9. Arrays

Suppose we want to write a program which will read three integers and calculate their sum. There are two approaches which could be taken, each with something different to offer. Firstly, our program could contain this fragment.

```
int sum, num1, num2, num3;

cin >> num1 >> num2 >> num3;
sum = num1 + num2 + num3;
```

In this approach each of the three numbers is stored and available for future use.

In the second approach, we notice that we want to do the same thing with each number and hence the code perhaps could be reduced by including one set of the instructions inside a loop. Here is a solution using the looping approach.

```
int sum, num, i;

sum = 0;
for (i=0; i<3; i++)
{
     cin >> num;
     sum += num;
}
```

In this case only the last value for `num` is stored and available for future use.

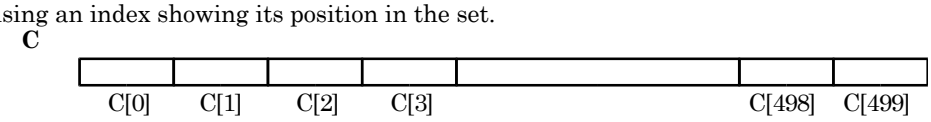
What happens if we extend this problem to one in which we wish to find the sum of 500 numbers? Obviously the second form of the program is more suitable than the first.

But what if we want not only to store their sum but also to store each of the numbers for future use? There must be a simple way of declaring 500 memory locations to be used to store these 500 values and hence make their values available later in the program.

C++, like all high-level languages, allows the user to declare a collection of variables of the same type and provides a simple mechanism to access them.

Consider a sequence of adjacent memory locations each capable of holding a value of the same type. Give the set of locations a name such as C.

To access any one of the individual memory locations we can refer to it by the name of the collection and specify which one by using an index showing its position in the set.



Such a data structure is called an **array** and each of the locations is called an **element** of the array and is accessed by the array **index**.

Note that, unlike many other languages, C++ arrays are always numbered from 0 to one less than the specified size.

Like all data types, a variable of this kind needs to be declared and must indicate the name of the array and the number of consecutive values that will be used for the index (and thus the number of memory locations to be reserved) and the type of data which each element can hold remembering that they are all of the same type.

For our example above which required the reading of 500 integer values we need to reserve 500 memory locations, each capable of holding an integer value and provide a name by which all of the elements can be referred.

Here's how:

```
int sum, i, num[500];
sum = 0;
for (i=0; i<500; i++)
{
    cin >> num[i];
    sum += num[i];
}
```

In this example each of the elements of the array `num` is an integer variable. The general element, the ith element, is `num[i]`, and we can use this array element just as any other integer type variable. The subscript can be any

integer expression.

The general form of an array declaration is:

```
type name[size]
```

where *size* is an constant integer expression. The number of elements of an array must be constant and is fixed at declaration time.

Examples of array declarations:

```
const int NumRakeOff = 25;
float MonthlySales[12];
double RakeOff[NumRakeOff];
char Letters[10];
```

`MonthlySales` is an array which can hold 12 `float` values stored in variables named `MonthlySales[0]`, `MonthlySales[1]` up to `MonthlySales[11]`.

`RakeOff` is an array which can hold 25 `double` values in variables named `RakeOff[0]`, `RakeOff[1] ... RakeOff[24]`.

`Letters` is an array which can hold 10 `char` values in variables named `Letters[0]`, `Letters[1] ... Letters[9]`.

Because an individual element of an array is a simple variable, the operations which can be carried out on the elements of an array are exactly the same as those which can be carried out on a simple variable of that type.

```
if(MonthlySales[5] < MonthlySales[6])
     RakeOff[5] = (MonthlySales[0] + MonthlySales[11])/2.0;
```

Further approaches to the manipulation of arrays will be encountered later.

Let's return to our summation program and expand it to include other computations which might be required.

```
const int howmany = 500;

int Sum, i, Ave;
int Num[howmany], Difference[howmany];

/* Initialise variables */
for (i=0; i<howmany; i++)
     cin >> Num[i];

Sum = 0 ;
for (i=0; i<howmany; i++)
     Sum += Num[i];

Ave = Sum/howmany;
for (i=0; i<howmany; i++)
     Difference[i] = Num[i] - Ave;
```

Some points to note:
At some stage in the design process the programmer must decide how much memory is reasonable to allocate for each array. This is based on the number of values each array is **expected** to hold. Obviously one option is to declare very large arrays on the off chance that each array will then be sufficiently large to handle any data provided. This is **not** a good option as it is very wasteful of memory.

Note that if you try to access a memory location as part of an array but that subscript does not fall within the range for which memory has been reserved you will **not** get an error at compile time. For example

```
float B[20];
B[100] = 6.0;
```

will not result in an error, or even a warning. (A more common fault will be the referencing of `B[20]`.) In fact, the computer may not even detect an error at execution time, but it will be referencing a memory location not part of the array you have declared. Should this memory location not be in the memory space allocated to your program, you may get a program crash or some other unexpected result. (On computers with memory protection, such as UNIX, the program will crash.) It is up to the programmer to avoid such occurrences by correct allocation of array sizes and by knowing the index values expected in the program.

Note that in the previous example a `const` was declared to set the maximum size of the array. Using this method and including the constant in your structure, it is a simple matter to change the value of the constant without having to change the rest of the program.

