7 Iteration

7.1 WHILE LOOPS

Loop constructs permit slightly more interesting programs. Like most Algol-family languages, C++ has three basic forms of loop construct:

- while-loops (*while condition X is true keeping doing the following ...*)
- for-loops (mainly a "counting loop" construct – *do the following ten times* – but can serve in more general roles)
- repeat-loops (*do the following ... while condition Z still true*)

Repeat-loops are the least common; for-loops can become a bit complex; so, it is usually best to start with "while" loops.

While loops

These have the basic form:

```
while ( Expression )
    body of loop;
```

Where *Expression* is something that evaluates to "True" or "False" and body of loop defines the work that is to be repeated. (The way the values True and False are represented in C/C++ is discussed more later.)

When it encounters a while loop construct, the compiler will use a standard coding pattern that lets it generate:

- instructions that evaluate the expression to get a true or false result,
- a "jump if false" conditional test that would set the program counter to the start of the code following the while construct,
- instructions that correspond to the statement(s) in the body of the loop,

and

- a jump instruction that sets the program counter back to the start of the loop.

Sometimes, the body of the loop will consist of just a single statement (which would then typically be a call to a function). More often, the body of a loop will involve several computational steps, i.e. several statements.

Compound statements

Of course, there has to be a way of showing which statements belong to the body of the loop. They have to be clearly separated from the other statements that simply follow in sequence after the loop construct. This grouping of statements is done using "compound statements".

C and C++ use { ("begin bracket") and } ("end bracket") to group the set of statements that form the body of a loop.

```
while ( Expression ) {
    statement-1;
    statement-2;
    ...
    statement-n;
    }
/* statements following while loop, e.g. some output */
cout << "The result is ";
```

(These are the same begin-end brackets as are used to group all the statements together to form a main-routine or another routine. Now we have outer begin-end brackets for the routine; and inner bracket pairs for each loop.)

Code layout

How should "compound statements" be arranged on a page? The code should be arranged to make it clear which group of statements form the body of the while loop. This is achieved using indentation, but there are different indenting styles (and a few people get obsessed as to which of these is the "correct" style). Two of the most common are:

```
while (Expression) {
  statement-1;
  statement-2;
  ...
  statement-n;
  }
```

```
while (Expression)
 {
     statement-1;
     statement-2;
     ...
     statement-n;
 }
```

Sometimes, the editor in the integrated development environment will choose for you and impose a particular style. If the editor doesn't choose a style, you should. Be consistent in laying things out as this helps make the program readable.

Comparison expressions

Usually, the expression that controls execution of the while loop will involve a test that compares two values. Such comparison tests can be defined using C++'s comparison operators:

```
 ==        equals operator  (note, two = signs; the single
           equals sign is the assignment operator that
           changes a variable)
!=         not equals operator
>          greater than operator
>=         greater or equal operator
<          less than operator
<=         less than or equal operator
```

7.2 EXAMPLES

7.2.1 Modelling the decay of CFC gases

Problem:

Those CFC compounds that are depleting the ozone level eventually decay away. It is a typical decay process with the amount of material declining exponentially. The rate of disappearance depends on the compound, and is usually defined in terms of the compound's "half life". For example, if a compound has a half life of ten years, then half has gone in ten years, only a quarter of the original remains after 20 years etc.

Write a program that will read in a half life in years and which will calculate the time taken (to the nearest half life period) for the amount of CFC compound to decay to 1% of its initial value.

Specification:

1. The program is to read the half life in years of the CFC compound of interest (the half life will be given as a real value).

2. The program is to calculate the time for the amount of CFC to decay to less than 1% of its initial value by using a simple while loop.

3. The output should give the number of years that are needed (accurate to the nearest half life, not the nearest year).

Program design

1. What data?
   Some are obvious:
   a count of the number of half-life periods
   the half life

   Another data item will be the "amount of CFC". This can be set to 1.0 initially, and halved each iteration of the loop until it is less than 0.01. (We don't need to know the actual amount of CFC in tons, or Kilos we can work with fractions of whatever the amount is.)

   These are all "real" numbers (because really counting in half-lives, which may be fractional, rather than years).

2. Where should data be defined?
   Data values are only needed in single main routine, so define them as local to that routine.

3. Processing:
    a) prompt for and then input half life;
    b) initialize year count to zero and amount to 1.0;
    c) while loop:
        terminates when "amount < 0.01"
            body of loop involves
                halving amount and
                adding half life value to total years;
    d) print years required.

Implementation

The variable declarations, and a constant declaration, are placed at the start of the main routine. The constant represents 1% as a decimal fraction; it is best to define this as a named constant because that makes it easier to change the program if the calculations are to be done for some other limit value.

```
int main()
{
    double        numyears;
    double        amount;
    double        half_life;

    const double limit = 0.01; // amount we want
```

The code would start with the prompt for, and input of the half-life value. Then, the other variables could be initialized; the amount is 1.0 (all the CFC remains), the year counter is 0.

```
    cout << "Enter half life of compound " ;
    cin >> half_life;

    amount = 1.0; // amount we have now
    numyears = 0.0; // Number of years we have waited
```

As always, the while loop starts with the test. Here, the test is whether the amount left exceeds the limit value; if this condition is true, then another cycle through the body of the loop is required.

```
    while(amount > limit) {
           numyears = numyears + half_life;
           amount = amount*0.5;
           }
```

In the body of the loop, the count of years is increased by another half life period and the amount of material remaining is halved.

