6 Sequence

The simplest programs consist just of sequences of statements with no loops, no selections amongst alternative actions, no use of subroutines (other than possibly those provided in the input/output or mathematical libraries).

They aren't very interesting as programs! They do things that you could usually do on a calculator (at least you could probably do with a calculator that had some memory to store partial results). Most often they involve simply a change of scale (e.g. you have an amount in $US and want it in ¥, or you have a temperature on the Celsius scale and want it in Fahrenheit, or you have an H+ ion concentration and want a pH value). In all such programs, you input a value, use some formula ("x units on scale A equals y units on scale B") and print the result.

Such programming problems, which are commonly used in initial examples, are meant to be so simple that they don't require any real thought as to how to solve the problem. Because the problem is simple, one can focus instead on the coding, all those messy little details, such as:

- the correct forms for variable names,
- the layout of variable declarations and statements,

and

- the properties of the "operators" that are used to combine data values.

6.1 OVERALL STRUCTURE AND MAIN() FUNCTION

The first few programs that you will write all have the same structure:

```
#include <iostream.h>
#include <math.h>
/*
  This program converts [H+] into
  pH using formula
     pH = - log10 [H+]
```

```
*/

int main()
{
    double Hplus;
    double pH;

    cout << "Enter [H+]  value";
    ...
    return 0;
}
```

#includes

The program file will start with `#include` statements that tell the compiler to read the "headers" describing the libraries that the program uses. (You will always use `iostream.h`, you may use other headers in a particular program.)

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

Introductory comments

It is sensible to have a few comments at the start of a program explaining what the program does.

```
/*
  This program converts [H+] into
  pH using formula
     pH = - log10 [H+]
*/
```

Program outline

The outline for the program may be automatically generated for you by the development environment. The outline will be in the form `void main() { ... }`, or `int main() { ... return 0; }`.

Own data definitions and code

You have to insert the code for your program between the `{` ("begin") and `}` ("end") brackets.

The code will start with definitions of constants and variables:

```
double Hplus;
double pH;
```

and then have the sequence of statements that describe how data values are to be combined to get desired results.

Libraries

What libraries do you need to "`#include`"? Someone will have to tell you which libraries are needed for a particular assignment.

Almost standard libraries!

The libraries are not perfectly standardized. You do find difference between environments (e.g. the header file for the maths library on Unix contains definitions of the values of useful constants like PI π = 3.14159..., the corresponding header file with Symantec C++ doesn't have these constants).

Libraries whose headers often get included are:

iostream	standard C++ input output library
stdio	alternate input output library (standard for C programs)
math	things like sine, cosine, tan, log, exponential etc.
string	functions for manipulating sequences of characters ("strings") – copying them, searching for occurrences of particular letters, etc.
ctype	functions for testing whether a character is a letter or a digit, is upper case or lower case, etc.
limits	constants defining things like largest integer that can be used on a particular machine
stdlib	assorted "goodies" like a function for generating random numbers.

6.2 COMMENTS

"Comments" are sections of text inserted into a program that are ignored by the compiler; they are only there to be read by the programmers working on a project. Comments are meant to explain how data are used and transformed in particular pieces of code.

Comments are strange things. When you are writing code, comments never seem necessary because it is "obvious" what the code does. When you are reading code (other peoples' code, or code you wrote more than a week earlier) it seems that comments are essential because the code has somehow become incomprehensible.

C++ has two constructs that a programmer can use to insert comments into the text of a program.

"Block comments"

"Block comments" are used when you need several lines of text to explain the purpose of a complete program (or of an individual function). These start with the character pair `/*` (no spaces between the `/` and the `*`), and end with `*/` (again, no space) e.g.

```
/*
    Program for assignment 5; program reads 'customer'
records from
    ....
*/
```

Be careful to match `/*` (begin comment) and `*/` (end comment) markers! Some very puzzling errors are caused by comments with wrongly matched begin-end markers. Such incorrectly formed comments "eat" code; the program looks as if it has coded statements but the compiler ignores some of these statements because it thinks they are meant to be part of a comment. (You won't get such problems with the better IDEs because they use differently coloured letters for comments making it much less likely that you would do something wrong.)

