

# Computer Science Department

## Introduction to Programming 1

### Brief Summary Notes by Eric Foxley

#### Chapter 1 : Background

You will be given duplicated notes each week as part of the programming course. You should supplement these by reading your own books, or books in the library.

##### 1.1. High level languages

There are many languages for programming computers; you may perhaps have used one already, at school or on a home computer. Which one have you used at school? BASIC? Pascal? None?

Old-fashioned serial languages include

BASIC

often used by beginners on home computers,

FORTRAN

old but still popular with some scientists and engineers, new versions are introduced every few years,

Algol

an elegant little-used internationally designed language, whose features are being incorporated into other languages,

COBOL

a widely used and well standardised language used in commerce,

APL

an interactive scientific language with a very mathematical notation,

PL/I

a failed attempt by IBM at achieving an all-purpose language, now almost dead,

Pascal

good for beginners, often taught as a first language,

Modula2

a development of Pascal to make it more realistic for large programs, and to enable modern program design techniques to be used,

C

a practical language, see below,

C++

a development of C, see below, and

ADA

a USA Department of Defense standard, now adopted by the UK Ministry of Defence also, aimed at *safe* programming for *real-time embedded* systems.

These languages are all essentially the same, they just involve different *syntactic sugar*, i.e. they are written using different formats, punctuation and grammars. The differences exist because the languages are aimed at different classes of users. For example those aimed at commercial users (COBOL, PL/I) will use more words and less symbols; we might see

```
rate_per_hour multiplied by hours_worked gives gross_pay
```

in COBOL, compared with

```
pay = rate * hours
```

in a language for engineers.

Languages aimed at scientific numerical users (FORTRAN, Pascal) offer facilities for simplifying the handling of vectors and matrices, and for performing very accurate arithmetic.

In addition to being aimed at different categories of user, languages may have other different objectives, such as aiming particularly at beginners (BASIC, Pascal), or being particularly safe for large projects (ADA, Modula2).

All of these languages are a formal notation for you to give sequential instructions to the computer. The instructions in all of the above languages are sequential ("Do this, then do that ...") and explicit.

There are now also many *fourth generation languages* or *4GLs*, aimed at retrieving information from modern databases.

There are other very different types of languages such a Prolog (taught at Nottingham to Computer Science 2nd years) which are of a completely different nature. In Prolog for example you essentially describe your problem, and leave the computer to decide how best to solve it.

## 1.2. Compilation versus interpretation

A typical BASIC or APL program is *interpreted*; that is, each line is decoded and interpreted by the computer each time it is executed. An instruction which occurs inside a repeated loop may have to be interpreted many times. This wastes computer time, and causes the program to run relatively slowly. It has the questionable advantage that parts of the program which are not executed do not need to be interpreted, so that an incorrectly typed line may not be detected until it comes to be executed.

Programs in most of the other languages mentioned above are *compiled*; that is, the whole program is first analysed and digested by a *compiler*, which converts it into a machine executable form. This machine executable form runs much faster than an interpreted program, since all of the analysis of the program statements has been completed before any execution starts. If you request it, the compiler will spend additional time making the compiled program as efficient as possible.

The compilation is often in two distinct stages, first *compiling* your program into an *object* module, and then *loading* the object module into an *executable* program. At the loading stage, items from a *library* to perform certain standard operations (to handle networks, or to draw pictures, for example) may be combined with the programmer's object module.

Language interpreters usually include some simple form of editor, such as the line numbering system in BASIC. Every line of the program starts with a number; the number determines the choice of a line to change, and the sequence in which lines are executed. Compiled languages use programs which have been stored in ordinary text files. You can use any editor to create the original text source file; the most commonly available general UNIX text editor is `vi`, the most commonly used editor on SUN computers is `emacs`.

## 1.3. A brief history of C++

Brian Kernighan et al (as they developed the UNIX computer operating system) required a language for writing a computer system. At that time (about 1970) most systems programs were written in *assembly language* for reasons of efficiency. This involves expressing the problem in terms of very low-level machine operations on a particular type of computer. The resulting program is long, tedious, error prone, non-portable and difficult to change. Brian Kernighan and his colleagues realised that the use of *Assembler* was to be avoided at all costs.

A language called BCPL had been developed in the UK specifically for writing computer systems. Brian Kernighan's first attempt at a language was based on BCPL, adding various features to make it more useful, and was called "B".

After some more experience, they decided that a new language was needed. A new language was therefore developed, and was called "C". This was used to write the next version of UNIX system software, which eventually became the world's first *portable* operating system.

C has now become a widely used professional language for various reasons.

- 1: It has high-level constructs.
- 2: It can handle low-level activities.
- 3: It produces efficient programs.
- 4: It can be compiled on a variety of computers.

Its main drawback is that it has poor error detection.

The standard for C programs was originally the features as set and implemented by Brian Kernighan. In order to make the language more internationally acceptable, an international standard was developed, ANSI C (American National Standards Institute).

Another group developed C to reflect modern developments in program design, in particular object-oriented programming. This language became "C++". C++ may be considered in several ways.

- 1: An extension of C.
- 2: A "data abstraction" improvement on C.
- 3: A base for "object-oriented" programming.

The first part of this course therefore looks rather like a C course, and covers the basic programming features of C++. The next module ("pr2") takes you on to the object oriented features. A third module ("oop") will teach object oriented program design techniques.

#### 1.4. A minimal C++ program

It is high time that we stopped talking about programming languages, and saw an actual C++ program; the simplest possible program might be as follows.

```
// Sample minimal program
// from EF's C++ notes

#include <iostream.h>

main() {
    cout << "Hi there!\n";
    exit ( 0 );
} // end of program
```

The text of the program as shown here would be stored in a file.

#### 1.5. Creating, compiling and running your program

The stages of developing your C++ program are as follows.

##### 1.5.1. Creating the program

Create a file containing the complete program, such as the above example, using any ordinary editor with which you are familiar such as **vi** or **emacs** .

The filename must by convention end ".C" (full stop, capital C), e.g. **myprog.C** or **progtest.C** . The contents must be as in the above example, starting with the line

```
// Sample ....
```

or a blank line preceding it, and ending with the line

```
} // end of program
```

or a blank line following it. The line

```
#include
```

must start with the "#" symbol in column 1.

## 1.5.2. Compilation

Compile your program with the command

```
g++ program.C
```

where `program.C` is the name of the file.

If there are obvious errors in your program (such as typing

```
main((
```

instead of

```
main()
```

or misspelling one of the key words or omitting a semi-colon ), the compiler will detect and report them. The compiler will tell you the number of the line where it detected the error; this may not be the line on which the error occurs, it is often the following line! Usually only the first reported error is significant; later error messages may be spuriously generated by the first genuine error. If errors occur, you must correct them using the editor, and then call the compiler again, and repeat this process until no errors are reported.

The compiler's error messages contain the word *Error*. There may be other messages from the compiler containing the word *Warning*. These represent constructions in your program which the compiler thinks suspicious. You do not have to correct them, but you should make sure that they are not significant.

There may, of course, still be logical errors that the compiler cannot detect. You may be telling the computer to do the wrong operations.

When the compiler has successfully digested your program, the compiled version, or *executable*, is left in a file called `a.out`. After a successful compilation, execute the command

```
ls -l
```

to see that a file `a.out` exists and has execute permission; the output from

```
ls -l
```

should now include a line such as

```
-rwxr-xr-x 1 ceilidh 286892 Jan 7 11:58 a.out
```

showing "read, write, execute" permission for the owner, "read, execute" for others. Observe its size, and compare it with that of the original program source.

## 1.5.3. Running the program

The next stage is to actually run your executable program. To run an executable in UNIX, you simply type the name of the file containing it, in this case

```
a.out
```

or you may need to type

```
./a.out
```

depending how your `PATH` is set up.

This executes your program, printing any results to the screen. At this stage there may be *run-time* errors, such as division by zero, or it may become evident that the program has produced incorrect output. If so, you must return to edit your program source, and recompile it, and run it again. For any serious program, the testing process must be thorough, and will involve careful planning of the number of tests required, and the test data chosen to exercise each part of the program.

## General points

Once you have an executable program in the file `a.out` , you can use `a.out` as just another UNIX command.

Any input requested by the program must be typed at the keyboard; there is no built-in prompt as in some other languages; the program just halts until you have typed the required input, terminated by end-of-line.

To take your input data from a file called *input\_file* , use the symbol "<" in the shell and type for example

```
a.out < input_file
```

Any program output normally comes to the screen; to send output to a file *output\_file* use the symbol ">" in the shell, and type for example

```
a.out > output_file
```

This overwrites any information which was previously in the file *output\_file* with the new output. To append the new output to any information already in the file without overwriting, use the ">>" operation as in

```
a.out >> output_file
```

To pipe output into another program use for example

```
a.out | wc
```

or

```
a.out < data_file | wc > out_file
```

to see how many lines, words and characters are in the program output.

It may be more convenient to use a "-o" and filename in the compilation as in

```
g++ -o program program.C
```

which puts the compiled program into the file `program` (or any file you name following the "-o" argument) instead of putting it in the file `a.out` .

You can then run it using the command

```
program
```

Alternatively you could have used the earlier compilation command, and then executed

```
mv a.out program
```

to rename the `a.out` file, or

```
cp a.out program
```

to take a copy.

You can now keep several different compiled programs. Beware though, they all occupy disc space, and you have an upper limit on your total disc occupation. Always do an

```
ls -l
```

from time to time to check your files. It is best to delete executables that you do not require immediately; they can always be quickly regenerated by compilation if you need them.

## C++ compilers

A compiler is itself just a program, albeit a large and complex one. There are (at least) two C++ compilers on the SUN computers. One is invoked by the command `g++` , the other by `cc` (in capitals). Both have the same options such as "-o" above, but use different methods to compile the program. Beware; they may have slightly different habits for their `#include` files. If `iostream.h` doesn't seem to work, replace it with `stream.h` and try again, and vice versa.

## 1.6. The Ceilidh† system

Programming is a practical subject, and can be learned only by practical experience. It is no use simply reading and listening how to do it, you MUST get your hands dirty and actually do it. The Ceilidh system was developed to make the practical work of programming courses as effective and productive as possible.

You will have weekly laboratory sessions for this course, and there will be at least one programming exercise every week. All of these exercises will be assessed, and the results form the basis of your end-of-course mark.

You will be expected to use the Ceilidh system for reading the coursework definition each week, and for marking your results. The intervening operations of editing, compilation and test runs can be performed either inside or outside the Ceilidh system. What follows is a brief summary; for full details see the Student Guide<sup>1</sup> to Ceilidh.

The simplest option is, at least early in the course, to perform all your programming work inside Ceilidh. In Ceilidh, each course is divided into a number of units. Each week you will be asked to complete certain exercises in certain units. Inside the Ceilidh system, you perform all operations by choosing items from menus, and will typically work in the following sequence after typing the command

```
ceilidh
```

```
Set the required course ("sc pr1")
  and unit ("su 1" for now).
Select the required exercise number ("sx 1" for now).
View the coursework question ("vq").
Setup your outline program ("set").
Edit the program ("ed") source so that
  it does what the question asks for.
Compile it ("cm").
Run it ("run") from the terminal.
Repeat the previous three steps until you are satisfied
  that the program is correct.
Ask the system to mark it ("sub").
Check that the submission is OK ("cks").
Quit Ceilidh ("q").
```

To start with, within course "pr1" try selecting unit 1, and then exercise 1 to try out the system. When you type

```
sub
```

to submit the program, the computer's reply will be something like

```
Analysis of Dynamic Correctness
      item:grade:weight
      Look for "Hi":  A  : 30
      Look for "there": A  : 40
      Look for "!":   A  : 30
Grade for Dynamic correctness is  A
```

These lines mean that it looked for the words **Hi** and **there** and an exclamation mark **!** in your output, and found them all. The right hand column gives the weight for each of the items it finds, 30% for finding "Hi", 40% for finding "there", and so on. The overall grade for this test was an A. We are testing that your program, when run, produces the correct output; this is called *Dynamic Correctness*.

```
Mark summary
      category:grade:weight
      Dynamic:  A  : 100
Overall grade awarded [range E-A] A
```

---

† "Ceilidh" stands for Computer Environment for Integrated Learning in Diverse Habitats.

In this case, there is only dynamic testing; at a later stage, other tests will be performed, and this part of the output summarises the overall assessment.

```
File for Unit 1 ex 1 suffix C copied to coursework directory
Mark for Unit 1 ex 1 copied to coursework directory
```

Your program source is copied to Ceilidh for future reference (perhaps by your tutor, or by an external examiner), and your marks are noted.

```
Do you want to comment on the mark [ny]?
```

You may comment on the mark; comments are sent as electronic mail to the course teacher; please don't pester us with trivia!

Whenever Ceilidh asks a question such as

```
Do you want to comment on the mark [ny]?
```

the possible replies are given in the square brackets, in this case "n" and "y". The default answer, which is assumed if you just hit return without typing a letter, is always the first one listed, "n" in this case.

To work outside the Ceilidh system, first use Ceilidh to read the coursework definition ("vq") and to set up your skeleton program ("set"). Then leave the system ("q"), and use **vi** or **emacs** to edit, a command such as **g++ -o prog11 prog11.C** to compile, and **prog11** to run your program, directly in your UNIX shell as required.

Then ALWAYS return to Ceilidh to issue your marking ("sub") command to mark and submit the work. In the early stages of the course, you are asked to show your working program to one of the laboratory demonstrators.

You must ALWAYS at some stage use the mark and submission command **sub** in Ceilidh to show that you have completed the work. If you forget this, the teacher will have no evidence that you have done the work. Do NOT hand sheets of paper to anyone, all work is monitored on-line. You should always use **cks** to check that the submission of your work has taken place satisfactorily.

You can enter course "pr1" directly without having to type

```
ceilidh
sc pr1
```

by calling

```
ceilidh -c pr1
```

The computer system should have a built-in command

```
pr1
```

which does this, but if not, look up your shell details, and set up your own "alias" such as

```
alias pr1 "ceilidh -c pr1"
```

to make "pr1" equivalent to "ceilidh -c pr1".

## 1.7. General

There are a number of books available on C++, including the ones by Neill Graham (Learning C++, McGraw-Hill, 1991), Kenneth Barclay & Brian Gordon (C++ Problem Solving and Programming, Prentice Hall, 1994), and Frank Friedman and Elliot Koffman (Problem Solving, Abstraction and Design Using C++, Addison Wesley, 1994). Choose any book with which you feel happy. The notes supplied with this course should be fairly comprehensive.

You may need to contact the course teacher at some point. Eric Foxley works jointly between the Computer Science and Mathematics departments; he may therefore be difficult to locate! He may be available in his Computer Science office (Tower building floor 11 room 1102, internal phone 4210) part of the time, or in his Mathematics department office (Maths/Physics building top floor room C111, internal phone 4953). To find which office he is in at any given time, you will of course use either

```
rwwho | grep ef
```

to see if **ef** is logged on, to which machine, and from where; or the

```
locate ef
```

command. It may be more convenient to use electronic mail to **ef** or the **co** (comment) facility in Ceilidh.

The marking for exercises in unit 1 of the course is purely on *dynamic correctness*. We run your program, and check that it produces the correct output.

© Eric Foxley November 12, 1996

## References

1. Steve Benford, Edmund Burke, and Eric Foxley, *Student's Guide to the Ceilidh System (2.4)*, 1995. LTR Report, Computer Science Dept, Nottingham University

## Chapter 2 : Elementary programming

We will now concentrate exclusively on the content of programs.

### 2.1. A simple program

We return to the elementary program used in the previous chapter, and discuss its features.

```
// Program written by EF
// October 1991

#include <iostream.h>

main() {
    cout << "Hi there!\n";
    // The "\n" represents a newline character
} // end of the main program
```

Notes:

- (i) Any text from `"/` to end of line is a comment, and is ignored by the compiler. There should be enough comments to make the program file understandable to someone who reads it. In the very short programs that you will write for your first few exercises, comments may not appear so important to you. In large realistic programs, comments are **very** important, since when a program needs amendment several years after it was written, the original writer may well have moved to another company, or at least will have forgotten the principles of the program. Any text to the left of the `"/` is part of the program, such as the `}` on the last line, and is read by the compiler.
- (ii) The `\n` at the end of the printed string represents a newline character. If you omit it, you will find the next prompt from the computer appearing on the same line as the `Hi there!` instead of being on the next line. Other special characters such as `"tabn"` and `"bleep"` are handled this way.
- (iii) The line `" #include ... "` must start in column 1. It causes a named file of C++ code to be inserted in the text at this point. This particular file `iostream.h` is in a system area of the computer, and contains certain standard items relevant to C++ programs. Further use of the `#include` facility for your own purposes will be essential during the second part of the course.
- (iv) The main program will (for now) always start with `main()` followed by an opening curly bracket. The reason for this convention will become obvious later.
- (v) The body of the program (the actual instructions to be executed by the computer) is contained between the `{` and `}` symbols. These symbols compare with the use of the keywords `BEGIN` and `END` in Pascal and other languages; in many aspects C uses features from other languages, but is always as terse as possible.
- (vi) Every statement which is an instruction to the computer to do something must end with a semi-colon. The symbols `{` and `}` and the `main()` are not instructions to "do something", but are part of the layout of the program; they do not have to be followed by a semi-colon. (This again is different from Pascal and other languages).
- (vii) No particular layout with newlines, tab characters and spaces (sometimes called "white space") is enforced, but you **MUST** make the program easily readable. The only place that extra space must **NOT** be inserted is inside words such as `"main"` and `"cout"`, and within the actual text to be printed (which will be printed exactly as given in the program, and must not contain a newline character). The end of line does not terminate a particular instruction; there must be a semi-colon present.

### 2.2. Output from the program

The identifier `cout` is used in combination with the `<<` operator in printing instructions as follows.

```
// Program written by EF
// October 1991
```

```

#include <iostream.h>

main() {

// to print several lines of text
cout <<
    "My room is\n1102 Tower\nUninott\n";
// or you could write
cout << "My room is\n"
    << "Room 1102\n"
    << "Uninott\n";

// and to combine values and text
cout << "One seventh is "
    << 1.0 / 7 << "\n";

} // end main

```

Text between double quotes is printed exactly as given, arithmetic values are evaluated and printed as numbers.

### 2.3. Constants and variables

We will wish to have identifiers to represent

- (i) constants whose value is required in the program; and
- (ii) variables which will be used for storing intermediate results during the running of the program.

For each identifier that we use, we must tell the computer the type of object which we wish to use it for.

```

// Program written by EF
// October 1991

#include <iostream.h>

// A float constant, value fixed
const float pi = 3.14159;

main() {

// A integer variable, initialised
// Its value may be reassigned
int number = 7;

// A float variable, initialised
// It can contain non-integral values
float reciprocal = 1.0 / number;

// An integer variable, not initialised
// to any particular value
int square;

// Now to print some results
cout << "The reciprocal of "
    << number << " is "
    << reciprocal << "\n";

cout << "The area of a unit circle is "
    << pi << "\n";

square = number * number;
cout << "Square is " << square << ".\n";

} // end main

```

The first print ( `cout <<` ) instruction will print the text

```
The reciprocal of 7 is 0.142857
```

The second prints

```
The area of ... is 3.14159
```

The third will print

```
Square is 49.
```

Each print instruction in the above examples ends with a newline character. You will generally want to end with a newline, unless you are printing a prompt requesting the user to enter a reply.

## 2.4. Identifiers

The identifiers you choose to represent constants and variables must satisfy various rules and recommendations.

- They must start with a letter, which can be followed by any number of letters, digits and underscores.
- They should be meaningful. In general, you should **not** use single characters such as "x", "y", "i" and "j"; these terse identifiers give no feel for the significance of the values they represent.
- Your identifiers must not clash with certain special words. If you use "float", for example, the compiler will become very confused.
- Upper and lower case characters (capital and small letters) in identifiers are distinct.

Examples of identifiers are:

```
fred total_91 total_92
Fred ProgWeek1A
```

It is common practice to use either

the underscore symbol to separate the component parts of an identifier, as in

```
April_total_pay
```

or

to start each part with an upper case letter, as in

```
AprilTotalPay
```

## Declarations

Declarations are program statements which tell the compiler which identifiers we intend to use. If an identifier occurs which we have not declared (either we forgot to declare it, or, more likely, we mistyped an identifier) the compiler will print an error message. Declarations such as those in the above program do two things:

- They tell the compiler about an identifier/type pair, so that the compiler can interpret later statements correctly.
- At run time, the running program must set aside the necessary space to store the specified type of object.

At a later stage, we may need to distinguish between these two effects by separating a declaration into two parts.

You may combine a declaration with the assignment of an initial value. If you do not initialise a variable, you cannot rely on it containing any particular value before you use it.

We will insist (although it is not strict in C++) for the moment that you put **const** declarations before the **main()** line, and variable declarations after it, before any executable program code.

## 2.5. Basic types

The basic variable types in C++ are given below, together with their sizes in bytes on some implementations.

type	PDP	68000	VAXVMS
char	1	1	1
int	2	4	4
long int	4	8	4
short int	2	2	2
float	4	4	4
double	8	8	8

char: This can contain one character, a letter or digit or punctuation character.

int: This can contain integral (whole-number) values. There is a limit to the size of the largest positive and negative numbers which can be stored, which depends on how many bytes are occupied.

long int:

This also contains integral values, and *may* use more space than, and hence be able to store larger numbers than, an `int` variable. It may actually be no larger than an ordinary `int` variable.

short int:

This also contains integral values, and *may* use less space than, and hence be able only to store smaller numbers than, an `int` variable. It may actually be no smaller than an ordinary `int` variable, but you may need to conserve space if possible.

float: This type of variable can store all numeric values, not just integral values. The accuracy to which they are stored, and the maximum and minimum values, depend of the particular hardware/software being used.

double: This type of variable is intended for storing more accurate numeric values than `float` variables.

## 2.6. Denotations

A denotation is a representation of a particular value. Typical denotations for constants of various types are as follows.

int: Typical denotations might be

```
25 -15 +99
0177 //leading zero denotes octal
0XFF //leading zero and X denotes hexadecimal
```

long int:

The integer denotations must be followed by the letter "L" to indicate a `long int` denotation.

```
1999L 0L
```

char: Single character as values are always written between prime symbols, so that they are not confused with identifiers.

```
'a' 'z' '+' '.' ' '
'\t' // tab
'\n' // newline
'\a' // alert = bleep
'\177' // ASCII octal code
'\0'
'\' ' // backslash
'\'' // prime
```

float: Float values are typed as numbers with a decimal point, and can be optionally followed by the letter "E" (for exponent) and an integer. The value before the "E" is assumed to be multiplied by 10 raised to the power of the integer after the "E".

```
111.222 1.2345E6
```

The latter denotation represents the value 1234500.

## 2.7. Comments

Comments are from a double slash to end-of-line.

```
// ... comment ...  
  
float velocity; // velocity in kph
```

There is another type of comment in C++, compatible with that used in C, but I will stick with the above convention.

## 2.8. Program layout

Lay out your program carefully, with plenty of white space, indented as appropriate. I don't mind which convention you use, but be consistent! As programs become larger, program layout becomes more and more important. Don't forget the general document<sup>1</sup> for details.

## 2.9. Input to the program

We use `cout` and the operator "<<" for output, and `cin` and the operator ">>" for input to the program.

### Example

```
// Program written by EF  
// October 1991  
  
#include <iostream.h>  
  
main() {  
    int number;  
    float reciprocal;  
  
    cout << "Type a number: ";  
  
    // "cin" expects a value valid  
    // for the given type.  
    cin >> number;  
  
    // If you use "1" instead of "1.0"  
    // it would be an integer result.  
    reciprocal = 1.0 / number;  
  
    cout << "Reciprocal of " << number  
        << " is " << reciprocal  
        << "\n";  
  
} // end main
```

Note that the prompt does not have a newline character, and has a space before the final closing quote.

### Example

This second program reads two values.

```
// Program written by EF  
// Calculate the area of a rectangle  
  
#include <iostream.h>
```

```

main() {

// Declare three float variables
float length, breadth, area;

cout << "Type length and breadth: ";

// Read two values
cin >> length >> breadth;

// Compute area
area = length * breadth;

// Print results
cout << "Length " << length
  << ", breadth " << breadth
  << ", area is " << area << "\n";

} // end main

```

You could avoid the variable "area" all together, and print the results with

```
cout << "Area is " << length * breadth << "\n";
```

but this is not good practice.

It is good practice at this stage of your course to print out all of the values you have read in, so that the output gives a complete picture of what has been calculated.

You may lose a few marks on the dynamic testing if you do not do this. You will certainly lose marks if you do not put explanatory text into your output.

