***Chapter 8* Functions**

Introduction

So far we have written very small programs - most of them have contained fewer than thirty lines of code. Programs of any real use contain a minimum of several hundred lines of code. Commercial programs may have hundreds of thousands of lines of code.

When you create something as complex as a large program it is essential that you break it down into smaller, simpler, more manageable parts. If you can do this, then you can design and build each (simpler) part in isolation, putting the parts together as you complete them. As a simple analogy, cleaning an entire house seems like a monumental task, but tackling the rooms one by one seems a lot easier, though having cleaned all the rooms one at a time results in the entire house having been cleaned.

There are **many advantages** to this **modular approach** of breaking a big problem down into **modules** or smaller sub-problems. We will mention some of them as we work through this chapter. You may also be able to think of other advantages which we won’t have time to mention.

The functionality of any complex program can be broken up into a number of separate pieces. COBOL programmers call them 'paragraphs'. In Visual Basic they are 'subroutines'. If you are a Pascal (or Delphi) programmer, you call them 'procedures' and 'functions'. As C++ programmers we just call them ***functions***.

So in C++ a function is a part of the program which carries out a particular function or sub-task of the whole program. It is in fact a sub-program - rather like a program in miniature. You have already used a number of functions supplied with the system – **getch()**, **toupper()** etc. Although hundreds of these are supplied with your C++ system, you will always need to write special ones for your own particular programs.

Suppose we have to design a program to produce a payslip for each employee in a company. The algorithm for this task could be expressed in pseudocode as follows:-

**get the next employee number**

**get the number of hours the employee has worked**

**calculate the standard hours worked**

**calculate the overtime hours worked**

**calculate the standard pay**

**calculate the overtime pay**

**calculate deductions for tax**

**calculate deductions for National Insurance**

**calculate deductions for pension contributions**

**calculate final take-home pay**

**print the payslip.**

Each of the lines in the above design could be considered a sub-task or function of the algorithm. The program code for each function could be written by a separate programmer to **speed up the production process**, and the whole lot could be put together when each programmer had finished **testing** his/her own function.

Note that many of the functions in this program have to communicate with other functions in the program. For example the function to calculate the standard pay needs to know how many standard hours the employee has worked, and that is worked out by another function - the one immediately preceding, in this case. Similarly, the function which prints the payslip needs information from several other functions - the standard pay, overtime pay, take-home pay and so on.

Functions communicate by means of data items - ***arguments*** (called ***parameters*** in other programming languages, and we may use this term interchangeably in C++ if we wish). Data can be passed to a function by means of an ***input******argument***, and a function can pass data back via an ***output******argument***. Some arguments can even be ***input-output******arguments*** where the same argument can take data into a function and bring it out again. But this is all getting a little abstract - let's see something concrete.

Our first user-defined function

The program below is supposed to ask the user for an integer. It will then write that many asterisks to the screen. The pseudocode would simply look like this:-

**ask how many stars**

**display the stars**

Each of the lines in the above could represent a function. For simplicity we'll just write a function to display the stars for the moment. We could write the pseudocode for the function as follows:-

**function** **display\_stars**

**start a new line**

**for as many stars as required**

**write a star to the screen**

**end for**

**start a new line**

**end function**

The lines to 'start a new line' are there simply to ensure that the line of stars stands on a line by itself, and doesn't attach itself to either the line before or the line after.

Type in the following program - **but** you will need to read the points listed on the following pages **before** you can compile and run it successfully:-

*// A first demonstration of a multi-function program*

*// Chap0801.cpp*

…

void display\_stars(int);

void main()

{

int numStars;

cout << "Line of Stars" << endl;

do {

*// Ask the user to specify the number of stars*

cout << endl << "Enter number of stars (0 to quit):- ";

cin >> numStars;

*// Call the function, passing as an argument*

*// the number of stars required*

display\_stars(numStars);

*// Stop when no stars requested*

} while (numStars > 0);

…

}

*// Function definition of display\_stars*

void display\_stars(int numStars)

{

int i;

*// Make sure the stars start on a new line*

cout << endl;

*// Show the asterisk appropriate number of times*

for (i = 1; i <= numStars; i++)

cout << "\*";

*// End the line and insert a blank line*

cout << endl;

**}**

There are several things to note about the preceding program:-

1. There are two functions in the program - **main()** and **display\_stars().** We have already met the function **main()** - all C programs have to have one.