A second "good programming practice" is to read in the number of elements which are to be used into the array. Provided this is combined with a check of the value read against the declared size of the array, a very general program can be constructed.

```
const int maxsize = 100;
int Sum, HowMany, i, Num[maxsize];

cin >> HowMany;
if (Howmany > maxsize)
{
     cout << "Too many values to be read. Only the first "
               << maxsize << " will be read." << endl;
     Howmany = maxsize;
}
Sum = 0;
for (i=0; i<HowMany; i++)
     cin >> Num[i];
```

9.1 Initialisation of arrays

We discussed earlier how we can initialise variables by incorporating an assignment within the declaration. Arrays can also be initialised using the form

type varname[*size*] = {$value_0$, $value_1$,...,$value_{size-1}$};

even though we cannot use a similar form of assignment for arrays in general. The list of values within the braces must be expressions of an acceptable type. So we can say

```
int DaysInMonth[12] = {31,28,31,30,31,30,31,
                                31,30,31,30,31};
float Heights[5] = {12.3,5.7,4.6,18.4,2.0};
char MonthLetter[12] ={'J','F','M','A','M','J',
                                'J','A','S','O','N','D'};
```

If the list is shorter than the size of the array, remaining elements will be initialized to zero. If the list is too long, a compiler error will be reported.

If the size of the array is omitted and the array is initialized, the size will be determined by the number of entries in the list of data values. For example

```
int LettersInMonthNames[] = {7,7,5,5,3,4,4,6,9,7,8,8};
```

would set the size of the array to 12.

Arrays can also be specified as named constants. The arrays `MonthLetter` and `LettersInMonthName` could be so declared, as they do not change in value.

9.2 Arrays of characters

One important application of arrays is handling strings of characters. We saw earlier that a character array is declared in the same manner as other basic data types. That is

```
char varname[size];
```

When we introduced constants, we described the character string (C-string) constant in the form "characters" where the characters appearing between the double quotes are terminated in memory by a zero byte called the null (control) character or NUL or '\0'. But we didn't indicate how to have a named character string constant. That's because we needed the character string variable to exist. Now we have it (almost) the character array. That is, as well as being able to access individual character elements of a character array, we can manipulate strings by storing them in character arrays.

In fact, a character array can be initialized by assigning a character string constant to it, as in

```
char Month[8] = "January";
char ProgrammerName[] = "James Bond";
```

But we must remember that the strings do contain the null character at their end. Thus, in the second example, the array would be allocated 13 bytes.

9.2.1 Input/Output of strings (Part 1)

We have already seen that type char is one of the basic types that can be handled by iostreams. That is

```
char FirstInitial;

cin >> FirstInitial;
cout << FirstInitial << endl;
```

To input or output a string, we can just use the array name, without a subscript.
So

```
char FirstInitial, Surname[20];

cin >> FirstInitial >> Surname;
cout << FirstInitial << ". " << Surname << endl;
```

However, although strings containing spaces can be output, if we try to use cin to read in a string, the reading is terminated by whitespace. Thus

```
char FullName[50];
cin >> FullName;
```

with input

```
James Bond<ret>
```

would only get "James", with a '\0' at the end of the 5 characters. (The quotes shown here are not part of the character array contents.) Alternately, if there are more than 49 characters in the input stream, the character array will overflow (use elements not declared) with the usual consequences.

10. Functions

As we have seen so far, C++ programs essentially involve the manipulation of variables (and constants) in the determination of the values of other variables. Back in our mathematical past, we encountered similar concepts of things which produce values based on the values of variables. These are called **functions**. For example

$$f(x) = x^2$$

is a function which says

"given the value of the variable x, we get the value of f(x) by squaring the value of x"

We can then create new functions in expressions such as

$$f(x+1)$$

where we say 'find the value of x+1 and then square it'. Or we could interpret this as using the 'symbol' x+1 so that the function results in an answer of $(x+1)^2$. While we can do such symbolic things in mathematics, where we have a language of symbols, called algebra, in computing we are restricted to the concept of the function which manipulates (numerical) values.

This is how we might create the above function, the square function, in C++.

We first consider what is involved in determining the value of the function the computation of x^2. We can do that in C++ using the expression

```
x*x
```

So, given x, our function must **return** to us, the value of x*x. Here is the structure of such a function.

```
float f(float x)
{
     return x*x;
}
```

The first line specifies that the function `f` has, as its **argument**, the `float` value `x`, and has a type of `float` itself the value of the function. Within the braces are the statements which calculate that function, and **return** that value in the `return` statement of the form

```
return expression;
```

Often the expression is placed inside a ( ) pair, just to make it clearer, as in

```
return (x*x);
```

although the parentheses are not required. In fact, here they make `return` look more like a function itself, which it isn't.

How do we use such a function? Let's return to the temperature conversion. As a function, we could write

```
float FtoC(float F)
{
     float C;

     C = 5.*(F-32.)/9.;
     return C;
}
```

Note the use of a variable within the function to store, temporarily, the calculated value.

Here's what our old program might now look like.

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

float FtoC(float);

int main()
{
     float degF, degC;

     cout << "Please enter a Fahrenheit temperature: ";
     cin >> degF;
```

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

float FtoC(float F)
{
      float C;

      C = 5.*(F-32.)/9.;
      return C;
}
```

Let's look at the changes step by step.
Firstly, we can now see that the **main** program that we've been using as 'magic' is in fact a function called, appropriately, `main`. It has no arguments – nothing within the `( )` pair, although an equivalent representation is to say the list of arguments is **void** by putting the keyword `void` between the parentheses. It has a (returned) value of type `int`, which is always returned as 0. This may seem altogether stupid as we aren't interested in the value of a function called `main` anyway. The concept of a value for `main` stems from mainframe computers where such values are interpreted upon completion of a program as to whether a program has successfully run – 0 means no problems. In fact, many C++ implementations allow the use of

```
void main()
{
```

to indicate no returned value. As this is not standard C++ we cannot.