You can now write simple programs to read in some values, evaluate a formula, and print the result.

## 2.10. Operators in more detail

We will now have a thorough tour of all of the available operators and their significance.

### 2.10.1. Arithmetic operators

The simple arithmetic operators you would expect to see are

```
+ - * /
```

representing addition, subtraction, multiplication and division, respectively. As everywhere in C++, types are important here. Between two "int"s the result is an "int", otherwise (between two "float"s, or between an "int" and a "float") the result is a "float". This is fairly obvious for addition, subtraction and multiplication.

Beware of "/" between integers; the result is an "int", which may not be what you expected. It gives the quotient as an integer rounded down towards zero if the result is positive. If you type

```
float x = 1 / 7;
float y = 6 / 7;
```

you may not get the expected result; both x and y will be set to zero! If the result would be negative, the language does not define exactly what will happen, for example whether "10 / -7" is -1 (which you would get if you rounded towards zero) or -2 (if you rounded down).

The operator "%" between integers gives the remainder when the first is divided by the second. Thus "72 % 10" evaluates to 2, and "30 % 13" evaluates to 4. If either of the operands are negative there are ambiguities similar to those in division. The value of "10 % -7" might be -3 or +4. See the unit 2 exercises on time conversion and pay per hour for examples. The official definition says that the value of the expression

```
( a / b ) * b + a % b
```

must always equal the value of a.

## Exponentiation

There is **no** operator in C++ for exponentiation (for raising any number to a given power).

## Example programs

In the example programs given from now on, we may give only the text within the "{" and "}" of the main program. To run the program, you would have to add the lines up to `main(){` and the final `}`. Further, we may not include the niceties of prompts and input/output if we are really demonstrating other types of statement.

- (i) Convert Fahrenheit temperature to Celsius

```
float fahrenheit, celsius;
cout << "Type Fahrenheit temperature: ";
cin >> fahrenheit;
celsius = (fahrenheit - 32) * 5 / 9;
cout << "Celsius is " << celsius << "\n";
```

- (ii) I have a certain number of bicycle spokes; I need 44 to make one wheel; how many wheels can I make? How many spokes will I have left over?

```
const int spokes_per_wheel = 44;
int spokes, wheels, left_over;
cout << "How many spokes? ";
cin >> spokes;
wheels = spokes / spokes_per_wheel;
left_over = spokes % spokes_per_wheel;
```

- (iii) We know the number of football matches won, drawn and lost by a given team; how many points do they have (3 for a win, 1 for a draw)?

```
integer won, drawn, lost, points;
cin >> won >> drawn >> lost;
points = won * 3 + drawn;
```

### 2.10.2. Comparison operators

There are many other operators besides those used for arithmetic evaluations. We will need these in the next chapter for use in "if ... then" constructs.

```
a > b    // greater than
a < b    // less than
a >= b   // g.t. or equal to
a <= b   // l.t. or equal to
a == b  // equals
a != b  // not equals
// watch for the double "=" sign
```

These can be between (almost) any types of object. The result delivered is zero for FALSE, one for TRUE. In appropriate places in C++ generally, zero is always interpreted as meaning FALSE, while non-zero is interpreted as TRUE.

Note that testing for equality "==" between floats or doubles is not sensible, because of the possibility of rounding errors. The compiler will permit you to do it, but it is considered bad practice. You should instead look for a small absolute difference between the two values.

## Examples

- (i) Is the Fahrenheit temperature above freezing point?

```
fahrenheit > 32
```

- (ii) Do I have enough spokes for 2 bicycle wheels?

```
spokes >= 2 * spokes_per_wheel
```

### 2.10.3. Logical operators

For combining the results of comparisons, we need general logical operators. In fact, we are not limited to combining the results of comparisons; we can combine any values, and any zero value will be interpreted as FALSE, and non-zero value as TRUE. We use "&&" for the logical "and" and "||" for "or".

```
int number;

// set "number" to some value ...

number >= 0 && number < 10
// "&&" is "and",
// so true if the number is from 0 to 9 inclusive
// false otherwise

number < 0 || number >= 10
// "||" is "or" (inclusive or)
// one or the other or both
// so true if the number is outside the range 0 to 9

!( number >= 0 && number < 10)
// "!" is logical "negation"
// this expression has the same value as the previous one
// a monadic operator
```

### Examples

(i) Is the Celsius temperature today within the expected band for this time of year, say 5 to 15 degrees?

```
5 < celsius && celsius < 15
```

(ii) Can I construct at least 2 wheels with less than 10 spokes left over?

```
spokes >= 2 * spokes_per_wheel
&& spokes % spokes_per_wheel < 10
```

### 2.10.4. Incremental operators

These are unique to C and C++. Their real significance and use will become apparent later.

```
++i // increment, deliver new value
i++ // increment, deliver old value
--i // decrement, deliver new value
i-- // decrement, deliver old value
```

The word "increment" means "increment by a suitable value". Compare

```
i = 0; cout << i++;
```

which prints the value 0, with

```
i = 0; cout << ++i;
```

which prints the value 1. In both cases, "i" takes the value 1 after the instruction.

Note that

```
j = p + i++;
```

is equivalent to

```
j = p + i; i = i + 1;
```

where

```
j = p + ++i;
```

or

```
j = p + (++i);
```

is equivalent to

```
i = i + 1; j = p + i;
```

You will often see the free-standing increment

```
i++; // increment i, could be ++i
```

to add 1 to i, used instead of writing

```
i = i + 1;
```

You will see examples of these operators later.

### 2.10.5. Assignment

The values being assigned will be cast or coerced (their types will be changed and their values converted between types) as required. Examples of assignments include the following.

```
int i, j;
char c, lc;

i = ( j + 2 ) / 3; // integer divide
c = 'X';
j = c - 'A';
// j is ordinal of char c
lc = c - 'A' + 'a';
// lc is lower case char
// for upper case char c
i += 3;
// "plus-and-becomes"
i -= j;
// "minus-and-becomes"
i *= 10;
// "times-and-becomes"
```

#### Delivered Result

Assignment is an operator, and delivers as its result the value just assigned. Thus

```
i = ( c = getchar() ) - 'A';
```

is equivalent to

```
c = getchar();
i = c - 'A';
```

To assign the same value to several variables use

```
i = j = k = 0;
```

which is interpreted as

```
i = ( j = ( k = 0 ) );
```

Note that in initialising declarations, you MUST still write in full

```
int i = 0, j = 0, k = 0;
```

#### Examples

These operations could all be performed as two separate instructions; express them as two statements if you feel happier that way.

- (i) How many bicycle wheels can I make, and how many spokes will I have used?

```
spokes_used =
    ( wheels = spokes / spokes_per_wheel )
    * spokes_per_wheel;
```

## A warning

Beware of using "=" instead of "==", such as writing accidentally

```
if ( i = j ) ....
```

with a single equals sign. This copies the value in "j" into "i", and delivers this value, which will then be interpreted as TRUE if it is non-zero.

### 2.10.6. The comma operator

The real significance and use of this operator will appear later. An expression consisting of statements separated by commas, as in

```
statement1,  
statement2,  
statement3
```

causes each statement to be executed in turn; the result finally delivered is that delivered by the last statement. Thus you can write

```
i = ( c = getchar(), j = i - 'A' ) + k;
```

or

```
if (   
    c = getchar(),  
    i = c - '0',  
    c != '\n'  
 ) { ...
```

The effect of this last statement is exactly the same as if you had typed

```
c = getchar();  
i = c - '0';  
if (   
    c != '\n'  
 ) { ...
```

### 2.10.7. Operator precedence

It is necessary to define carefully the meaning of such expressions as

```
a + b * c
```

to define the effect as either

```
( a + b ) * c
```

or

```
a + ( b * c )
```

All operators have a priority, and high priority operators are evaluated before lower priority ones. Operators of the same priority are evaluated from left to right, so that

```
a - b - c
```

is evaluated as

```
( a - b ) - c
```

as you would expect.

Exact details are in any book on C++. There are many operators here that we have not yet met, but they are all entered here for completeness.

If you are ever in doubt, use extra parentheses to ensure the correct order of evaluation, and (equally important) to ensure the easy readability of the program.

From high priority to low priority the order is

```
( ) [ ] -> .
! ~ - * & sizeof cast ++ --
    (these are right->left)
* / %
+ -
< <= >= >
== !=
&
^
|
&&
||
?:      (right->left)
= += -= (right->left)
,      (comma)
```

Thus

```
a < 10 && 2 * b < c
```

is interpreted as

```
( a < 10 ) && ( ( 2 * b ) < c )
```

and

```
a =
b =
c =
    spokes / spokes_per_wheel
    + spares;
```

is interpreted as

```
a =
( b =
( c =
( spokes / spokes_per_wheel )
+ spares
)
);
```

## Program layout

We strongly suggest that you lay out your program as in the above examples, with the lines between "main" and the final "}" indented about 4 spaces, all other lines starting at the left hand edge, and a comment after the last closing "}". You should also put additional helpful comments at appropriate intervals throughout the program. The following example follows those rules.

```
// Program written by EF
// Calculate the area of a rectangle

#include <iostream.h>

main() {

// Declare three float variables
float length, breadth, area;

cout << "Type length and breadth: ";

// Read two values
cin >> length >> breadth;

// Compute area
```

```
    area = length * breadth;

// Print results
cout << "Length " << length
    << ", breadth " << breadth
    << ", area is " << area << "\n";

} // end main
```

### Coursework exercise marking

The exercises in this unit are marked partly on dynamic correctness and partly on *Typographic layout*. For dynamic correctness we run the program against samples of test data, and look in the output for the correct results; we also check for words in your output, so that you should print for example

```
123 degrees Fahrenheit corresponds to 65.34 degrees Celsius
```

rather than merely print a few numbers. In general you should also at this stage print out the values you have just read in, so that the output is complete in itself.

Your program itself should be easily "readable" to others. The "typographic" marker checks that your program is laid out as suggested above and in lectures; that it is indented correctly, that it contains a reasonable amount of "white space" (blank lines and spaces within lines), that you have a reasonable number of comments in the program, that the identifiers you have chosen are meaningful, and that you have done all the other things mentioned above to make a program readable to others.

The exact typographic marking metrics vary from exercise to exercise, but as a rough guide the typographic mark is formed as follows. The marking looks for correct indentation, approximately 20 to 30 characters per line, 30 to 60% comment in the program with comments after every "}", 20 to 50% blank lines, 20 to 30% white space in a line, and an average identifier length between 5 and 10.

© Eric Foxley December 6, 1996

### References

1. Cleveland Gibbon, *Writing Typographically Sound Programs with Ceilidh*, 1995. LTR Report, Computer Science Dept, Nottingham University

## Chapter 3 : Conditionals

We now get down to some typical program statements. This chapter is concerned with constructs which cause the program to take different paths depending on the values which have been assigned to program variables during the running of the program.

### 3.1. If statements

#### 3.1.1. Simple "if" statements

The simplest form of "if" statement causes selected statements to be executed if a certain condition holds at that point in the program.\*

```
int Radius, Result = 0;

cin >> Radius;

if ( Radius > 0 ) {
    Result = Radius * Radius;
    cout << "Radius is positive";
} // end if Radius > 0

cout << Radius << Result;
```

The condition to be tested appears in parentheses after the word "if", and the statement or statements whose execution is determined by the condition are contained within curly braces. If the condition does not hold, everything up to the next "}" is ignored, and program execution continues after the "}". In this example, if the radius value read in is positive, the program then executes in order the statements

```
Result = Radius * Radius;
cout << "Radius is positive";
cout << Radius << Result;
```

If the value read in is not positive, the statement in the braces following the "if" condition will be ignored, and the program will execute the single statement

```
cout << Radius << Result;
```

#### 3.1.2. "If ... else ..." statements

If the condition does not hold, we may wish to execute some different statements as alternatives to the "if" statements. We add the keyword "else" and the additional statements contained between curly braces.

```
int Money;
int Deposits = 0;
int Withdrawals = 0;
int Number = 0;

cin >> Money;

if ( Money > 0 ) {
    cout << "Deposit.\n";
    Deposits += Money;
    Number++;
} else { // if Money <= 0
    cout << "Withdrawal.\n";
    Withdrawals -= Money;
} // end if else Money > 0
```

---

\* From now on, we will not generally show complete programs, but just those internal statements under discussion. To complete the program you would need extra lines such as `#include ...` and `main() ...` at the top and `} // end`

```
cout << "Transaction noted.\n";
```

If the quantity read is positive, the program then executes

```
cout << "Deposit.\n";
Deposits += Money;
cout << "Transaction noted.\n";
```

If the quantity is zero or negative, the program executes

```
cout << "Withdrawal.\n";
Withdrawals -= Money;
cout << "Transaction noted.\n";
```

The careful layout of programs becomes more important as the programs become more structured. Always indent the statements between curly braces more than the lines containing the braces. Some people prefer to put the braces on separate lines as in

```
if ( Radius > 0 )
{
    ....;
    ....;
} // end of Radius > 0
else
{ // if Radius <= 0
    ....;
    ....;
} // end if else
```

Choose a method of laying out programs which you feel happy with, and keep to it. In legal jargon, I would say that the closing curly brace must line up with the first visible character on the line containing the opening curly brace.

### 3.1.3. Further alternatives

We can add further alternatives to an "if" statement if we require.

```
if ( Radius > 100 ) {
// Radius > 100
    ....;
} else if ( Radius > 10 ) {
// Radius > 10 and Radius <= 100
    ....;
} else if ( Radius > 1 ) {
// Radius > 1 and Radius <= 10
    ....;
} else {
// Radius <= 1
    ....;
} // end if Radius various values
```

The tests in the if statements are executed exactly in the order in which they are encountered. The first test here is "Radius > 100"; if this is false, the second test is executed, testing whether "Radius > 10", so that this is effectively the test "Radius <= 100 && Radius > 10", since we know that the first test is false.

### 3.1.4. Possible conditions (logical expressions)

We looked at the format of conditions in chapter 2, when we were discussing expressions in general.

```
// Test equality, use double equals
if ( code == 3 ) ....

// Two tests ANDed together
if ( Radius >= 0 && Radius <= 10 ) ....

// Two tests ORed together
```

```
if ( code == 0 || code == 1 ) ....
```

Remember all the points on conditions from chapter 2, in particular the interpretation of the value zero as FALSE, and non-zero as TRUE; the danger of using assignment instead of equality; and the danger of testing equality between "float"s.

Note that the AND ("&&") and OR ("||") used above are *lazy left-to-right* operators. This means that the expressions to be AND-ed together are evaluated from left to right, one at a time, and as soon as one is found which is FALSE, evaluation finishes and a FALSE result is returned. The remaining expressions are not evaluated at all. Only the minimum amount of work is done to determine the result. It is then perhaps safe to write

```
if ( Div > 0.001 && Total/Div < 100 ) { ...
```

since we can never get division be zero.

Similarly with OR, the expressions are evaluated left-to-right, and as soon as one is found which is TRUE, evaluation ceases and a TRUE result is returned.

### 3.1.5. The comma operator

You may find the comma operator useful when the test you wish to perform involves several calculations, as in

```
if ( Radius = 1.0 / i, circ = 2 * pi * Radius, circ > 5 ) {  
    ....;  
}
```

To make the program more readable, you may choose to lay this out on more lines as

```
if (  
    Radius = 1.0 / i,  
    circ = 2 * pi * Radius,  
    circ > 5  
) {  
    ....;  
}
```

The first two statements could be written before the "if" if you prefer, so that we could also write

```
Radius = 1.0 / i;  
circ = 2 * pi * Radius;  
if ( circ > 5 ) {  
    ....;  
}
```

You may not find the comma operator helpful at first. There are many philosophical arguments about programs and their structure, and the argument in favour of the comma here is that all the calculations involved in the test should be grouped together within the "if" condition.

### 3.1.6. Nesting "if" statements

Your "if" statements can be nested to your heart's content if that is what the program logic requires.

```
if ( Radius > 0 ) {  
    if ( Result > 0 ) {  
        // Radius > 0 and Result > 0 here  
        ....;  
    } else {  
        // Radius > 0 and Result <= 0 here  
        ....;  
    } // end if else Result > 0  
} else { // not Radius > 0  
    if ( Result > 0 ) {  
        // Radius <= 0 and Result > 0 here  
        ....;  
    }  
}
```

```

    } else {
// Radius <= 0 and Result <= 0 here
    ....;
    } // end if else Result > 0
} // end if else Radius > 0

```

### 3.1.7. Ambiguity

We have specified above that you must always use curly braces after an "if" condition and after the "else". The C++ official definition states that the curly braces are essential only if there is more than one statement to be executed as a result of the condition. Many commercial users of C++ insist, as we do, that the curly braces should always be there.

If you don't use curly braces, the following is ambiguous.

```

if ( Radius > 0 )
    if ( Result > 0 )
        xxx;
    else
        yyy;

```

The above could mean either

```

if ( Radius > 0 ) {
    if ( Result > 0 ) {
        xxx;
    } else {
        yyy;
    }
}

```

or

```

if ( Radius > 0 ) {
    if ( Result > 0 ) {
        xxx;
    }
} else {
    yyy;
}

```

These two have quite different effects. In the case when, for example, "Radius > 0" and not "Result > 0", the first will execute "yyy", the second will have no effect. If we have "Radius <= 0" (the first condition is FALSE) and "Result > 0", the first example will have no effect, while the second would execute "yyy". The two are thus quite different in their effect.

It is a good principle always to use curly braces, and to put a comment after any closing curly brace which is not close to its opening partner. An example might be

```

if ( Radius > 0 ) {
    ...;
    ...;
} // end if Radius > 0

```

Our rule (enforced by the Ceilidh marking system) will be that a closing curly brace must have a comment if it is more than 10 lines after its opening curly brace. This ensures that you see a comment on the computer terminal screen if the complete "if" statement is unlikely to fit onto one screenful of information.

See exercises such as the one on student-staff ratios or that on pay for problems requiring "if...then" constructs.

## 3.2. Switch statements

The "if" statement essentially gives a choice between two alternatives. We may sometimes need a choice between a larger number of possibilities. The "switch" construct illustrated below allows for any number of different actions to be taken dependent of the value of an integer calculation.

### 3.2.1. Switch example

```
int input_value;

cin >> input_value;

switch ( input_value ) {

    case 0 :
// if "input_value" is 0
    do_this();
    break;

    case 3 :
    case 4 :
// if "input_value" is 3 or 4
    do_that();
    break;

    case 7 :
// if "input_value" is 7
    do_the_other();
    break;

    default :
// if "input_value" is anything else
    yet_else();

} // end switch ( input_value )
```

### Case values

The value after the word "case" must be a constant, you could not put

```
case PartType :
```

where "PartType" is an "int" variable. You must put either an explicit constant (the numeric value 7), or a declared const (where you have declared for example

```
const int ins_per_ft = 12;
```

as a global constant).

Two "case" labels can be adjacent. In this case, the two specified values will cause the same code to be executed.

Don't forget the "break;" statements if you need them. You will normally want control to leave the "switch" statement at the end of each separate "case".

The **default:** entry is optional, but will most often be included.

The value in the "switch" parentheses must be an integer variable or expression. It cannot be a "float" value.

See the exercise on "Day of the week" for an example of an integer switch.

### 3.2.2. Character switch example

```
char command_char;

cin >> command_char;

switch( command_char ) {
// Perhaps 'e' for "edit"
  case 'e' :
    edit();
    break;

// Perhaps 'l' for list/print
  case 'l' :
  case 'p' :
    print();
    break;

// Some other character
  default :
    cout << "Don't understand "
         << command_char << "\n";

} // end switch ( command_char )
```

See the exercise on road travel for an example requiring a character switch.

Note again the careful indentation, and the comment after the closing curly brace.

Think also of omitting the **break** statements on the rare occasions when you wish to cause code to follow through as in this example.

```
case 'e' :
  edit();
case 'c' :
  compile();
case 'r' :
  run();
  break;
```

In this case,

```
'r' causes "run"
'c' causes "compile" then "run"
'e' causes "edit" then "compile" then "run"
```

### 3.2.3. Notes

The expression in brackets after "switch" must deliver an integral value, not a float or double. Integers, characters and long integers are permitted.

### 3.3. Program layout

As we learn new constructs, programs become more complex, and the neat layout of programs becomes more important. Our automatic marking system will check your layout.

The suggested form of layout for "if" statements and "switch" statements is as shown in the above examples, but for your convenience we repeat some of them here. Generally, we indent each set of statements from "{" to "}" about 4 spaces more than the outer code.

```
if ( Radius > 0 ) {
    Result = Radius * Radius;
    cout << "Radius is positive";
} // end if Radius > 0
```

If you prefer, use

```
if ( Radius > 0 )
{
    Result = Radius * Radius;
    cout << "Radius is positive";
} // end if Radius > 0
```

The system will award this full marks too.

```
if ( Money > 0 ) {
    cout << "Deposit.\n";
    Deposits += Money;
} else { // if Money <= 0
    cout << "Withdrawal.\n";
    Withdrawals -= Money;
} // end if else Money > 0

if ( Money > 0 ) {
    cout << "Deposit.\n";
    Deposits += Money;
} else if ( Money > 100 ) {
    cout << "Money greater than 100.\n";
} else { // if Money <= 0
    cout << "Withdrawal.\n";
    Withdrawals -= Money;
} // end if else Money > 0

if ( x > 10 ) {
    cout << "x positive\n";
    if ( y > 0 ) {
        cout << "y positive too\n";
    } else {
        cout << "x is but y isn't\n";
    }
} // end if x positive

switch ( input_value ) {

    case 0 :
// if "input_value" is 0
    do_this();
    break;

    case 3 :
    case 4 :
// if "input_value" is 3 or 4
    do_that();
    break;

    case 7 :
// if "input_value" is 7
    do_the_other();
    break;

    default :
// if "input_value" is anything else
    yet_else();

} // end switch ( input_value )
```

Ceildh marking has the same rules for "{" and "}" and for "(" and ")". If the matching symbols will fit on the same line, all is well. You are permitted

```
Cash = Number * ( CostPerItem + Overhead );
if ( x > 0 ) { cout << "Message\n"; }
```

If the matching pair will not fit on one line, then the closing one must start a line further down, lining up with the start of the line containing the opening one.

```
Cash = Number * (  
    CostPerItem + Overhead // perhaps a longer expression  
);  
if ( x > 0 ) {  
    cout << "A long message\n";  
} // and a comment here  
if (  
    a long test condition &&  
    another condition &&  
    another  
) {  
    a long statement  
} // a terminating comment
```

### Coursework exercise marking

The exercises in this unit are marked partly on dynamic correctness, partly on typographic layout, and sometimes you will see a heading *Features*. For dynamic correctness and typographic layout, earlier comments apply (see the end of unit 2 notes<sup>1</sup> and the Typographic Layout paper<sup>2</sup> and the program layout section above).

The *features* mark is problem specific, and concerns the items which a teacher would look for by eye if marking your programs by hand. For example, if the problem involves converting centimetres to inches, the explicit conversion factor 2.54 should appear as a single global constant, it should NOT appear more than once, it should NOT appear in the program. If you are converting minutes to hours, the constant 60 should appear only once, as a global constant, and it is bad practice to use the constant 59. Always use

```
>= 60
```

rather than

```
> 59
```

when the value which represents the factor which concerns us (the number of minutes in an hour) is 60, not 59.

So, if you loose marks on *features* you must stop and think of all the good programming practices you have been taught, and decide which principle you have broken.

© Eric Foxley December 5, 1996

### References

1. Eric Foxley, *Programming in C++ - Basics*, 1996. LTR Report, Computer Science Dept, Nottingham University
2. Cleveland Gibbon, *Writing Typographically Sound Programs with Ceilidh*, 1995. LTR Report, Computer Science Dept, Nottingham University

## Chapter 4 : Loops

### 4.1. Introduction

In the programs we have written so far, the statements in the program have been executed in sequence, from the start of the program to the end, omitting sections of "if" and "switch" constructs which have not been selected. The real power of computers comes from their ability to execute given sets of statements many times.

The repetition may be required for several reasons.

- (i) We wish to repeat the calculation once for each one of a number of items of data. We may wish to compute the profit for a number of different months of a company's sales. We may wish to compute the stress in each part of the structure of a bridge. We may wish to look at each word in a file of text. In some cases, we may know in advance exactly how many times the calculation will need to be repeated.
- (ii) We may wish to repeat a certain operation until a particular condition is satisfied. For example, we may read numbers from a keyboard, analyse each one and perform some action on it, until a particular value is typed. Another example is that many computer programs keep reading commands and executing them until the user types "quit". In this case, we do not know in advance how many times we will have to go round the loop.
- (iii) We may wish to keep attempting a certain operation (such as obtaining data from a remote computer over a network) until either we succeed (all is well, we have obtained the data, so we proceed to use that data), or until a specified number of attempts have failed. (In this case the whole process must be abandoned.)