2. When we talk about a function we always put the brackets after it - this emphasizes that we mean a function and not a variable.

3. Each function has a ***header*** and a ***body***. The header is the first line of the function which begins **void** … It declares the name of the function and any arguments the function might use. The body is the part of the function between the braces which contains the executable code - the part that does the job, so to speak.

All functions look like this - you will be familiar with the appearance of a function already since you have already seen many **main()** functions.

Note that a function header is NOT terminated by a semi-colon.

4. The ***argument******list*** in the function header specifies a ***name*** or ***identifier*** for each argument and the ***type*** of each argument (**int**, **float** etc).

**display\_stars()** uses a single integer argument called **numStars**.

Note that **main()** has no arguments in this or any other program you have written, though it can have some. When a function has no arguments, we can put the word **void** in the argument list brackets – simply leave them empty.

5. The line of code:-

display\_stars(numStars);

in function **main()** is said to ***call*** the function **display\_stars()***.* We mean that function **display\_stars()** is executed. Moreover, in making a call to **display\_stars()***,* the value of the variable **numStars** is said to be ***passed*** to the function as an argument. We say that **main()** is the ***caller*** or the ***calling******function*** in this case.

If you write a line of code which calls a function, but doesn't pass to it the right number of arguments, you'll get a compiler error. Moreover, the arguments must be in the right order (same as in the function header), and must be of the right type (again, same as in the function header).

Now try running the program listed above.

You'll find that you get a **compilation error** of the same type as you get if you forget to put **#include** at the top of your program. The compiler wants to see a ***prototype*** for the function defined at the top of the program. With supplied functions such as **toupper()**, the prototype is held in a header file - such as **ctype**.**h**. When you write your own function you must put its prototype at the top of your program.

**What is a prototype?**

When the compiler comes across a call to a function, it's supposed to check that the programmer has supplied the right number and type of arguments in the right order. It really needs to look at the function header to check these things. If the first mention of a function is in a call to it, as in the above program, then the compiler is unable to make such checks. So you supply a prototype at the top of the program which is roughly a copy of the function's header. There are two differences:-

- the prototype does not need to know the name of the argument - just its type

-the prototype IS terminated with a semi-colon.

So the top of the above program should look like this:-

…

void display\_stars(int);

void main()

{

*etc* …

Now the compiler has already seen the function definition by the time it meets a call to the function, and so can do the relevant checks.

Update your program (by inserting the missing prototype) and get it working.

One way to avoid having to write prototypes for all your own functions is to change the physical order you code them in your program. Instead of putting **main()** first you put it last in the program (in Pascal programming the ‘main’ part always comes at the end). This does make the program slightly less easy to read and understand however and is not particularly recommended - other C++ programmers may expect **main()** to be at the top of your programs.

If you do it the traditional way with prototypes then it doesn’t matter what order your functions appear in your program code (**physical sequence**) because at run time C++ will call and execute them in the correct sequence (**logical sequence**).

There’s another reason you might prefer to leave **main()** at the top. It arises if you use structure charts or hierarchy charts to show the relationships between the different functions in your program. Your chart will show **main()** at the top so it makes sense to have it there in the code too.

Let's now discover some more about structure charts and hierarchy charts.

Program structure and structure charts - a first look

An algorithm, written in pseudocode, shows the logic of a program section or function - what happens when and in what sequence. This is very useful in designing a complex function, or indeed an entire program. It is also very useful as a means of documenting the function or program for reference by other programmers.

Another useful aid in program design and documentation is the ***structure******diagram***. This is a diagram which shows, not the logic flow, but the relationships between functions in the program. It gives us the ‘**big picture**’. Our program so far could be represented by the following structure diagram:-

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | **main** | |  |  |  |
|  |  |  |  |  |  |  |
|  |  | **display\_stars** | |  |  |  |

The information contained in the above diagram is that the function **main()** calls a function **display\_stars()***.* No other information is presented. This simple form is more correctly known as a ***hierarchy chart***.

