibn Musa al-Kwarismi al-Magusi* .

The book was a collection of recipes or rules for using the (then) new mathematics which combined the decimal number system of the Hindu with the Arab invention of the *zypher* or place-holder (zero). The author's name gave us the name we now use for a set of step by step instructions to carry out a task an algorithm or algorism.

This concept of an algorithm is an important one for computer science since it is essential that the task given to the computer must be precisely stated and completely unambiguous. Later in your course you will encounter many traditional algorithms for searching and sorting.

In every problem solution it is essential to state the algorithm which will be used to solve each sub-problem. This can be done at the highest level of abstraction using **structured English**. At a more detailed level it is often expressed in **pseudocode** (which is something like a programming language but without the very precise syntactic rules) or by **flowcharts** (of which there are a number of varieties).

The three categories of statement execution mentioned above can be represented by all of these forms. Let's use simple flowcharts in which a **process** (something to be done) is shown in a rectangle, a **decision** (choice to be made) is shown in a diamond and the **flow of control** is shown by connecting lines along which the execution order passes from top to bottom and from left to right unless otherwise indicated by arrowheads on the lines.

The basic flow structures can be made to fit into one of the following shapes. Only one way in and one way out of each structure is allowed. Any process box can be replaced by any of the basic structures thereby leading to complex algorithms.

Let's look at each of these structures in turn.

8.1 Consecutive execution (sequence)

This is the most common form of program flow. Each statement is executed before the next. We saw earlier that, to create a program which repeats a process involves duplication of code.

8.2 Conditional execution (selection)

This structure indicates that the block of statements making up the process are only executed if the test produces a certain result. That is, the process is conditional upon that test result. But what is a test? It is the determination of the value of a logical expression which has only two possible values **true** or **false**. If the expression is true, the process is performed; otherwise, the process is skipped.

The C++ structure to perform this conditional execution is the **if** statement of the form

```
if (logical_expression)
     statement
```

where `statement` can be a block. For example

```
if (var_a < var_b)
     cout << "var_a is smaller than var_b\n";
```

or

```
if (b*b-4.*a*c < 0.)
{
     answer = 0.;
     cout << "The quadratic equation has no real solutions.\n";
}
```

If the process is one statement it can appear on the same line as the `if()`. However, as a matter of style, it is usually placed on the next line, indented by one position. Note that the `{` and `}` are not indented – the block is indented instead. But, again, this is just style.

***** WARNING *****
Here is one of C++'s major dangers for the novice programmer. Although the specification for the if statement indicates that the expression in the parentheses is a logical expression, many expressions in C++ can (and will) be automatically cast to a logical value. For example, an integer expression with a zero value will become a `false` Boolean value, while any non-zero value will be interpreted as `true`. Thus, you can consider the expression in such a way that the `statement` is executed if the value of the expression is non-zero. Only if the expression is zero is the `statement` not executed. This can be used to our advantage as we will see later.

Thus, the statement:

```
if (var_a == var_b)
     cout << "var_a is equal to var_b\n";
```

is correct but, by accident, we may write

```
if (var_a = var_b)
{
     cout << "var_a is equal to var_b\n";
}
```

Leaving out that second = sign changes a logical expression of equality to an assignment expression (which has a value) and, in this case, means the output message is always `true` (since `var_a` is assigned the value of `var_b` when the assignment expression is evaluated) but is printed if `var_b` is any non-zero value, but not if it is zero.

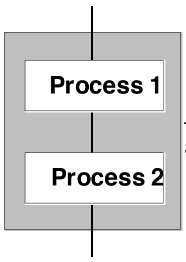
Again, many compilers provide a warning message if an assignment operator is located inside the parentheses of an `if` statement.

8.3 Alternate execution

Here the result of the test determines one of two alternate paths of execution. If the test expression is true (non-zero), then Process 1 is performed, otherwise Process 2 is executed.