To repeat sets of instructions there are three main loop constructs in C++.

### 4.2. "while" loops

A "while" loop repeats a given set of instructions until a given condition holds. The notation is as follows.

```
int Number = 10;

while ( Number >= 0 ) {
    cout << "Number is " << Number << "\n";
    Number--;
} // end while Number >= 0

cout << "Loop ended\n";
```

The condition to be tested is contained in parentheses (round brackets) after the word "while", and the body of the loop is in curly braces after the condition. There is no semi-colon after the closing curly brace.

The sequence of operations in the loop (after the initialisation of the value of "Number" to 10, for example) is

- (i) Test whether "Number >= 0".
- (ii) If it is FALSE, abandon the loop, and continue with the statements after the closing curly brace. In this case, the next statement to be executed would print out the "Loop ended" message.
- (iii) If the result of "Number >= 0" is TRUE, execute the statements in the body of the loop (between the curly braces, in this case first print a message, and then decrement the value of "Number" by 1), and then return to step (i) above.

The test is TRUE to continue the loop, FALSE to leave it. The test occurs before the loop is executed; the loop may not be executed at all if the test result is FALSE the very first time that it is encountered. The pattern of execution is thus

```
test;
```

or

```
test; loop; test;
```

or

```
test; loop; test; loop, test;
```

and so on. The last test on each line must have delivered the result FALSE; earlier tests must have delivered the result TRUE.

When the loop finishes, control passes to the next instruction in the program, following the closing curly brace of the loop.

### Some simple examples of "while" loops

To execute a loop for an integer variable "Number" taking the values 1, 2, ..., 10 you may use any one of the following four possible "while" loop constructions.

Version 1

```
int Number = 1;

while ( Number <= 10 ) {
    cout << Number << "0;
    ....;
    Number++;
}
```

Version 2

```
int Number = 0;

while ( Number < 10 ) {
    Number++;
    cout << Number << "0;
    ....;
}
```

Version 3

```
int Number = 0;

while ( Number++ < 10 ) {
    cout << Number << "0;
    ....;
}
```

Version 4

```
int Number = 0;

while ( ++Number <= 10 ) {
    cout << Number << "0;
    ....;
}
```

In the first two examples, it is immaterial whether you write "Number++;" or "++Number;"; to increment to value of "Number" either of these will do. In the third and fourth examples, you must use "++" as shown. All of these examples are coded in exercise "loops" in which the skeleton is a complete program.

### Some points to observe

1 If what you really want is to execute the loop 10 times, write the condition (as above)

```
Number < 10
```

and **not** as

```
Number <= 9
```

The numeric denotations (actual numeric values) appearing in your program should be exactly the numbers you would talk about in describing what the program is required to do. In this case, the value 9 should not appear. If you are converting seconds to minutes and hours, the value 59 should not appear, only the value 60. The Ceilidh automatic marking system checks program features such as this.

- 2 In general, specific values such as "10" should not appear within the body of your program. They would probably be needed in more than one place, since you may have several loops processing the same number of data items. You should therefore declare them as "const"s at the top of the program as typified by the example

```
const int repeats = 10;

main () {
    ....
    while( Number < repeats ) {
        ....
    } // end the loop
} // end main program
```

The Ceilidh marking system checks that constant values other than, for example "0" and "1" do not appear in the body of the program, and appear exactly once in a "const" declaration. My examples in the notes may not follow this rule.

- 3 In C++ generally you would more likely want to loop not from 1 to 10, but from 0 to 9. All counting in C++ tends to start at zero rather than one. This is a convention that most C++ programmers adopt. A **for** loop will thus start

```
for ( i = 0; i < 10; i++ ) {
```

with a "strictly less than" test.

### More examples of "while" loops

To add together the sequence

$1 + 1/2 + 1/4 + 1/8 + \dots$

until the terms we are adding together are smaller than 0.00001 the program might be as follows.

```
// We need float variables
float Term = 1.0, Total = 0.0;
int Counter = 0;

while ( Term > 0.00001 ) {

// Add the next term to the total
    Total += Term;
// Halve the term
    Term /= 2.0; // or *= 0.5
// Count them
    Counter++;

} // end while Term > 0.00001 loop

cout << "Total " << Total << "\n";
cout << "Number" << Counter << "\n";
```

The value of "Term" is halved each time round the loop; each of these values is added to the total.

You could use

```
Total = Total + Term;
Term = Term / 2.0;
```

instead of

```
Total += Term;
Term /= 2.0;
```

if you feel happier for now.

For debugging we suggest that you put a print statement inside the loop, such as

```
cout << Total << Term << "\n";
```

to verify that the loop is functioning correctly. You can delete this line later, or simply put a `//` at the start of it.

To read positive integer numbers in, terminated by a zero, and print the biggest one.

```
int NextNumber = 1, Biggest = 0;

while ( NextNumber != 0 ) {
    cin >> NextNumber;
    if ( Biggest < NextNumber ) {
        Biggest = NextNumber;
    }
} // end while NextNumber non-zero

cout << "Biggest was " << Biggest << "\n";
```

Note that the terminating zero is still processed by the statements in the second part of the loop. In this example, it will have no effect on the final result. If we were counting how many positive numbers we had read, we would have to be careful not to include the terminating zero.

If we were doing something with each of the numbers in the sequence (perhaps finding the biggest number, or printing the square of each number read in), we would probably not want to perform the operation on the terminating zero; the zero is there as a terminating value rather than as a data item. The operation on each number would need to be preceded by a

```
if ( NextNumber != 0 ) {
```

test. We would then need

```
int NextNumber = 1, Biggest = 0;

while ( NextNumber != 0 ) {
    cin >> NextNumber;
    if ( NextNumber != 0 ) {
        if ( Biggest < NextNumber ) {
            Biggest = NextNumber;
        } // end of "if Biggest < ..."
    } // end of the "if next != 0"
} // end while NextNumber non-zero

cout << "Biggest was " << Biggest << "\n";
```

Note the following possible alternative coding, which has the drawback that the input instruction has to be repeated.

```
int NextNumber, Biggest = 0;

cin >> NextNumber;

while ( NextNumber != 0 ) {
    if ( Biggest < NextNumber ) {
        Biggest = NextNumber;
    }
    cin >> NextNumber;
} // end while NextNumber non-zero

cout << "Biggest was " << Biggest << "\n";
```

Within the loop, we read the next number at the end of the loop. We must read the very first number before we enter the loop. We show a better solution to this problem later in this chapter.

Look at the exercises on averages and Halberstam sequences for ideas.

### 4.3. "do" loops

The "while" loops above performed the test first, and then executed the loop. Sometimes you may wish to test at the end of the loop, after the execution of the statements in the body of the loop (and hence to execute the loop body always at least once). In C++ we use what is referred to as a "do" loop, written as follows.

```
int Number = 1;

do {
    ....;
    Number++;
} while ( Number <= 10 );
```

In this case the value of "Number" would be 1 the first time round the loop, and 10 the last time.

The condition (exactly as in a "while" loop) is still contained in round brackets, and is still TRUE to continue with another execution of the loop body, and FALSE to leave the loop. Remember that there is a semi-colon after the condition, terminating the whole statement. We have one more semi-colon overall than the equivalent "while" loop.

The pattern of execution in this case can be summarised as follows.

```
loop; test
loop; test; loop; test
loop; test; loop; test; loop; test
```

The code in the above programming example could also be written

```
int Number = 1;

do {
    ....;
} while ( ++Number <= 10 );
```

This is the form of combined "increment and test" that most C++ programmers would use. The "++" must, of course, be in front of the "Number" in this case.

It is generally safer to test at the start of a loop; "while" loops are generally safer and more common than "do" loops.

### More examples of "do" loops

To read in positive numbers until a zero is encountered, and print the biggest one.

```
int NextNumber, Biggest = 0;

do {
    cin >> NextNumber;
    if ( Biggest < NextNumber ) {
        Biggest = NextNumber;
    }
} while ( NextNumber != 0 );

cout << "Biggest " << Biggest << "\n";
```

## The use of exit

We may wish to abandon the program from within the body of the loop if some error condition occurs.

```
int NextNumber;

do {
    cin >> NextNumber;
    if ( NextNumber < 0 ) {
        cout << "Error, negative number\n";
        cout << "Value "
             << NextNumber << "\n";
        exit( 0 );
    }
    .. process the number ..
    .. which must be >= 0 ..
} while ( NextNumber > 0 );
```

## Use of the comma operator

You can use the comma operator in the condition and write statements such as

```
int this_one = 10, that_one = 0;

while ( this_one--, that_one++, this_one > that_one ) {
    ....;
} // while this_one > that_one
```

At the start of the loop we decrement "this\_one", then increment "that\_one", and then test whether "this\_one > that\_one" before proceeding with the body of the loop.

It would perhaps be clearer to lay this out as

```
int this_one = 10, that_one = 0;

while (
    this_one--,
    that_one++,
    this_one > that_one
) {
    ....;
} // while this_one > that_one
```

This shows each of the statements and tests on separate lines for clarity.

This type of construction in which the "while" condition involves several statements, effectively gives us a loop which exits in the middle. A simple "while" loop tests and exits at the top, a simple "do ... while" loop tests and exits at the bottom, and a "while" loop with commas tests and exits in the middle of the loop.

## Examples of the comma operator

The cleanest way to, for example, read integer values until a zero is encountered, is to use the comma operator in the following construct.

```
while (
    cin >> NextNumber,
    NextNumber != 0
) {
    ....
} // end while next value non-zero
```

With this program structure, the terminating zero is not processed; this is generally what we require.

The use of the comma in a **while** loop condition can be considered as allowing a new type of loop. Essentially **while** loops are a repeated set of statements with the test condition at the beginning. A **do** loop has the test at the end. A **while** loop with commas has the test in the middle of the loop, preceded by all the statements separated by the commas.

The construct

```
while(
    s1,
    s2,
    test
) {
    s3;
    s4;
}
```

causes sequences such as a minimum of

```
s1; s2
```

then

```
s1; s2; s3; s4; s1; s2
```

then

```
s1; s2; s3; s4; s1; s2; s3; s4; s1; s2
```

Note the layout of the program; we treat parentheses rather like curly braces. If a matching opening and closing parentheses fit onto one line, that is fine. If they do not, then they should line up with each other and be indented in the same way as curly braces. The Ceilidh automatic marking system checks that program layout conforms to this pattern.

In my solutions to the exercises on encryption and word counts I use the comma operator.

#### 4.4. "for" loops

There is a third type of loop in C++, called a "for" loop. It is written as follows.

```
int Counter, Number;

for ( Counter = 0; Counter < 10; Counter++ ) {
    ....;
}

for ( Number = 10; Number > 0; Number-- ) {
    ....;
}

const float e = 0.00001;
float term;

for (
    term = 1.0;
    term > e;
    term *= 0.5
) {
    ....;
}
```

The general form of a "for" loop is

```
for ( initialise; test; execute after loop ) {
    ....;
}
```

The initialise statement is carried out once only, at the start of the very first time that the loop is entered. The test is executed before each execution of the body of the loop, including a test before the very first execution of the loop. The first time will be immediately after the initialisation, and hence there will be perhaps no executions of the loop body if the test fails at this stage. The third expression is a statement executed after every execution of the loop body, before the next test. It typically increments a counter.

The sequence is now

```
init; test;
init; test; loop; incr; test;
init; test; loop; incr; test; loop; incr; test;
```

Again note that the increments in the examples above could be written with the "++" before or after the variable identifier; in this case it does not matter.

## Readability

One of the important advantages of a "for" loop is its readability. All of the essential loop control is grouped together at the top of the loop. We can see at a glance the initial values which are set up, the test to be satisfied for loop exit, and the main variable increments. You should make maximum use of this readability.

The "for" loop could be written as a "while" loop in the form

```
initialise;
....

while ( test ) {
    ....;
    incr;
}
```

In this layout, the loop control is not so clearly seen.

## Defaults

Defaults are obvious; any or all of the three control statements can be omitted. The construct

```
for ( ; ; ) {
    ....;
}
```

gives no initialisation, assumes a TRUE test result, and performs no incrementing.

## The dreaded comma again

You may find the comma operator useful again, particularly in the initialisation and increment parts of the loop control.

```
for (
    this = 10, that = 0;
    this > that;
    this--, that++
) {
    ....;
}
```

Look at the exercises on interest and bicycle gears for examples of programs requiring a "for" loop.

## 4.5. General points on loops

### 4.5.1. Nesting of loops

Loops may, of course, be nested to any depth in any combination as required.

```
int month, year;

for ( year = 1900; year < 2000; year++ ) {

    for ( month = 0; month < 12; month++ ) {
        ....; // execute 1200 times ...
    } // end month loop for each year
}
```

```
    } // end year loop
```

The loop executes with "month" and "year" taking the pairs of values [1900,0], [1900,1], [1900,2], ..., [1900,11], [1901,0], [1901,1], ..., [1901,11], ..., [1999,11] in turn in that order.

#### 4.5.2. The "break" statement

In any of the above loops, the special statement "break" causes the loop to be abandoned, and execution continues following the closing curly brace.

```
while ( i > 0 ) {
    ....;
    if ( j == .... ) {
        break; // abandon the loop
    }
    ....;
} // end of the loop body

cout << "continues here ... \n";
```

The program continues after the end of the loop.

Within a nested loop, "break" causes the innermost loop to be abandoned.

#### 4.5.3. The "continue" statement

In any of the above loops, the statement "continue" causes the rest of the current round of the loop to be skipped, and

- a "while" or "do" loop moves directly to the next test at the head or foot of the loop, respectively; and
- a "for" loop moves to the increment expression, and then to the test.

#### 4.5.4. Example of "break" and "continue"

We wish to write a loop processing integer values which we have read in. If the value we have read is negative, we wish to print an error message and abandon the loop. If the value read is great than 100, we wish to ignore it and continue to the next value in the data. If the value is zero, we wish to terminate the loop.

```
while ( cin >> value, value != 0 ) {

    if ( value < 0 ) {
        cout << "Illegal value\n";
        break;
    } // Abandon the loop

    if ( value > 100 ) {
        cout << "Invalid value\n";
        continue;
    } // Skip to start loop again

    // Process the value read
    // guaranteed between 1 and 100
    ....;
    ....;
} // end while value != 0
```

#### 4.5.5. Comments

It is good practice to comment the end closing curly brace of any loop which extends over more than a few lines. The Ceilidh system expects a comment after every closing brace which appears more than 10 lines from its opening curly brace.

#### 4.5.6. Curly braces

If there is only a single statement in the loop body, the curly braces are not obligatory in C++. It is however recommended that you always use them.

#### 4.5.7. Empty loop bodies

You may find examples of programs in which the complete work of a loop is performed within the parentheses of the test; such a loop may have an empty body.

```
while (
    Total += term,
    term *= 0.5,
    term > 0.00001
) {
    ;
}
```

This is quite legal; it may not be clear to a reader at first glance exactly what is happening.

### 4.6. Other general loop examples

#### 4.6.1. Sum, minimum, and maximum of data

Read positive "float" values from the input, and print their sum, how many numbers were read, the largest value and the smallest value.

```
int count = 0;
float maximum = 0.0, minimum = 1e6;
float Total = 0.0, Number;

cout << "Type positive values ended by zero or negative: ";
while ( cin >> Number, Number > 0 ) {

    // add numbers together
    Total += Number;

    // check minimum so far
    if ( minimum > Number ) {
        minimum = Number;
    }

    // check maximum
    if ( maximum < Number ) {
        maximum = Number;
    }

    // count numbers
    count++;
} // while Number > 0

cout << Total << count << "\n";
cout << minimum << maximum << "\n";
```

Be careful if you wish to print the average of the numbers, and use

```
if ( count > 0 ) {
    cout << "Ave " << Total / count << "\n";
} else {
    cout << "No data to average\n";
} // end if count > 0
```

You must ensure that you can never attempt a division by zero. Wherever there is a division sign in a program you must be able to prove that the denominator cannot be zero.

#### 4.6.2. Sum of squares

To sum the square of all the integer values from 1 squared up to 99 squared.

```
int next_val, sum = 0;

// add squares up to 99 squared
for (
    next_val = 1;
    next_val < 100;
    next_val++
) {
    sum += next_val * next_val;
}

cout << "Sum " << sum << "\n";
```

#### 4.6.3. Decreasing powers of 2

To print (in decimal) the decreasing powers of 2 (1, 1/2, 1/4, 1/8, ...) you would write:

```
int count = 1;
float x = 1.0;

while ( x > 0.0001 ) {
    cout << "Count " << count
        << ", x " << x << "\n";
    x *= 0.5;
    count++;
}
```

It may be clearer to write

```
int count = 1;
float x;

for ( x = 1.0; x > 0.0001; x *= 0.5 ) {
    cout << "Count " << count
        << ", x " << x << "\n";
    count++;
}
```

#### 4.6.4. Read a sentence of text

To read text until a full stop (period) is encountered, and to count the number of occurrences of the letter "e", you could write:

```
char ch;
int count = 0;

while (
    cin >> ch,
    ch != '.'
) {
    if ( ch == 'e' || ch == 'E' ) {
        count++;
    }
} // end while ch != '.'

cout << "There were " << count << " letter e's\n";
```

#### 4.6.5. Increasing and decreasing integers

To read a series of integers, terminated by a zero, and print how many times an integer was larger than its predecessor, and how many times it was smaller. Ignore the terminating zero.

```
int next, previous;
int bigger = 0, smaller = 0;

// Set up the first value
cin >> previous;

// Read data up to a zero
while ( cin >> next, next != 0 ) {

// Was it bigger than the previous?
    if ( next > previous ) {
        bigger++;
    }
// Was it smaller?
    if ( next < previous ) {
        smaller++;
    }
// Store this value for next loop
    previous = next;
}

cout << "Bigger " << bigger << "\n";
cout << "Smaller " << smaller << "\n";
```

#### Coursework exercise marking

The exercises in this unit are marked partly on dynamic correctness, partly on typographic layout (see<sup>1</sup> again) partly on features (see earlier units for details of all of these), and may now have two new sub-headings, *Complexity* and *Structure*.

The complexity marker compares the complexity of the structure of your solution (the number of conditional, the number of loop,s, the depth of the loops, etc) with the teacher's model solution. Your program should not differ too far from the teacher's.

The structure marker looks for program weaknesses such as variables declared and not used, variables with values assigned but not used, and other points such as those revealed by using the verbose compilation option.

© Eric Foxley 1996

#### References

1. Cleveland Gibbon, *Writing Typographically Sound Programs with Ceilidh*, 1995. LTR Report, Computer Science Dept, Nottingham University

## Chapter 5 : Functions

### 5.1. Motivation: the need for functions.

The basic property of functions is that they enable us to break a program down into a number of smaller units. There are several different reasons why programs should be broken down in this way. Different users attach different importance to these reasons; some of the considerations below may be considered irrelevant by some practitioners.

- (i) One approach starts from the problem which the program is intended to solve. Most problems naturally break down into sub-problems. Many methods for problem solving or system analysis rely on this property. A large problem is broken down into sub-problems; each non-trivial sub-problem is broken down into smaller sub-sub-problems ... Each solution of a sub-problem will be represented by an appropriate piece of program called a function. This may be called "top-down" problem analysis.
- (ii) Most problems in the real world of industry or commerce are large and complex. Large problems must be implemented by a team, not an individual. The problem therefore needs to be broken down in a way appropriate for team implementation. This may be done by looking at the different objects involved in the problem and their associated operations; this may be called an "object-oriented" approach to programming.
- (iii) When you write a program, you will often find that similar or identical code is required at several places in the program. The use of a function allows this code to be written just once, and to be called up wherever it is required.
- (iv)

If a solution has been created for a particular sub-problem (an implementation of the solution involving means for storing the related data items, and the means of accessing them in an appropriate way) may be useful in more than one problem. An example is the sorting of a number of items into a particular order; sorting is a basic operation which is needed at many points in many computer problems.

Units which solve frequently occurring sub-problems can therefore be re-used in appropriate places in a variety of larger problems.

- (i) They may be used in order to save effort in re-solving that particular problem, and in re-programming the solution.
- (ii) They may be used in order to produce better quality systems, on the assumption that the solution being re-used was written and tested (by someone else) to a high quality standard when it was originally written.

In this C++ course we teach just the programming techniques needed for producing useful re-usable code; we are not emphasising just WHY you may choose to break down your problem into these particular units. We need what are called "functions" and "structures" in C++; in the next course you will learn techniques for using functions and structures as the basis of an object-oriented approach to the design of programs; they can be used in other design methodologies too.

In this section of the course, we teach the techniques for implementing functions. For those who have used other programming languages, the concept of a "function" in C++ corresponds to a "procedure" or "subroutine" elsewhere.

### 5.2. A simple example of a function

```
// First the declaration and definition
// of two functions
void dothis () {
    cout << "Hello\n";
} // end of dothis

void dothat () {
```

```

        cout << "Goodbye\n";
    } // end of dothat

    // Now the program calls them
    main () {
    // These are the calls
        dothis ();
        ....;
        dothat ();
    } // end of main

```

The parentheses following the function identifier in the function calls in the main program indicate that "dothis" and "dothat" are identifiers representing functions to be executed. The functions may, of course, be called as many times as you wish within the program.

### 5.3. Declaration versus definition

For each function the compiler needs two distinct items of information.

- (i) A declaration, to define (to tell the compiler) what a call of the function will look like, and the identifier to be used.
- (ii) A definition, to define exactly what is to be done (the actions to be executed) whenever the function is called.

This should be compared with the declarations of variables, where again there are two distinct aspects to any declaration, one to inform the compiler about the type and identifier for compile-time, and one to arrange to claim space at run-time.

In the above simple example, declaration and definition are combined.

These two parts (declaration and definition) of a function can be separated in C++ as in the example

```

// declarations
void dothis ();
void dothat ();

main () {
// calls
    dothis ();
    ....;
    dothat ();
} // end main

// definitions
void dothis () {
    cout << "Hello\n";
} // end of dothis

void dothat () {
    cout << "Goodbye\n";
} // end of dothat

```

The first declaration `void dothis();` is to warn the compiler what to expect as calls of the function, for example by introducing the identifier to be used. The compiler will not now be surprised to see the calls of the function when it encounters them in the main program. Although the compiler knows that `dothis` is the identifier of a function, not of a variable, it still requires parentheses after the identifier in function calls in the program.

The later definition `void dothis() { ... }` with executable statements within curly braces will define between those curly braces the code to be executed whenever the function is called. It will involve executable C++ code.

### 5.3.1. The function declaration

The function declaration needs to inform the compiler of

- 1: the type of result to be returned;
  - 2: the identifier of the function;
  - 3: the number and types of any parameters.
- 1 Some functions will return a result, some will not. Functions to compute the square root of a number will return a numeric result for use in the program requesting the square root. A function to print output will not return a numeric value.
  - 2 The identifier of a function follows exactly the same rules as identifiers for variables. It must not clash with the special words of the language.
  - 3 We will see later that we will sometimes need to pass values into functions when we call them. With the "square root" example above, each time we call it we must give the value of the number whose square root we require. The number which we pass to it is sometimes called a *parameter* and sometimes an *argument*.

#### Examples of function declarations

A function to compute the square root of a given value.

```
// identifier "sqrt"  
// takes one double parameter  
// gives double result  
double sqrt ( double );
```

A function to compute the larger of two integer values.

```
// identifier "max"  
// takes two integer parameters,  
// gives integer result  
int max ( int, int );
```

A function to print an error message involving an integer value.

```
// identifier "error"  
// takes integer parameter  
// gives no returned result  
void error ( int );
```

#### Examples of function calls

The above functions would be called from elsewhere (perhaps from the main program, perhaps from another function) by statements such as the following.

Square root would take a "double" parameter and return a double result.

```
double y = sqrt ( 2.0 );  
  
if ( sqrt ( x * 5 ) > 2 ) ....  
  
cout << "Sqrt " << sqrt( 2.0 )  
    << "\n";
```

We would normally expect to use the returned result as a value.

Maximum of two integers

```
int this, that;  
int biggest, mark;  
  
cin >> this >> that;  
  
biggest = max( this, that );  
mark = max( this, 40 );
```

```

cout << "Larger value is "
    << max ( this, that );

non_negative = max( this, 0 );

```

Reporting an error

```

if ( n < 0 ) {
    error ( 5 ); // no result
}
if ( n > 100 ) {
    error ( 6 ); // no result
}

```

### The syntax of a function declaration

A function declaration consists of

```

<returned type> <identifier> (
    <param type>,
    <param type>,
    ....
); // Note the semicolon

```

The returned type may be any ordinary C++ type such as `int` , `float` and so on, or the type `void` if no value is being returned to the main program.