"Line comments"

C++ shares "block comments" with C. C++ also has "line comments". These start with `//` (two `/` characters, no space between them) and continue to the end of a line. They are most commonly used to explain the roles of individual variables:

```
int count1; // Count of customers placing orders today
int count2; // Count of items to be delivered today
int count3; // Count of invoices dispatched
```

6.3 VARIABLE DEFINITIONS

A variable definition specifies the type of a variable, gives it a name, and sometimes can give it an initial value.

"Built in" types

You will start with variables that are of simple "built in" types. (These types are defined in the compiler.) These are:

type	**(short name)**	**data**
short int	short	integer value in range -32768 to +32768
long int	long	integer value in range ±2,000million
float		real number (low accuracy)
double		real number (standard accuracy)
char		character (a letter, digit, or other character in defined character set)

Integer variables are often declared as being of type `int` rather than `short` or `long`. This can cause problems because some systems use `int` as synonymous with `short int` while others take it to mean `long int`. This causes programs using `int`s to behave differently on different systems.

Example definitions:

```
short   aCount;
double  theTotal;
char    aGenderFlag; // M = male, F = Female
```

You can define several variables of the same type in a single definition:

```
long  FailCount, ECount, DCount, CCount, BCount, ACount;
```

Definitions do get a lot more complex than this. Later examples will show definitions of "derived types" (e.g. an array of integers as a type derived from integer) and, when structs and classes have been introduced, examples will have variables of programmer defined types. But the first few programs will use just simple variables of the standard types, so the complexities of elaborate definitions can be left till later.

Since the first few programs will consist solely of a `main()` routine, all variables belong to that routine and will only be used by that routine. Consequently, variables will be defined as "locals" belonging to `main()`. This is achieved by putting their definitions within `main()`'s `{` (begin) and `}` (end) brackets.

```
int main()
{
   long aCount;
   double theTotal;
   ...
}
```

6.4 STATEMENTS

We begin with programs that are composed of:

"input statements"	these get the "`cin` object" to read some data into variables,
"assignment statements"	these change values of variables,
"output statements"	these get the "`cout` object" to translate the data values held in variables into sequences of printable characters that are then sent to an output device.

In fact, most of the early programs are going to be of the form

```
main()
{
    get some data        // Several input statements, with
                         // maybe some outputs for prompts
    calculate with data  // Several assignment statements
    output results       // Several more output statements
}
```

Input

These statements are going to have the form:

```
cin >> Variable;
```

where *Variable* is the name of one of the variables defined in the program. Examples

```
short     numItems;
double    delta;
char      flag;
...
cin >> numItems;
...
cin >> delta;
...
cin >> flag;
...
```

If several values are to be read, you can concatenate the input statements together. Example

```
double    xcoord, ycoord, zcoord;
...
cin >> xcoord;
cin >> ycoord;
cin >> zcoord;
...
```

or

```
double    xcoord, ycoord, zcoord;
...
cin >> xcoord >> ycoord >> zcoord;
...
```

The `cin` object has to give data to each variable individually with a >> operator.

Common error made by those who have programmed in other languages

You may have seen other languages where input statements have the variable names separated by commas e.g. `xcoord, ycoord`. Don't try to follow that style! The following is a legal C++ statement:

```
cin >> xcoord, ycoord;
```

but it doesn't mean read two data values! Its meaning is actually rather odd. It means: generate the code to get `cin` to read a new value and store it in `xcoord` (OK so far) then generate code that fetches the current value of `ycoord` but does nothing with it (i.e. load its value into a CPU register, then ignores it).

cin's handling of >> operation

The `cin` object examines the characters in the input stream (usually, this stream comes from the keyboard) trying to find a character sequence that can be converted to the data value required. Any leading "white space" is ignored ("white space" means spaces, newlines, tabs etc). So if you ask `cin` to read a character, it will try to find a printable character. Sometimes, you will want to read the whitespace characters (e.g. you want to count the number of spaces between words). There are mechanisms (explained later when you need them) that allow you to tell the `cin` object that you want to process whitespace characters.

Output

Output statements are going have the form

```
cout << Variable;
```

where *Variable* is the name of one of the variables defined in the program, or

```
cout << Text-String;
```

where *Text-String* is a piece of text enclosed in double quotes. Examples:

```
   cout << "The coords are: ";

   cout << xcoord;
```

Usually, output operations are concatenated:

```
   cout << "The coords are:  x " << xcoord << ", y  "
              << ycoord << ", z " << zcoord;
```

(An output statement like this may be split over more than one line; the statement ends at the semi-colon.)