The next addition is the statement

```
float FtoC(float);
```

preceding the `main` statement. This is called a **prototype**. It informs the C++ compiler of the forthcoming use of a function called `FtoC` with a `float` argument and a returned `float` value. The compiler can't look ahead to where we have defined the function, as it is interpreting each statement as it occurs. (The prototype could also be placed inside the main function, near the variable declarations.)

The next change is where we **call** the function in the statement

```
      degC = FtoC(degF);
```

Note that, as in mathematics, we are saying "return to me the value of the function FtoC when given the value of degF so that I can give that value to the variable degC"

Even though we defined the function with the argument `x`, we can send that function the value of **any** variable, or in fact, an expression.

When a call to a function is encountered during the execution of a program, the value of the **actual** argument in the call is used as the value of the **formal** argument in the function. We often say the argument is **passed by value**.
Given $f(x) = x^2$, we know that $f(y) = y^2$ and $f(23) = 23^2$; similarly we know that

```
      FtoC(degF)
```

will use the value of `degF` as the value of `F` in the function `FtoC`.

The concept of actual versus formal argument may seem confusing. For example, we may define the function `max` to return the larger of two argument values as in

```
int max(int x, int y)
{
      if (x > y)
            return x;
      else
            return y;
}
```

and then call it in another function (or `main`) using

```
      int x, y;
      . . .
      cout << max(y,x);
```

Here the correspondence is between actual argument `x` and formal argument `y`, and actual argument `y` and formal argument `x`.

What does the prototype do?
Suppose our program uses the reference

```
FtoC(36.5)
```

As we know, real constants are considered to be of type `double` in C++. Thus, the value 36.5 is a `double` whereas the formal argument `F` of `FtoC` is of type `float`. C++ performs the type conversion for us. We could even say

```
FtoC(36)
```

and C++ would convert the integer to a `float` for us because (through the prototype) it knows what `FtoC` expects (and because it knows how to do the conversion).

The types of the actual and formal arguments must be **type consistent**, that is, the type of the actual argument must be **capable** of being converted to the type of the formal argument.
C++ uses knowledge of the type of the return value in a similar manner.

10.1 Pass by Value

When a formal argument receives the value of an actual argument, it has a copy of that value. Consider

```
int diff(int x)
{
     int oldx = x++;

     return x*x - oldx*oldx;
}
```

This function takes the passed value, sets the variable `oldx` to that value and increments `x`. Then the function returns the difference between the square of `x+1` and the square of `x`. That is, the function is $(x+1)^2 - x^2$.

Because the formal argument `x` stores a **copy** of the value passed to it, the value of the actual argument is unchanged. So, if the variable `alpha` has the value 5, then

```
diff(alpha)
```

has the value 11, but `alpha` is unchanged by the function call.

10.2 Library functions

One of the benefits of functions is that, once someone has written a function to perform some task, like square root for example, we don't need to write it provided we have access to it. As with most higher-level languages, C++ provides **libraries** of pre-defined functions.

Libraries come in two parts. First there is the code (or executable) portion which performs the task the **implementation**. This may have originally been written in the same language, or in another language. It probably now exists in an intermediate form between the higher-level language and machine code usually called **relocatable code**. We must ensure that our compiled program incorporates this component by the inclusion of the library code in our executable.

The second component is the **interface**. This provides C++ with the way in which the programmer can access the library. This provides the explanation of prototypes for the functions and possibly the declaration of constants and variables. This interface is contained in a **header** file.

We've already seen the header file `iostream` which is included in our main program. This is the interface for the `iostream` library and defines such terms as `cin`, `cout`, the operators `<<` and `>>`, and the keyword `endl`.

The mathematical library **`cmath`** provides access to those functions that mathematicians hold dear, such as square root, log, sin, exp and so on. The header file `cmath` provides the interface. Thus, we can incorporate expressions such as

```
sqrt(b*b-4.*a*c)
```

into a quadratic program solver since `sqrt` is the name of the square root function in the cmath library. If we didn't include the header file, however, C++ would have no knowledge of what type of argument `sqrt` expected nor what type of value it returned. The C++ compiler would object by saying that it had not seen the prototype for `sqrt`.

When we say

```
#include <iostream>
```
or
```
#include <cmath>
```

we are asking the C++ **pre-processor** to add the contents of the text file `iostream` or `math` to our program's statements, and then compile them as if they were part of our program. Again, remember that it is unlikely that there are any executable statements in the header. They merely identify usable functions by prototype.

The angle brackets are telling the preprocessor that these are **system library** headers and should be looked for in certain places on the computer's disks. Later we'll create our own header files for our own functions at which time we'll use " pairs about the header file's name to identify the header as a personal header.

(The header `cmath` used to be called `math.h` and is really a header for a C function library. When the standard was changed recently, all the C-style libraries had their `.h` removed and a `c` prefixed. We will see others later. Headers such as `iostream` which are pure C++ libraries just had their `.h` removed. You may still find C++ books that refer to the old `.h` type headers – and they won't have the `using...` statement either.)

Here are some of the prototypes to be found in `cmath` for C++. The comments are not in the header.

```
double fabs(double);            // absolute value
double pow(double, double);     // exponentiation
double sqrt(double);            // square root
double exp(double);             // e to a power
double log(double);             // natural logarithm
double sin(double);             // sine of angle (radians)
double cos(double);             // cosine
double tan(double);             // tangent
double asin(double);            // inverse sine
double acos(double);            // inverse cosine
double atan(double);            // inverse tangent
```

Note that the function

```
int abs(int);
```

is not in `cmath`, but is in the library `cstdlib`.