In some older programming languages, the word "function" was used only when a result was to be delivered; the word "procedure" or "subroutine" was used if there was no result to deliver. In C++ all such objects are called functions.

#### 5.3.2. The function definition

The function definition needs to inform the compiler of the actions to be taken when the function is called. In the declaration we described only the types of the parameters. The compiler needed this information to enable it to handle the parameters correctly wherever the function is called. In order to describe the actions to be taken when the function is called, we need to identify each of the parameters, so that we can refer to them and use them from within the code which forms the body of the function.

#### The "max" function

We want a function to deliver the larger of the values of the two values given as parameters.

```

// deliver the larger of the 2 params
int max ( int i, int j ) {
    if ( i > j ) {
        return i;
    }
    return j;
} // end max

```

The "return" keyword acts both to specify the particular value to be returned by the call of the function, and to cause dynamic exit from the function.

If the function returns type `void` we leave the function using the statement

```
return;
```

with no value specified. If the function returns a non-void type, the `return` must be followed by a value which the compiler can force into the required type.

Notice that we do not need an "else" statement in this example because of the dynamic exit caused by the `return i;` statement.

## The "error" function

We want a function to print an error message and then abandon the program.

```
void error ( int errno ) {
    cout << "Error number " << errno
        << " has occurred\n";
    cout << "Program abandoned\n";
    exit ( 1 );
} // end error
```

In this example, the "exit" causes the whole program to terminate; if we merely wished to report the error and continue the program, we would use a

```
return;
```

statement with no value given.

```
void error ( int errno ) {
    cout << "Error number " << errno
        << " has occurred\n";
    cout << "Program continues\n";
    return;
} // end error
```

If the function has its delivered type declared as `void` its code does not return values, and the function is left by using

```
return;
```

## The *exit* function

Convention with the `exit` function is that

```
exit( 0 ); // indicates normal exit
exit( 1 ); // indicates error exit
exit( 2 ); // indicates error exit
```

The UNIX shell can use the value returned by a terminating program to determine whether it terminated successfully or not.

The Ceilidh system will, in some exercises, require you to use the "exit" function to indicate errors in the data.

## The "sqrt" function

We need to calculate the square root of the given parameter value, and return it. Please forgive the crude method!

```
double sqrt( double x ) {
    // code not guaranteed, I'm in a hurry!
    const double accuracy = 0.00000001;
    double low = 1, high = x, mid = x / 2;
    if ( x < 0 ) {
        error( 5 ); // error if negative
    }
    if ( x < 1 ) {
        low = 0;
        high = 1;
    }
    while( high - low > accuracy ) {
        if ( mid * mid > x ) {
            high = mid;
        } else {
            low = mid;
        } // end if guess too large
        mid = ( low + high ) / 2;
    } // end while accuracy not reached
}
```

```

    return mid;
} // end sqrt

```

## The syntax of a function definition

A definition consists of

```

<returned type> <identifier> (
    <param type> <param ident>,
    <param type> <param ident>,
    ....
) { // No semi-colon after the ")"
    ....; // body
} // usually an end comment here

```

### 5.4. Returned types

The compiler needs to know the type of the value to be returned by the function so that it can be treated correctly in the call of the function. If no value is to be returned (cf "error" or "dothis" above) then the return type is given as "void".

The particular value to be returned is specified by the line

```
return <value>;
```

With a `void` function write simply

```
return;
```

The compiler will do type conversions where necessary. If the function declaration says that it returns a `float` and the function includes `return 1;` then the compiler knows to sort this out.

### 5.5. Parameter types

The parameter type sequence in the declaration and the definition must agree. The compiler will make the necessary type conversions to the parameters and to the delivered result in a call of the function. Again, the compiler will perform type conversions if the function declaration specifies a `float` parameter, and if the call includes an *actual* integer parameter.

The parameters in the function definition are called the *formal parameters*. The parameter values supplied in calls of the function are called *actual parameters*.

### 5.6. Default parameters

A function defined with, say, 3 parameters can be called with less than three parameters if default values have been set. Default values are set using what looks like initialised declarations for the formal parameters.

A possible example might be

```

int max(
    int i = -1000000,
    int j = -1000000,
    int k = -1000000
) {
    if ( i >= j ) {
        if ( i >= k ) {
            return i;
        } else {
            // i >= j && i < k
            return k;
        }
    } else {
        // i < j
        if ( j >= k ) {
            return j;
        }
    }
}

```

```

    else
// i < j && j < k
    return k;
} // end if else i >= j
} // end max

```

This could now be called as

```
m = max ( p, q, r );
```

to find the maximum of the three values `p`, `q`, `r` or as

```
m = max( p, q );
```

in which case the last (third) parameter in the definition assumes its default value; we effectively get the maximum of the first two parameters.

A drawback of using this technique is that errors in giving the wrong number of parameters to a function may not be detected by the compiler.

Another example could be written as in

```

float volume (
    float height = 1.0,
    float width = 1.0,
    float depth = 1.0
) {
    return height * width * depth;
} // end volume

```

The function can then be called as in

```

float x;
x = volume ( 2.3, 4.5, 6.7 );
// returns 2.3 × 4.5 × 6.7

x = volume ( 2.3, 4.5 );
// returns 2.3 × 4.5

x = volume ( 2.3 );
// returns 2.3

x = volume ();
// returns 1.0

```

## 5.7. Function overloading

You can use the same function identifier for more than one function, providing that the compiler can determine the particular one to be chosen by the types (and number) of its parameters.

```

int square ( int i ) {
// int squared is an int
    return i * i;
}

double square ( double x ) {
// double squared is a double
    return x * x;
}

double square ( double x, double height ) {
// return the volume of a square prism
    return x * x * height;
}

```

The calls might be

```
i = square( 3 );
```

```

double z;
if ( square( z ) > 3.14159 ) { ....
vol = square( z, 1.27 );

```

The compiler would then look at each call of the function, and determine from the types of the parameters which of the definitions was to be invoked.

The strict rules to define what parameter types can be distinguished are very complicated; the best advice is just to make the difference very obvious!

## 5.8. Program structure

The main program and function definitions can appear in any order in your program. It is normally clearest to read and understand if the main program appears before the function definitions. Reading the main program gives an overview of the sequence of operations.

Function declarations **MUST** appear before the function is called. They are usually put together before the main program.

## 5.9. Local variables

We can declare local variables at the start of our function code if we require additional variables for computations within the function. The declarations appear after the opening curly brace, just as they do in the main program.

If a local identifier clashes with a global constant, or with the name of another function, then that global constant or other function becomes inaccessible within this function. The local object will be referred to by the identifier.

```

const float pi = 3.1416;

int fred( int );

main() {
    ....
    ... pi ... // global constant
} // end main

int fred( int i ) {
    int pi; // local variable
    ....
    pi = ...; // local variable
} // end fred

```

## 5.10. Parameters called by value

When a function is called, the values of the (actual) parameters at the point where the function is called are calculated, and passed to the function definition for execution. Within the function, the parameters act like variables, initialised to the value of the corresponding actual parameter at the call, and their values can be changed by ordinary assignment.

Thus if a function definition is

```

int fred( int number ) {
    ....;
    number = number + p;
    ....;
} // end function fred

```

the identifier "number" can be considered as a local variable to the function. Changing its value inside the function as shown in the above example has no effect on the outside world.

```

int count( int value ) {
    ....;
    for ( ; value > 0; value-- ) {

```

```

        ....
    } // end for loop executed "value" times
    ....;
} // end function fred

```

If the call of the function is

```

int counter = ...;

fred( counter );

```

the value of the variable `counter` in the calling program will not be changed. We could also call the function by

```

fred( 23 );

```

which would pass the value 23 to the formal parameter. This way of passing parameters is referred to as "passing by value".

There are some cases where we wish to change the value of a variable in the world from which the program is called; the use of a "swap" function to interchange two integer variable values is an example. We wish to pass to the function the names of two variables as parameters, and require the function to interchange the values in the two named variables. These are variables of the calling program, not of the function. We would want to call the function as in

```

int p1, p2;
...;
swap( p1, p2 ) // swap the values

```

If we declared it as

```

void swap ( int p, int q ) {
    int temp = p;
    p = q;
    q = temp;
} // end swap

```

then the above call would not have any effect on the world outside the function. It would interchange only the values of the internal variables "p" and "q", not the program variables "p1" and "p2".

To have the effect we require we must change the declaration to

```

void swap ( int &, int & );

```

and the first line of the definition to

```

void swap ( int & p, int & q ) {

```

If the function parameters are shown just as normal, without the "&" symbol, the only effect of the function is to interchange the two "local" variable values. To ensure that the actual variables whose identifiers are passed as parameters to the call of the function are swapped, the "&" sign shown in the example is necessary. The call will now interchange the values of the main program variables named as arguments. This is referred to as a parameters being "called by address" or "called by name" or "called by reference".

A mathematical example might be a function to solve quadratic equations (non-mathematicians don't panic ...). The input to the function is three float values ("a", "b" and "c"), and the output is two float values (the roots, "x" and "y"). We cannot deliver *two* values directly as the result of a function. Instead we use two address parameters, so that the function specification is

```

void quadratic(
    float, float, float,
    float &, float &
);

```

and a call might be

```

float x1, x2;
...;

```

```
quadratic( 1.2, 3.4, 5.6, x1, x2 );
```

The values of roots will appear in "x1" and "x2".

The definition might be

```
void quadratic (
    float a, float b, float c,
    float & x, float & y
) {
    float d = sqrt( b * b - 4 * a * c );
    x = ( -b + d ) / ( 4 * a * c );
    y = ( -b - d ) / ( 4 * a * c );
} // end quadratic
```

A less mathematical example; we are given a number of bicycle wheel rims and spokes in stock, and as a global constant the number of spokes needed to make one wheel. How many wheels can we make, how many rims and spokes will be left over? There are three computed values to pass back to the calling program. We choose in the example below to return as the result of the function the number of wheels it is possible to make, and to store in two address parameters the numbers of remaining rims and spokes.

```
int wheels(
    int rims0, int spokes0,
    int & rims1, int & spokes1
) {
    int made = rims0;
    if ( spokes0 < rims0 * spokes_per_rim ) {
        made = spokes0 / spokes_per_rim;
    }
    rims1 = rims0 - made;
    spokes1 = spokes0 - made * spokes_per_rim;
    return made;
} // end wheels
```

A call might be

```
int rim, spoke;
int r, s, number;

// 10 rims and 150 spokes
number = wheels( 10, 150, rim, spoke );
// left-overs in "rim", "spoke"

// "r" rims and "s" spokes
number = wheels( r, s, r, s );
// left-overs in "r" and "s"
```

We could instead have had a `void` function, and pass three values back through address parameters.

### Warning 1

Do not use the "&" sign unless it is really needed. The fact that most parameters are "called by value" is a safety feature, to ensure that a function cannot accidentally change the values of program variables unintentionally.

### Warning 2

If you have nominated a parameter to be called by address, it does then not make sense to substitute a specific value in a call, as in

```
swap( i, 3 );
```

or

```
swap( i + 1, .... );
```

since the object "3" or "i+1" is not something which can be swapped.

### 5.11. Jargon reminder

The parameters as specified at the start of the function definition are referred to as "formal parameters".

The parameters substituted in any particular call of the function are referred to as "actual parameters".

### 5.12. A complete example

The complete program with function declaration, main program and function definition will look roughly as follows. We use an example with the Halberstam function.

```
// Program example
// Halberstam function being declared
// then used
// then defined

#include <stream.h>

// Declare the function
int halberstam( int );

// The main program
main()
{
    int numb = 0, calcs = 0;

// Loop reading integers until we meet a zero
    while (
        cout << "Enter the number now please: ",
        cin >> numb,
        numb != 0
    ) {

// This is the call
        calcs = halberstam( numb );
        cout << "numb " << numb << " result " << calcs << "0";

    } // end while read number > 0

} // end main program

// Now for the definition
int halberstam( int numb ) {
    int calcs = 0;

// Error exit
    if ( numb <= 0 ) {
        return -1;
    }

// Loop counting how many times
    while ( numb != 1 ) {
        if ( numb % 2 ) { // number is odd
            numb *= 3;
            numb++;
        } else { // number is even
            numb /= 2;
        }
        calcs++;
    } // End of while loop
    return calcs;
} // end function halberstam
```

## Note

In general functions will perform operations on data, and not perform their own input output except

- (i) if the main purpose of the function is for input/output; or
- (ii) to print error messages.

## 5.13. Examples

### 5.13.1. Circle radii and areas

We show here two functions taking `float` parameters, and giving `float` results. The first takes a radius length, and delivers the area of a circle of that radius. The second takes an area, and delivers the radius of a circle with that area.

```
const float pi = 3.14159;

float area ( float radius ) {
    return pi * radius * radius;
} // end radius to area

float radius ( float area ) {
    if ( area < 0 ) {
        cerr << "Sqrt invalid param "
              << area << "\n";
        // You might choose to exit here with
        // exit( 1 );
        return -1;
    }

    return sqrt ( area / pi );
} // end area to radius
```

Calls of these functions might be

```
float a1, r1, a2, r2;

cin >> r1;
a1 = area( r1 );

cin >> a2;
r2 = radius( a2 );

if ( r2 < 0 ) { ...
```

### 5.13.2. Summation

This function sums the series

$$1 + x + x^2/2 + x^3/2.3 + x^4/2.3.4 + \dots$$

for a given value of "x". We keep adding terms until we reach a term whose value is less than 0.001.

```
float expl( float x ) {
    // sum the exponential series
    float total = 0;
    float term = 1;
    int count = 0;

    // now loop
    while ( term > 0.001 ) {
        total += term;
        count++;
        term *= x/count;
    }
}
```

```

// return the total
return total;

} // end expl

```

Calls of this function might be

```
cout << "Value is " << expl( 1 );
```

### 5.13.3. The OK function

We require a function to print out a given message as a question, and return TRUE if the user replies "y" and FALSE otherwise. The type of a message (a string of text contained within double quotes) is `char *` (the reasons for this will appear later), and the definition of the function might be:

```

int ok( char * message ) {
    char ch;
    cout << message << " [Type y or n]? ";
    cin >> ch;
    return ch == 'y';
}

```

In a real situation, the function body might repeat the question until the given answer is "y" or "n". In the above example, we respond with TRUE if the answer is "y" and FALSE otherwise.

Note that we do not need to say

```

if ( ch == 'y' ) return 1;
return 0;

```

We can return the comparison result directly.

Calls of this function might be:

```

if ( ok( "Was that correct" ) ) {
    ....;
} else {
    ....;
}

if (
    ok ( "Overwrite the file" )
    && ok ( "Are you sure" )
) {
    ....;
}

```

Observe the subtlety of the last example. The program first asks

Overwrite the file [Type y or n]?

and, because the "&&" operator is lazy, only if the reply to the first question is "y" asks for confirmation

Are you sure [Type y or n]?

If the reply to the first question is "n", the second question is not asked.

### 5.14. Mathematical functions

A number of standard mathematical functions are available. For details try

```

man exp
man sqrt
man trig

```

or

```
ls /usr/man/man3/*.3m
```

### **5.15. Typographic layout**

Your program layout should now be as typified by the above examples. The function declarations are near the top, all starting at the left hand edge.

The function definitions follow the program. The first line and the final "}" of each definition start in the first column. The internal statements are indented, the final "}"s are followed by a comment on the same line.

© Eric Foxley 1994

## Chapter 6 : Miscellany

We now fill in a variety of missing details before moving to further fundamentally new facilities of C++. Some of these extra details are just included for completeness, and will not be studied extensively.

### 6.1. Program layout

It is vitally important that programs are laid out carefully. One of the most important considerations in programming is the readability of the program. In industry and commerce one would want a readable style, and each company will have its own style so that programs written in one department can be read by users in another. There are programs ("program formatters" or "beautifiers") which can lay out a given program to comply with certain rules. Such programs cannot, of course, insert comments, or ensure that identifiers are meaningful, but they can ensure that the indentation is correct, and that there is a reasonable amount of spacing.

There are also "program editors" or "structure editors" which enforce program layout as you create the program. They operate in different ways, I may describe some in the lecture.

A number of ordinary text editors such as "vi" and "emacs" have optional facilities for automatically laying out programs.

When marking programs, we will look for

- neat and regular indentation
- related to
  - braces { }
  - parentheses ( )
  - and square brackets [ ]
- sufficient comments for readability
- comments after every closing curly brace
- the use of meaningful identifiers
- plenty of white space within statements
- and some blank lines

In the content of the program, we will look for

- helpful output with words around it
- detect errors where possible
- appropriate use of functions

For the moment (see later in this document) we also recommend

- no global variables
- global const declarations where possible

### 6.2. Basic types

A slightly longer list of basic variable types in C++ is given below, together with their sizes in bytes on some implementations.

type	PDP	68000	VAXVMS
char	1	1	1
int	2	4	4
long int	4	8	4
short int	2	2	2
float	4	4	4
double	8	8	8
unsigned int	2		

unsigned long	4			
unsigned short	2			
unsigned char	1			
(pointers	2	4	4	

The "unsigned" types are exactly what they say. They are used to store positive quantities, and can (in general) store a quantity twice as large as the corresponding signed type. You will be aware of the techniques used for the binary representations of integers. A 2-byte "int" variable would be able to store values ranging from  $-2^{15}$  (approx -30,000) to  $2^{15}-1$  (approx 30,000). A 2-byte "unsigned int" will be able to store values from 0 to  $2^{16}-1$  (approx 60,000). Many quantities in real applications are known to be positive (the number of items in stock, the address of a block on disc, the length of a queue) so that unsigned variables are ideal.

### sizeof

The compile-time operator `sizeof` gives the size of a supplied object in bytes, as in the example

```
main() {
    int i;
    cout << "Size is an int is "
          << sizeof i << "\n";
} // end main
```

By "compile-time" we mean that the calculated value is substituted into the program at the appropriate point by the compiler, not calculated at run-time.

Further, if in a program we had the expression

```
20 * sizeof i
```

(an operator with fixed values as its operands) we would expect the compiler to perform the multiplication once only at compile time; there would be no multiplications in the run-time of the program.

### 6.3. Lazy programming

In C++ you may use a numeric value wherever a logical (Boolean) condition would be expected, because "zero" is always interpreted as FALSE, and "non-zero" as TRUE. If `n` is an "int", then the code

```
if ( n ) ...

while ( ! n ) ...
```

is exactly equivalent to

```
if ( n != 0 ) ...

while ( n == 0 ) ...
```

You will often see this terse version in C++ code. The two versions should be exactly equivalent when compiled.

### 6.4. Lazy operators

The word "lazy" has a proper meaning in the compilation of programs. The "and" and "or" logical operators ("&&" and "||") are lazy, i.e. the operands are evaluated from left-to-right, and stop as soon as the result is determined.

```
i >= 0 && i < 10 && funct( i ) == 0
// evaluated left to right "lazy"
// stop as soon as "false" is encountered

ok( "Overwrite file?" ) && ok( "Sure?" )
// Second prompt only if first ok
```

```
x < 0 || x >= 10 || ...
```

The "and" operator stops as soon as a false expression is encountered; the "or" stops as soon as a true expression is encountered.

## 6.5. Bit-wise operators (keenies only)

This facility is just for the freaks, all normal mortals can ignore it. The following operators act upon a variable considered as a pattern of bits.

```
a << 2      // shifted left
a >> 3      // shifted right
a & 0177   // bit-by-bit AND
a | 0100   // bit-by-bit OR
a ^ 0100   // bit-by-bit excl OR
~ a       // bit-by-bit negation
```

Such operations are normally performed on "unsigned" objects.

## 6.6. Declarations

Declarations within a main program or function in C programs had to be placed at the start of the program/function, before any executable statements. That is the pattern I have followed in all my examples so far. This restriction has been relaxed in C++.

We refer to any section of the program or of a function from an opening curly bracket to the corresponding closing curly bracket as a "block".

C++ allows declarations to occur (almost) anywhere; each declaration is valid until the closing curly bracket matching the most recent previous opening one, i.e. to the end of the current block.

For example

```
main() {
// start outer block
  int counter;
  cin >> counter;

  float recip = 1.0 / counter;
  float total = 0.0;

  while ( counter > 0 ) {
// start new block
    float x = recip / counter;
    total += x;
    cout << counter << x << total << "\n";
    counter--;
// x exists up to here
  } // end while

  cout << total;
// other variables exist up to here
} // end main
```

Each declaration is valid from the declaration until the closing curly bracket ending the block in which it was declared. By the word "valid" above we mean two things.

- (i) The program can refer to the variable within this part of the code; and
- (ii) the variable will exist at run-time until the enclosing block is left.

If an identifier is declared which is the same as an identifier in any outer enclosing block, the outer declaration becomes temporarily inaccessible until after the end of the inner block.

The declarations of "recip" and "x" above can be made "const" declarations, since the variable is not reassigned a different value. The declaration of "total" cannot be made constant, since its value is reassigned each time round the loop. If a value is not going to be reassigned, it is good practice for safety to use a

"const" declaration to avoid it being accidentally overwritten. If you did this, you cannot accidentally write

```
if ( x = value )
```

since assignment to "x" would not be permitted.

## Global declarations

You can also declare variables outside the main program and all of the functions. Such declarations are valid from the declaration right through to the end of the file. They can be referred to from anywhere in program or functions from the declaration to the end of the file. They will exist for the whole duration of the program.

If any identifier is re-declared in the program or a function (or in an inner block) the outer declaration again becomes inaccessible for that block.

Global variables are a convenient but dangerous way of communicating between functions, or between a function and the main program. For the moment, we recommend against their use. All information should be passed into and out of functions through their parameters.

## 6.7. Goto's and assignments considered harmful

This paragraph can be ignored by those who have not done programming before, and/or those who do not know what the word "GOTO" means (I certainly haven't mentioned it); you should skip to the following paragraph.

The preceding paragraph brings us to an interesting challenge. It has long been accepted that, generally speaking, a "goto" type of instruction is messy. I have not mentioned that C++ has a "goto" statement, and even if you realise that it has, you are not permitted to use it! There may however be a few occasions where it is helpful (it occurs several times in the huge amount of UNIX source code) but generally it encourages messy and tangled programming. The appearance of "goto"s generally indicates a poorly thought-out program. The use of "goto"s certainly makes the theoretical analysis of programs more difficult. For more discussion see Dijkstra's letter to the Editor of the Communications of the ACM, published March 1968, entitled "GO TO Statement Considered Harmful".

It spawned other articles such as "The Case Against the GOTO" by W A Wulf, (Proceedings of the 25th National ACM Conference, 1972, pp 791--), and "The Case for the GOTO" by M E Hopkins at the same conference. Most modern computer languages are designed to minimise the use of GOTO.

All readers start here! There are those in the real world who suggest that the use of assignment is equally harmful. You can perhaps appreciate that assignment makes proving things about programs more difficult, since the value of a given variable may change as we proceed.

The C++ feature that enables us to put declarations anywhere, and to use constant declarations, encourages a healthy train of thought to write complete programs using only constant declarations. The only other form of assignment permitted will be the use of "return" in a function, and, of course, the initialisation of constants is really a form of assignment.

The "Halberstam Function" of an integer value is defined as follows:

```
Given a positive integer value, generate the sequence
  If it is even, half it.
  Otherwise multiply by 3 and add 1.
```

The Halberstam value of an integer value "n" is the number of times this operation needs to be repeated before the value 1 (one) is reached. It is guaranteed that you will eventually reach the value 1. If we start with the value 3, for example, the sequence goes 10, 5, 16, 8, 4, 2, 1; 7 steps. If we start with the value 7, the sequence goes 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1; 16 steps.

The old-fashioned solution declared as a function might typically be something like this:

```
int halberstam ( int n ) {
    int value = n, count = 0;
    while ( value != 1 ) {
        if ( value % 2 == 0 ) { // Even
```

```

        value = value / 2;
    } else { // "value" is odd
        value = 3 * value + 1;
    }
// Count the times round the loop
    count++;
} // while value not yet 1
return count;
} // end halberstam

```

This has two internal variables whose value is re-assigned each time round the loop.

A cleaner way might be on the following lines:

```

int halberstam( int n ) {
    if ( n <= 1 ) {
        return 0;
    }
    if ( n % 2 == 0 ) {
        return 1 + halberstam( n / 2 );
    }
    return 1 + halberstam( 3 * n + 1 );
} // end halberstam

```

The above leads naturally to the concept of "recursion". "Recursion" occurs when you use a function within its own definition. Obviously you must be careful not to let the definition recurse indefinitely (infinitely), so a recursive definition will usually have an "escape clause" such as the `if ( n <= 1 )` ... above. We may study the ideas of recursion in more detail later.