As previously explained (in the section 5.2), the `iostream` library defines something called "`endl`" which knows what characters are needed to get the next data output to start on a new line. Most concatenated output statements use `endl`:

```
   cout << "The pH of the solution is " << pHValue << endl;
```

Long text strings

In C++, you aren't allowed to define text strings that extend over more than one line. So the following is not allowed:

```
   cout << "  Year    Month    State    Total-Sales   Tax-
   due" << endl;
```

But the compiler allows you to write something like the following ...

```
   cout << "  Year    Month    State    "
       "Total-Sales   Tax-due" << endl;
```

It understands that the two text strings on successive lines (with nothing else between them) were meant as part of one longer string.

Special control characters

Sometimes you will want to include special characters in the text strings. For example, if you were producing an output report that was to be sent to a printer you might want to include special control characters that the printer could interpret as meaning "*start new page*" or "*align at 'tab-stop*'". These characters can't be typed in and included as part of text strings; instead they have to be coded.

You will often see coded characters that use a simple two character code (you will sometimes see more complex four character codes). The two character code scheme uses the "backslash" character `\` and a letter to encode each one of the special control characters.

Commonly used two character codes are:

```
   \t          tab
   \n          newline
   \a          bell sound
   \p          new page?
   \\          when you really want a \ in the output!
```

Output devices won't necessarily interpret these characters. For example a "tab" may have no effect, or may result in output of one space (or 4 spaces, or 8 spaces), or may cause the next output to align with some "tab-position" defined for the device.

Generally, '\n' causes the next output to appear at the start of new line (same effect as endl is guaranteed to produce). Consequently, you will often see programs with outputs like

```
cout << "Year    Month    Sales\n"
```

rather than

```
cout << "Year    Month    Sales"  << endl;
```

The style with '\n' is similar to that used in C. You will see many examples using '\n' rather than endl; the authors of these examples probably have been programming in C for years and haven't switched to using C++'s endl.

Calculations

Along with input statements to get data in, and output statements to get results back, we'd better have some ways of combining data values.

The first programs will only have assignment statements of the form

```
Variable = Arithmetic Expression;
```

where *Variable* is the name of one of the variables defined for the program and *Arithmetic Expression* is some expression that combines the values of variables (and constants), e.g.

```
double s, u, a, t;
cout << "Enter initial velocity, acceleration, and time : ";
cin >> u >> a >> t;

s = u*t + 0.5*a*t*t;

cout << "Distance travelled " << s << endl;
```

The compiler translates a statement like

```
s = u*t + 0.5*a*t*t;
```

into a sequence of instructions that evaluate the arithmetic expression, and a final instruction that stores the calculated result into the variable named on the left of the = (the "assignment operator").

"lvalue"

The term "*lvalue*" is often used by compilers to describe something that can go on the left of the assignment operator. This term may appear in error messages. For example, if you typed in an erroneous statement like "3 = n - m;" instead of say "e = n - m;" the error message would probably be "*lvalue required*". The compiler is trying to say that you can't have something like 3 on the left of an assignment operator, you have to have something like the name of a variable.

As usual, C++ allows abbreviations, if you want several variables to contain the same value you can have a concatenated assignment, e.g.

```
   xcoord = ycoord = zcoord = 1.0/dist;
```

The value would be calculated, stored in `zcoord`, copied from `zcoord` to `ycoord`, then to `xcoord`. Again be careful, the statement "`xcoord, ycoord, zcoord = 1.0/dist;`" is legal C++ but means something odd. Such a statement would be translated into instructions that fetch the current values of `xcoord` and `ycoord` and then does nothing with their values, after which there are instructions that calculate a new value for `zcoord`.

Arithmetic "operators"

Arithmetic expressions use the following "operators"

```
   +        addition operator
   -        subtraction operator (also "unary minus sign")
   *        multiplication operator
   /        division operator
   %        remainder operator (or "modulo" operator)
```

Note that multiplication requires an explicit operator. A mathematical formula may get written something like

```
   v = u  +  a  t
```

But that isn't a valid expression in C++; rather than `a t` you must write `a*t`.

You use these arithmetic operators with both real and integer data values (% only works for integers).

Arithmetic expressions work (almost) exactly as they did when you learnt about them at school (assuming that you were taught correctly!). So,

```
   v = u + a * t
```

means that you multiply the value of `a` by the value of `t` and add the result to the value of `u` (and you don't add the value of `u` to `a` and then multiply by `t`).