The program would end with the output statements that print the result.

```
    cout << "You had to wait " << numyears <<
```

```
                    " years" << endl;
```

7.2.2 Newton's method for finding a square root

Problem:

The math library, that implements the functions declared in the `math.h` header file, is coded using the best ways of calculating logarithms, square roots, sines and so forth. Many of these routines use power series polynomial expansions to evaluate their functions; sometimes, the implementation may use a form of interpolation in a table that lists function values for particular arguments.

Before the math library was standardized, one sometimes had to write one's own mathematical functions. This is still occasionally necessary; for example, you might need a routine to work out the roots of a polynomial, or something to find the "focus of an ellipse" or any of those other joyous exercises of senior high school maths.

Successive approximation algorithm

There is a method, an algorithm, due to Newton for finding roots of polynomials. It works by successive approximation – you guess, see whether you are close, guess again to get a better value based on how close you last guess was, and continue until you are satisfied that you are close enough to a root. We can adapt this method to find the square root of a number. (OK, there is a perfectly good square root function in the maths library, but all we really want is an excuse for a program using a single loop).

If you are trying to find the square root of a number `x`, you make successive guesses that are related according to the following formula

```
new_guess = 0.5 ( x / guess + guess)
```

This new guess should be closer to the actual root the original guess. Then you try again, substituting the new guess value into the formula instead of the original guess.

Of course, you have to have a starting guess. Just taking x/2 as the starting guess is adequate though not always the best starting guess.

You can see that this approach should work. Consider finding the square root of 9 (which, you may remember, is 3). Your first guess would be 4.5; your second guess would be 0.5(9/4.5 + 4.5) or 3.25; the next two guesses would be 3.009615 and 3.000015.

You need to know when to stop guessing. That aspect is considered in the design section below.

Now, if you restrict calculations to the real numbers of mathematics, the square root function is only defined for positive numbers. Negative numbers require the complex number system. We haven't covered enough C++ to write any decent checks on input, the best we could do would be have a loop that kept asking the user to enter a data value until we were given something non-negative. Just to simplify things, we'll make it that the behaviour of the program is undefined for negative inputs.

Specification:

1. The program is to read the value whose root is required. Input is unchecked. The behaviour of the program for invalid data, such as negative numbers is not defined.

2. The program is to use half of the input value as the initial guess for its square root.

3. The program is to use a while loop to implement the method of successive approximations. The loop is to terminate when the iterative process converges (the way to test for convergence is discussed in the design section below).

4. The program is to print the current estimate for the root at each cycle of the iterative process.

5. The program is to terminate after printing again the number and the converged estimate of its square root.

Program design

1. What data?
   the number whose root is required
   the current estimate of the root
   maybe some extra variables needed when checking for convergence

   These should all be double precision numbers defined in the main program.

2. Processing:
   a) prompt for and then input the number;
   b) initialize root (to half of given number)
   c) while loop:
   test on convergence
   body of loop involves
   output of current estimate of root
   calculation of new root
   d) print details of number and root

The only difficult part of the processing is the test controlling iteration. The following might *seem* suitable:

```
x   the number whose root is required
r   current guess for its square root

    while(x != r*r)
```

This test will keep the iterative process going while x is not equal to r^2 (after all, that is the definition of r being the square root of x). Surprisingly, this doesn't work in practice, at least not for all values of x.

The reason it doesn't work is "roundoff error". That test would be correct if we were really working with mathematical real numbers, but in the computer we only have floating point numbers. When the iteration has progressed as far as it can, the value of root will be approximated by the floating point number that is closest to its actual real value – but the square of this floating point number may not be exactly equal the floating point value representing x.

The test could be rephrased. It could be made that we should keep iterating until the difference between x and r^2 is very small. The difference x - r^2 would be positive if the estimate r was a little too small and would be negative if `r` was a little too large. We just want the size of the difference to be small irrespective of whether the value difference is positive or negative. So, we have to test the absolute value of the difference. The maths library has a function `fabs()` that gets the absolute value of a double number. Using this function, we could try the following:

```
const double VERYSMALL = 1.0E-8;
…
…
while(fabs(x-r*r) > VERYSMALL)
```

This version would be satisfactory for a wide range of numbers but it still won't work for all. If you tried a very large value for x, e.g. 1.9E+71, the iteration wouldn't converge. At that end of the number range, the "nearest floating point numbers" are several integer values apart. It simply isn't possible to get the fixed accuracy specified.

The test can't use an absolute value like 1.0E-8. The limit has to be expressed relative to the numbers for which the calculation is being done.

The following test should work:

```
const double SMALLFRACTION = 1.0E-8;
…
…
while(fabs(x - r*r) > SMALLFRACTION*x) {
```

Implementation

The implementation is straightforward. The code would be:

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

int main()
{
    const double SMALLFRACTION = 1.0E-8;
    double x;
    double r;
    cout << "Enter number : ";
    cin >> x;
```

```
    r = x / 2.0;
    while(fabs(x - r*r) > SMALLFRACTION*x) {
           cout << r << endl;
           r = 0.5 *(x / r + r);
           }

    cout << "Number was : " << x << ", root is "
                 << r << endl;
    return EXIT_SUCCESS;
}
```

Test data are easy:

```
Enter number : 81
40.5
21.25
12.530882
9.497456
9.013028
9.000009
Number was : 81, root is 9
```

7.2.3 Tabulating function values

Problem

If you are to compare different ways of solving problems, you need to define some kind of performance scale. As you will learn more later, there are various ways of measuring the performance of programs.

Often, you can find different ways of writing a program that has to process sets of data elements. The time the program takes will increase with the number of data elements that must be processed. Different ways of handling the data may have run times that increase in different ways as the number of data elements increase. If the rate of increase is:

linear	time for processing 20 data elements will be twice that needed for 10 elements
quadratic	time for processing 20 data elements will be 4x that needed for 10 elements
cubic	time for processing 20 data elements will be 8x that needed for 10 elements
exponential	time for processing 20 data elements will be about one thousand times that needed for 10 elements

Usually, different ways of handling data are described in terms of how their run times increase as the number of data elements (N) increases. The kinds of functions encountered include

log(N)	logarithmic (very slow growth)

N	linear
N * log(N)	log linear (slightly faster than linear growth)
$\mathbf{N^2}$	quadratic
$\mathbf{2^N}$	exponential (unpleasantly fast growth)
N!	factorial (horribly fast growth)
$\mathbf{N^N}$	disgustingly fast growth

When you study algorithms later, you will learn which are linear, which are quadratic and so on. But you need some understanding of the differences in these functions, which you can get by looking at their numeric values.

The program should print values for these functions for values N=1, 2, 3, ... The program is to stop when the values get too large (the function N^N, "N to the power N", grows fastest so the value of this function should be used in the check for termination).

Specification:

1. The program is to print values of N, N^2, N^3, log(N), N*log(N), 2^N and N^N for values of N = 1, 2, 3, 4, ...

2. The loop that evaluates these functions for successive values of N is to terminate when the value of N^N exceeds some large constant.

Program design

1. What data?
   Value N, also variables to hold values of N^2, N^3, log(N) etc

   Some could be integers, but the ones involving logarithms will be real numbers. Just for simplicity, all the variables can be made real numbers and represented as "doubles".

2. Where should data be defined?
   Data values are only needed in single main routine, so define them as local to that routine.

3. Processing ...
   a) initialize value of N, also variable that will hold N^N, both get initialized to 1
   b) while loop
      terminates when value in variable for N^N is "too large"
         body of loop involves
            calculating and printing out all those functions of N;

incrementing N ready for next iteration;

Implementation

The implementation is again straightforward. The code would be:

```
#include <iostream.h>
#include <math.h>
/*
    Program to print lists of values ...

N    log(N)    N*log(N)  N^2   N^3   2^N   N^N

for N = 1, 2, 3, ...
values to be printed while N^N is not "too large".

*/

int main()
{
    const char TAB = '\t';
           // Used to help tidy output a little
    const double TOO_LARGE = 10000000000000000.0;
           // Arbitrary large value
    double N, LGN,N_LGN, NSQ, NCUBE,TWO_POW_N, N_POW_N;
    N = 1.0;
    N_POW_N = 1.0;

    while (N_POW_N < TOO_LARGE) {
           NSQ = N*N; NCUBE = N*NSQ;
           LGN = log(N); N_LGN = N*LGN;
           TWO_POW_N = pow(2.0,N);
           N_POW_N = pow(N,N);
           cout << N << TAB << LGN << TAB << N_LGN << TAB <<
                  NSQ <<  TAB << NCUBE <<
                  TAB << TWO_POW_N << TAB <<
                  N_POW_N << endl;
           N = N + 1.0;
           }
}
```

Note the lines with multiple statements, e.g. `NSQ = N*N; NCUBE = N*NSQ;`. This is perfectly legal. Statements are terminated by semicolons, not newlines. So you can have statements that extend over several lines (like the monstrous output statement), or you can have several statements to a line.

The program produced the following output:

```
1      0      0      1      1      2      1
2      0.693147       1.386294       4      8      4      4
3      1.098612       3.295837       9      27     8      27
4      1.386294       5.545177       16     64     16     256
5      1.609438       8.04719        25     125    32     3125
6      1.791759       10.750557      36     216    64     46656
```

```
7      1.94591 13.621371       49      343     128     823543
…
…
13    2.564949     33.344342      169    2197   8192  3.0287e+14
14    2.639057     36.946803      196    2744   16384  1.1112e+16
```

By the time N=14, N^N is already a ridiculously large number!

Algorithms (i.e. prescriptions of how to solve problems) whose run times increase at rates like 2^N, or N!, or N^N are really not much use. Such "exponential" algorithms can only be used with very small sets of data. There are some problems for which the only known exact algorithms are exponential. Such problems generally have to be solved by ad hoc approximate methods that usually work but don't always work.

Even with the use of tabs to try to get output lined up, the printed results were untidy! You will later learn about additional capabilities of the `cout` object that can be used to tidy the output (Chapter 9.7).

7.3 BLOCKS

In the last example program, the variables `N` and `N_pow_N` were initialized outside the while loop; all the others were only used inside the body of the while loop and weren't referenced anywhere else.

If you have a "compound statement" (i.e. a `{` begin bracket, several statements, and a closing `}` end bracket) where you need to use some variables not needed elsewhere, you can define those variables within the compound statement. Instead of having the program like this (with all variables declared at start of the main function):

```
int main()
{
    const char TAB = '\t';
    const double TOO_LARGE = 1000000000000000.0;
    double N, LGN,N_LGN, NSQ, NCUBE,TWO_POW_N, N_POW_N;
    N = 1.0;
    …
    while (N_POW_N < TOO_LARGE) {
           NSQ = N*N; NCUBE = N*NSQ;
           …
           }
}
```

You can make it like this, with only `N` and `N_pow_N` defined at start of program, variables `NSQ` etc defined within the body of the while loop:

```
void main()
{
    const char TAB = '\t';
    const double TOO_LARGE = 1000000000000000.0;
    double N, N_POW_N;
    N = 1.0;
```