Note that all these functions are defined for type `double`, so `float` arguments are type-converted, while the results are also `double` and would force `float`s in expressions with them to `double`.
In the example

```
float x = 35.6;
float y = 10.;
float z;

z = y*sqrt(x);
```

the last statement would involve a conversion of the value of `x` to a `double`, the calling of `sqrt` with that value, the conversion of `y`'s value to a `double`, the evaluation of the product (as a `double` expression) and a final conversion of the result back to a `float` for storage in `z`. In the precedence of operators, function calls rank with parenthesized expressions before any of the usual mathematical operators.

One of the above functions deserves mention – `pow`. We mentioned earlier that C++ does not have an operator for exponentiation, such as x^y. Instead, we have to say

```
pow(x,y)
```

Note that both arguments are `double`s. This function does not necessarily use successive multiplication to achieve its aim. It might use logs and antilogs(i.e. exp). As such, it should **not** be used for simple powers where the power is a small integer value.

Thus, calculate x^3 as `x*x*x`, not `pow(x,3)`.

10.3 Explicit Type Conversion

As we said above, when we call a function which has a formal argument of one type with an actual argument of a different type C++ automatically converts the type for us. This can lead to some surprising results. Consider the following (artificial) example.

We would like to know whether the cube of a certain integer number is divisible by 4. The value is stored in the integer variable `testval`. So, we (stupidly) use the function `pow` to cube the value and then test whether the

result is divisible by 4 as in

```
if (pow(testval,3)%4 == 0)
     cout << "Cube is divisible by 4" << endl;
```

Imagine our surprise when this won't even compile! Even though pow's actual arguments are integer, its returned value is of type double, thereby making the expression with operator % a double expression for which % is invalid.

The underlying structure of C++ provides a set of built-in functions, with names like int, float, double, char and so on, which can be used to explicitly convert one type to another. For example

```
if (int(pow(testval,3))%4 == 0)
     cout << "Cube is divisible by 4" << endl;
```

would now work (and do what we require), since we have converted the result of pow to an integer value before its use with the % operator. (Unfortunately, due to real arithmetic inaccuracies, the result of the final conversion might be one off.)

This approach of intentionally converting types is called **type casting** and the functions can actually be used as unary operators by placing their name in parentheses.

Thus the example

```
cout << char(100) << " and " << int('A') << endl;
```

would print out

```
d and 65
```

When learning to program, mixed mode expressions and automatic type conversion of function arguments cause many of the logic errors in programs. Using type casting to avoid such problems can ease some of the pain.

Recent changes to the standard now provide a more formal way of casting. To cast an expression to a particular built-in type, use

```
static_cast<type>(expression)
```

so the above example would read

```
cout << static_cast<char>(100) << " and " << static_cast<int>('A') << endl;
```

This method is now considered the preferred manner of casting, but the other method will still be found in most texts.

10.4 Functions for program structure

You may wonder what the use of a function of type void is. That is, a function with no return value. Consider the following void function.

```
void quad_solver(float a, float b, float c)
{
     float x1, x2, disc;

     disc = b*b-4.*a*c;   // discriminant
     if (disc < 0)
          cout << "There are no real roots" << endl;
     else if (disc == 0)
          cout << "The one root is " << -0.5*b/a << endl;
     else
     {
          disc = sqrt(disc);
          x1 = 0.5*(-b+disc)/a;
          x2 = 0.5*(-b-disc)/a;
          cout << "The roots are " << x1 << " and "
                    << x2 << endl;
     }
}
```

The prototype for the above function would be

```
void quad_solver(float,float,float);
```

specifying that a call such as

```
quad_solver(x,34.5, a+b);
```

would elicit in C++ the correct type conversions and value transfers.

This is in fact how such a `void` function has to be referenced as a statement on its own as it cannot appear in any expression for lack of a value.

(The above function is incomplete – what if `a`'s value were 0?)

Why would we write the solution of a quadratic as a function in the first place?

As we increase the complexity of the tasks, we require our programs to perform, programs will soon get more difficult to design and even harder to follow. One of the tenets of good program design is that of **functional decomposition**. This involves breaking the whole problem the program has to solve into smaller subproblems. Each of these subproblems can then be solved within an individual function. These functions may not need to return a calculated value.

One of the important criteria for good program construction is **functional cohesion**. This means that an individual function should perform one task only, should only need as input those values required for that task, and should only return that value for which it is designed. If you cannot describe what a function does without words like "and" or "or", then the function is probably doing too many things and should be broken into more functions.

10.5 Functions with more than one returned value

Consider our quadratic equation solver. When pressed to describe what it does, we would have to say that it solves the quadratic whose coefficients are passed in **and** prints the answer(s) out. Whoops! Not cohesive. So, we have to avoid printing out the answer. That is, the function should just solve the quadratic.

But quadratics may have more than one solution. How do we get the function to return more than one value? We use **output arguments**, whose values are altered by the function.

Here is a suitable description of a more suitable (and cohesive) quadratic solver: we pass the function three values (the coefficients); we get passed back two values (the answers). Since we also need to know how many solutions were found, we'll use the function's value to return the number.

Recall that arguments that are passed by value have their value **copied** to the formal argument, thus eliminating the possibility of altering the actual argument's value. To be able to change the actual argument's value, the function must know **where** that value is stored and use that location instead of the copy. This is called **pass by reference**. Such formal arguments do not actually occupy any (extra) memory locations but use the formal argument name as an **alias** to the actual argument.