Operator precedence

The order in which operators are used is determined by their *precedence*. The multiply (`*`) and divide (`/`) operators have "higher precedence" than the addition and subtraction operators, which means that you do multiplies and divides before you do addition and subtraction. Later, we will encounter more complex expressions that combine arithmetic and comparison operators to build up logical tests e.g. when coding something like "if sum of income and bonus is greater than tax threshold or the interest income exceeds value in entry fifteen then …". The operators (e.g. `+` for addition, `>` for greater than, `||` for or etc) all have defined precedence values; these precedence values permit an unambiguous interpretation of a complex expression. You will be given more details of precedence values of operators as they are introduced.

"Arithmetic expressions work (almost) exactly as they did when you learnt them at school". Note, it does say "almost". There are actually a number of glitches. These arise because:

- In the computer you only have a finite number of bits to represent a number

  This leads to two possible problems
  your integer value is too big to represent
  your real number is only represented approximately

- You have to be careful if you want to combine integer and real values in an expression.

You may run into problems if you use short integers. Even if the correct result of a calculation would be in the range for short integers (-32768 to +32767) an intermediate stage in a calculation may involve a number outside this range. If an out of range number is generated at any stage, the final result is likely to be wrong. (Some C++ compilers protect you from such problems by using long integers for intermediate results in all calculations.)

Integer division is also sometimes a source of problems. The "/" divide operator used with integers throws away any fraction part, so 10/4 is 2. Note that the following program says "result = 2" ...

```
void main()
{
    int i, j;
    double result;
    i = 10;
    j = 4;
    result = i / j;
    cout << "result = " << result << endl;
}
```

Because the arithmetic is done using integers, and the integer result 2 is converted to a real number 2.0.

6.5 EXAMPLES

6.5.1 "Exchange rates"

Specification:

1. The program is to convert an amount of money in Australian dollars into the corresponding amount given in US dollars.

2. The program will get the user to input two values, first the current exchange rate then the money amount. The exchange rate should be given as the

amount of US currency equivalent to $A1, e.g. if $A1 is 76¢ US then the value given as input should be 0.76.

3. The output should include details of the exchange rate used as well as the final amount in US money.

Program design

1. What data?

Seems like three variables:

exchange rate
aus dollars
us dollars

If these are "real" numbers (i.e. C++ floats or doubles) we can use the fraction parts to represent cents.

2. Where should variables be defined?

Data values are only needed in a single `main()` routine, so define them as local to that routine.

3. Pseudo-code outline

```
a) prompt for and then input exchange rate;
b) prompt for and then input Australian dollar amount;
c) calculate US dollar amount;
d) print details of exchange rate and amount exchanged;
e) print details of US dollars obtained.
```

4. Test data

If the exchange rate has $A1.00 = $US0.75 then $A400 should buy $US300. These values can be used for initial testing.

Implementation

Use the menu options in you integrated development environment to create a new project with options set to specify standard C++ and use of the `iostream` library. The IDE should create a "project" file with a main program file (probably named "main.cp" but the name may vary slightly according to the environment that you are using).

The IDE will create a skeletal outline for `main()`, e.g. Symantec 8 gives:

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

```
#include <iostream.h>

int main()
{
    cout << "hello world" << endl;

    return EXIT_SUCCESS;
}
```

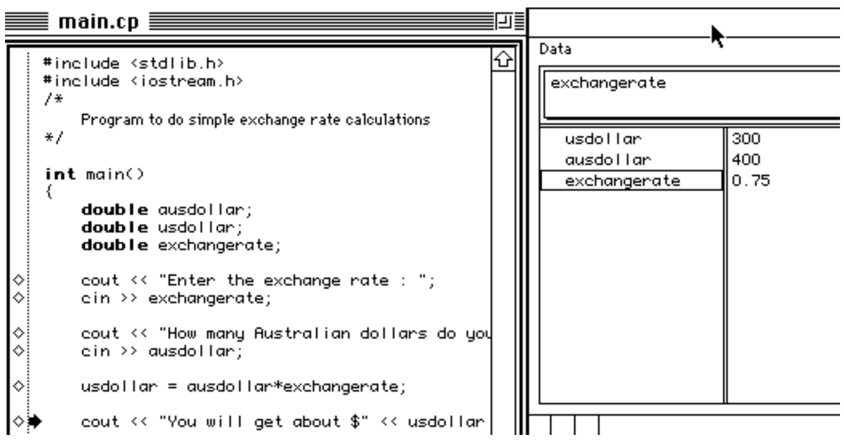
(Symantec has the name `EXIT_SUCCESS` defined in its stdlib; it just equates to zero, but it makes the return statement a little clearer.) Replace any junk filler code with your variable declarations and code:

```
int main()
{
    double ausdollar;
    double usdollar;
    double exchangerate;

    cout << "Enter the exchange rate : ";
    cin >> exchangerate;
    cout << "How many Australian dollars do you want "
                  "to exchange? ";
    cin >> ausdollar;

    usdollar = ausdollar*exchangerate;
    cout << "You will get about $" << usdollar <<
           " less bank charges etc." << endl;

    return EXIT_SUCCESS;
}
```