```
        …
        while (N_POW_N < TOO_LARGE) {
                double  LGN,N_LGN, NSQ, NCUBE,TWO_POW_N;
                NSQ = N*N; NCUBE = N*NSQ;
                …
                }
}
```

It is usually sensible to define variables within the compound statement where they are used; the definitions and use are closer together and it is easier to see what is going on in a program.

Defining a variable within a compound statement means that you can only use it between that compound statement's { begin bracket and } end bracket. You can't use the variable anywhere else. Because they had to be initialized and used in the test controlling the iterative loop, the variables N and N_pow_N had to be defined at the start of the program; they couldn't be defined within the compound statement.

Compound statements that contain definitions of variables are called "blocks". The body of a function forms a block. Other blocks can be defined inside as just illustrated.

Where should variables be defined

In all the examples so far the variables have been defined at the start of the block before any executable statements. This is required in C (and in languages like Pascal and Modula2). C++ relaxes the rules a little and allows variables to be defined in the body of a block after executable statements. Later examples will illustrate the more relaxed C++ style.

Scope

The part of the text of the program source text where a variable can be referenced is called the "scope" of that variable. Variables defined in a block can only be referenced in statements following the definition up to the end of block bracket.

7.4 "BOOLEAN" VARIABLES AND EXPRESSIONS

7.4.1 True and False

In C/C++, the convention is that

```
the (numeric) value 0 (zero) means false,
any non-zero numeric value means true.
```

This is the reason why you will see "funny looking" while loops:

```
int n;
// read some +ve integer specify number of items to process
cin >> n;
while(n) {
    ...
    ...
    n = n - 1;
    }
```

The `while(n)` test doesn't involve a comparison operator. What is says instead is "keep doing the loop while the value of n is non-zero".

Representing the true/false state of some condition by a 1 or a 0 is not always clear. For example, if you see code like:

```
int married;
…
…
married = 1;
```

It is not necessarily obvious whether it is updating a person's marital status or initiating a count of the number of times that that person has been married. Other languages, like Pascal, Modula2 and modern versions of FORTRAN have "Boolean" (or "logical") variables for storing true/false values and have constants `true` and `false` defined.

Code like

```
bool married;
…
…
married = true;
```

is considerably clearer than the version with integers.

C "hacks" for boolean types

C never defined a standard "Boolean" data type. So, every programmer working with C added their own version. The trouble was there were several different ways that the boolean type (and the constants true and false) could be added to C. Programmers used different mechanisms, and these different mechanisms got into library code.

One way used by some programmers was to use `#define` macro statements. These macro statements simply instruct the compiler to replace one bit of text by another. So you could have

```
#define Boolean int
#define True 1
#define False 0
```

The compiler would then substitute the word `int` for all occurrences of `Boolean` etc. So you could write:

```
Boolean married;
…
married = True;
```

and the compiler would switch this to

```
int married;
…
married = 1;
```

before it tried to generate code.

Another programmer working on some other code for the same project might have used a different approach. This programmer could have used a "typedef" statement to tell the compiler that boolean was the name of a data type, or they might have used an "enumeration" to specify the terms { false, true }. (Typedefs are explained later in Chapter 11; enumerated types are introduced in Chapter 16). This programmer would have something code like:

```
enum boolean { false, true };
…
married = true;
```

Of course, it all gets very messy. There are *Booleans*, and *booleans*, and *True* and *true*.

bool type in new C++ standard

The standards committee responsible for the definition of C++ has decided that it is time to get rid of this mess. The proposed standard for C++ makes `bool` a built in type that can take either of the constant values `true` or `false`. There are conversions defined that convert 0 values (integer 0, "null-pointers") to `false`, and non-zero values to `true`.

A compiler that implements the proposed standard will accept code like:

```
bool married;
…
…
married = true;
```

Most compilers haven't got this implemented yet.

The older C hacks for `#define Boolean int`, or `typedef int boolean`, or `enum boolean` etc are everywhere. For years to come, older code and libraries will still contain these anachronistic forms.

7.4.2 Expressions using "AND"s and "OR"s

The simple boolean expressions that can be used for things like controlling while loops are:

1. a test on whether a variable is 0, false, or non zero, true; e.g.

   ```
   while(n) …
   ```

2. a test involving a comparison operator and either a variable or a constant; e.g

   ```
   while(sum < 1000) …
   ```

   ```
   while(j >= k) …
   ```

Sometimes you want to express more complex conditions. (The conditions that control `while` loops should be simple; the more complex boolean expressions are more likely to be used with the `if` selection statements covered in Chapter 8.)

For example, you might have some calculation that should be done if a double value x lies between 0.0 and 1.0. You can express such a condition by combining two simple boolean expressions:

```
(x > 0.0) && (x < 1.0)
```

The "&&" AND operator

The requirement for `x` being in the range is expressed as "x is greater than 0.0 AND x is less than 1.0". The AND operation is done using the `&&` operator (note, in C++ a single `&` can appear as an operator in other contexts – with quite different meanings).

The "||" OR operator

There is a similar OR operator, represented by two vertical lines `||`, that can combine two simpler boolean expressions. The expression:

```
(error_estimate > 0.0001) || (iterations < 10)
```

is true if either the value of `error_estimate` exceeds its 0.0001 OR the number of iterations is less than 10. (Again, be careful; a single vertical line `|` is also a valid C++ operator but it has slightly different meaning from `||`.)

The "!" NOT operator

Along with an AND operator, and an OR operator, C++ provides a NOT operator. Sometimes it is easier to express a test in a sense opposite that required; for example, in the following code the loop is executed while an input character is *not* one of the two values permitted:

```
char ch;
…
cout << "Enter patient's gender (M for Male or"
           " F for Female)" << endl;
cin >> ch;
while(!((ch == 'M') || (ch == 'F'))) {
    cout << "An entry of M or F is required here. "
                  "Please re-enter gender code" << endl;
    cin >> ch;
    }
```

The expression `((ch == 'M') || (ch == 'F'))` returns true if the input character is either of the two allowed values. The NOT operator, `!`, reverses the truth value – so giving rise to the correct condition to control the input loop.

Evaluation of boolean expressions

Expressions involving AND and OR operators are evaluated by working out the values of the simple subexpressions one after the other reading the line from left to right until the result is known (note, as explained below, AND takes precedence over OR so the order of evaluation is not strictly left to right). The code generated for an expression like:

```
A  ||  B  || C
```

(where A, B, and C are simple boolean expressions) would evaluate A and then have a conditional jump to the controlled code if the result was true. If A was not true, some code to evaluate B would be executed, again the result would be tested. The code to evaluate C would only be executed if both A and B evaluated to false.

Parentheses

If you have complex boolean expressions, make them clear by using parentheses. Even complex expressions like

```
(A || B) && (C || D) && !E
```

are reasonably easy to understand if properly parenthesised. You can read something like

```
X || Y && Z
```

It does mean the same as

```
X || (Y && Z)
```

but your thinking time when you read the expression without parentheses is much longer. You have to remember the "precedence" of operators when interpreting expressions without parentheses.

Expressions without parentheses do all have meanings. The compiler is quite happy with something like:

```
i < limit && n != 0
```

It "knows" that you don't mean to test whether `i` is less than `limit && n`; you mean to test whether `i` is less than `limit` and if that is true you want also to test that `n` isn't equal to zero.

The meanings of the expressions are defined by operator precedence. It is exactly the same idea as how you learnt to interpret arithmetic expressions in your last years of primary school. Then you learnt that multiply and divide operators had precedence over addition and subtraction operators.

The precedence table has just been extended. Part of the operator precedence table for C++ is:

Operator precedence

`++, --`	*increment and decrement operators*
`!`	*NOT operator*
`*, /, %`	*multiply, divide, and modulo*
`+, -`	*addition and subtraction*
`<, <=, >, >=`	*greater/less comparisons*
`==, !=`	*equality, inequality tests*
`&&`	*AND operator*
`||`	*OR operator*
`=`	*assignment operator*

You can think of these precedence relations as meaning that the operators higher in the table are "more important" and so get first go at manipulating the data. (The ++ increment and -- operators are explained in section 7.5.)

Because of operator precedence, a statement like the following can be unambiguously interpreted by a compiler:

```
res = n == m > 7*++k;
```

(It means that res gets a 1 (true) or 0 (false) value; it is true if the value of n (which should itself be either true or false) is the same as the true or false result obtained when checking whether the value of m exceeds 7 times the value of k after k has first been incremented!)

Legally, you can write such code in C++. Don't. Such code is "write only code". No one else can read it and understand it; everyone reading the code has to stop and carefully disentangle it to find out what you meant.

If you really had to work out something like that, you should do it in steps:

```
++k;
int temp = m > 7*k;  // bool temp if your compiler has bool
res = n == temp;
```

Readers should have no problems understanding that version.

Caution

There are many traps for the unwary with complex boolean expressions. For example, you might be tempted to test whether x was in the range 0.0 to 1.0 inclusive by the following:

```
0.0 <= x <= 1.0
```

This is a perfectly legal C++ test that will be accepted by the compiler which will generate code that will execute just fine. Curiously, the code generated for this test returns true for x = -17.5, x = 0.4, x = 109.7, and in fact for any value of x!

The compiler saw that expression as meaning the following:

```
compare x with 0.0
    getting result 1 if x >= 0.0, or 0 if x < 0.0
compare the 0 or 1 result from last step with 1.0
    0 <= 1 gives result 1, 1 <= 1 also gets result 1
```

So, the code, that the compiler generated, says "true" whatever the value of x.

Another intuitive (but WRONG) boolean expression is the following (which a student once invented to test whether a character that had been read was a vowel)

```
ch == 'a' || 'e' || 'i' || 'o' || 'u'
```

Once again, this code is perfectly legal C++. Once again, its meaning is something quite different from what it might appear to mean. Code compiled for this expression will return true whatever the value in the character variable `ch`.

The `==` operator has higher precedence than the `||`. So this code means

```
(ch == 'a') || 'e' || …
```

Now this is true when the character read was 'a'. If the character read wasn't 'a', the generated code tests the next expression, i.e. the 'e'. Is 'e' zero (false) or non-zero (true)? The value of 'e' is the numeric value of the ASCII code that represents this character (101); value 101 isn't zero, so it must mean true.

The student wasn't much better off when she tried again by coding the test as:

```
ch == ('a' || 'e' || 'i' || 'o' || 'u')
```

This version said that no character was a vowel. (Actually, there is one character that gets reported as a vowel, but it is the control character "start of header" that you can't type on a keyboard).

The student could have expressed the test as follows:

```
((ch == 'a') || (ch == 'e') || (ch == 'i') || (ch == 'o')
            || (ch == u))
```

It may not be quite so intuitive, but at least it is correct code.

7.5 SHORT FORMS: C/C++ ABBREVIATIONS

As noted in other contexts, the C and C++ languages have lots of abbreviated forms that save typing. There are abbreviations for some commonly performed arithmetic operations. For example, one often has to add one to a counter:

```
count_customers = count_customers + 1;
```

The += operator

In C and C++, this could be abbreviated to

```
count_customers += 1;
```

The "++" increment operator

or even to

```
count_customers++;
```

Actually, these short forms weren't introduced solely to save the programmer from typing a few characters. They were meant as a way that the programmer could give hints that helped the early C compilers generate more efficient code.

An early compiler might have translated code like

```
temp = temp + 5;
```

into a "load instruction", an "add instruction", and a "store instruction". But the computer might have had an "add to memory instruction" which could be used instead of the three other instructions. The compiler might have had an alternative coding template that recognized the += operator and used this add to memory instruction when generating code for things like:

```
temp += 5;
```

Similarly, there might have been an "increment memory" instruction (i.e. add 1 to contents of memory location) that could be used in a compiler template that generated code for a statement like

```
temp++;
```

Initially, these short forms existed only for addition and subtraction, so one had the four cases

`++`	increment
`+=` *Expression*	add value of expression
`--`	decrement (i.e. reduce by 1)
`-=` *Expression*	subtract value of expression

Later, variants of `+=` and `-=` were invented for things like `*` (the multiplication operator), `/` (the division operator), and the operators used to manipulate bit-patterns ("BIT-OR" operator, "BIT-AND" operator, "BIT-XOR" operator etc --- these will be looked at later, Chapter 18). So one can have:

```
x *= 4;   // Multiply x by 4, short form for x = x * 4

y /= 17; // Short form for y = y/17;
```

(Fortunately, the inventors of the C language didn't add things like `a**` or `b//` – because they couldn't agree what these formulae would mean.)

These short forms are OK when used as simple statements:

```
N++;
```

is OK instead of (maybe even better than)

```
N = N + 1;

Years += Half_life;
```

is OK instead of (maybe even better than)

```
Years = Years + Half_life;
```

The short forms can be used as parts of more complex expressions:

```
xval = temp + 7 * n++;
```

this changes both `xval` and `n`. While legal in both C and C++, such usage should be avoided (statements with embedded increment, `++`, or decrement, `--`, operators are difficult to understand and tend to be sources of error).

Pre-op and post-op versions of ++ and --

Expressions with embedded `++` and `--` operators are made even more complex and confusing by the fact that there are two versions of each of these operators.

The `++` (or `--`) operator can come before the variable it changes, `++x`, or after the variable, `x++`. Now, if you simply have a statement like

```
x++;          or          ++x;
```

it doesn't matter which way it is written (though `x++` is more common). Both forms produce identical results (`x` is incremented).

But, if you have these operators buried in more complex expressions, then you will get different results depending on which you use. The prefix form (`++x`) means "*increment the value of x and use the updated value*". The postfix form (`x++`) means "*use the current value of x in evaluating the expression and also increment x afterwards*". So:

```
n = 15;
temp = 7 * n++;
cout << "temp  " << temp << endl;
cout << "n " << n << endl;
```

prints `temp 105` and `n 16` while

```
n = 15;
temp = 7 * ++n;
cout << "temp  " << temp << endl;
cout << "n " << n << endl;
```

prints `temp 112` and `n 16`.

General advice

Only use short form operators `++`, `--`, `+=`, `-=` etc in simple statements with a single operator. The following are OK

```
i++;          years += half_life;
```

anything more complex should be avoided until you are much more confident in your use of C++.

Why C++?

Now you understand why language is called C++:

take the C language and add a tiny bit to get one better.

7.6 DO …WHILE

The standard while loop construct has a variation that is useful if you know that a loop must always be executed at least once.

This variation has the form

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

Usually, the body of the loop would be a compound statement rather than a simple statement; so the typical "do" loop will look something like:

```
do {
    …
    …
} while (expression);
```

The expression can be any boolean expression that tests values that get updated somewhere in the loop.

A contrived example –

```
// Loop getting numbers entered by user until either five
// numbers have been entered or the sum of the numbers
// entered so far exceeds limit
const int kLIMIT = 500;
int entry_count = 0;
int sum = 0;
do {
    cout << "Give me another number ";
    int temp;
    cin >> temp;
    entry_count++;
    sum += temp;
} while ((entry_count < 5) && (sum < kLIMIT));
```

7.7 FOR LOOP

Often, programs need loops that process each data item from some known fixed size collection. This can be handled quite adequately using a standard while loop –

```
// Initialize counter
int loop_count = 0;
// loop until limit (collection size) reached
while(loop_count < kCOLLECTION_SIZE) {
    // Do processing of next element in collection
    …
    // Update count of items processed
    loop_count++;
    }
```

The structure of this counting loop is: initialize counter, loop while counter less than limit, increment counter at the end of the body of the loop.

It is such a common code pattern that it deserves its own specialized statement – the `for` statement.