To specify that a formal argument is to be an alias, we place an & after the type of the argument in **both** the implementation of the function and in the prototype. Here is what our new quadratic solver now looks like.

```
int quad_solver2(float a, float b, float c, float& x1, float& x2)
{
     float disc;          // discriminant
     int nsol;            // number of solutions

     disc = b*b-4.*a*c;
     if (disc < 0)
          nsol = 0;
     else if (disc == 0)
     {
          x1 =  -0.5*b/a;
          nsol = 1;
     }
     else
     {
          disc = sqrt(disc);
          x1 = 0.5*(-b+disc)/a;
          x2 = 0.5*(-b-disc)/a;
          nsol = 2;
     }
     return nsol;
}
```

Note that the variables `x1` and `x2` are no longer local to the function, but are arguments passed by reference.

However, they are used within the function as if they were local variables. Note also the use of the local variable `nsol` to store the number of solutions so that only one return statement is needed.

Note also that if there are no solutions, the two output arguments are unchanged; if there is one solution, it is returned in the first of the two output arguments.

How is it called? Here is a calling program. Often these simple `main` programs are used to test whether the function works often called a **driver program**.

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

int main()
{
     float A, B, C, first_root, second_root;
     int quad_solver2(float,float,float,float&,float&);

     cout << "Please enter the three coefficients: ";
     cin >> A >> B >> C;
     switch (quad_solver2(A,B,C,first_root,second_root))
     {
          case 0:
               cout << "There are no real roots" << endl;
               break;
          case 1:
               cout << "The repeated root is " << first_root
                         << endl;
               break;
          case 2:
               cout << "The roots are " << first_root
                         << " and " << second_root << endl;
     }
     return 0;
}
```

We've used the alternate location of the prototype here.

Note that, as output arguments are required to have a location into which results can be put, or where changes can be made, a variable **must** appear as the actual argument. Thus we could not call `quad_solver2` in an expression involving

```
quad_solver2(1.,2.,1., ans1,123.)
```

even if we know, in this case, that the second output argument is not going to be needed.

10.5.1Functions and Header Files

The use of separate compilation provides the mechanism for isolation of functions from one another as well. Each function, or group of functions with a common purpose, can be placed in their own `.cpp` file. Then they can be compiled independently, and re-used in other programs.

Only the interface need be known to the caller – a **header** file.

As in the case of library header files, we use the `#include` pre-processor line. However, our header files are referenced slightly differently from standard libraries.

For a standard library header we use

`#include <header>`

(and `using namespace std;`)

For example:

```
#include <iostream>
```

The angle brackets < and > indicate to the pre-processor that the file between them can be found in a standard location.

For our own headers, we usually place the header file in the same location as the source file, so we use

```
#include "header.h"
```

(In fact, the pre-processor has a **search path** it uses to find the file.)

Note that for our header files, we use the traditional `.h` extension. This is really a matter of convenience. It means we can identify a file in our directory as a header file merely by the extension. In CSCI204 (or maybe even in CSCI121) you will learn about namespaces and the reason for the `using namespace std;`

For now, just keep using it.

10.6 Character functions

10.6.1 Standard library for character manipulation

The library `cctype` (with associated header file `cctype`) contain useful functions for the manipulation of characters. Here are the prototypes, along with their purpose.

```
int isalnum(int);    // non-zero(true) if arg is alpha or num
int isalpha(int);    // non-zero if arg is an alphabetic character
int iscntrl(int);    // non-zero if arg is a control character
int isdigit(int);    // non-zero if arg is '0'..'9'
int isgraph(int);    // non-zero if arg is printable character but is not space
int islower(int);    // non-zero if arg is lower case letter
int isprint(int);    // non-zero if arg is printable character including space
int ispunct(int);    // non-zero if arg is printable but not alphabetic or numeric
int isspace(int);    // non-zero if arg is whitespace
int isupper(int);    // non-zero if arg is upper case letter
int isxdigit(int);   // non-zero if arg is a hex digit ('A'-'F','a'-'f','0'-'9')
int tolower(int);    // if arg is upper case, returns lower case equivalent, or arg
int toupper(int);    // if arg is lower case, returns upper case equivalent, or arg
```

Remember that **whitespace** is space, newline, line feed, form feed, carriage return, tab, or vertical tab. Even though listed with `int` arguments (because of their C origins), these functions are usually used with `char` type as argument. Apart from the last two (converters), the others are usually used in logical expressions.

10.6.2 Input/Output of strings (Part 2)

When input of strings was introduced in the last chapter, it was indicated that standard stream input using `cin` could not input spaces. To be able to read in strings and treat the spaces (or even newlines) as significant, or to protect from overflow, requires the use of a new function (actually two) from the `iostream` library.
The function

```
cin.get(char_var);
```

returns the next character in the stream in the character variable *`char_var`*. The `cin.` identifies this function as related to the input stream `cin` as opposed to perhaps some other stream. Thus

```
const int MaxLen = 39;
char Line[MaxLen+1];       // space for the null
int i;

i = -1;
do
{
     i++;
     cin.get(Line[i]);
} while (Line[i] != '\n' && i <= MaxLen);
Line[i] = '\0';            // the null - could just say 0
```

The function

```
cin.getline(charArray,MaxChar,termChar);
```

returns up to *MaxChar*–1 characters into the character array *charArray*, where reading is terminated by the character *termChar* or by reaching the *MaxChar*–1 characters, or by encountering an end-of-file. The function puts a NUL after the last character read (the *termChar* does not go into the array).

Thus, the previous loop can be replaced by