Let's say we choose to move the following lines from **main()** to become a separate function:-

cout << "How many stars (0 to quit)? ";

cin >> numStars;

Then the structure chart would look like:-

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | **main** | |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| **get\_no\_of\_stars** | |  |  | **display\_stars** | |

This structure diagram shows that **main()** calls the functions **get\_no\_of\_stars()** and **display\_stars()**. We could say it shows that the program consists of two sub-tasks (functions) which together constitute the main functionality of the program.

We shall see more of structure diagrams in future. The two examples above are for programs so simple that you don’t need a diagram to remind you how the functions relate to each other. But in real programs you may have dozens of functions and the structure chart can make clear the hierarchy of calls.

Adding a second argument to the function

Let's try adding another argument to our function **display\_stars()**. Actually what we'll do next is to modify our function so that it will display not just a line of stars, but of any character we ask for. So we'll change the name of the function to **line\_of\_chars()**. We'll want to specify two arguments - the character to display and the number of those characters to display. The first of these arguments will be of type **char** and the second will be an integer as before.

The function could look like this:-

void line\_of\_chars(char userCh, int userNum)

{

int i;

cout << endl;

*// Show the character the appropriate number of times*

for (i = 1; i <= userNum; i++)

cout << userCh;

cout << endl << endl;

}

Our **main()** function must be changed too of course:-

void main()

{

int numChars;

char inCh;

cout << "Line of Characters" << endl << endl;

do

{

*// Ask the user to specify the number of chars*

cout << "Enter a number (0 to quit):- ";

cin >> numChars;

if (numChars > 0)

{

*// Ask which character to display*

cout << endl << "Enter a character:- ";

cin >> inCh;

*// Call the function, passing both arguments*

line\_of\_chars(inCh, numChars);

}

*// Stop when no chars requested*

} while (numChars > 0);

…

}

Try to get this program working (Chap0802.cpp).

Do you have to pass variables as arguments to functions?

No, you don't have to pass variables as arguments to functions. You might use the function we just created to put lines of characters on the screen as part of something else. You might know in advance where you want the lines to go, how long they are to be and what character should make up the line. You could have calls to the function like the following:-

**line\_of\_chars('\*', 50);**

line\_of\_chars('-', 78);

line\_of\_chars('#', 10);

In other words, you are passing ***literal******values*** rather than variables to the function. Whether you pass variables or values simply depends on what you are trying to do at the time. You will make that decision when you need to. This is a simple example of **re-cycling a function** so as to get two different uses from one chunk of code – which we see as a major advantage of this modular approach to program development.

Try inserting these lines into your program and check that the output is as you would expect i.e. fifty stars for the first one etc.

Let's add a third argument . . .

Just for practice let's add another argument. All the lines of characters so far start at the left margin. Suppose we wanted to be able to start them at some distance in from the margin, under our control - an indent. We could simply get the function to write some spaces before the main character with pseudocode as follows:-

**function line\_of\_characters**

**start a new line**

**for as many spaces as required**

**write a space to the screen**

**end for**

**for as many chars as required**

**write a star to the screen**

**end for**

**start a new line**

**end function**

Note that there should not be a new line after the spaces.

Try it - again you'll need to tweak **main** to get it to work (Chap0803.cpp).

So far we've only used input arguments to a function - ones that bring data in to the function. As mentioned earlier, functions can also pass data back to the calling function - a function does this by means of an ***output******argument***.

Suppose we want a function which will double any integer we give it. Let's call it **num\_doubler**. The function pseudocode might be as follows:-

**function num\_doubler( in number, out doublednumber )**

**set doublednumber to 2 times number**

**end function**

So the actual program code for the function is as follows:-

void num\_doubler(int origNum, int& doubledNum)

{ *// & denotes an output argument*

doubledNum = 2 \* origNum;

}

This is extremely simple code - but nevertheless demonstrates an important point. In order to pass data out of a function we need to indicate that the argument which will carry the output data is an output argument. This can be done by suffixing the argument type with an ampersand **&**, as in **int&** **doubleNum**. Passing data back to the caller is then simply a case of assigning the data to the output argument.

What’s actually going on here is that instead of giving **num\_doubler()** its own copy of the data we are telling it where to find **main()**’s original version of the data. This is just a glimpse into a complex but very important aspect of C++ called **pointers** – which luckily we don’t have to worry about on this course!