Insert some comments explaining the task performed by the program. These introductory comments should go either at the start of the file or after the `#include` statements.

```
#include <stdlib.h>
#include <iostream.h>
/*
    Program to do simple exchange rate calculations
*/

int main()
{
    …
```

Try running your program under control from the debugger. Find how to execute statements one by one and how to get data values displayed. Select the data variables `usdollar` etc for display before starting program execution; each time a variable is changed the value displayed should be updated in the debugger's data window. The values initially displayed will be arbitrary (I got values like 1.5e-303); "automatic" variables contain random bit patterns until an assignment statement is executed.



```
#include <stdlib.h>
#include <iostream.h>
/*
    Program to do simple exchange rate calculations
*/

int main()
{
    double ausdollar;
    double usdollar;
    double exchangerate;

    cout << "Enter the exchange rate : ";
    cin >> exchangerate;

    cout << "How many Australian dollars do you
    cin >> ausdollar;

    usdollar = ausdollar*exchangerate;

    cout << "You will get about $" << usdollar
```

```
exchangerate

usdollar        300
ausdollar       400
exchangerate    0.75
```

Figure 6.1 Screen dump of program run through the debugger.

Figure 6. 1 is a screen dump showing the program being executed in the Symantec 8 environment.

6.5.2 pH

Specification:

1. The program is to convert a "pH" value into a H+ ion concentration ([H+]) using (in reverse) the definition formula for pH:

   pH = $-\log_{10}$ [H+]

   If you have forgotten you high school chemistry, the [H+] concentration defines the acidity of an aqueous solution; its units are in "moles per litre" and values vary from about 10 (very acidic) down to 0.00000000000001 (highly alkaline). The pH scale, from about -1 to about 14, is just a more convenient numeric scale for talking about these acidity values.

2. The user is to enter the pH value (should be in range -1..14 but there is no need to check this); the program prints both pH value and [H+].

Program design

1. Data:
   Just the two values, pH and HConc, both would be "doubles", both would be defined as local to the main program.

2. The formula has to be sorted out:

   pH = -log10 [H+]

so

[H+] = 10^-pH

(meaning 10 to power -pH).

How to calculate that?
This can be calculated using a function provided in the maths support library `math.h`; a detailed explanation is given at the start of "Implementation" below.

3. Program organization:

   Need to "#include" the maths library in addition to `iostream.h`.

4. Pseudo code outline

```
a) prompt for and then input pH;
b) calculate HConc;
d) print details of pH and corresponding [H+] value.
```

Implementation

Finding out about library functions

Many assignments will need a little "research" to find out about standard library functions that are required to implement a program.

Here, we need to check what the `math.h` library offers. Your IDE will let you open any of the header files defining standard libraries. Usually, all you need do is select the name of a header file from among those listed at the top of your program file and use a menu command such as "Open selection". The header file will be displayed in a new editor window. Don't change it! Just read through the list of functions that it defines.

The library `math.h` has a function

```
double pow(double d1, double d2)
```

which computes d1 to the power d2. We can use this function.

Getting details about library functions

The Borland IDE has an integrated help system. Once you have a list of functions displayed you can select any function name and invoke "Help". The help system will display information about the chosen function. (Symantec's IDE is slightly clumsier; you have to have a second "Reference" program running along with the IDE. But you can do more or less the same name lookup as is done directly in the Borland system; `pow` is in Symantec's Reference database called SLR – standard library routines.)