```
cin.getline(Line,MaxLen,'\n');
```

Be aware that, without the null terminator, any output of a character array will not terminate until it encounters a zero byte in the memory for the array **or** any following it, if the space for the array doesn't contain one.

An analogous function provides character-by-character output using

```
cout.put(char_exp);
```

to output the value of the character expression *char_exp*. Again, note the identifying prefix `cout.` to link this function to the appropriate stream.

10.6.3 Manipulation of strings (Part 1)

Because each individual character array element is just a simple variable, even if it is referenced with an index, it can be used as a typical character variable. This is evident in the one-argument `cin.get` example above, where the argument is just a character variable. The fragment:

```
char Surname[30];

cin >> Surname;
if (islower(Surname[0])) Surname[0] = toupper(Surname[0]);
```

converts the first letter of the string `Surname` to uppercase.

However, assignment and general operations on strings (and arrays in general) is not possible. For strings, there is a standard library called `string` with interface `string.h` which contains many functions useful for the manipulation of strings. But before we can discuss this library, we must describe how arrays in general are handled in functions. We'll get back to the string library in a little while. For now, let's just say that we cannot just assign strings, nor can we just compare strings.

10.7 Arrays and functions

Because standard C++ operators such as `<, =, +, *` can be used only to manipulate one array element at a time, we'll usually see array names followed by its subscript. One place that array names appear on their own is as function arguments. Here's how arrays are passed as arguments.

A formal array argument is specified by appending empty brackets after the name, as in

```
double array_max(double array[], int size)
```

This does not allocate space on the stack for the entire array of `size` elements however. All it does is says space for the location of the array (its **address**) is going to appear on the stack. When the function is called, the location of the actual argument (as in pass by reference) is passed to the function via the stack. If we also do not want to change the contents of the actual array within the function, we can use the keyword `const` preceding the formal argument's type to protect the values.

Our example to find the largest value in an array can now be written as a function.

```
double array_max(const double array[], int size)
{
     int i;
     double maxval = array[0];

     for (i=1; i<size; i++)
     {
          if (array[i] > maxval)
               maxval = array[i];
     }
     return maxval;
}
```

To call this function, we could then say, for example

```
     cout << "The maximum array value is "
                          << array_max(salary,20);
```

Suppose we wanted to find the largest value in only a part of the array. If the array `salary` above contained 50 salaries, then the above call would find the largest in the first 20. If we wanted to find the largest in the array elements 20 to 44, we could call using

```
     array_max(&salary[20],25)
```

The use of the `&` is 'similar' to its use in pass by reference arguments in that it converts the expression `salary[20]` from the **value** of that variable to the **location** of that variable.

This is actually a throwback to 'old' C, in which we have to pass arguments by reference in this way.

C requires the type of a formal pass by reference argument to be written as *`type`* `*`, instead of *`type`*`&`, and you will often find this form in prototypes for C libraries. Many of the standard libraries used in C++ are, in fact, C libraries `cmath, cctype, cstring`, for instance. We'll see this in the next section.

A more C++ friendly way to avoid this form is to provide an initial index for the array as an argument, as in

```
     double array_max(double array[], int first_index,
                                  int num_entries)
```

10.8 Manipulation of strings (Part 2)

The interface `cstring` contains prototypes for functions to handle character arrays holding C-strings. Again remember that the array's string content must be terminated by the `'\0'` character. Here are some of the useful ones.

`size_t strlen(const char *);`
: returns, as the value of the function, the number of characters from the start of the array argument to the null character. The type of the function, `size_t`, is a special type used by C++ to define the size of objects. Use it as if it were an `int`. A reminder that the argument type `const char *` is the same as `const char[]`. So
```
strlen("Fred")
```
would be 4, while if
```
char surname[20]
```
contains `"Smith"` then
```
strlen(surname)
```
would be 5.
String constants can be used for arguments of type `const char *` but not those for which a pass by reference is required (those without the `const`).

`char *strcpy(char *, const char *);`
: copies the string second argument to the string first argument. This includes the null at the end. The value of the function is the address of the target (first) argument but we will not use this information.
Thus
```
strcpy(surname,"Jones");
```
would copy the 6 characters in the constant string into the first six positions of the array `surname`.

`char *strncpy(char *, const char *, size_t);`
: same as `strcpy` except only the first `n` characters (if there are that many) are copied where `n` is the value of the third argument. If less than `n` are copied, a null is placed on the end. Return value is address of target array.

```
char *strcat(char *, const char *);
```
finds the first null in the target (first) array, and then concatenates the string second argument to that string. So, if `FullName` currently contains `"P. T. "`, then

```
strcat(FullName,"Barnum");
```

would result in `FullName` containing `"P.T. Barnum"`. Return value is address of target array.

```
char *strncat(char *, const char *, size_t);
```
same as `strcat`, but only `n` characters from second string appended to first string, where `n` is value of third argument. Here the target is always appended by a null character. For example

```
strncat(FullName,"Barnum",3);
```

would produce `"P.T. Bar"`.

```
int strcmp(const char *, const char *);
```
compares the two strings character by character. If first string is less than second in alphabetic ordering, return value is negative; if the strings are the same, return 0; otherwise return a positive integer. Thus

```
if (strcmp(A,B) == 0)
      cout << "Strings are the same.\n";
```

or

```
if (!strcmp(A,B))
      cout << "Strings are the same.\n";
```

So

`strcmp("Alpha","Beta")` would be negative
`strcmp("Alpha","Alph")` would be positive

```
int strncmp(const char *, const char *, size_t);
```
same as `strcmp` but only compare up to `n` characters.