## 6.8. Error output

In ordinary UNIX commands, there are two distinct output streams for each command, the "standard output" and the "standard error" streams. When you divert output using redirection to a file or down a pipe, the error stream is not diverted; this ensures that error messages still come to the screen, to make sure the user is aware of the error.

In C++ programs, any output sent to "cout" is standard output, any sent to "cerr" is error output.

```

int total = 0, number;
while (
    cin >> number,
    number != 0
) {
    if ( number < 0 ) {
        cerr << "Invalid data "
             << number << "\n";
        continue;
    }
    total += number;
} // end while number non zero

cout << "Total of valid data "
     << total << "\n";

```

Both of "cout" and "cerr" are declared in the standard "iostream.h" (or is it "stream.h"?) file.

Some later Ceilidh exercises may require that you print certain messages on the error stream, and other on the standard output. Don't forget to end "cerr" messages with a newline character. Output (on "cout" or "cerr") may not be printed if the terminating newline is missing.

## 6.9. Strings

Although we have not yet studied enough about arrays (collections of many objects of identical type, vectors) and pointers (addresses of objects) it will be useful to describe briefly how to use strings. When we study arrays later, all of this should make more sense.

## Declaration

The declaration

```
char name[] = "Eric Foxley";
```

gives an object called "name" initialised to the value given between quotes. You might print it using its identifier as in

```
cout << "My name is " << name << "\n";
```

If you declare an uninitialised string you must give it a size as in

```
char name[50];
```

which allows for a string of maximum length 49 characters; the fiftieth is used to contain a terminating null character. You can now read a string in using

```
cin >> name;
```

This will read characters from the input into the named string until white space (a space or tab or newline) is encountered. No checks are made to confirm that the length of the declared string is sufficient, so beware of chaos if the string read in is longer than the declared length.

## The string handling library

There are library functions to do most of the things you might wish to do with strings. A few of the most useful are as follows.

strlen

This gives the length of a string given as parameter.

```
int strlen( char [] );

char name[50];
cin >> name;
cout << "length " << strlen(name) << "\n";
```

strcmp

The function "strcmp" is for comparing strings. It is given two strings as arguments, and returns zero if they are identical. If the strings are not identical, it returns a positive value if the second string is alphabetically before the first, and negative if it is after it.

```
int strcmp( char[] , char[] );

char command[50];

while(
  cout << "Type a command: ",
  cin >> command,
  1
) {

  if (
    strcmp( command, "edit" ) == 0
  ) {
    ... call editor ...
  } else if (
    strcmp( command, "print" ) == 0
  ) {
    ... call printer ...
  } else if (
    strcmp( command, "quit" ) == 0
  ) {
    cout << "Bye!\n";
    exit( 0 );
  }
}
```

```
}
```

If you wish just to look at the leading characters of the string, use

```
strncmp
```

to limit the number of characters which are compared as in the following example.

```
int strncmp( char[], char[], int );

if (
    strncmp( command, "edit", 2 ) == 0
) {
    ... call editor ...
} else if (
    strncmp( command, "quit", 1 ) == 0
) {
    ....;
}
```

The additional parameter limits the number of characters which will be compared. There is no need to put the full string into the program, but it helps readability.

**strcpy**

This copies one string to another, as in

```
char one[50], two[50];
cin >> one
// copy string "one" into "two"
strcpy( two, one );
```

It copies from the second parameter to the first.

**strcat**

This is similar to **strcpy** but concatenates the right hand parameter string to the end of whatever is already in the first parameter. The strings must all be declared long enough to store the resulting strings.

For details of all such string functions, type

```
man string
```

at your terminal.

The compiler needs to know the declarations (as opposed to the definitions) of any string library functions which you use. Instead of inserting these yourself, you should add the line

```
#include <string.h>
```

at the head of your program.

### **Strings as function parameters**

To pass strings to a function, the function declaration might be

```
void error( char[], int );
```

and the corresponding definition

```
void error( char message[], int number ) {
    cerr << "Error " << message
        << " number " << number << "0";
    exit( 1 );
}
```

This function would be called as

```
error( "Negative number", 5 );
```

The parameter type given in the declaration is "char[]", and in the definition the parameter is specified as "char <name> []".

## 6.10. The "system" function

For keenies only, and please do not abuse it! You can use the `system` function in C++ to execute any UNIX command while your program is executing. If you write

```
system( "who" );
```

the program will execute the `who` command at this point just as if you had typed it from the keyboard.

The parameter can, of course, be the identifier of a string variable in which you have stored an appropriate command.

## 6.11. File inclusion

This is a feature which we will need on the near future.

The "`#include`" line which you already use is a general facility. The complete contents of the named file are substituted into your program instead of the "`#include`" line.

If the filename given is enclosed in "<" and ">" signs, the named file is searched for in a special system directory associated with the compiler and its function libraries. If the filename is enclosed in quote symbols, as in

```
#include "myheader.h"
```

then the named file is found in your directory and inserted in the text at this point in your program. The name can include a full path, as in

```
#include "/staff/eric/ceilidh.h"
```

## 6.12. Formatted output

To print your output more neatly, you may wish to use formatted output techniques. This corresponds to the use of "`printf`" in C. The function "`form`" takes as arguments a format string, and the values of expressions to be inserted in it. It is simplest to explain by example.

To print decimal values:

```
int i, j;

cout << form( "%d times %d is %d\n", i, j, i*j );
```

The values following the format string are picked up and inserted for each "%" in order in the format. The rest of the format string is printed as it stands. The letter after the "%" specifies the type of format required.

To print floating values:

```
int i;
float x;

cout << form( "%d times %f is %f\n", i, x, i*x );
```

To print characters as such:

```
char ch;

cout << form( "Char %c has ASCII code %d\n", ch, ch );
```

To print an integral value in octal:

```
int i;

cout << form( "%d in octal is %o\n", i, i );
```

Possible other field definitions are

```
%3d // decimal, allow for 3 digits
%03d // zero fill on left
```

```
%-5d // allow 5 columns but left justify
%10.3f // float, width 10, 3 DPs
%s // string, see later
%30s // string, 30 spaces, right justified
%-30s // string, 30 spaces, left justified
```

Further examples:

```
cout << form(
    "hours %d\nmins %d\nsecs %d\n",
    h, m, s );
```

To print the time nicely:

```
cout << form( "time %2d:%02d:%02d",
    h, m, s );
```

© Eric Foxley December 6, 1996

## Chapter 7 : Arrays and structures

So far we have declared variables one at a time. We will often need many variables in a program, and "array"s are a technique for declaring many variables of the same type in one statement, and for being able to consider the whole collection as a single object for appropriate operations.

A "structure" is a means for combining several related items which may be of different types as a single object, while still being able to refer to the separate individual components if we wish to.

### 7.1. Arrays

#### 7.1.1. Examples of the requirement for arrays

We are analysing the annual rainfall over a period of 90 years; we require 90 `float` variables to store the information.

We are studying a piece of English text; we require 10000 `char` variables to store the characters.

We are recording the numbers of students in each department of the university; we need (depending on how many departments there are) 100 `int` variables.

#### 7.1.2. Declaration of arrays

The above examples would be declared as

```
// 90 float variables
float annual_rain[ 90 ];
// 10000 char variables
char text[ 10000 ];
// 100 int variables
int stud_nos[ 100 ];
```

The number of elements requested in the declaration must be fixed at compile time.

#### 7.1.3. Calls of array elements

To get at a particular variable (element) in an array, we write the identifier of the array followed by the subscript in square brackets. In C++ if we declare

```
float annual_rain[ 90 ];
```

then the subscripts run from 0 (zero) to 89 inclusive; this gives us the required number (90) of variables.

To access the 23rd of these variables, we would put the subscript in square brackets after the array identifier, and use

```
... annual_rain[ 22 ] ...
```

The subscript can be any integer expression.

To store a value in this variable we may use

```
annual_rain[ 22 ] = ....;
```

or perhaps

```
cin >> annual_rain[ 22 ];
```

and to use the value stored there

```
if ( annual_rain[ 22 ] > minimum ) {
    total = total + annual_rain[ 22 ];
}
```

The real use of arrays is when we refer to each element of the array not by a specific constant subscript, but access all elements in turn or choose a particular element dynamically. To access the variables in turn, for example to read 90 values from the data into the 90 locations, we would use

```

int year;
for( year = 0; year < 90; year++ ) {
// Variable "year" goes from 0 to 89
  cin >> annual_rain[ year ];
} // for year

```

The 90 values would have to appear in the data stream in the correct order, separated by "white space", i.e. spaces, tabs or newlines.

Having read the 90 values in, we may wish to calculate the total and average rainfall over these 90 years. For this we would use

```

float total = 0;
for( year = 0; year < 90; year++ ) {
// Each array element is a "float"
  total += annual_rain[ year ];
} // for year
cout << "Total " << total << "\n";
cout << "Average " << total / 90 << "\n";

```

In the above examples, the constant value 90 keeps appearing. We should really take this out as a constant, so that the actual value appears at only one point in the program. The program now becomes as follows.

```

// Global constant
const int duration = 90;
main() {
  float annual_rain[ duration ];
  int year;
  for( year = 0; year < duration; year++ ) {
    cin >> annual_rain[ year ];
  } // for year
  float total = 0;
  for( year = 0; year < duration; year++ ) {
    total += annual_rain[ year ];
  } // for year
  cout << "Total " << total << "\n";
  cout << "Avege " << total / duration << "\n";
} // end main

```

We will not now have problems when the number of years duration of the rainfall analysis needs to be changed; all values of the duration will change in step together when we change the value of the global constant. If we had not done this, we might have forgotten to change some of the occurrences of "90" to another value.

Note that the value of the "const" denoting the number of elements in the array must be a genuine integer constant! When the compiler is digesting your program must know exactly how many array elements are required. The constant must not be one which is determined as the result of some calculation while the program is running.

Note also the typical C++ loop, starting at zero, and limited by "counter strictly less than number of elements". This gives us the range of values 0, ..., 89 if there are 90 elements. There is NO element with subscript 90.

#### 7.1.4. Other array examples

To count the number of occurrences of the letter 'e' in a piece of text, we will write a program to read the characters of the text into an array of characters, and then search the array for occurrences of the letter 'e'. We will need an array of `char` elements. We will read characters from the input until we encounter a full stop.

```

// Set maximum number of characters
const int max = 100;
// Grab space for 100 characters
char sentence[ max ];
int i = 0;
// Read up to a full stop

```

```

while(
    cin >> sentence[ i ],
    sentence[ i ] != '.'
) {
    if ( ++i >= max ) {
        cerr << "Error sentence overflow\n";
        exit( 1 );
    }
} // while read character not full stop
const int no_of_chars_read = i;

```

We should declare the array long enough to hold all likely sentences, and check that the data does not overflow it. Another "safe" way to code the loop might be

```

for ( i = 0; i < 200; i++ ) {
    cin >> sentence[i];
    if ( sentence[i] == '.' ) {
        break;
    }
}

```

If we ran the program, and typed

```
The cat sat on the mat.
```

at the terminal, the code would set

```

sentence[ 0 ]   to the value   'T'
sentence[ 1 ]   to the value   'h'
sentence[ 2 ]   to the value   'e'

```

and so on.

Note that "cin >> character" ignores all white space (spaces, tabs and newlines), so that we would have set

```

sentence[ 3 ]   to the value   'c'
sentence[ 4 ]   to the value   'a'
sentence[ 5 ]   to the value   't'

```

even though we typed a space after "The".

To search through the array once it has been read in looking for occurrences of the letter 'e', we use

```

int count = 0;
// Now go through the array counting
for( i = 0; sentence[ i ] != '.'; i++ ) {
    if( sentence[ i ] == 'e' ) {
        count++;
    } // end if letter is e
} // end search sentence
cout << "Number of e's is "
    << count << "\n";

```

Or we could use

```
for ( i = 0; i < numberofchars; i++ ) {
```

to control the loop

We again control the loop by looking for the full stop '.'. It would be possible instead to count the total number of characters as they are read in, and use this count to control the upper limit of the loop.

Beware of looking for a pair of consecutive letters with, for example

```

if (
    sentence[i] == 'h'
    && sentence[i+1] == 'e'
) ...

```

to look for the string "he". The "i+1" suffix means that you must terminate the loop one position earlier. If

you terminate the loop by

```
for( i = 0; sentence[i+1] != '.'; i++ )
```

you will miss the terminator on an empty sentence.

## Sorting numbers

We wish to read integers into an array (of `int` variables), sort them into ascending order by swapping, and then print them out. We will assume that the data is terminated by a zero value. First the declarations:

```
// the maximum number of ints
const int max_n = 100;
int array[ max_n ];
```

Then we read the data in:

```
int i = 0;
while(
    cin >> array[ i ],
    array[ i ] != 0
) {
    if ( ++i >= max_n ) {
        cerr << "Error array overflow\n";
        exit( 1 );
    }
} // while read up to zero
// note how many we read in
const int number = i;
```

Then we sort the numbers into ascending order by swapping:

```
int j;
for( i = 0; i < number; i++ ) {
    for( j = 0; j < i; j++ ) {
        if( array[ j ] > array[ i ] ) {
            // swap if out of order
            const int swap = array[ i ];
            array[ i ] = array[ j ];
            array[ j ] = swap;
        } // if out of order
    } // for j up to i-1
} // for i in the array
```

Then we might print the results:

```
for( i = 0; i < number; i++ ) {
    cout << i << array[ i ] << "\n";
} // for i
```

To print the results ten entries per line

```
// npl = number per line
const int npl = 10;
for( i = 0; i < number; i++ ) {
    cout << array[ i ];
    if ( (i+1) % npl == 0 ) {
        cout << "\n"; // newline
    }
} // for i
cout << "\n";
```

## The complete program

The above program when put together becomes as follows.

```
#include <iostream.h>
// the maximum number of ints
const int max_n = 100;
// npl = number printed per line
const int npl = 10;
main() {
    int array[ max_n ];
    int i = 0;
    while(
        cin >> array[ i ],
        array[ i ] != 0
    ) {
        if ( ++i >= max_n ) {
            cerr << "Error array overflow\n";
            exit( 0 );
        } // if check array overflow
    } // while read up to zero
    // note how many numbers we read in
    const int number = i;

    int j;
    // Now order them
    for( i = 0; i < number; i++ ) {
        for( j = 0; j < i; j++ ) {
            if( array[ j ] > array[ i ] ) {
                // swap if out of order
                const int swap = array[ i ];
                array[ i ] = array[ j ];
                array[ j ] = swap;
            } // if out of order
        } // for j up to i-1
    } // for i in the array

    // Now print them
    for( i = 0; i < number; i++ ) {
        cout << i << array[ i ];
        if ( (i+1) % npl == 0 ) {
            cout << "\n"; // newline
        }
    } // for i
    cout << "\n";
} // end main program
```

### 7.1.5. Array elements and indexes

Always be careful to distinguish between the operation

find the value of the largest element in the array ...

and

find the position of the largest element ...

To find the maximum value in an array you may write

```
float values[100];
// Assume that the array is set up
// with values terminated by a zero.
int sub; // Subscript
float max = values[0];
// note that we start here at subscript 1
for ( sub = 1; values[ sub ] >= 0; sub++ ) {
    if ( max < values[ sub ] ) {
        max = values[ sub ];
    }
}
```

```

    }
} // for sub in array
cout << max << "\n";

```

We start by assuming that the first element is the largest, and compare each of the remaining elements (subscripts from 1 upwards) with it in turn.

We now have the largest value in the "float" variable "max". The type of the variable "max" will be the same as the type of the array elements.

To find the position (the "index") of the largest element, write

```

float values[100];
// Assume the array is set up
// as before.
int sub;
int maxpos = 0;
for ( sub = 1; values[ sub ] >= 0; sub++ ) {
    if ( values[ maxpos ] < values[ sub ] ) {
        maxpos = sub;
    }
} // for sub in array
cout << "Position " << maxpos << "\n;"
cout << "Largest value " << values[ maxpos ] << "\n;"

```

We assume the position of the largest element is zero, and compare each other element with it in turn. We now have the position of the largest element in the "int" variable "maxpos". The type of the variable "maxpos" will always be `int`.

Note that there is always a unique maximum value for an element of an array; there may not be a unique position of the largest element if there are several equal largest values. The above code gives us the position of the first of the equal maxima; if we replaced the

```

if ( values[ maxpos ] < values[ sub ] ) {

```

by

```

if ( values[ maxpos ] <= values[ sub ] ) {

```

we would get the position of the last of the equal maxima.

### 7.1.6. Searching for words

To search for the occurrences of a particular word such as "the" in the text, we must search for occurrences of the 3 characters 't', 'h' and 'e' in adjacent positions in the array.

```

count = 0;
// now go through the array counting
for( i = 0; sentence[ i ] != '.'; i++ ) {
    if ( i < 2 ) {
        continue;
    }
    if(
        sentence[ i-2 ] == 't' &&
        sentence[ i-1 ] == 'h' &&
        sentence[ i ] == 'e'
    ) {
        count++;
    } // end if word "the"
} // end search sentence

```

Note that in this case we start the subscript at 2 rather than zero.

We could move the subscript from zero upwards using

```

count = 0;
// now go through the array counting
if ( sentence[0] != '.' && sentence[1] != '.' ) {

```

```

for( i = 0; sentence[ i+2 ] != '.'; i++ ) {
    if(
        sentence[ i ] == 't' &&
        sentence[ i+1 ] == 'h' &&
        sentence[ i+2 ] == 'e'
    ) {
        count++;
    } // end if word "the"
} // end search sentence
} // end if not early dot

```

In either case, we must be careful not to run off either end of the array.

If we were actually looking for the word "the" for serious linguistic reasons, we would need to check that there were non-letters at both ends of the string "the" wherever we find it, using perhaps

```

if (
    ( i < 3 || sentence[ i-3 ] not a letter )
    &&
    sentence[ i+1 ] not a letter
) { ...

```

There is a standard library function `is_alpha` to check whether a character given as parameter represents a letter. In addition we would probably accept either a leading upper case 'T' or lower case 't'. using a construction such as

```

if (
    sentence[ i-2 ] == 'T'
    || sentence[ i-2 ] == 't'
) { ...

```

Instead of looking for a specific word such as "the", we may wish to look for a general word (string). We would then store the word we are looking for in a second character array. To search our stored sentence for a word of length "l\_word" stored in a character array "word" (assuming that the value in `l_word` has been set up to equal the length of the word stored in the character array `word` ) the second part of the program would have to be turned into a loop to compare characters in the word with characters in the array. The code might be:

```

count = 0;
int found;
// Now go through the sentence counting
for(
    i = l_word-1; // Start at length of word
    sentence[ i ] != '.'; // Stop at full stop
    i++
) {
    found = 1; // 1 means true
    for ( j = 0; j < l_word; j++ ) {
        if(
            sentence[ i-l_word+j+1 ] != word[ j ]
        ) {
            found = 0; // 0 means false
        } // end if next letter found
    } // end for all chars in word
    if( found ) { // if ( found == 1 ) would do
        count++;
    } // end if found
} // end search sentence

```

The above example would be a little more easily readable if we declared

```

const int TRUE = 1;
const int FALSE = 0;

```

and use the values TRUE or FALSE later in the program.

## String functions

Any operation like this will be much more easily performed by using the string library functions, perhaps "strncmp" to compare strings in this case:

```
count = 0;
// Now go through the array counting
for( i = 0; sentence[ i ] != 0; i++ ) {
    if(
        strncmp(
            word,
            sentence + i,
            l_word
        ) == 0
    ) {
        count++;
    } // end if found
} // end search sentence
```

For serious programming, always use library functions whenever they are available. Type

```
man string
```

for details.

All of the string library functions assume that the characters stored in the array are terminated by a null (zero) character. Such arrays could be printed by "cout", which will print characters from a character array until a zero element is encountered.

If you read a string, you should use

```
char sentence[100];
int subscript;
subscript = 0;
while (
    cin >> s[subscript],
    s[subscript] != '.'
) {
    subscript++;
}
s[subscript] = 0;
```

to insert the terminating null.

### 7.1.7. Initialised arrays

If you wish the values of an array to be initialised, you can write ONLY IN GLOBAL

```
int primes[ ] = { 1, 2, 3, 5, 7, 11 };
```

with the initial values comma-separated, between curly braces. You need not put a value for the length of the array between the square brackets, since the compiler can count how many elements you have declared! This would set up an array of six elements starting at suffix zero with

```
primes[0] = 1
primes[1] = 2
etc
```

If you do put a value between the square brackets, as in

```
int primes[ 10 ] = { 1, 2, 3, 5, 7, 11 };
```

an array of 10 elements will be declared, with the remaining elements (4 in this case) not initialised. The number given between the square brackets must be greater than or equal to the number of initialising values given.

To initialise a character array, you could write of course

```
char word[ ] = { 'E', 'r', 'i', 'c', 0 };
```

An alternative notation has been devised because initialised strings are required so often. You can also write

```
char word[ ] = "John";
```

This actually initialises a 5-character array, with an additional zero element at the end; this is provided so that programs using the array can keep looping until they find the zero element. The form of a loop using this array of characters (an array with a terminating zero element) is now

```
int sub; // Our subscript
for( sub = 0; word[ sub ] != 0; sub++ ) {
    ....;
}
```

We could, of course, omit the "!= 0".

Notice that "word" is an ordinary array of character elements. If we execute

```
word[2] = 'a';
```

the word becomes "Joan".

### The size of an initialised array

To operate on the elements of an initialised array you will need to know how many elements it has. It is bad practice to have a separate constant giving the number of elements, declared separately from the initialised array declaration, since there is always a possibility that the length value might not agree with the actual length.

One possibility is to use the compile-time operator `sizeof` (deliver the size in bytes of an object) which was mentioned earlier, and write

```
const int arr_length =
    sizeof annual_rain / sizeof annual_rain[0];
```

and then

```
for ( sub = 0; sub < arr_length; sub++ ) { ...
```

Alternatively you can put a special marker element at the end of the array, and loop through the elements until you encounter that special value. For example

```
int primes[ ] = { 1, 2, 3, 5, 7, 11, 0 };
for( i = 0; primes[ i ] != 0; i++ ) {
    ....;
} // for i loop
```

Adding extra prime numbers into the declaration at a later stage will not affect the correct functioning of the loop, as long as the last element is always a zero.

This is exactly the way that strings are scanned by library functions; you may well see the lazy version written as

```
for( i = 0; sentence[ i ]; i++ ) {
    ....;
} // for i loop
```

where the " != 0 " is omitted.

### Efficiency of array initialisation

To initialise a globally declared array take NO time while the program is running. The appropriate locations have already been initialised to the correct values in your executable file.

To initialise an array declared inside a function requires

1. a copy of the values for initialisation must be kept somewhere, to be used each time the function is entered; and

2. on each entry to the function, the values must be copied into the newly declared array. This can take a lots of time if the function is called many times, and if the array is large.

Large initialised arrays are usually seen only in global declarations.

Arrays and variables declared in global, and not otherwise initialised, are all initialised to zero.

## 7.2. Printing large arrays

For large arrays, you will not wish to

- (i) print all values on separate lines, or
- (ii) print one value per line.

Use a construct such as

```
for( sub = 0; sub < MAX; sub++ ) {
    if ( sub % 10 == 0 ) {
        cout << "\n";
    }
    cout << array[ sub ];
}
cout << "\n";
```

Entries 0 to 9 will appear on one line, entries 10 to 19 on the next ...

To make the values appear in neat columns, you would need to format the output as in

```
cout << form( "%5d", array[sub] );
```

### 7.2.1. Two-dimensional arrays

#### Declaration

The above arrays are one-dimensional, and can store a single "row" or "column" or "vector" of values. Suppose we wish to store a table (two-dimensional) of values; these might be the rainfall for each of 12 months for each of 90 years, or the IQs of each of the 11 people in each of 22 football teams, or the marks for each of 20 exercises for each of 100 students, or the names (30 characters long) of each of 100 students. We now need two subscripts for each element, and would write

```
// 90 * 12 variables
const int n_yrs = 90;
const int n_mths = 12;
float month_rain[ n_yrs ][ n_mths ];

int IQ[ 22 ][ 11 ];

int mark[ 100 ][ 20 ];

char names[ 100 ][ 30 ];
```

#### Use of 2-dimensional arrays

To read rainfall data in, we might use

```
int year, month; // for subscripts
// Read in all the 12 * 90 values
for( year = 0; year < n_yrs; year++ ) {
    for( month = 0; month < n_mths; month++ ) {
        cin >> month_rain[ year ][ month ];
    } // Month loop
} // Year loop
```

The numbers in the data must be given in the correct order required by the program; if the loops are nested as above, we would require the twelve numbers for the first year, then the twelve numbers for the second year, ...