```
int loop_count;
for(loop_count=0;
           loop_count < kCOLLECTION_SIZE;
                  loop_count++) {
    // Do processing of next element in collection
    …
    }
```

The `for` statement has the following structure:

```
for(  First part, do things like initialize counts ;
          Second part, the termination test ;
                 Third part, do things like updating
```

```
                    counters (done after body of loop
                    has been executed)
)
            Statement of what is to be done ;
```

There are three parts to the parenthesised construct following the keyword `for`. The first part will be a simple statement (compound statements with begin `{` and `}` end brackets aren't allowed); normally, this statement will initialize the counter variable that controls the number of times the loop is to execute. The second part will be a boolean expression; the loop is to continue to execute as long as this expression is true. Normally, this second part involves a comparison of the current value of the loop control variable with some limit value. Just as in a `while` loop, this loop termination test is performed before entering the body of the loop; so if the condition for termination is immediately satisfied the body is never entered (e.g. `for(i=5; i<4; i++) { … }`). The third part of the `for(…;…;…)` contains expressions that are evaluated after the body has been executed. Normally, the third part will have code to increment the loop control variable.

A for loop "controls" execution of a statement; usually this will be a compound statement but sometimes it will be a simple statement (generally a call to a function). The structures are:

```
for(…; …; …)                for(i=0;i<10;i++)
        statement;                  sum += i;
```

(note the semicolon *after* the controlled statement.) and

```
for(…; …; …) {              for(i=0;i<10;i++) {
        statement;                  cout << i << ":";
        statement;                  …
        …                           …
        }                           }
```

You can have loops that have the loop control variable decreasing –

```
for(j = 100; j > 0; j--) {
    // work back down through a collection of data
    // elements starting from top
    …
    }
```

Don't change the loop control variable in the body of the code

If you are using a `for` as a counting loop, then you should not alter the value of the control variable in the code forming the body of the loop. It is legal to do so; but it is very confusing for anyone who has to read your code. It is so confusing that you will probably confuse yourself and end up with a loop that has some bug associated with it so that under some circumstances it doesn't get executed the correct number of times.

Defining a loop control variable in the for statement

As noted earlier, section 7.3, variables don't have to be defined at the start of a block. If you know you are going to need a counting loop, and you want to follow convention and use a variable such as `i` as the counter, you can arrange your code like this where i is defined at the start of the block:

```
int main()
{
    int i, n;
    cout << "Enter number of times loop is to execute";
    cin >> n;
    for(i=0; i < n; i++) {
           // loop body with code to do something
           // interesting
           …
           }
    cout << "End of loop, i is " << i << endl;
    …
}
```

or you can have the following code where the loop control variable `i` is defined just before the loop:

```
int main()
{
    int n;
    cout << "Enter number of times loop is to execute";
    cin >> n;
    int i;
    for(i=0; i < n; i++) {
           …
           }
    cout << "End of loop, i is " << i << endl;
    …
}
```

or you can do it like this:

```
int main()
{
    int n;
    cout << "Enter number of times loop is to execute";
    cin >> n;
    for(int i=0; i < n; i++) {
           …
           }
    cout << "End of loop, i is " << i << endl;
    …
}
```

Note area of possible language change

A loop control variable can be defined in that first part of the `for(…;…;…)` statement. Defining the control variable inside the `for` is no different from defining it just before the `for`. The variable is then defined to the end of the enclosing block. (There are other languages in the Algol family where loop control variables "belong" to their loops and can only be referenced within the body of their loop. The new standard for C++ suggests changing the mechanism so that the loop control variable does in fact belong with the loop; most compilers don't yet implement this.)

You are not restricted in the complexity of the code that you put in the three parts of the for(…;…;…). The initialization done in the first part can for example involve a function call: `for(int i = ReadStartingValue(); …; …)`. The termination test can be something really elaborate with a fearsome boolean expression with lots of `&&` and `||` operators.

You aren't allowed to try to fit multiple statements into the three parts of the `for` construct. Statements end in semicolons. If you tried to fit multiple statements into the for you would get something like

```
// Erroneous code in initialization part of for
for( sum = 0; i= 1;
    …; // termination test
           // Erroneous code in update part of for
           i++; sum+= i*val; …)
```

The semicolons after the extra statements would break up the required pattern **`for(… ; … ; …)`**.

The comma operator

The C language, and consequently the C++ language, has an alternative way of sequencing operations. Normally, code is written as *`statement`*`;` *`statement`*`;` *`statement`*`;` etc. But you can define a sequence of expressions that have to be evaluated one after another; these sequences are defined using the comma "," operator to separate the individual expressions:

```
expression , expression, expression
```

Another oddity in C (and hence C++) is that the "assignment statement" is actually an expression. So the following are examples of single statements:

```
i = 0, max = SHRT_MAX, min = SHRT_MIN, sum = 0;
i++, sum += i*val;
```

The first statement involves four assignment expressions that initialize `i` and `sum` to `0` and `max` and `min` to the values of the constants `SHRT_MAX` etc. The second statement has two expressions updating the values of `i` and `sum`. (It was the comma operator that caused the problems noted in section 6.4 with the erroneous input statement `cin >> xcoord, ycoord;`. This statement actually consists of two comma separated expressions – an input expression and a "test the value of" expression.)

Although you aren't allowed to try to fit multiple statements into the parts of a `for`, you are permitted to use these comma separated sequences of expressions. So you will frequently see code like:

```
for(i=0, j=1;                    // initialization part
           i<LIMIT;              // termination test
                  i++, j = ++j % 5) // update part
    {
    …
    }
```

(It isn't totally perverse code. Someone might want to have a loop that runs through a set of data elements doing calculations and printing results for each; after every 5th output there is to be a newline. So, the variable `i` counts through the elements in the collection; the variable `j` keeps track of how many output values have been printed on the current line.)

Because you can use the comma separated expressions in the various parts of a for, you can code things up so that all the work is done in the `for` statement itself and the body is empty. For example, if you wanted to calculate the sum and product of the first 15 integers you could write the code the simple way:

```
int i, sum = 0, product = 1;
for(i = 1; i<= 15; i++) {
    sum += i;
    product *= i;
    }
cout << "Sum " << sum << ", product " << product << endl;
```

or you can be "clever":