```
char *strchr(const char *, int);
```
returns the address of the first occurrence in the string first argument of the character in the second argument (note again the use of `int` type for a character argument). Returns a zero address if it does not appear in the string. See below for an example.

```
char *strrchr(const char *, int);
```
returns the address of the last occurrence in the string first argument of the character in the second argument. Returns a zero address if it does not appear in the string. See below for an example.

```
char *strpbrk(const char *, const char *);
```
returns the address of the first occurrence in the string first argument of any of the characters in the string second argument. Returns a zero address if none of the characters appear in the string. See below for an example.

```
char *strstr(const char *, const char *);
```
returns the address of the first occurrence in the string first argument of the string second argument. Returns a zero address if it doesn't appear.

The last four functions are usually used in association with the address of the first character in the searched string, which of course is the name of the string.

Thus if `sentence` is `"The quick brown fox"` then

```
strchr(sentence,'q') - sentence
```

would evaluate to 4 (the subscript for the character `q`). And

```
strrchr(sentence,'o') - sentence
```

gives 17.

```
strpbrk(sentence,"iou") - sentence
```

would yield 5, the location of the first `u`. Finally

```
strstr(sentence,"row") - sentence
```

would tell us that the string `"row"` appear starting from `sentence[11]`, but

```
strstr(sentence,"of") -sentence
```

would be negative, since the first term would produce a null address (0).

This is how these four functions should be used, at least at the present time.

10.9 The mechanics of functions

Further investigation of functions requires some explanation of the operational mechanics of how a function is called. The default sequence of execution of a program is to start at the first statement of the main program and proceed to execute the statements one after the other.
The control structures described earlier modify that sequence:

if and case statements select which of alternative sets of statements are executed, then the program continues in sequence;

loops allow for control to skip to an earlier statement for repetition;

break (and continue) allow for termination of a sequence of statements to skip to a point further down the program;

functions transfer control to a separate set of statements which are performed to completion and then control is returned to the calling statement.

Here is a simplified description of what (usually) happens when a function is called. Before control is transferred to the function, two groups of actions take place:
(i) the actual argument list is finalized. Any expressions are determined. The order of evaluation of expressions is undefined.

If the argument is to be passed by value, the **value** of the argument is placed in a (newly allocated) memory location suitable for the type of the formal argument. This memory location is usually on a **stack**.
If the argument is to be passed by reference, the **location** of the argument (must be a variable, not an expression) is placed on the stack.
(ii) the position of the instruction next to be executed in the calling program is stored.

Control is now transferred to the function. At this point, formal arguments know where their memory is located. It's on the stack in a position relative to a position called a **stack frame**, which the function knows. Arguments passed by value use the location on the stack as the location of that variable. Formal arguments passed by reference must be manipulated slightly differently. Their location on the stack provides a reference for finding the location of the value. Thus to find the value for such a variable, we must get the memory location for its value from the stack and then go there. This is called **dereferencing** or **indirection**.

All the variables declared within the function, but not formal arguments, are provided space on the stack as well. These variables which are automatically allocated space upon the calling of the function are called **automatic variables**. (Those that are allocated space at the start of execution of the program are called **static variables**, which will be described later.)

The function is then executed. If there are arguments called by reference, the locations of the actual arguments are manipulated. All other arguments and automatic variables exist on the stack and are manipulated there. Upon the encountering of a `return` statement or the closing } of the function, execution of the function is terminated. The part of the stack used to store arguments and automatic variables is then freed up for its next use. The variables are considered to be **destroyed**. For non-`void` functions the return value is placed in a location known to the calling function (often a register), and control is transferred to the memorized location of the next instruction.

Thus every call of a function has an **overhead**, a number of operations to be performed that are not part of the actual execution of the statements of the program.
Here's an example.

```
int abs(int x)
{
     if (x < 0)
          return (-x);
     else
          return x;
}
```

Calling this function, as in

```
     y = abs(z);
```

would involve the following steps.

(i) store the location of the instruction to do the assignment;
(ii) put the value of `z` on the stack (to be used in the function as `x`);

(iii) transfer control to the function;

(iv) perform the statements in the function;

(v) put the return value in a register;

(vi) destroy the location of the value of `x` (and hence where we put the value of `z`);

(vii) return to the assignment statement and continue.

Here's an alternative for very simple functions.

10.9.1 Inline functions

Suppose a function can be simply described by a small number of statements (like one). By placing this function in the calling function (at the top of the function with the other declarations) preceded by the keyword `inline`, we create an **inline** function. For example

```
inline int abs(int x)
{
     return (x<0 ? -x: x);
}
```

When the compiler encounters a call to this function, it replaces the function call by the instructions to perform that function inline (in that position). The arguments for the function call are substituted for the function's actual arguments. Successive calls would duplicate these instructions.

Thus, the decision as to whether a function should be inline should be a weigh-up between the time saved by elimination of execution overhead, and the program space wasted by duplication of the instructions of the function. The `inline` keyword is only a **request** to the compiler to place the function inline. If the compiler thinks the function is too complicated, it might ignore the request.

10.10 Functional abstraction, local and global variables

The discussion earlier about breaking programming problems into smaller problems, then solving each of these subproblems within one function is called **procedural** or **functional abstraction**. To take this concept to its natural limit is to write functions to perform each of these tasks **independently** of each other. That is, the only information a function should need is the data for the task, and the only output of the function should be the solution of the subproblem.

Our discussions of functions above have indicated that those variables within a function are called automatic variables. They are created when the function is called and destroyed when we leave the function. This concept of the locality of variables is actually finer than this. In fact, a variable can be declared within **any block**. In the obvious case, this means within any pair of braces `{` and `}`.

For example