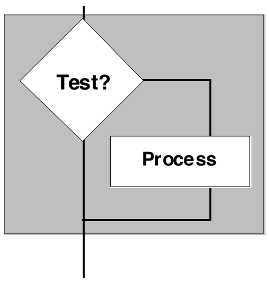
The C++ form is

```
     if (expression)
                statement1
          else
                statement2
```



Again, the indenting is style. *Statement1* is executed if *expression* is non-zero, and *statement2* is executed otherwise. For example,

```
if (b*b-4.*a*c == 0.)
     cout << "The only solution is " << -0.5*b/a << endl;
else
{
  if (b*b-4.*a*c < 0.)
       cout << "No real solutions." << endl;
  else
       cout << "There are two solutions." << endl;
}
```



Note that the statements in an `if()` can include another control structure as the alternate statement. Because the second `if` above is one statement, the `{` and `}` could be deleted without a change in the meaning. Thus

```
          if (b*b-4.*a*c == 0.)
               cout << "The only solution is " << -0.5*b/a << endl;
          else
               if (b*b-4.*a*c < 0.)
                     cout << "No real solutions." << endl;
     else
          cout << "There are two solutions." << endl;
```

There may, however, be cases in which there's some confusion, not necessarily by the compiler, but by a human reading the program. It is thus good for readability to include the `{` and `}` even if they're not required.

The above example illustrates one of the greatest strengths of these control structures: the ability to **nest** one inside another to practically any depth. The major problem with such nesting is that, due to the indenting of the content of each structure, there is less and less space on each line of the program. This makes readability of the program a concern.

In fact. in the above example, because the first item in the alternate block is another `if`, that `if` can be included on the same line as the `else` thus:

```
     if (b*b-4.*a*c == 0.)
          cout << "The only solution is " << -0.5*b/a << endl;
     else if (b*b-4.*a*c < 0.)
          cout << "No real solutions." << endl;
     else
          cout << "There are two solutions." << endl;
```

This structure is often called an **if-else if** control structure. It can also be considered as a **sieve**, as each successive `if` narrows down the possible range of the cases covered by the later alternatives.

Consider finding the largest of three values.

```
     cin >> a >> b >> c;
     if (a > b)           // either a or c is the largest
     {
          if (a > c)
               cout << "a is the largest." << endl;
          else
               cout << "c is the largest." << endl;
     }
     else if (b > c)      // we know b >= a from the first if
          cout << "b is the largest." << endl;
     else
          cout << "c is the largest." << endl;
```

When successive if tests involve the same calculation, such as the quadratic equation solver earlier, we could pre-evaluate the test value as follows.

```
     disc = b*b-4.*a*c;
     if (disc == 0.)
          cout << "The only solution is " << -0.5*b/a << endl;
     else if (disc < 0.)
          cout << "No real solutions." << endl;
     else
          cout << "There are two solutions." << endl;
```

or use the assignment operator inside the logical expression, as in

```
     if ((disc = b*b-4.*a*c) == 0.)
          cout << "The only solution is " << -0.5*b/a << endl;
     else if (disc < 0.)
          cout << "No real solutions." << endl;
     else
          cout << "There are two solutions." << endl;
```

Note the use of parentheses to ensure precedence of the assignment before the equality. Without the `(` and `)`

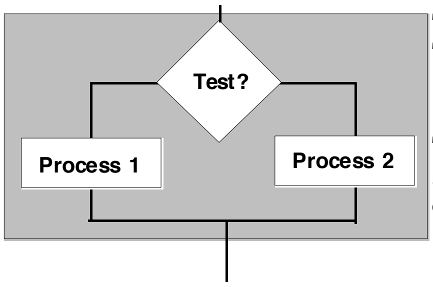
```
     disc = b*b-4.*a*c == 0.
```

would calculate `b*b-4.*a*c == 0.` first, yielding 1 for true and 0 for false and set the variable `disc` to be that value.

8.3.1 The conditional expression

Often the if-else structure is used to set a variable to two alternate values, as in

```
     if (x < 0)
          y = -x*x;
     else
          y = x*x;
```



There is a shorthand method of providing alternate values of an expression. The general form is

```
     condition ? expression1 : expression2
```

This is interpreted as follows: if the expression *`condition`* evaluates to non-zero (`true`) then the whole expression has the value of *`expression1`*, otherwise it equals *`expression2`*. Thus, our `if` above is equivalent to

```
     y = (x<0 ? -x*x : x*x);
```

Again the spaces are style. The parentheses are often included, especially when the conditional expression is part of a larger expression, because of its low precedence.