```
int i, sum, product ;
for(sum = 0, product = 1, i = 1;
           i<= 15;
                 sum += i, product *= i, i++) ;
cout << "Sum " << sum << ", product " << product << endl;
```

A form like:

```
for(…; …; …);
```

(with the semicolon immediately after the closing parenthesis) is legal. It means that you want a loop where nothing is done in the body because all the work is embedded in the `for(…;…;…)`. (You *may* get a "warning" from your compiler if you give it such code; some compiler writer's try to spot errors like an extraneous semicolon getting in and separating the `for(…;…;…)` from the body of the loop it is supposed to control.)

While you often see programs with elaborate and complex operations performed inside the `for(…;…;…)` (and, possibly, no body to the loop), you shouldn't write code like this. Remember the "KISS principle" (Keep It Simple Stupid). Code that is "clever" requires cleverness to read, and (cleverness)^2 to write correctly.

A `while` loop can do anything that a `for` loop can do.

Conversely, a `for` loop can do anything a `while` loop can do.

Instead of the code given earlier (in 7.2.2):

```
r = x / 2.0;
while(fabs(x - r*r) > SMALLFRACTION*x) {
    cout << r << endl;
    r = 0.5 *(x / r + r);
    }

cout << "Number was : " << x << ", root is "
           << r << endl;
```

You could make the code as follows:

```
r = x / 2.0;
for( ;            // an empty initialization part
    fabs(x - r*r) > SMALLFRACTION*x;  // Terminated?
    ) {           // Empty update part
    cout << r << endl;
    r = 0.5 *(x / r + r);
    }
cout << "Number was : " << x << ", root is "
                  << r << endl;
```

or, if you really want to be perverse, you could have:

```
for(r = x / 2.0;
    fabs(x - r*r) > SMALLFRACTION*x;
    r = 0.5 *(x / r + r))
           cout << r << endl;
cout << "Number was : " << x << ", root is "
                  << r << endl;
```

(It is legal to have empty parts in a `for(…;…;…)`, as illustrated in the second version.)

Most readers prefer the first version of this code with the simple `while` loop. Write your code to please the majority of those that must read it.

7.8 BREAK AND CONTINUE STATEMENTS

There are two other control statements used in association with loop constructs. The `break` and `continue` statements can appear in the body of a loop. They appear as the action parts of `if` statement (section 8.3). The conditional test in the `if` will have identified some condition that requires special processing.

break

A `break` statement terminates a loop. Control is transferred to the first statement following the loop construct.

continue

A `continue` statement causes the rest of the body of the loop to be omitted, at least for this iteration; control is transferred to the code that evaluates the normal test for loop termination.

Examples of these control statements appear in later programs where the code is of sufficient complexity as to involve conditions requiring special processing (e.g. use of `continue` in function in 10.9.1).

EXERCISES

1. Implement versions of the "square root" program using the "incorrect" tests for controlling the iterative loop. Try finding values for x for which the loop fails to terminate. (Your IDE system will have a mechanism for stopping a program that isn't terminating properly. One of the command keys will send a stop signal to the program.

Check the IDE documentation to find how to force termination before starting the program.)
If your compiler supports "long doubles" change the code of your program to use the long double data type and rerun using the same data values for which the original program failed. The modified program may work. (The long double type uses more bits to represent the mantissa of a floating point number and so is not as easily affected by round off errors.)

2. Write a program that prints the values of a polynomial function of x at several values of x. The program should prompt for the starting x value, the increment, and the final x value.

   A polynomial function like $6x^3 - 8x^2 - 3x + 4.5$ can be evaluated at a particular x value using the formula:

   ```
   val = 6.0 * pow(x, 3.0) - 8.0*pow(x,2.0) -3.0*x + 4.5;
   ```

   of

   ```
   val = 6.0 * x * x * x   -8.0 * x *x  -3.0 *x  + 4.5;
   ```

   or

   ```
   val = ((6.0*x -  8.0)*x - 3.0)*x + 4.5;
   ```

   The first is the most costly to evaluate because of the `pow()` function calls. The last is the most efficient (it involves fewer multiplication operations than the second version).

   (For this exercise, use a fixed polynomial function. List its values at intervals of 0.5 for x in range -5.0 to + 5.0.)

3. Generalize the program from exercise 2 to work with any polynomial of a given maximum degree 4, i.e. a function of the form $c4\ x^4 + c3\ x^3 + c2\ x^2 + c1\ x + c0$ for arbitrary values of the coefficients c4, c3, c2, c1, and c0.

   The formula for evaluating the polynomial at a given value of x is

   val = (((c4 * x + c3)*x + c2)*x + c1)*x + c0

   The program is to prompt for and read the values of the coefficients and then, as in exercise 2 it should get the range of x values for which the polynomial is to be evaluated.

4. An object dropped from a height is accelerated to earth by gravity. Its motion is determined by Newtons laws. The formulas for its velocity and distance travelled are:

   ```
   v = 32.0 * t; // velocity in feet per second
   s = 16.0 * t * t;    // distance fallen in feet
   ```

   Write a program that reads the height from which the object is dropped and which prints a table showing the object's velocity and height at one second intervals. The loop

printing details should terminate when the distance fallen is greater than or equal to the specified height.

5. Write a program that "balances transactions on a bank account".

   The program is to prompt for and read in the initial value for the funds in the account. It is then to loop prompting for and reading transactions; deposits are to be entered as positive values, withdrawals as negative values. Entry of the value 0 (zero) is to terminate the loop. When the loop has finished, the program is to print the final value for the funds in the account.