```
for (i=0; i<n; i++)
{
     int x;
     cin >> x;
     sum += x;
}
```

means that, within the loop there is a variable called `x`, which is destroyed upon completion of the loop. If there is a variable `x` outside the loop, it is ignored while the inner `x` is in existence, but can then be used afterwards.

A variable declared in a block is available to other blocks inside this one, but not outside it.
The use of variables local to a block should be avoided as it is an invitation to danger.

The range of blocks and functions over which a variable or, in fact, any identifier, is available is called the **scope** of that identifier.

The concept, however, of automatic variables local to a function, is an integral part of good program design. It is called **data hiding**. Why should one function need to know the working variables from another function. It does mean that you cannot use the value of an automatic variable from one function (or the main) within another function.

In fact, functions can also be hidden from functions in other files.

Once this concept is understood, the use of separate compilation can be utilized. Each function, or group of similar functions, can be placed within their own .cpp file. Then they can be compiled independently, and re-used in other programs.

If a particular variable is needed, it can be made **global** to the functions within the one file, by placing its declaration outside all the functions in the file. That means it precedes all the functions within which it is required.

For example,

```
int x;

int main()
{
     . . .
     x = 3;     // assigned here
     . . .
}

int sub()
{
     . . .
     y = x;     // known here
     . . .
}
```

Another situation which is often needed is the retention of the value of a variable local to a function between calls of that function. As automatic variables are destroyed upon completion of the function, and then created on the next call, any variable values are not kept. Here is one situation. A function needs to know how many times it is called.

```
int func(int arg)
{
     static int TimesCalled = 0;
     . . .
     . . .
     TimesCalled++;
}
```

When this function is first called the variable `TimesCalled` is initialized to 0. During the function this variable is incremented. Because the variable's declaration includes the keyword `static`, the variable is **not** destroyed at completion of the function. It is also not re-initialised upon re-entry. Thus, the variable is available whenever we are inside the function. It is still not known outside the function, but is no longer automatic.

If a variable is declared outside all functions in a file global to the functions in the file and obviously not automatic its value is also continually available. If that variable appears in another file as a global then it is assumed to be a duplicate of the same variable. This is generally reported as a **link error**, discovered when the individual program elements (source files, libraries) are joined together.

If these identically named variables are supposed to be the same variable, all but one of the declarations must use the qualifier `extern` before their type. This allows for the transfer of (usually large amounts of) variable values between files a practice usually avoided by the use of arguments and counter to the concept of data hiding. It is C++ programming practice to use variable names preceded by the letter `g` to indicate global variables this is a matter of style.
Should a variable global to a file be required to be hidden from other files, the qualifier `static` (yes, the same keyword that makes non-automatic variables) hides such variables from outside functions.

10.10.1 Initialization of global variables

Variables declared outside functions are usually created at the time the source files are compiled. The compiler determines the amount of memory required and places these variables in an area of memory separate from the stack. Any initialisation of such variables must be by constant expressions. These values are then placed in these memory locations during compilation, and become part of the compiled code.

If a global variable is not explicitly initialized, C++ ensures that the variable is zeroed before execution begins. This is quite often exactly what is required. However, getting out of the habit of initialising variables and assuming that they have the value zero is dangerous. You may forget to set an automatic variable to zero before it is used in an expression, and get **any value** that happened to be in the memory location allocated to that variable.

10.11 A Word about Functions and Header Files

The use of separate compilation provides the mechanism for isolation of functions from one another as well. Each function, or group of functions with a common purpose, can be placed in their own `.cpp` file. Then they can be compiled independently, and re-used in other programs.

Only the interface need be known to the caller – a **header** file.

As in the case of library header files, we use the `#include` pre-processor line. However, our header files are referenced slightly differently from standard libraries.
For a standard library header, we use
`#include <header>`
(*and* `using namespace std;`)
For example
`#include <iostream>`

The angle brackets < and > indicate to the pre-processor that the file between them can be found in a standard location.

For our own headers, we usually place the header file in the same location as the source file, so we use

```
#include "header.h"
```

(In fact, the pre-processor has a **search path** it uses to find the file.)

Note that for our header files, we use the traditional `.h` extension. This is really a matter of convenience. It means we can identify a file in our directory as a header file merely by the extension. Later, you will learn about namespaces and the reason for the `using namespace std;`

For now, just keep using it.

10.12 Recursion I

One of the most powerful benefits of the mechanics of functions is the ability of functions to call themselves, either directly, or via other functions. We'll see this in more detail later. Suffice to show here one example. Consider the calculation of n!, factorial n. For non-negative integers, this is defined as

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ \ldots \cdot 3 \cdot 2 \cdot 1$$

But we can define it recursively as

$$n! = n \cdot (n-1)!$$

provided at some stage, usually at 0! we give the answer explicitly, as 1. Then we can write the function as

```
unsigned int fact(unsigned int n)
{
     if (n == 0)
          return n;
     else
          return n*fact(n-1);
}
```

which can then be called using

```
     cout << "n\! = " << fact(23) << endl
```

or

```
     y = fact(x);
```

We will discuss the uses (and pitfalls) of recursion in more detail later. For now, note the inclusion in the code of

(i) a return value for a simple case;

(ii) a recursive call of the function with a value closer to the simple case than the value on entry to the function.