Using simple mathematical functions

The use of functions is examined in more detail in Chapters 11 and 12. But the standard mathematical functions represent a particularly simple case. The maths functions include:

Function prototypes of some maths functions

```
double cos(double);          cosine function
double sin(double);          sine function
double sqrt(double);         square-root
double tan(double);          tangent
double log10(double);        logarithm to base 10
double pow(double, double);     power
```

These are "function prototypes" – they specify the name of the function, the type of value computed (in all these cases the result is a double real number), and identify the data that the functions need. Apart from `pow()` which requires two data items to work with, all these functions require just one double precision data value. The functions `sin()`, `cos()` and `tan()` require the value of an angle (in radians); `sqrt()` and `log10()` must be given a number whose root or logarithm is to be calculated.

You can understand how to work with these functions in a program by analogy with the similar function keys on a pocket calculator. If you need the square root of a number, you key it in on the numeric keys and get the number shown in the display register. You then invoke the sqrt function by activating its key. The sqrt function is implemented in code in the memory of the microprocessor in the calculator. The code takes the value in the display register; computes with it to get the square root; finally the result is put back into the display register.

"Passing arguments to a function"

It is very much the same in a program. The code for the *calling* program works out the value for the data that is to *be passed t*o the function. This value is placed on the stack (see section 4.8), which here serves the same role as the display in the calculator. Data values passed to functions are referred to as "arguments of the function". A subroutine call instruction is then made (section 2.4) to invoke the code of the function. The code for the function takes the value from the stack, performs the calculation, and puts the result back on the stack. The function returns to the calling program ("return from subroutine" instruction). The calling program collects the result from the stack.

Using a mathematical function

The following code fragment illustrates a call to the sine function passing a data value as an argument:

```
double angle;
double sine;
cout << "Enter angle : ";
cin >> angle;
sine = sin(angle*3.141592/180.0);
cout << "Sine(" << angle << ") = " << sine << endl;
```

The implementation of the pH calculation program is straightforward. The IDE's skeletal outline for `main()` has to be modified to include the `math.h` library, and the two variables must be declared:

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

int main()
{
    double pH;
    double HConc;
```

The body of the main function must then be filled out with the code prompting for and reading the pH value and then doing the calculation:

```
cout << "Enter pH : ";
cin >> pH;

HConc = pow(10.0, -pH);

cout << "If the pH is " << pH << " then the [H+] is " <<
       HConc << endl;
```

6.6 NAMING RULES FOR VARIABLES

C++ has rules that limit the forms of names used for variables.

- Variables have names that start with a letter (uppercase or lowercase), or an underscore character (_).

- The names contain only letters (abcdefghijklmnopqrstuvwxyzABC...YZ), digits (0123456789) and underscore characters _.

- You can't use C++ "reserved words".

In addition, you should remember that the libraries define some variables (e.g. the variable `cin` is defined in the `iostream` libray). Use of a name that already has been defined in a library that you have #included will result in an error message.

Avoid names that start with the underscore character "_". Normally, such names are meant to be used for extra variables that a compiler may need to invent.

C++'s "reserved words"

C++'s "reserved words" are:

```
asm           auto         bool       break       case
catch         char         class      const       const_cast
continue      default      delete     do          double
dynamic_cast  else         enum       extern      false
float         for          friend     goto        if
inline        int          long       mutable     namespace
new           operator     private    protected   public
register      reinterpret_cast        return      short
signed        sizeof       static     struct      switch
template      this         throw      try         typedef
typeid        typename     union      unsigned    using
virtual       void         volatile   wchar_t     while
```

The following words (classed as "alternative representations for operators") are also reserved:

```
bitand        and          bitor      xor         compl
and_eq        or_eq        xor_eq     not         not_eq
```

You can't use any of these words as names of variables. (C++ is still undergoing standardization. The set of reserved words is not quite finalized. Some words given in the list above are relatively new additions and some compilers won't regard them as reserved.)

Unlike some other languages, C++ (and C) care about the case of letters. So, for example, the names `largest`, `Largest`, `LARGEST` would be names of *different* variables. (You shouldn't use this feature. Any coworkers who have to read your program will inevitably get confused by such similar names being used for different data items.)

6.7 CONSTANTS

You often need constants in your program:

> gravitational constant (acceleration by earth's gravity,
> approximately 32 feet per second per second)
> speed of light
> π
> marks used to define grades (45, 50, 65, 75, 85 etc)
> size of "window" to be displayed on screen (width = ..., height = ...)
> …

Don't embed "magic numbers" in code!

It is a *bad* idea to type in the value of the constant at each place you use it:

```
    /* cannonball is accelerated to earth by gravity ... */
    Vert_velocity = -32.0*t*t;
    ...
```

If you need to change things (e.g. you want to calculate in metres rather than feet) you have to go right through your program changing things (the gravitational constant is 9.8 not 32.0 when you change to metres per second per second).

It is quite easy to make mistakes when doing such editing. Further, the program just isn't clear when these strange numbers turn up in odd places (one keeps having to refer back to separate documentation to check what the numbers might represent).

C++ allows you to define constants – named entities that have an fixed values set before the code is run. (The compiler does its best to prevent you from treating these entities as variables!) Constants may be defined local to a particular routine, but often they are defined at the start of a file so that they can be used in any routine defined later in that file.

Definition of a constant

A constant definition has a form like

```
const   Type  Name = Value;
```

Type is the type of the constant, short, long, double etc. *Name* is its name. *Value* is the value (it has to be something that the compiler can work out and is appropriate to the type specified).

"const"

Examples

```
const double g = 32.0; /* Gravitational constant in fpsps */
const double Neutral_pH = 7.0; /* pH of pure water */
const double Pool_pH = 7.4;
          /* Swimming pool is slightly alkaline */
const int PCMARK = 45;
const int PMARK = 50;
const int CREDITMARK = 65;

const char TAB = '\t';
const char QUERY = '?';

const double PI = 3.1415926535898;
const double SOMENUM = 1.235E75;

const double DEGREES_TO_RADIANS = PI / 180.0;
```

As shown in last example, the value for a constant isn't limited to being a simple number. It is anything that the compiler can work out.

Character constants, e.g. `Query`, have the character enclosed in single quote marks (if the character is one of those like `TAB` that is represented by a special two or four character code then this character code is bracketed by single quotes). Double quote marks are used for multi-character text strings like those already shown in output statements of example programs.

Integer constants have the sign (optional if positive) followed by a digit sequence. Real number constants can be defined as `float` or `double`; fixed and scientific formats are both allowed: 0.00123, or 1.23E-3.

You will sometimes see constant definitions with value like `49L`. The `L` after the digits informs the compiler that this number is to be represented as a long integer. You may also see numbers given in hexadecimal, e.g. 0xA3FE, (and even octal and binary forms); these usually appear when a programmer is trying to define a specific bit pattern. Examples will be given later when bitwise operations are illustrated.

#define The keyword `const` was introduced in C++ and later was retrofitted to the older C language. Before `const`, C used a different way of defining constants, and you will certainly see the old style in programs written by those who learnt C a long time ago. C used "*#define macros*" instead of `const` definitions:

```
#define g 32.0
#define CREDITMARK 65
```

(There are actually subtle differences in meaning; for example, the `#define` macros don't specify the type of the constant so a compiler can't thoroughly check that a constant is used only in correct contexts.)

6.7.1 Example program with some const definitions.

Specification

The program it print the area and the circumference of a circle with a radius given as an input value.

Program design

1. Data:
   The three values, radius, aread, and circumference, all defined as local to the main program.

2. Thevalue π has to be defined as a const. It can be defined inside main routine as there are no other routines that would want to use it.

```
int main()
{
    const double PI =  3.1415926535898;

    double radius;
    double area;
    double circumference;

    cout << "Enter radius : ";
    cin >> radius;

    area = PI*radius*radius;
    circumference = 2.0*PI*radius;

    cout << "Area of circle of radius " << radius << " is "
                  << area << endl;
    cout << "\tand its circumference is " << circumference
                  << endl;
}
```

6.8 INITIALIZATION OF VARIABLES

When you plan a program, you think about the variables you need, you decide how these variables should be initialized, and you determine the subsequent processing steps. Initially, you tend to think about these aspects separately. Consequently, in your first programs you have code like:

```
int main()
{
    double x, y, z;
    int n;
    …
    // Initialize
    x = y = z = 1.0;
    n = 0;
    …
```

The initialization steps are done by explicit assignments after the definitions.

This is not necessary. You can specify the initial values of variables as you define them.