8.4 Case statement

This is a particular inplementation of the sieve, where one test separates execution into one of n possible alternatives. The C++ structure is called the **`switch`** statement.

It's form is:

```
     switch (expression)
     {
          case const_expr1:     statements1
                                break;
          case const_expr2:     statements2
                                break;
          . . .

          case const_exprn:     statementsn
                                break;
          default:         statements
     }
```

The *`expression`* within the `switch` statement must be of integral type. Each *`const_expr`* is converted to the same type. Once the expression has been evaluated, control is transferred to the case where the *`const_expr`* equals the value of the expression. The statements after that case are then executed. No braces are required to identify a block here, as the `case` labels delineate the blocks. The `break` statements cause control to transfer to the statement following the `switch` statement's closing `}`. Without the `break`, execution would continue with the next `case`'s statements.

The (optional) `default` statement is executed if the value of the expression does not equate to any of the specific cases.

Here is an example.

```
switch ((3*num+27)/4)
{
     case 1:
          cout << "The answer is 1" << endl;
          break;
     case 2:
          cout << "The answer is 2" << endl;
          break;
     default:
          cout << "The answer is neither 1 nor 2" << endl;
}
```

Note the use of indents for style purposes.

8.5 Repetition or iteration

There are two alternate structures leading to three C++ constructs.
These structures provide a way of repeating the execution of a statement (or block or any compound structure).

8.5.1 The while-do

This is an example of a **guarded loop** in which a process is or is not repeatedly executed depending upon the truth of some condition.

The loop guard is tested **first** to see whether one execution of the loop's process should be allowed. If so the statements in the body of the loop are executed, control is passed back to the start of the loop and the test carried out again.

The form of this statement in C++ is

```
while (expression)
     statement
```

Here is an example of its use.

```
test = 1.0;
newsmall = 1.0;
factor = 0.5;   // This factor is calculated once only
while ((test + newsmall) != test )
{
     oldsmall = newsmall;
     newsmall = newsmall * factor;
}
cout << "Precision is " << oldsmall << endl;
```

Since control always returns to the start of the loop to re-evaluate the expression before the next execution, it should be obvious that one thing that **must** be provided in the body of the loop is a possibility for the guard condition to be changed.

In the example above the **guard condition** is

```
test + newsmall != test
```

By providing a statement within the body of the loop which reassigns a value to `newsmall`

```
newsmall = newsmall * factor;
```

there is the possibility that the condition will change and hence that the loop will terminate.

It is **essential** in designing a "good" algorithm, to ensure that any loop is guaranteed to terminate and the programmer must be able to state under what conditions that will happen.

The while-do loop is often called a **pre-test** loop since the condition is tested before allowing execution of the loop body. It is usually better to design loops which are guarded.

Because of this pre-test it is possible that the body of the loop is **never** executed.

Here are some more examples.

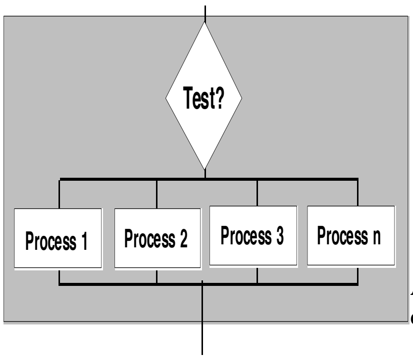
```
      i = 0;
      sum = 0;
      while ((i*i - 27*i + 4 ) < 500)
      {
           sum = sum + 1;
           i = i + 1;
      }
```
or
```
      cin >> num;
      while (num > 0)
           num = num * (num - 1);
/*    Depends on value of num read in */
```

What is wrong with the following program fragment?

```
      i = 0;
      while (i > 0)
      {
           i = i + 1;
           num = i * (i + 3) - 4;
      }
      cout << "The value of the number (num) is " << num;
```

Answer: Since `i` is set to zero before the loop is tested the first time, the loop is never executed.

What is wrong with this one?



```
i = 0;
Difference = 'A' - 'a';
cin >> Ch;
while (Ch >= 'a' && Ch <= 'z')
{
   Ch1 = Ch + Difference;
   i = i + 1;
   cout << Ch1 << " has appeared " << i << "times" <<endl;
}
```

Answer: Since `Ch` is never altered inside the loop, the guard condition never changes. The loop will run forever.