If the loops had been nested the other way round (just interchange the two "for" lines) the data would have had to consist of the 90 January figures, then the 90 February figures, ...

### Accessing the elements

To calculate the annual totals for each of the 90 years we might write

```
// Calculate the year totals
float total;
for( year = 0; year < n_yrs; year++ ) {
    total = 0;
    for( month = 0; month < n_mths; month++ ) {
        total += month_rain[ year ][ month ];
    } // for month
    annual_rain[ year ] = total;
    cout << "year " << year + 1900
         << " rain " << total << "\n";
} // for year
```

We could have used "annual\_rain[ year ]" instead of "total" for our addition above; it would be marginally slower on the computer, since it would have to look up the array subscript each time.

To print the average rainfall over the 90 years for each of the 12 months separately, we might write

```
// print monthly averages
for( month = 0; month < n_mths; month++ ) {
    total = 0;
    for( year = 0; year < n_yrs; year++ ) {
        total += month_rain[ year ][ month ];
    } // for year
    cout << month << total / n_yrs << "\n";
} // for month
```

To find the first student whose name starts with the letter 'S', we might use

```
int student = -1, i;
for ( i = 0; i < n_students; i++ ) {
    if ( names[ i ][ 0 ] == 'S' ) {
        student = i;
        break;
    }
}
```

leaving `student` set to -1 if no such student was found.

### Initialising 2-dimensional arrays

You could write (only in global) for example

```
int table[ ][ ] = {
    { 1, 2, 3, 4, 5 },
    { 5, 4, 3, 2, 1 },
    { 1, 3, 5, 3, 1 }
};
```

This would give us a 3 by 5 array of integers.

There is an alternative "flattened" form

```
int table[3][5] = {
    1, 2, 3, 4, 5,
    5, 4, 3, 2, 1,
    1, 3, 5, 3, 1
};
```

In the flattened form we have to insert the bounds, since the compiler could not know whether we intended a 3 by 5 array or a 5 by 3 array.

The particular case of initialised two-dimensional character arrays which will be explained more fully later, but is introduced here since you may find it useful. We are concerned with initialising an array of words.

## Keywords

When looking at C++ programs, we may wish to search for all keywords. We would declare

```
char *keywords[] = {
    "int",
    "float",
    "double",
    "char"
    ""
};
```

This is effectively a 2-dimensional array of characters. Each row of this array is of a different length, the length of that keyword + one for the terminating zero which is always added to a string.

We could now have code to look for each keyword in turn, the element "keyword[i][j]" is the j-th character of the i-th keyword. We search along each keyword until we encounter the terminating zero.

This subject is dealt with more fully later under the subject of pointers.

### 7.2.2. Higher dimensional arrays

You can, of course, declare and use 3-dimensional, 4-dimensional and higher dimensioned arrays by extending the above notation.

In three dimensions

```
float mass[10][10][10];
int x_co_ord, y_co_ord, z_co_ord;
for ( x_co_ord = 0; x_co_ord < 10; x_co_ord++ ) {
    for ( y_co_ord = 0; y_co_ord < 10; y_co_ord++ ) {
        for ( z_co_ord = 0; z_co_ord < 10; z_co_ord++ ) {
            if ( mass[x_co_ord][y_co_ord][z_co_ord] >= min ) ...
        } // z loop
    } // y loop
} // x loop
```

Beware that it is very easy to eat up huge amounts of storage (the above example declares 1000 variables) with many-dimensional arrays. The declaration

```
float mass[100][100][100][100];
```

would occupy 100,000,000 floats. For your programs you may assume a megabyte of space, say 250K integers or 100k floats.

## 7.3. More array examples

### 7.3.1. Reflect lines

We read text from standard input until end-of-file is encountered, printing each line as it is read in reflected from left to right. We use the function "getchar()" to read characters instead of "cin", since it reads spaces and newline characters. The "getchar()" function returns a negative value when it reaches end-of-file, so that the loop is controlled by a "while ... > 0" mechanism.

```
#include <iostream.h>
#include <stdio.h>
main() {
    // Store each line as it is read
    char line[100];
    char ch;
    int i = 0;
    while ( ch = getchar(), ch > 0 ) {
        if ( ch == '\n' ) {
            // End of line, print in reverse
```

```

        while( --i >= 0 ) {
            cout << line[ i ];
        }
        cout << "\n";
        i = 0;
    } else {
// Ordinary character, store it
        line[ i++ ] = ch;
    } // end if else end of line
} // end while not end of file
} // end main

```

Observe that we have a single loop to read the characters, which takes a special action when it encounters a newline character.

Observe that if the compiled program is called "reflect", then typing

```
cat any_file | reflect | reflect
```

or

```
reflect < any_file | reflect
```

should show the original file.

### 7.3.2. Arrays as function parameters

Arrays can be passed as function parameters, subject to a few new ideas.

When the identifier of an array is used as a parameter, the identifier (which is a single value representing the location of the array in memory) is passed over; this reference to where the array is stored is passed "by value" in terms of our earlier jargon. This means that, inside the function, individual elements of the array can be modified. Thus in the program

```

// declaration
void functn( int a[] );
main() {
    int eric[10];
    int fred[1000];
    functn( eric );
    functn( fred );
} // end main
// definition
void functn( int a[] ) {
    int temp = a[0];
    a[0] = a[1];
    a[1] = temp;
} // end functn

```

the call of "functn" in the main program will have the effect of swapping over the values in "eric[0]" and "eric[1]". It is the *identifier* of the array which is passed over, not the contents of the whole array. To do the latter would cause great inefficiency in terms of storage space and processor time if the arrays were large.

If you wish to ensure (for safety reasons) that the array elements cannot be accidentally overwritten, then the function declaration becomes something like

```
void functn( const int a[] ) { ....
```

The "const" keyword should ensure that any attempt to overwrite array elements from within the function is prevented (reported as a compile-time error) by the compiler. The "g++" compiler appears merely to report a warning, not an error.

If the function needs to know the size of the array, you must pass it as an integer parameter, as in

```

// declaration
void functn( int a[], int length );
main() {

```

```

    int eric[10];
    int fred[1000];
    functn( eric, 10 );
    functn( fred, 1000 );
} // end main
// definition
void functn( int a[], int length ) {
    int temp;
    for( temp = 0; temp < length; temp++ ) {
        ...
    }
} // end functn

```

## 7.4. Structures

The purpose of structures is to group together a number of related items. The items do NOT need to be of the same type.

### 7.4.1. Examples of the requirement for structures

- (i) We are dealing with payroll in a company. For each person employed we need their
  - name (string)
  - address (string)
  - works number (integer)
  - tax code (integer?)
  - rate of pay / hour (integer)
  - hours worked this week (float?)
  - total pay so far this year (integer)
- (ii) We are analysing the properties of a proposed bridge design. For each component of the structure we need its
  - strength (float)
  - dimensions (3 floats?)
  - weight (float?)
  - cost (integer)
  - name (string)
- (iii) We are working with geographical data. Each item of data consists of
  - a data value of type integer
  - two position co-ordinates of type float
- (iv) We are working with train timetables. For each entry in the timetable we need
  - departure time (int)
  - arrival time (int)
  - special features (Fridays only?, Buffet car?)
- (v) We are working with a number of countries. For each country we need
  - name (char array)
  - population (int)
  - area (float)

### 7.4.2. Declarations of structure types

We will declare first the layout (components) required of a particular type of structure; the declaration of the structure objects themselves will come later elsewhere.

A bridge component structure type might need

```
struct element {
    float strength;
    float length, breadth, height;
    int cost;
    float weight;
    char name[50];
    int stock;
};
```

A general purpose structure for handling dates might be

```
struct date {
    int day_no, week_no, month_no;
    char name[4];
    long int secs;
};
```

A structure for handling geographical data might be

```
struct city {
    char name[50];
    float latitude, longitude;
};
struct country {
    char name[50];
    city capital;
    int population;
    float area;
}
```

A structure for integer data values each of which is associated with an (x,y) co-ordinate (the x and y could represent geographical latitude and longitude or Ordnance survey co-ordinates) might be

```
// The structure type
struct data_item {
    float x, y;
    int value;
};
```

A structure for railway timetable entries might be

```
struct t_table {
    int depart;
    int arrive;
    int Fri_only;
    int buffet;
};
```

A structure for a country might be

```
struct country {
    char name[20];
    int popn;
    float area;
};
```

A structure for a string might be

```
struct string {
    char data[200];
    int length;
};
```

```
}
```

where the integer field tells us how long the stored string is.

The separate components of a structure are called its *fields*.

### 7.4.3. Declaration of structure objects

The above declarations are of structure types, not of structure variables for storing data values. To actually declare objects of the above structures, we might write

```
data_item this, that, result[200];
```

Here we have declared two single structures called `this` and `that` and an array of 200 structures; the whole array is called `result`.

Do not confuse the structure type declaration (uses no space, gives an identifier to the type of structure) and the variable declarations (they occupy space in the program's memory when the program is running). Typically the structure type declarations would go into a header file.

This is similar in some ways to the distinction between a function declaration and definition. The function declaration merely informs the compiler of the details of how function calls are to be handled, the definition includes the actual code to be executed.

### 7.4.4. Accessing elements of a structure

In order to access individual fields of a structure we use the structure variable identifier, followed by a full stop, then the field identifier from the structure type declaration. Given

```
data_item this, that, result[200];
```

declared earlier, we may write

```
this.x = 10.5; // float
this.y = 20.7; // another float
this.value = 15; // an int
if ( this.x > 0 ) { ....
for ( i = 0; i < 200; i++ ) {
    cin >> result[i].x >> result[i].y;
}

// "girder" is a "element" structure
element girder;
// "all_stock" is an array of 100 structures
element all_stock[ 100 ];
girder.name = "Box girder type 10A"; // array of char
girder.strength = 25.93; // float
girder.stock = 0; // int
if ( all_stock[ 10 ].stock < minimum ) {
    all_stock[ 10 ].required = minimum;
}
for ( stock = 0; stock < 100; stock++ ) {
    tot_requd += all_stock[ stock ].required;
} // for stock loop
```

The compiler will not be confused by the fact that `stock` is used both as a structure field identifier, and as an array identifier. The field identifier will always follow a full stop.

```
// "nott_london" is an array of "t_table"
// for the Nottm to London timetable
for( i = 0; i < NNTS; i++ ) {
    if ( nott_london[ i ].depart < .... ) {
        ....;
    }

// "result" is an array of "data_item"s
// add data values for all points within
// given "radius" circle
```

```

int sum = 0;
for ( i = 0; ..... ) {
    if (
        result[ i ].x * result[ i ].x
        + result[ i ].y * result[ i ].y
        < radius * radius
    ) {
        sum += result[ i ].value;
    }
} // for i loop

```

To declare an array of countries use

```
country asia[50];
```

("country" is the structure type, "asia" is the object identifier) and to access fields

```

// the area of the first one
asia[0].area // a float
// the latitude of the capital of the next one
asia[1].capital.latitude

```

To copy a whole structure use:

```
all_structs[0] = girder;
```

or if you require merely to copy certain fields use:

```

all_stock[0].strength = girder.strength;
all_stock[0].length = girder.length;
all_stock[0].breadth = girder.breadth;
all_stock[0].height = girder.height;

```

### 7.4.5. Initialising structures

To initialise structures we use a notation similar to that for initialising arrays, with the values between curly braces, as in

```

element girder = {
    1234.56,
    15.7, 32,6, 99,9,
    7600,
    50.05,
    "Box girder"
};

```

For an array of structures, use either the nested braces as in

```

data_item value[] = {
    { 1.0, 2.0, 1323 },
    { 1.5, 1.0, 4523 },
    { 0.0, 2.9, 4373 },
};

```

or use the flattened form as in

```

data_item value[] = {
    1.0, 2.0, 1323,
    1.5, 1.0, 4523,
    0.0, 2.9, 4373,
};

```

## Geographic example

For geographical data use

```
country europe[] = {
    "UK", { "London", 52, 0 }, 50000000, 2000000,
    etc.
```

The length of (the number of elements in) an initialised array of structures can be calculated (a compile-time constant) by using the `sizeof` function to divide the total size of the array by the size of one element, as in

```
const int nvalues =
    sizeof value / sizeof value[0];
```

Do not forget always to divide by the size of a single element of the array. You may then use

```
for( sub = 0; sub < nvalues; sub++ ) {
```

to control various loops.

For a simplified BritRail timetable, use perhaps

```
struct t_table {
    int depart, arrive ;
};
```

for the structure type, and

```
t_table nott_lond[] = {
    { 537, 727 },
    { 738, 909 },
    { 837, 1023 },
    { 1037, 1216 },
    { 1237, 1412 },
    { 1437, 1623 },
    { 1637, 1819 },
    { 2037, 2229 },
    { 2214, 512 }
};
```

for the initialised declaration. You will also need

```
const int num_trains =
    ( sizeof nott_lond / sizeof nott_lond[0] );
```

In this case the structure type "t\_table" represents a single timetable entry, whereas the array of structures "t\_table" represents a whole timetable.

### 7.4.6. Bit-fields in structures (keenies only)

Ordinary mortals do not need to know that structure notation can be used to break a single computer word down into small bit-fields for storing small data items.

```
struct line_type {
    int i;
    float f;
    unsigned line : 5;
    unsigned col : 6;
    unsigned mode : 2;
    unsigned t : 2;
};
```

The numbers after each field indicate the number of bits allocated. All fields must be of type `unsigned`.

### 7.4.7. The use of arrays of structures

Whenever you have related items, they should be grouped in a structure. If a program contains two arrays of the same length, it can usually be inferred that the corresponding elements are related.

You should replace

```
float x[100], y[100];
int value[100];
for ( i = 0; i < 100; i++ )
    cout << x[i] << y[i];
```

by

```
struct data {
    float x, y;
    int value;
};
data point[100];
```

and use by

```
for ( i = 0; i < 100; i++ )
    cout << point[i].x << point[i].y;
```

Program style checkers look for arrays of the same length, and recommend that they be turned into a structure.

## 8. Marking

Your program layout should now treat square brackets like any others; if the open and close fit on the same line, that's OK. If they don't, use the same rules as for curly braces (but of course no comment necessary after the close).

```
average +=
    this_val[
        ( total * factor + base ) / per_unit + offset
    ];
```

© Eric Foxley December 10, 1996

## Chapter 8 : File input/output

It is comparatively easy to perform simple serial input and output using files in C++. We will teach only the straightforward aspects.

### 8.1. Introduction

Perhaps we should first remember that input and output can be diverted from and to files in operating systems such as UNIX or MS-DOS without any program changes, provided that we are talking about *all* of the input or *all* of the output. Thus we can write

```
prog32 < data_file
```

to run the program `prog32` taking all of its input from the file `data_file`. This is the basis of the way your programs are tested for dynamic correctness in the Ceilidh command against various sets of test data.

Conversely, we can write

```
prog32 > output_file
```

to store all of the (standard) output in the named file. This is how your program output is saved when Ceilidh tests the program. The dynamic testing system then searches the saved output for the keywords or phrases which the teacher has specified.

The error output (anything using the `cerr` stream) would still come to the terminal, and would not be diverted to the file.

This simple approach for diverting program input or output will satisfy some simple situations where we wish to use files instead of the terminal for input or output. However, for most realistic applications, we will need to communicate with standard input and output (the user's terminal) as well as one or more files. We will generally need to interact with a user (print a prompt such as "When does the person arrive at Nottingham station?" on the terminal, and read the reply from the terminal) as well as interacting with the files of data (reading from a file containing the actual train timetables, and perhaps writing to a file of seat reservations).

### 8.2. The basics of file access

The approach to file input/output in most modern shared computer systems involves the program (i.e. any process wishing to access a file) in the following sequence of activities.

- (i) Open the file. The name of the file is given, and the type of access required (e.g. read, write, both, append, seek, ...). When the program runs, the system checks that the type of access requested is permitted; in UNIX this involves the ownership of the file, the access permissions for that file defined by its owner (perhaps modified using the `chmod` command, and defined in terms of "read" or "write" or "execute" for the "owner", their "group" and the "others"), the identity of the user running the program, and perhaps even in special cases (look up the concept of SUID facilities) the owner of the program. In more secure computer systems, access permission may be a much more complicated concept related to the user's management seniority and areas of responsibility.
- (ii) After opening, the file can be accessed by the program as required. The program will not be allowed to read the file if it has asked only for "write" permission. The program now refers to the file not by name, but by a special code returned by the opening process. Information is now read from the file, or written to it. There is usually no consideration of the fact that the permissions on the file could be changed after the open has taken place; once the file has been successfully opened, access is permitted until the file is closed or the process terminates.
- (iii) The file is closed when access is no longer required. This is often built into the program termination activity, so that the user does not need to take any closing action. There may be upper limits on the number of files simultaneously open on certain systems, so this may necessitate closing files as soon as access to them is no longer required.

### 8.3. Files in C++

Files are made under UNIX and MS-DOS to appear very similar to any other form of serial input/output device. Thus in the ordinary command shell, output can be directed to the terminal, or to a file, or to another process, with no fundamental change to the process producing the output. The operating system is hiding from the user what are in reality great differences between the physical processes of sending characters to a terminal and sending them to file.

A typical C++ program to send output to a file is

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

// The name of the file
const char * f = "/tmp/data_out_file";

main() {

    ofstream fout;

    // Open the file
    fout.open ( f );
    if ( fout.fail() ) {
        cerr << "Cannot open file"
             << f << "\n";
        exit( 1 );
    }
    int i;
    // Write to the file
    for ( i = 6; i >= 0 ; i-- ) {
        fout << i << "\n";
    }
    // Close it
    fout.close();
    if ( fout.fail() ) {
        cerr << "Close file error\n";
        exit( 1 );
    }
} // end main
```

The declaration of an `ofstream` requests an "output file stream", to be called `fout`. The call of `fout.open` then asks that it be connected to the file `"/tmp/data_out_file"`. If the request is successful, then "fout" will appear to us just like "cout" for performing output, but output will go to the file, not the terminal. The particular identifier chosen ("fout" in this case) is entirely up to the programmer. Before proceeding any further with the program, we must check that the named file was opened successfully; this is done with the

```
if ( fout.fail() )
```

part of the code. There is no point in the program continuing if the open was not successful. At this point we would typically print a message to `cerr` and exit with non-zero status.

Having opened our output file stream successfully, we then write data and strings to the file using exactly the same techniques as we have been using for `cout`, but using the output stream identifier `fout` instead.

We do not need to close the stream when we have finished with the file, but can leave closing to be performed automatically during the end of program activities. It is however generally good practice to close files no longer in use. The status of the `close` should also be checked with the `fout.fail()` command.

## Reading from a file

If we had wanted to read from the file, we might have written

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

const char * f = "data"; // name of file

main() {

    ifstream fin;

    // Open for reading
    fin.open( f );
    // Check as before
    if ( fin.fail() ) {
        cerr << "Error opening "
             << f << "\n";
        exit(1);
    }

    int total = 0, number;
    // Read from "fin"
    while ( fin >> number, number != 0 ) {
        total += number;
    }
    // Close the file
    fin.close();
    cout << "Total: " << total << "\n";
} // end main
```

e start by declaring an `ifstream` (input file stream) and using `fin.open( f )` to connect it to file `f`. We again use `fin.fail()` to check that the open was successful. If the request is successful, then `fin` becomes an input stream to be treated just like `cin`. In this case the file is presumed to contain a series of integer values terminated by a zero. It could have been created using a text editor such as "vi" or "emacs", or as the output of an earlier program using

```
prog8xxx > outfile
```

or by a program using it as an output file stream directly.

The choice of identifier `fin` is up to you.

## Simultaneously open files

We can have a need for several files to be open at the same time, typically at least one for reading and one for writing. There will usually be an upper limit imposed by the system, perhaps 20 files.

In UNIX the file close takes place automatically when the program terminates if the file has not previously been closed. In other systems, information being written to a file may be lost if the file is not closed properly.

## 8.4. The use of files in general

Updating files is the essence of commercial programming. A file will contain details of

- all personnel, pay to date, tax to date, tax codes, etc
- all stock in the warehouse, current and minimum levels, etc
- all bank accounts, the owner, the balance, the maximum debt, etc
- all flights by the airline, booked and free seats, destination, timing, etc

In commerce, each set of related data (one person's record, the data for one type of stock item in the warehouse) is referred to as a "record". Each complete set of related data and the means for accessing it from

within a program would be represented by a structure inside a C++ program.

Each day or week or month (or instantly on receipt of an interactive transaction) the file will be updated, and a new file produced. For security reasons, a firm will keep a limited number of old copies of the file together with details of all subsequent transactions, so that the latest file can be re-created if it gets corrupted.

The information inside most files will be held in a definite order, e.g. ordered by personnel works numbers, warehouse stock number, bank customer account number, flight departure time, etc.

### Updating small files

A typical program to update a file would, if the file is small,

- 1 read the whole of the latest master file into an array (open for reading, read the entire file, and then close it)
- 2 interact with the user (or use information stored in a data file) to update the various entries in the array (interact using "cin" and "cout") (to add this week's pay, to decrement or increment the current warehouse stock values, to change the current credit in the bank accounts, to reserve a seat on a flight)
- 3 write the updated information stored in the array into a new file (open for writing, write the entire file, and close).

If all has gone smoothly, the new file is now the master copy, the previous master file becomes the backup copy.

The program sequence might be

```
Declare a structure type suitable for
the information in each record
Declare a big enough array of
these structures
Open the existing master file
for reading
Read all the information from the
existing file into the array
Close the file
```

Then interact with the user using:

```
while (
  Ask "Any more updates? ",
  Reply isn't no
) {
  Ask "person? ", read person
  Find array subscript for this person
  Ask "details? ", read details
  Amend entry values in array of structures
} // end while more updates
```

Now finish off with

```
Open a new file for writing
Write the complete array to the
new file name
Close the new file
Print any summaries as required
```

## Updating large files

For larger files, it may not be possible to read the whole file into memory. The program would first order the transactions so that they are in the same order as the entries in the master file; we would assume that the transactions are now held in a file rather than input from a keyboard. We then read the existing master file one entry at a time, see if that entry needs updating, and write that entry to the new master file. In this case both old and new files (and the file of transactions) are open, and only one record is held in the program at a time. The program outline might be as follows.

```
Declare a structure type for each record
Declare one structure variable
Open existing master file for reading
  say "fin"
Open new master file for writing
  say "fout"

while (
  Not at end of transaction file
) {
  Read next transaction from transaction file
  Read records from master file "fin",
    copying to new master file "fout"
  until this person's record found
  Check transaction details
  Amend record values
  Write this person's new record to
    the new master file "fout"
}

Copy the remainder of old master
  file "fin" to new master file "fout"
Both files should be closed here

Print any summaries ...
```

Alternatively, the while loop could be controlled by the reading from the input file, as in

```
Declare structure type for each record
Declare one structure variable
Open existing file for reading
Open new file for writing
Ask "First person to amend? "

while (
  Not reached end-of-input-file
) {
  Read record from existing file
  if ( not person we're looking for ) {
    Write record to output file
    continue
  }
  Ask "details? ", read details
  Amend record values
  Write this person's record to output file
  Ask "Next person to search for? "
} // end loop to end of file

Both files could be closed here
Print summary ...
```

### **Interactive transactions**

For interactive transactions (such as airline bookings) there must be a way of **locking** an individual record; it must not be possible for two customers to simultaneously request a spare seat, find that there is one, and attempt to both occupy the same single remaining seat! We are then into a new level of complexity.

© Eric Foxley December 3, 1996

## Chapter 9: Miscellany 2

### 9.1. Reading characters

There are many programming applications in which we have to read text a character at a time to analyse it in some way.

1. A word processor. We are reading text from the user, and laying it out on a page according to certain rules, perhaps padding out lines to a set width using additional spacing.
2. A spelling checker. We are reading text, splitting it into separate words, and comparing the words with those in a "dictionary" (just a list of words, on UNIX it is stored in the file `/usr/dict/words` which is publicly readable and should contain about 30000 words) to check if they occur in the dictionary.

On UNIX, use the command

```
spell -b file
```

to check the words in the named file against British spelling.

3. A compiler. The compiler is reading a program, and converting it into equivalent executable instructions for a particular processor.
4. Analysis of Ceilidh program output under test. The program is run against some test data, and the output is search to see if expected words and phrases occur.
5. A text editor such as "ed" or "vi" or "emacs".