```
int main()
```

```
{
    double x = 1.0, y = 1.0, z = 1.0;
    int n = 0;
```

Of course, if a variable is immediately reset by an input statement there isn't much point in giving it an initial value. So, in the example programs above, things like the radius `r` and the `pH` value were defined but not initialized.

Beware of uninitialized variables

However, you should be careful about initializations. It has been noticed that uninitialized variables are a common source of errors in programs. If you forget to initialize an "automatic variable" (one of the local variables defined in a function) then its initial value is undefined. It usually gets to have some arbitrary numeric value based on whatever bit pattern happened to be found in the memory location that was set aside for it in the stack (this bit pattern will be something left over from a previously executing program). You are a bit safer with globals – variables defined outside of any function – because these should be initialized to zero by the loader.

It is easy to miss out an initialization statement that should come after the definition of a local variable. It is a little safer if you combine definition with initialization.

6.9 CONVERTING DATA VALUES

What should a compiler do with code like the following?

```
int main()
{
    int    n;
    double d;
    …
    cin >> n;
    …
    d = n;
```

The code says assign the value of the integer `n` to the double precision real number `d`.

As discussed in Chapter 1, integers and real numbers are represented using quite different organization of bit pattern data and they actually require different numbers of bytes of memory. Consequently, the compiler cannot simply code up some instructions to copy a bit pattern from one location to another. The data value would have to be converted into the appropriate form for the lvalue.

The compilers for some languages (e.g. Pascal) are set to regard such an assignment as a kind of mistake; code equivalent to that shown above would result in a warning or a compilation error.

Such assignments are permitted in C++. A C++ compiler is able to deal with them because it has an internal table with rules detailing what it has to do if the type of the lvalue does not match the type of the expression. The rule for converting integers to reals are simple (call a "float" routine that converts from integer to real). This rule would be used when assigning an integer to a double as in the assignment shown above, and also if the compiler noticed that a function that needed to work on a double data value (e.g. `sqrt()`) was being passed an integer. Because the compiler checks that functions are being passed the correct types of data, the following code works in C++:

```
double d:
…
d = sqrt(2);
```

Since the `math.h` header file specifies that `sqrt()` takes a `double`, a C+ compiler carefully puts in extra code to change the integer 2 value into a `double` 2.0.

Conversions of doubles to integers necessarily involve loss of fractional parts; so the code:

```
…
int    m, n;
double x, y;
…
x = 17.7; y = - 15.6;
m = x; n = y;
cout << m << endl;
cout << n << endl;
```

will result in output of 17 and -15 (note these aren't the integers that would be closest in value to the given real numbers).

The math library contains two functions that perform related conversions from doubles to integers. The `ceil()` function can be used to get the next integer greater than a given double:

```
x = 17.7; y = - 15.6;
m = ceil(x); n = ceil(y);
cout << m << endl;
cout << n << endl;
```

will print the values 18 and -15.

Similarly there is a `floor()` function; it returns the largest integer smaller than a given double value. If you want to get the closest integer to real number `x`, you can try `floor(x + 0.5)` (this should work for both positive and negative `x`).

Minor note regarding default output formats

The code fragments above all printed data values one per line. What should you get from code like this:

```
int    n, m;
…
n = 55;
m = 17;
…
cout << n << m << endl
```

You get a single line of output, with apparently just one number; the number 5517.

The default setting for output to `cout` prints numbers in the minimum amount of space. So there are no spaces in front of the number, the sign is only printed if the number is negative, and there are no trailing spaces printed after the last digit. The value fifty five can be printed using two 5 digits, the digits 1 and 7 for the value seventeen follow immediately.

Naturally, this can confuse people reading the output. ("Shouldn't there be two values on that line?" "Yes. The values are five and five hundred and seventeen, I just didn't print them tidily.")

If you are printing several numeric values on one line, put spaces (or informative text labels) between them. At least you should use something like:

```
cout << n << ", " << m << endl;
```

so as to get output like "55, 17".

EXERCISES

Programs that only use sequences of statements are severely limited and all pretty much the same. They are still worth trying – simply to gain experience with the development environment that is to be used for more serious examples.

1. Write a program that given the resistance of a circuit element and the applied voltage will calculate the current flowing (Ohm's law). (Yes you do remember it – V = IR.)

2. Write a program that uses Newton's laws of motion to calculate the speed and the distance travelled by an object undergoing uniform acceleration. The program is to take as input the mass, and initial velocity of an object, and the constant force applied and time period to be considered. The program is to calculate the acceleration and then use this value to calculate the distance travelled and final velocity at the end of the specified time period.

3. Write a program that takes as input an investment amount, a percentage rate of compound interest, and a time period in years and uses these data to calculate the final value of the investment (use the pow() function).