We can deliberately make a loop run forever by specifying a loop condition which is always true such as `while(true)...` But would we ever want to do that?

Well, yes. There may be some other condition which is determined within the body of the block which will trigger a termination of the loop.

To terminate a loop prematurely, we can use the statement

```
      break;
```

That is, use the same method as used for the `switch`. When this is encountered, for example

```
      if (b*b-4*a*c < 0) break;
```

the loop is terminated and control passes to the next statement to be performed after the loop's completion.

The statement

```
      continue;
```

provides for the sequence of statements within a loop's block to be curtailed, and the loop continued from the next repetition.

These statements can be used in all the loop constructs.

8.5.2 The for loop

This is an alternative implementation of the while-do structure, although it is used more frequently in a specific form.

The format of the C++ **for** statement is:

```
for (init_expression;test_expression;update_expression)
      statement
```

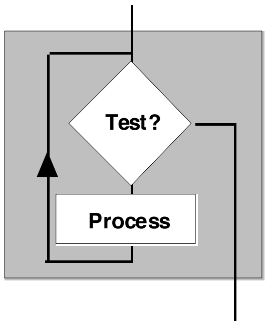
This is equivalent to the following while-do construct.

```
init_expression;
while (test_expression)
{
      statement;
      update_expression;
}
```

That is, the *init_expression* is evaluated and the *test_expression* is checked. If it is non-zero, the *statement* is executed followed by the *update_expression*. Then execution is returned to the calculation of the *test_expression*. Thus, any while-do can be written as a for loop.

The more common form of the for loop is used when the number of times the loop is to be executed is more precisely known. A loop variable is involved, which is initialised to some starting value in the *init_expression*, which will look something like

```
loop_var = starting_value
```



The test_expression is then an expression like

```
loop_var <= final_value
```

while the *update_expression* is of the form

```
loop_var = loop_var + step
```

Here are some examples. It is advisable that you can work out, for any for loop, the number of times the loop will be executed and the values that the loop index will take on.

```
sum = 0;
for (i=1;i<=100;i=i+1)     // loop executed 100 times
      sum = sum + count;

sum = 0;
for (i=100;i>0;i=i-1)      // loop executed 100 times
      sum = sum + count;
```

and

```
for (Ch='A';Ch<='z';Ch=Ch+5)
      cout << Ch;
/* outputs every 5th character from A to ? */
```

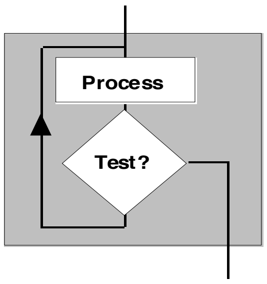
8.5.3 The do-while

This is an example of an **unguarded loop** in which a process is executed and then repeated depending upon the value of some expression.

The loop guard is tested **last** to see whether entry to the loop should be allowed again. If so, control is passed back to the start of the loop and the statements in the body of the loop are executed and the test carried out again.

The form of this statement in C++ is:

```
do
      statement
while (expression);
```



The do-while loop is also called a **post-test** loop since the condition is tested after execution of the loop body. Because of this post-test the body of the loop is **always** executed at least once. Another form of the do-while is the **repeat-until** in which the test condition being true terminates the loop. There is no repeat-until in C++, although using ! (logical not) on the condition in the do-while is equivalent.

As with all loops, it is essential that within the body of the loop there is provision for the guard condition to be changed.

```
test = 1.0;
```

```
newsmall = 1.0;
factor = 0.5;
do
{
     oldsmall = newsmall;
     newsmall = newsmall * factor;
}
while ((test + newsmall) != test);
cout << "Precision is " << oldsmall << endl;
```

In the following code, what is the logic error in the way the program is structured?

```
do
{
     cout << "Enter a value (>300.0 to terminate): ";
     cin >> degF;
     degC = 5.*(degF - 32.)/9.;
     cout << degF << " degrees F converts to " <<
          degC << " degrees C" << endl;
}
while (degF <= 300.);
```

Answer: The condition to terminate the loop (`degF > 300.`) is tested at the end of the loop and hence the sentinel value will be processed.