The simple use of the standard input "cin" to read characters as in

```
char ch;  
  
while(  
    cin >> ch,  
    cin != '.'  
) { ...
```

has the logical but inconvenient habit that all white space is ignored. Any spaces, tab characters or new-lines in the input are completely invisible to the program. The loop in the above example read characters from the input until a specific character (a full stop / period) is encountered.

In all of the applications discussed above, we need to be aware of white space characters. We must use the function

```
cin.get( ch )
```

to read each character, and we need a new technique to detect the end of the data. To indicate the end of data from a terminal, you type "control-D". This is referred to as "end of file" or "EOF" whether we are reading from the terminal (and encounter a control-D) or from a genuine file and encounter the end of the file. The C++ feature for this purpose is the function

```
cin.eof()
```

which delivers TRUE if we are at the end of the data, and FALSE otherwise. This detects end-of-file when reading from "cin" as in

```
prog99 < data_file
```

If you are reading from an file which you have opened from within the program, you may need to use "fin.eof()" instead.

We thus have the program outline

```
while ( ! cin.eof() ) {  
    cin.get( ch );  
    if (
```

```

        ch == ' '
        || ch == '\t' // tab
        || ch == '\n' // newline
    ) { ...
        ....;
    } // end if white space
    ....;
} // end while not EOF

```

or

```

while ( ! cin.eof() ) {
    cin.get( ch );
    switch( ch ) {
        case 'p' : ....; break;
        case 'q' : ....; break;
        default : ....;
    } // end switch
    ....;
} // end while not EOF

```

or

```

while ( ! cin.eof() ) {
    cin.get( ch );
    if ( ch >= 'a' && ch <= 'z' ) {
        ....;
    }
    ....;
} // end while not EOF

```

or

### What is a word?

While reading characters, many of the applications above require us to split the input text into words. If we are word processing (laying out text on a page), each word includes its terminating punctuation. If a word and its punctuation cannot be fitted onto a line, it must all be moved to the start of the next line. We determine the end-of-word by looking for white space.

If we are writing a spelling checker, we split the text into words and compare them with words in a dictionary. In this case, the terminating punctuation is not part of the word. We determine end-of-word by finding a non-alpha character, i.e. one which is not a letter.

The definition of a word thus depends on the application.

### Program structure

Note the possible codings for reading a word at a time from the data. One possibility often seen is

```

while( ! cin.eof() ) {
    while( get char, it isn't a letter ) {
        skip it;
    }
    while( get char, it is a letter ) {
        store it;
    }
    process the word just read;
} // end while ! eof loop

```

The inner loops should both contain checks for end of file. The problem just does not need nested loops.

A much better and safer solution is

```

while( ! cin.eof() ) {
    cin.get( ch );
    if ( ch is a letter ) {
        store it;
    }
}

```

```

    } else if ( ch is the first non-letter ) {
        process the word now stored;
    } else {
        skip it;
    }
} // end while ! eof loop

```

## 9.2. Arrays of strings

It is often useful to have a number of strings stored as an array, so that we can print the i-th string of a set, or search for a command name among a set of alternatives.

We declare (in global, because it initialises an array)

```

char *months[] = {
    "January",
    "February",
    "March",
    "April",
    ""
};

```

This gives us

```

months[0]  is the string  "January"
months[1]  is the string  "February"
months[2]  is the string  "March"
months[3]  is the string  "April"
months[4]  is the string  ""

```

We could thus print the name of the i-th month using

```
cout << months[i] << "\n";
```

The first character of the name of the the i-th month is

```
months[i][0]
```

We could search through the strings for a particular string

```

char word[20];
cin >> word;
for( i = 0; months[i][0]; i++ ) {
    if ( strcmp( word, months[i] ) == 0 ) {
        // found it ...
    }
}

```

Although this study properly belongs to the area of pointers (which is covered properly in the next course, but is summarised without exercises in an extra unit at the end of this course) some useful applications are described below.

### Arguments to the program.

When you type a UNIX command as in

```
prog99 this that other
```

the system generates two arguments which the program can access if it wishes. To access the given parameters, the main program should start

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

The program is then supplied by the system on startup with two arguments.

The first is an integer, and is set to the number of arguments plus one (i.e. the number of words on the command line including the command name itself).

The second is a "char \*[]" containing as strings the command name and the arguments.

Thus in the above example, the arguments are set up as if we had included

```
int argc = 4;

char *argv[] = {
    "prog99",
    "this",
    "that",
    "other",
    0
};
```

The convention is to use "argc" (argument count) and "argv" (argument values) as the argument names, although such names are purely local to your main program.

Thus we now have the value of argv[0] as the string "prog99", of argv[1] as the string "this", etc.

### Accessing the arguments

We can now check that there is at least one argument using

```
if ( argc > 1 ) { ...
```

(the "cp" command for example always checks that it has at least two arguments) and we can access its value by

```
cout << argv[1] ...
```

Each of the strings in the array will have a terminating zero on it; and the array of strings finishes with a zero.

The program can loop through all the parameters in turn with

```
int argno;

for( argno = 1; argno < argc; argno++ ) {
    cout << argv[ argno ] << " ";
}
cout << "\n";
```

This will print the arguments on a single line separated by spaces. This correspond to the "echo" command in UNIX.

Note that these are not global variables; they are parameters to the main program. They can therefore be accessed only from within the main program, or by being passed as parameters to other functions.

### Wild-cards in arguments

Note that if you type, for example

```
prog99 *.C
```

then the UNIX shell first expands the "\*.C" into the names of all files in the current directory ending ".C", and passes all of these over to the program. The program may thus find large numbers of parameters being passed to it. The asterisk generally will not appear in the program's parameter. Thus the command

```
echo *.C
```

echoes the names of all files ending ".C".

If there is NO file matching the requested pattern, the Bourne shell and its derivatives pass over the parameter as a string containing the asterisk.

## UNIX-type flags

A program can detect flag arguments (arguments starting with a '-') by a construct such as

```
int argno;

for ( argno = 1; argv[ argno ][ 0 ] == '-'; argno++ ) {
// argument "argno" is a flag
  switch( argv[ argno ][ 1 ] ) {
    case 'l' : ...; break;
    case 't' : ...; break;
    ....;
  } // end switch
} // for all arguments
```

The arguments starting with a '-' are examined in turn, and actions taken depending on the letter following the '-'. We leave the loop with "argno" indicating the first argument NOT starting with a '-'.

## Values from arguments

It is possible to read numeric values from arguments. For example, you may wish to give the rate of pay and hours worked as integer arguments, and type

```
prog32 152 45
```

instead of typing the values as data) the above arrangements would set "argv[1]" to the string "152", and "argv[2]" to the string "45".

The program could then use the library function "atoi" (ASCII to integer) and write

```
if ( argc > 2 ) {
  rate = atoi( argv[1] );
  hours = atoi( argv[2] );
}
```

There is a similar function delivering floating point values "atof".

Always check that there are enough arguments before trying to access them.

## The environment

There is a third argument available if you wish, containing details of the program's running environment.

If you write

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

as the program heading, the additional third parameter to the main program is another array of strings, this time set to a value such as

```
char *envp[] = {
  "USER=ef",
  "HOME=/staff/ef",
  "TERM=vt100",
  "EDITOR=emacs",
  "SHELL=bash",
  0
}
```

Each string in the array is of the form

```
<environment variable>=<value>
```

You could search this array to find the settings for any variable in which you are interested.

To make life easier, you can declare

```
char *getenv( char * );
```

and use the library function "getenv" as in

```
char *term = getenv( "TERM" );
char *edit = getenv( "EDITOR" );
```

This is straying into the territory of pointers, which properly belongs to the next course.

### 9.3. Identifier conventions

Every large company has its conventions for choosing identifiers. We have not enforced any particular convention. Some companies have conventions with which we violently disagree (such as "identifiers for integer variables must start with "i" or "j" or "k" ...).

Identifier names should certainly be meaningful. They will therefore consist of several words. Some users compose with capitalised initial letter for each word (such as `TotalCostPerHour` for example), while other prefer `total_cost_per_hour` with underscores separating the components.

Older compilers sometimes limited the length of identifiers to eight significant characters (if the identifier was longer, only the first eight characters were significant) but there is no known C++ compiler with such a restriction. That is an advantage of a modern language.

### 9.4. The use of #define

There are occasions where the "#define" facility of the C++ preprocessor can be useful.

If you write

```
#define MAX 150
```

near the top of a program, then everywhere that the identifier "MAX" occurs in the text, it is substituted by the string "150" before the program is passed to the compiler.

The string which is substituted can be absolutely any string of characters. Thus if you write

```
#define NU "The University of Nottingham"
```

the given string (including the quote symbols) will be substituted at every occurrence. This will result in many occurrences of the actual string. For a specific value such as this, you would normally use a global constant in C++.

If you define

```
#define TOTAL (n_small + n_medium + n_large)
```

then every occurrence of "TOTAL" will be substituted by the given expression, which will be compiled and re-evaluated at every run-time encounter in the program.

Consider

```
#define EVER ;;
```

and

```
for ( EVER ) { ...
```

### 9.5. Debugging your program

What can I say? This is a skill you must develop yourself.

Develop your program in small steps. Don't write 200 lines of code and expect it all to work and be easy to debug. Create the program in stages, testing each stage as you develop it. You will learn with experience how much it is wise to add at a time.

Put additional printing statements into the program until you are sure that it works. This way you can check intermediate results, and convince yourself that results are correct.

Re-use previously developed and test code wherever possible. This may mean your own code, but more often means the use of library code. It is usually worth the effort of looking up library functions for many operations.

## 9.6. Quotations

Your programs are intended to be useful, and to be economic to develop and run. In this context

Perfection is the enemy of quality.<sup>1</sup>

© Eric Foxley December 5, 1996

### Appendix: Programming standards

I have appended for interest a summary of the C++ programming standards used by one industrial company, "Ellemtel". The actual description of the standards occupies 82 pages, this is just a one-line summary of each of them. We hope that one day Ceilidh will enforce ALL of them!

They are divided into "Rules" (general application), "Recommendations" (recommended, not mandatory) and "Portability Rules" (only needed for applications which need to be portable, which for any large organisation would mean all applications). Summary of Rules

Rule 0 Every time a rule is broken, this must be clearly documented.

Rule 1 Include files in C++ always have the file name extension ".h".

Rule 2 Implementation files in C++ always have the file name extension ".cc".

Rule 3 Inline definition files always have the file name extension ".icc".

Rule 4 Every file that contains source code must be documented with an introductory comment that provides information on the file name and its contents.

Rule 5 All files must include copyright information.

Rule 6 All comments are to be written in English.

Rule 7 Every include file must contain a mechanism that prevents multiple inclusions of the file.

Rule 8 When the following kinds of definitions are used (in implementation files or in other include files), they must be included as separate include files:

- classes that are used as base classes,
- classes that are used as member variables,
- classes that appear as return types or as *argument types* in function/member function prototypes.
- function prototypes for functions/member functions used in inline member functions that are defined in the file.

Rule 9 Definitions of classes that are only accessed via pointers (\*) or references (&) shall not be included as include files.

Rule 10

**Never** specify relative UNIX names in **#include** directives.

Rule 11

Every implementation file is to include the relevant files that contain:

- declarations of types and functions used in the functions that are implemented in the file.
- declarations of variables and member functions used in the functions that are implemented in the file.

Rule 12

The identifier of every globally visible class, enumeration type, type definition, function, constant, and variable in a class library is to begin with a prefix that is unique for the library.

Rule 13

The names of variables, constants, and functions are to begin with a lowercase letter.

Rule 14

The names of abstract data types, structures, **typedefs**, and enumerated types are to begin with an uppercase letter.

Rule 15

In names which consist of more than one word, the words are written together and each word that

follows the first is begun with an uppercase letter.

Rule 16

Do not use identifiers which begin with one or two underscores (‘\_’ or ‘\_\_’).

Rule 17

A name that begins with an uppercase letter is to appear directly after its prefix.

Rule 18

A name that begins with a lowercase letter is to be separated from its prefix using an underscore (‘\_’).

Rule 19

A name is to be separated from its suffix using an underscore (‘\_’).

Rule 20

The public, protected, and private sections of a class are to be declared in that order (the public section is declared before the protected section which is declared before the private section).

Rule 21

No member functions are to be defined within the class definition.

Rule 22

Never specify public or protected member data in a class.

Rule 23

A member function that does not affect the state of an object (its instance variables) is to be declared **const**.

Rule 24

If the behaviour of an object is dependent on data outside the object, this data is not to be modified by const member functions.

Rule 25

A class which uses “new” to allocate instances managed by the class, must define a **copy constructor**.

Rule 26

All classes which are used as base classes and which have virtual functions, must define a virtual destructor.

Rule 27

A class which uses “new” to allocate instances managed by the class, must define an **assignment operator**.

Rule 28

An assignment operator which performs a destructive action must be protected from performing this action on the object upon which it is operating.

Rule 29

A public member function must never return a non-const reference or pointer to member data.

Rule 30

A public member function must never return a non-const reference or pointer to data outside an object, unless the object shares the data with other objects.

Rule 31

Do not use unspecified function arguments (ellipsis notation).

Rule 32

The names of formal arguments to functions are to be specified and are to be the same both in the function declaration and in the function definition.

Rule 33

Always specify the return type of a function explicitly.

Rule 34

A public function must never return a reference or a pointer to a local variable.

- Rule 35  
Do not use the preprocessor directive **#define** to obtain more efficient code; instead, use inline functions.
- Rule 36  
Constants are to be defined using **const** or **enum**; never using **#define**.
- Rule 37  
Avoid the use of numeric values in code; use symbolic values instead.
- Rule 38  
Variables are to be declared with the smallest possible *scope*.
- Rule 39  
Each variable is to be declared in a separate declaration statement.
- Rule 40  
Every variable that is declared is to be given a value before it is used.
- Rule 41  
If possible, always use initialization instead of assignment.
- Rule 42  
Do not compare a pointer to NULL or assign NULL to a pointer; use 0 instead.
- Rule 43  
Never use *explicit* type conversions (casts).
- Rule 44  
Do not write code which depends on functions that use implicit type conversions.
- Rule 45  
Never convert pointers to objects of a derived class to pointers to objects of a virtual base class.
- Rule 46  
Never convert a **const** to a not-**const**.
- Rule 47  
The code following a **case** label must always be terminated by a **break** statement.
- Rule 48  
A **switch** statement must always contain a **default** branch which handles unexpected cases.
- Rule 49  
Never use **goto**.
- Rule 50  
Do not use **malloc**, **realloc** or **free**.
- Rule 51  
Always provide empty brackets (“[]”) for **delete** when deallocating arrays.

### Summary of Recommendations

- Rec. 1 Optimize code only if you know that you have a performance problem. Think twice before you begin.
- Rec. 2 If you use a C++ compiler that is based on Cfront, always compile with the +w flag set to eliminate as many warnings as possible.
- Rec. 3 An include file should not contain more than one class definition.
- Rec. 4 Divide up the definitions of member functions or functions into as many files as possible.
- Rec. 5 Place machine-dependent code in a special file so that it may be easily located when porting code from one machine to another.
- Rec. 6 Always give a file a name that is unique in as large a context as possible.
- Rec. 7 An include file for a class should have a file name of the form <class name> + extension. Use uppercase and lowercase letters in the same way as in the source code.

- Rec. 8 Write some descriptive comments before every function.
- Rec. 9 Use `//` for comments.
- Rec. 10  
Use the directive `#include "filename.h"` for user-prepared include files.
- Rec. 11  
Use the directive `#include <filename.h>` for include files from libraries.
- Rec. 12  
Every implementation file should declare a local constant string that describes the file so the UNIX command `what` can be used to obtain information on the file revision.
- Rec. 13  
Never include other files in an `.icc` file.
- Rec. 14  
Do not use typenames that differ only by the use of uppercase and lowercase letters.
- Rec. 15  
Names should not include abbreviations that are not generally accepted.
- Rec. 16  
A variable with a large scope should have a long name.
- Rec. 17  
Choose variable names that suggest the usage.
- Rec. 18  
Write code in a way that makes it easy to change the prefix for global identifiers.
- Rec. 19  
Encapsulate global variables and constants, enumerated types, and typedefs in a class.
- Rec. 20  
Always provide the *return type* of a function explicitly.
- Rec. 21  
When declaring functions, the leading parenthesis and the first argument (if any) are to be written on the same line as the function name. If space permits, other arguments and the closing parenthesis may also be written on the same line as the function name. Otherwise, each additional argument is to be written on a separate line (with the closing parenthesis directly after the last argument).
- Rec. 22  
In a function definition, the *return type* of the function should be written on a separate line directly above the function name.
- Rec. 23  
Always write the left parenthesis directly after a function name.
- Rec. 24  
Braces (“{ }”) which enclose a block are to be placed in the same column, on separate lines directly before and after the block.
- Rec. 25  
The flow control primitives `if`, `else`, `while`, `for` and `do` should be followed by a block, even if it is an empty block.
- Rec. 26  
The dereference operator `*` and the address-of operator `&` should be directly connected with the type names in declarations and definitions.
- Rec. 27  
Do not use spaces around `.'` or `->`, nor between unary operators and operands.
- Rec. 28  
Use the `c++` mode in GNU Emacs to format code.

- Rec. 29  
Access functions are to be inline.
- Rec. 30  
Forwarding functions are to be inline.
- Rec. 31  
Constructors and destructors must not be inline.
- Rec. 32  
Friends of a class should be used to provide additional functions that are best kept outside of the class.
- Rec. 33  
Avoid the use of global objects in constructors and destructors.
- Rec. 34  
An assignment operator ought to return a *const* reference to the assigning object.
- Rec. 35  
Use operator overloading sparingly and in a uniform manner.
- Rec. 36  
When two operators are opposites (such as `==` and `!=`), it is appropriate to define both.
- Rec. 37  
Avoid inheritance for parts-of relations.
- Rec. 38  
Give derived classes access to class type member data by declaring protected access functions.
- Rec. 39  
Do not attempt to create an instance of a class template using a type that does not define the member functions which the class template, according to its documentation, requires.
- Rec. 40  
Take care to avoid multiple definition of overloaded functions in conjunction with the instantiation of a class template.
- Rec. 41  
Avoid functions with many arguments.
- Rec. 42  
If a function stores a pointer to an object which is accessed via an argument, let the argument have the type pointer. Use reference arguments in other cases.
- Rec. 43  
Use constant references (`const &`) instead of call-by-value, unless using a pre-defined data type or a pointer.
- Rec. 44  
When overloading functions, all variations should have the same semantics (be used for the same purpose).
- Rec. 45  
Use `inline` functions when they are really needed.
- Rec. 46  
Minimize the number of temporary objects that are created as return values from functions or as arguments to functions.
- Rec. 47  
Avoid long and complex functions.
- Rec. 48  
Pointers to pointers should whenever possible be avoided.
- Rec. 49  
Use a `typedef` to simplify program syntax when declaring function pointers.

- Rec. 50  
The choice of loop construct (**for**, **while** or **do-while**) should depend on the specific use of the loop.
- Rec. 51  
Always use **unsigned** for variables which cannot reasonably have negative values.
- Rec. 52  
Always use inclusive lower limits and exclusive upper limits.
- Rec. 53  
Avoid the use of **continue**.
- Rec. 54  
Use **break** to exit a loop if this avoids the use of flags.
- Rec. 55  
Do not write logical expressions of the type **if (test)** or **if (!test)** when **test** is a pointer.
- Rec. 56  
Use parentheses to clarify the order of evaluation for operators in expressions.
- Rec. 57  
Avoid global data if at all possible.
- Rec. 58  
Do not allocate memory and expect that someone else will deallocate it later.
- Rec. 59  
Always assign a new value to a pointer that points to deallocated memory.
- Rec. 60  
Make sure that fault handling is done so that the transfer to exception handling (when this is available in C++) may be easily made.
- Rec. 61  
Check the fault codes which may be received from library functions even if these functions seem foolproof.

### Summary of Portability Recommendations

- Port.Rec. 1  
Avoid the direct use of pre-defined data types in declarations.
- Port.Rec 2  
Do not assume that an **int** and a **long** have the same size.
- Port.Rec 3  
Do not assume that an **int** is 32 bits long (it may be only 16 bits long).
- Port.Rec 4  
Do not assume that a **char** is **signed** or **unsigned**.
- Port.Rec 5  
Always set **char** to **unsigned** if 8-bit ASCII is used.
- Port.Rec 6  
Be careful not to make type conversions from a “shorter” type to a “longer” one.
- Port.Rec 7  
Do not assume that pointers and integers have the same size.
- Port.Rec 8  
Use explicit type conversions for arithmetic using signed and unsigned values.
- Port.Rec 9  
Do not assume that you know how an instance of a data type is represented in memory.

Port.Rec 10

Do not assume that **longs**, **floats**, **doubles** or **long doubles** may begin at arbitrary addresses.

Port.Rec 11

Do not depend on underflow or overflow functioning in any special way.

Port.Rec 12

Do not assume that the operands in an expression are evaluated in a definite order.

Port.Rec 13

Do not assume that you know how the invocation mechanism for a function is implemented.

Port.Rec 14

Do not assume that an object is initialized in any special order in constructors.

Port.Rec 15

Do not assume that static objects are initialized in any special order.

Port.Rec 16

Do not write code which is dependent on the lifetime of a temporary object.

Port.Rec 17

Avoid using shift operations instead of arithmetic operations.

Port.Rec 18

Avoid pointer arithmetic.

## References

1. M Tvrđiková and J Tvrđík, *Human Factors and the Design of Interactive Applications*, December 1993. Proceedings of the International Conference on Computer Based Learning in Science, Vienna

## Chapter 10: Pointers

Pointers are not included in this course, but you are welcome to read these notes. There are no examples or exercises for this unit. Pointers are dealt with fully in the next C++ course.

Pointers are for making direct reference to stored objects, without necessarily having to refer to them by identifier.

### 10.1. Simple pointers

```
int i, *pointer;
    // "pointer" is a variable
    // for storing pointer to int
pointer = &i;
    // "pointer" assigned
    // the address of i
*pointer = 2;
    // "int" pointed at by "pointer"
    // is assigned the value 2
```

The (monadic) operator "&" is "address of", while "\*" is "object pointed at by". The variable name is "pointer" (no star).

Pointers are typed by the object they point at; you can't assign the address of a **long int** into a pointer to an **int**.

#### Note well

When you declare a pointer, it is not initialised to point at anything, unless you set it. To declare

```
int *point;
```

and to then use

```
*point = ....
```

will generate an (unpredictable) error. If you initialise the value by

```
int total;
int *point = &total;
```

```
*point = ....
```

then all is OK.

#### Note

The notation for an initialised pointer declaration

```
int *point = &total;
```

is perfectly acceptable with the meaning of

```
int *point; point = &total;
```

but can be confusing. In the abbreviated version, we are declaring “\*point” but assigning to “point”. Although it is written “\*point = ...” it actually means “point = ...”.

#### 10.1.1. Pointers and arrays

```
int array[ 10 ], *arr_pt;

arr_pt = &array[ 0 ];
arr_pt = array; // identical,
// since "array" is pointer to "array[0]"
// *arr_pt is now equivalent to array[0].
```

```

arr_pt++;
// increment arr_pt
// to point at next object

*arr_pt = 2; // assigns to array[1]

*( arr_pt + 3 ) = 2; // assigns to array[4]

```

In general "<pointer> plus-or-minus <integer>" gives a pointer; and "<pointer> minus <pointer>" gives an integer.

### 10.1.2. Pointers for loops

To sum the elements of an initialised array, use a terminating zero.

```

int array[] = { 1, 2, 3, 5, 7, 11, 0 };
int *point, sum = 0;

for( point = array; *point; point++ ) {
// the "*point" means "*point != 0"
    sum += *point;
} // for point in array
cout << "total << sum << "\n";

```

To count characters in an array of characters terminated by a '.' use:

```

char buffer[ 100 ], *pc;
int e_count = 0;

for( pc = buffer; *pc != '.'; pc++ ) {
    if ( *pc == 'e_count' ) {
        e_count++;
    }
} // for pc in buffer

```

### 10.1.3. Priorities

The expression "\*p++" is interpreted as "(p++)", i.e. increment "p", deliver what it previously pointed at.

Whereas "(\*p)++" increments whatever "p" points at. Given

```
char c, *p, a[10];
```

then

```
p = a; // point p at a[0]
c = *p++;
```

is equivalent to

```
{ c = *p; p++; }
// now c is a[0], p points at a[1]
```

but

```
p = a; // point p at a[0]
c = (*p)++;
```

is equivalent to

```
{ c = a[0]; a[0]++; }
// now p still points at a[0]
```

## Some pointer examples

We will give two simple example programs, each in two forms, firstly using subscripts in the arrays, then using pointers.

To read two arrays of *floats*, and form their scalar product, first using subscripts:

```
// scalar product

main() {

    float a[10], b[10];
    float scalar_prod = 0.0;
    int i;
    for ( i = 0; i < 10; i++ ) {
        cin >> a[i];
    }
    for ( i = 0; i < 10; i++ ) {
        cin >> b[i];
    }
    for ( i = 0; i < 10; i++ ) {
        scalar_prod += a[i] * b[i];
    }
    cout << "Scalar prod "
         << scalar_prod << "\n";
}
```

Now using pointers:

```
main() {
    float a[10], b[10];
    float scalar_prod = 0.0;
    int i;
    float *pa, *pb;

    for (
        i = 0, pa = a; i < 10; i++
    ) {
        cin >> *pa++;
    }
    for (
        i = 0, pb = b; i < 10; i++
    ) {
        cin >> *pb++;
    }
    for (
        i = 0, pa = a, pb = b; i < 10; i++
    ) {
        scalar_prod += *pa++ * *pb++;
    }
    cout << "Scalar prod "
         << scalar_prod << "\n";
}
```

To print the first few lines of a Pascal triangle, first using subscripts:

```
// Print Pascal's triangle
const int MAX = 100;

main ()    {

    int old[ MAX ], new[ MAX ], i, l;
    int n = 5;
    old[0] = 1;

    for ( l = 1; l < n; l++ ) {
        new[0] = 1;
```

```

    for ( i = 1; i <= l; i++ ) {
        new[i] = old[i-1] + old[i];
    }
    for ( i = 0; i <= l; i++ ) {
        old[i] = new[i];
        cout << new[ i ] << " ";
    }
    cout << "\n";
} // for l = 0 to n
} // end main

```

Then using pointers:

```

const int MAX = 6;

main () {

    int old[ MAX ], new[ MAX ], l;
    int *pold, *pnew;
    *old = 1;

    for ( l = 1; l < MAX - 1; l++ ) {
        *new = 1;
        for (
            pold = old+l, pnew = new+l;
            *pnew++ = *(pold-1) + *pold;
            pold++
        ) {
            ;
        }
        for (
            pold = old, pnew = new; *pnew;
        ) {
            cout << *pnew << " ";
            *pold++ = *pnew++;
        }
        cout << "\n";
    } /* for l = 0 to n */
}

```

#### 10.1.4. Double pointers (pointers to pointers)

We will assume that we are given an int variable "i" and an array "a" of 9 ints

```
int i, a[9];
```

We then declare an array "pa" of 3 pointers to ints, either as

```
int *pa[] = { a, a + 3, a + 6 };
```

or exactly equivalent

```
int *pa[] = { &a[0], &a[3], &a[6] };
```

We then declare a single pointer to pointer to int, either as

```
int **ppa; ppa = pa;
```

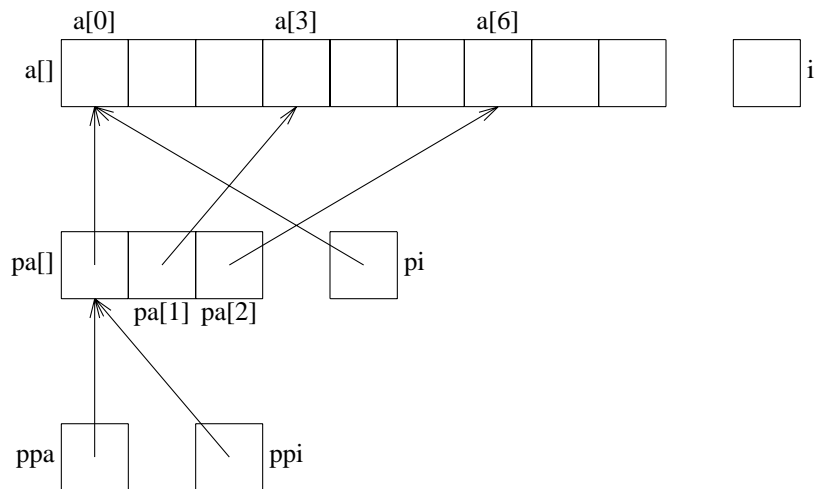
or exactly equivalent

```
int **ppa; ppa = &pa[0];
```

Now add two more declarations of general pointers to int and pointer to pointer to int.

```
int *pi = a; // i.e. pi = a
int **ppi = ppa;
```

The above data can be represented pictorially as follows.



I have laid them out so that the top row are all “int”s, the next row “pointers to int” or “int \*”, the bottom row “pointers to pointers to int” or “int \*\*”.

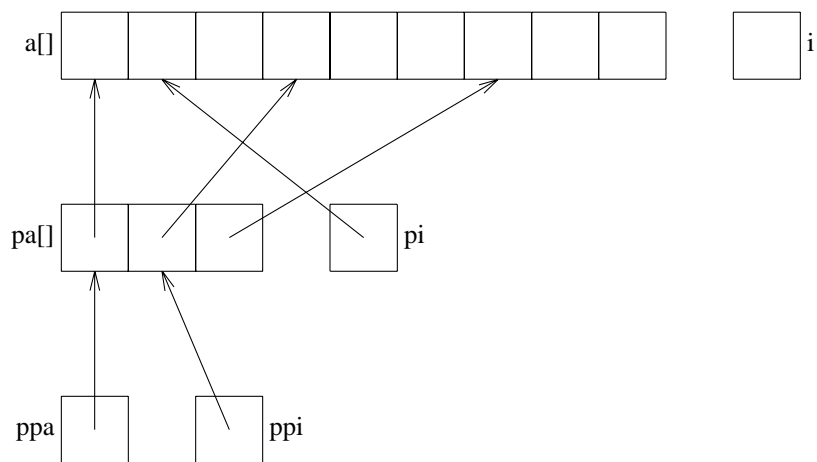
We can now get the value of "pa[0]" by either "`*ppa`" or "`*ppi`".

Using double pointers, we can refer to "a[0]" as "`**ppa`" or "`**ppi`", or with single pointers as "`*pa[0]`" or "`*pi`".

If we execute

```
ppi++;
pi++;
```

we get the revised picture



The value of "`*ppi`" is now "`pa[1]`", "`**ppi`" is now "`a[3]`", and "`*pi`" is now "`a[1]`".

Note the importance of order of evaluation in

```
pi = *ppi++;
```

which is equivalent to

```
pi = *ppi; ppi++;
```

and

```
i = *++pi;
```

which is equivalent to

```
pi++; i = *pi;
```

and

```
pi = (*ppi)++;
```

which is equivalent to

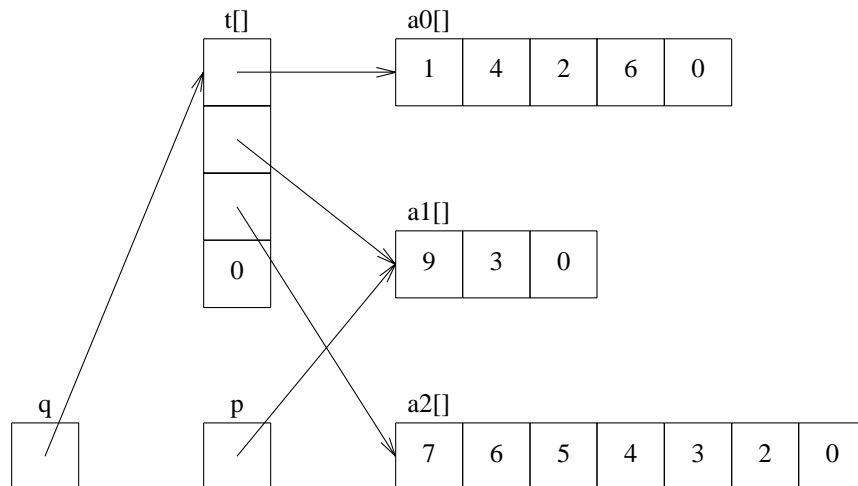
```
pi = pa[1]; pa[1]++;
```

## Ragged Arrays

Another way of obtaining a structure similar to the above is as follows.

```
int a0[] = { 1, 4, 2, 6, 0 };
int a1[] = { 9, 3, 0 };
int a2[] = { 7, 6, 5, 4, 3, 2, 0 };
int *t[] = { a0, a1, a2, 0 };
// t can be used like a [3][?] array
```

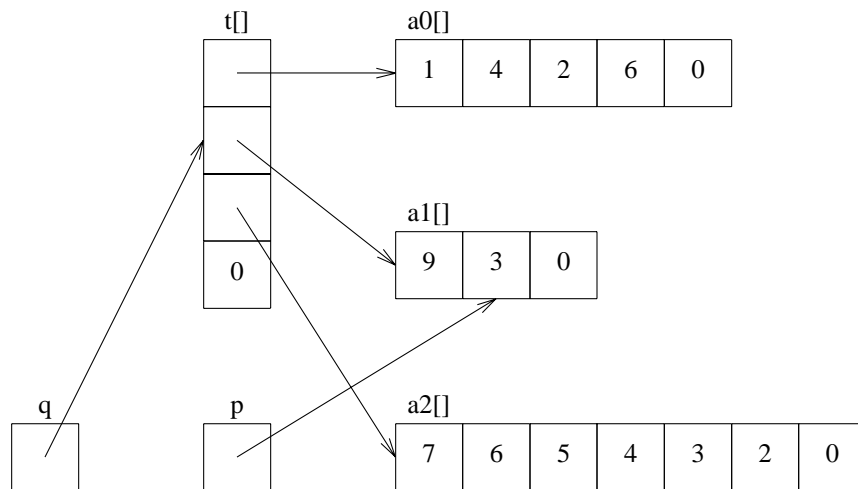
```
int *p, **q;
p = t[1];
// p points to the start of a1
q = t;
// q points to value of a0
```



```
*p = ...
// *p is a1[0]
**q = ...
// *q is t[0] or a0
// **q is a0[0]
```

If we execute

```
p++;
// *p is now a1[1]
.. = *q++;
// delivers t[0]
// but *q is now t[1]
// **q is now a1[0]
```



```

>(*q++)++;
// **q is now t[2][1]

```

A typical for loop might now be:

```

for( q = t; *q; q++ ) {
    // scan q through t
    for( p = *q; *p; p++ ) {
        // scan *p over elements of t
        ...
    } // p loop
} // q loop

```

## 10.2. Strings

Strings are handled specially in C++.

The denotation "eric" (including the quote signs) is valid anywhere (not just in global), and delivers a pointer to a preassigned string of characters

```
{ 'e', 'r', 'i', 'c', 0 }
```

Note the terminating zero. Thus we can have

```
char *name;
name = "eric";
```

or

```
char *name = "eric";
```

Because of the terminating zero convention we can write

```
char *p;

for( p = name; *p; p++ ) {
    ... *p ...
}
```

### 10.2.1. Strings in C++

All strings in C++ are assumed to be character arrays which end with a null, for example in

```
printf( "His name is %s", name );
```

the variable "name" must be a character pointer, and characters are printed until a null is found. (What's the difference between "printf( name )" and "printf( "%s", name )"?)

To copy the string pointed at by "q" into the area pointed at by "p" (they must both be of type "char \*"), we use

```
while ( *p++ = *q++ )
    ;
```

To compare two strings (until a null is encountered) pointed at by p and q

```
while( *p ) {
    if ( *p++ != *q++ ) {
        return 0;
    }
    // return 0 is FALSE return
if ( *q ) {
    return 0;
}
    // if not at end of string "q"
return 1;
    // return 1 is TRUE exit
```

### 10.2.2. String libraries

The library functions for strings include

```
strcmp(a, b ) // compare two strings
strcpy(a, b ) // copy b to a
strlen(a)     // deliver the length
strcat(a, b ) // concatenate b onto end of a
strncmp(a, b, n) // compare at most n characters
etc
```

See the on-line manual "man string" for further details.

### 10.2.3. The "system" routine (Unix only)

The call

```
system( "who" );
```

causes the program to be suspended while the command "who" is executed (with its standard input and output connected to the terminal), after which program execution continues. The "system" call actually calls a shell to interpret the string.

```
system( "a.out" ); // infinitely recursive, nasty!

cout << "The date is "; system( "date" );

system( "g++ prog.C; a.out < data_file; rm a.out" );

system( "rm *.o" );

strcpy( p, "ed " ); strcat( p, file );
system( p ); // edit the named file

system( "stty cbreak" );
...
system( "stty -cbreak" );
```

```
system( "cd /tmp; ....; ...." );
```

### 10.2.4. Arrays of strings

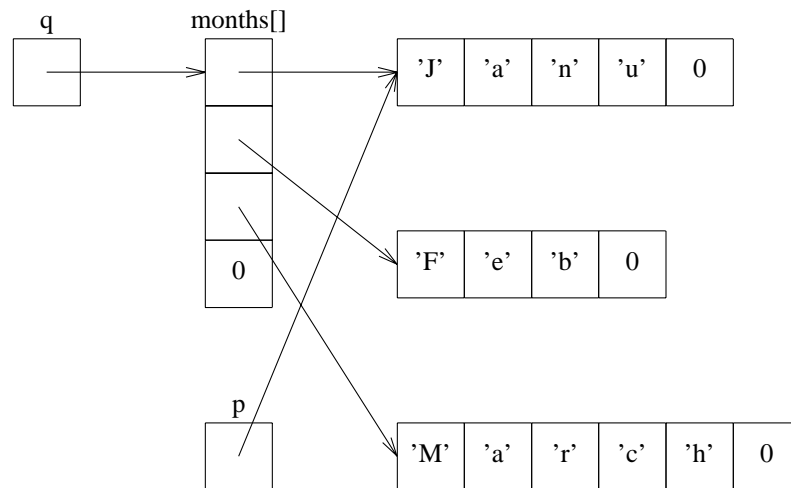
Arrays of strings (of assorted lengths) are often necessary, in looking up command names, people's names, names of months, ... They are declared as

```
char *months[] = {
    "January",
    "February",
    "March",
    "April",
    "May",
    "June",
    0 };
```

This sets up an array of pointers to characters, with a null pointer at the end of each string **and** at the end of the array of pointers. Now add

```
char **q, *p;

q = months; p = *q;
```



Boxes at right are "char".

Boxes in middle are "pointer to char" or "char \*".

Box at left are "pointer to pointer to char" or "char \*\*".

### 10.2.5. Using the "char \*[]"

We can do the following:

```
// print the names
q = months;
while( *q ) {
    cout << *q++, "\n";
}

// look for a name
while( *q && strcmp( "April", *q++ ) ) {
    ;
}
```

```

}
if ( *q == 0 ) {
    ... // not found
} else {
    q--; // *q is the one found    ...
}

q = months;
// *q points to "January",
// **q is 'J'
q++;
// *q points to "February",
// **q is 'F'
(*q)++;
// *q points to "ebruary",
// **q is 'e'

```

### 10.2.6. sprintf and sscanf

The same as printf and scanf, but an extra first parameter, a "char \*", used instead of actual i/o.

```

sprintf( s, "value is %d", rate );
// sets in s "value is 123"
// s must point at a large enough space

sscanf( s, "%d", &i );
// looks for a decimal value in s

sprintf( s, "cc %s.c", progname );
system( s );

```

### 10.3. Arguments to the program.

When a command (calling a program) is issued with arguments, as in

```
a.out this that other
```

the system generates an integer `mIargcm-I` counting the arguments (four in this case) and an object `mIargvm-I` set up to represent the command line, as in

```

int argc = 4;
char *argv[] = {
    "a.out",
    "this",
    "that",
    "other",
    0
};

```

("argc" means argument count, "argv" means argument values.) Both of these data items can be picked up from the main program.

#### 10.3.1. Accessing the arguments

Use

```

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

```

The parameter "argv" is the assembled array of strings (including the command name, "a.out" in this case), while "argc" is an integer, the number of (non-null) entries. ("argc" is not essential, since its value could be found by scanning through "argv"; it is just convenient; you must include it.) The data in the variables is set up by the system.

### 10.3.2. The echo command

To print out the arguments (which is what the "echo" command does), the program would be

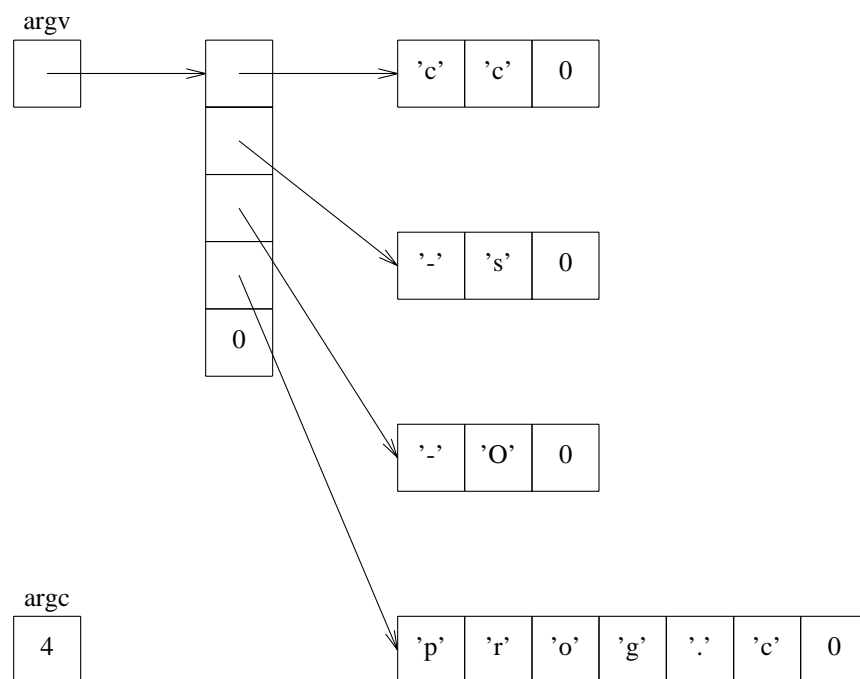
```
main( int argc, char *argv[] ) {
    int i;
    for ( i = 1; i < argc; i++ ) {
        cout << "Arg no " << i << " is " << argv[i] << "\n";
        i, argv[ i ] );
    } // end for i
} // end main
```

or, with a more pointer-oriented method,

```
main( int argc, char *argv[] ) {
    argv++;
    while ( *argv ) {
        cout << "Next arg is " << *argv++ << "\n";
    } // end while
} // end main
```

### 10.3.3. Unix-type flags

To work out the code to pick up the flags in "g++ -s -O prog.c", we first look at the "argv" data as set up by the system.

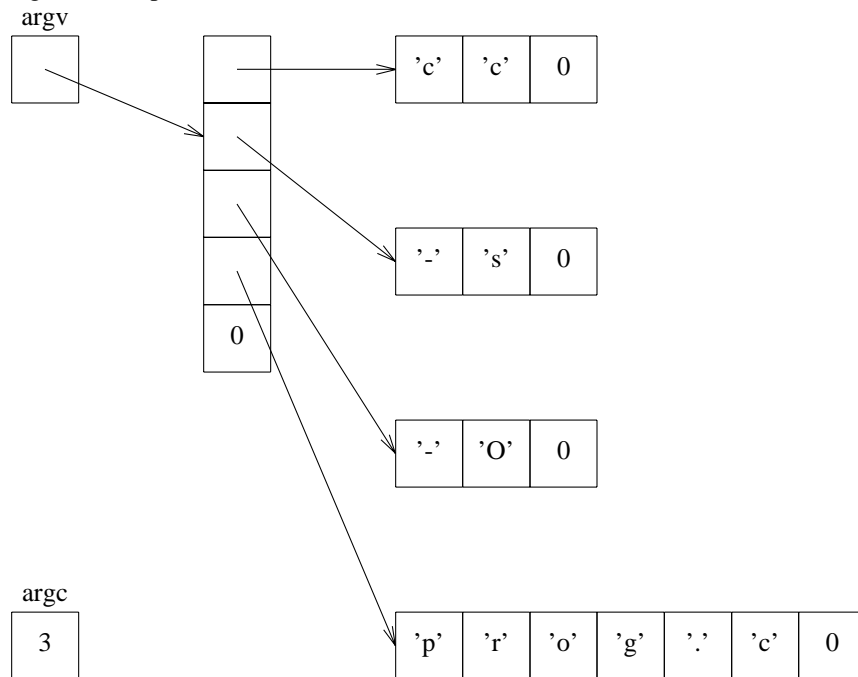


The code to look for flags would be:

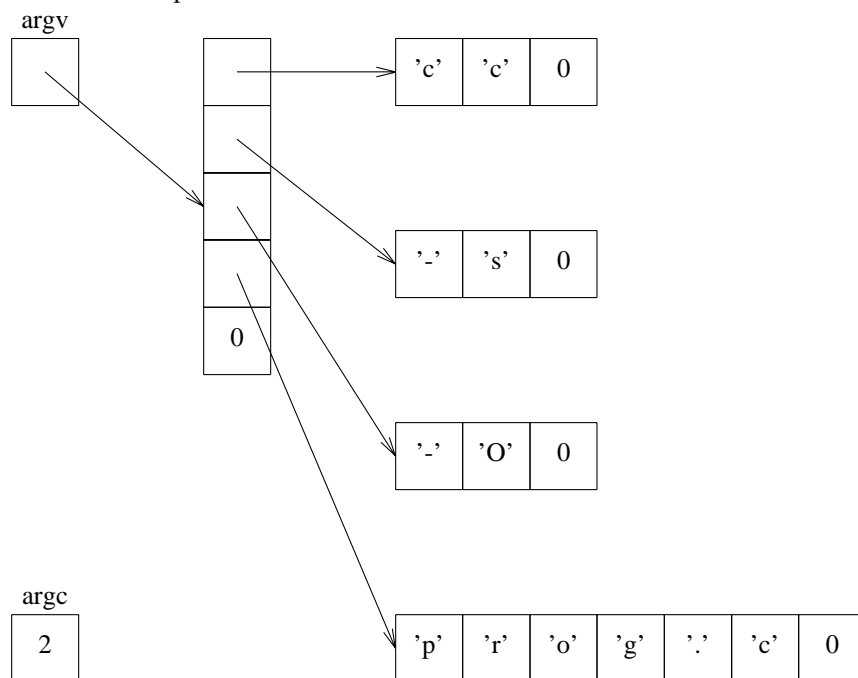
```
while( argc > 1 && argv[1][0] == '-' ) {
    switch( argv[1][1] ) {
        case 's' : strip = 1; break;
        case 'O' : optimise = 1; break;
        default :
            cerr << "Don't recognise ...";
    }
    argc--; argv++;
}
```

You could, of course, do all this with pointers instead of subscripts.

After a single "argc--; argv++;" the picture becomes



After the loop is finished we have the picture



### 10.3.4. Other flags

The above code handles separate flags such as "g++ -s -c prog". If the flags occur several together, as in "ls -lrt" use

```

if( argc>1 && argv[1][0] == '-' ) {
    i = 1;
    while( c = argv[1][i++] ) {
        switch( c ) ... as above
    }
}

```

### 10.3.5. Values from arguments

To read numeric values from arguments, use the "atoi" (ASCII to integer) function as follows.

```
if ( argc > 2 ) {
    rate = atoi( argv[1] );
    hours = atoi( argv[2] );
}
```

or

```
argv++; argc--;
if ( argc > 0 ) {
    rate = atoi( *argv++ ); argc--;
}
if ( argc > 0 ) {
    hours = atoi( *argv++ ); argc--;
}
```

### 10.3.6. The environment

A third parameter of the same type as "argv" can be used to pick up your UNIX environment ...

```
main( int argc, char *argv[], char *envp[] ) {
    ....;
}
```

where "envp" contains pointers to strings such as "USER=ef", "TERM=vt100", and so on, and is null terminated. The library function "getenv( "USER" )" then delivers a pointer to the string following "USER=" if it can find one starting "USER=".

## 10.4. Pointers and structures

### 10.4.1. Basics

Pointers can be used with structures, using all the techniques explained above. There is one significant extension relating to the operator "->".

```
date today, week[7], *p;
p = &today;
p = week;
(*p).name = "Mon";
p -> name = "Mon";
p++;
p = week + 6;

p = (struct date *) 0177756; // cast
```

What is meant to look like an arrow is formed of a '-' and a '>'. It is preceded by a **pointer to** a structure, and followed by a fieldname.

### 10.4.2. Structures for lists

```
struct cell {
    int data; struct cell *next;
};
for( p = head; p != NULL; p = p->next ) {
    ....;
}
```

### 10.4.3. Planting Structures

```
struct disc {
    int drive, sector, block;
    char error, mode;
    unsigned data;
};

disc *d;
d = (struct disc *) 01777756;

if ( d->error < 0 ) ...
d->sector = ...
```

© Eric Foxley 1992