A suitable **main()** function for trying out the **num\_doubler()** function is:-

void main()

{

int inNum, doubleNum;

cout << "Number Doubler" << endl << endl;

cout << "Enter a number (0 to quit):- ";

cin >> inNum;

while (inNum != 0)

{

num\_doubler(inNum, doubleNum); *// no & needed on the call*

cout << endl << inNum << " doubled is " << doubleNum;

cout << endl << endl;

cout << "Enter a number (0 to quit):- ";

cin >> inNum;

}

…

**}**

Try it. Again some tweaking may be required (Chap0804.cpp).

A note on naming conventions for parameter types

Argument types, for clarity have been labelled input and output. In most C++ literature they are normally called ***value*** and ***reference*** arguments respectively. We shall continue for a while to refer to them as input and output arguments.

*Try a few yourself ...*

Now try to write the following functions (with a suitable **main()** to test them):-

8.1 A function which accepts an input argument (a float) and outputs, through an output argument, a value which is the square of the input argument.

8.2 A function which accepts an input argument (a float) and outputs, through an output argument, a value which is the cube of the input argument.

8.3 A function which accepts two integers, and outputs, through an output argument, a value which is the smaller of the two inputs.

*If you have time ...*

8.4 Write a program which asks the user for the height and width of a rectangle, and which then draws a solid rectangle of 0s on the screen, 10 spaces in from the left. Use the **line\_of\_characters** function (the one with the **three** arguments) without modification. In other words **main()** should call **line\_of\_characters()***.*

If you didn’t get the three-argument version working yourself you can find it in Chap0803.cpp.

8.5 Write a function which accepts three integers, and outputs, through an output argument, the smallest of the three.

8.6 Write a function which is called repeatedly and accepts a continuous sequence of numbers from the user until zero is input as a terminator. Output the highest input.

This looks at first sight like a rather tougher problem than the previous one – because that was limited to three inputs and this problem has to cope with many. But actually you could solve it with a simple function which accepts just two arguments – ‘next user input’ and ‘highest so far’ and outputs the new value of ‘highest so far’.

**Input-output arguments**

We've already seen how input arguments allow data to be passed into functions, and output arguments allow a function to pass data back to the caller. There is often a need for a combination of the two - an **input-output argument** - where the same argument carries data both into *and* out of a function.

Let's take the example used before of a function which doubles a number supplied to it. The function we used was:-

void num\_doubler(int origNum, int& doubledNum)

{ *// & denotes an output argument*

doubledNum = 2 \* origNum;

}

This could be rearranged so that the same argument is used to both take in the number to be doubled, and to pass out the doubled number, like this:-

void num\_doubler(int& aNum)

{

aNum = 2 \* aNum;

}

In C++ an output argument can always be used as an input argument as well. You could just make all your arguments outputs, but that would be bad practice for reasons which we can leave for the moment. So don't!

The **main()** function which calls **num\_doubler()** needs to be modified slightly too:-

void main()

{

int inNum;

cout << "Another Number Doubler" << endl << endl;

cout << "Enter a number (0 to quit):- ";

cin >> inNum;

while (inNum != 0)

{

cout << endl << inNum << " doubled is ";

num\_doubler(inNum);

cout << inNum << endl << endl;

cout << "Enter a number (0 to quit):- ";

cin >> inNum;

}

…

}

Notice that once**main()** has sent **inNum** to **num\_doubler()** the value in **inNum** has been permanently changed (i.e. to double its previous value) so the original value is lost. The output (**cout**) lines are changed to reflect this (it only displays one variable). But there is some simplification of both **main()** and **num\_doubler()** as a result of using the input-output argument. (Chap0805.cpp)

**Scope of variables**

When a variable is declared inside a function we say that the ***scope*** of the variable is from the point where the variable is declared to the end of the function. Whenever the function is called the variable is ‘born’. While the function is executing the variable exists - but when the function terminates the variable ‘dies’ and no longer exists. It therefore follows that any other function is unaware of the variable - the variable cannot be ‘seen’ from outside the function in which it is declared.

Variables which are declared inside a function and are invisible from the outside are called ***local******variables***- they are **local** because they are declared inside a function and therefore are available only to that function.

Let's look at an example:-

void main()

{

**x and y are local to main()**

int x, y;

------

**my\_funct() is being passed the values of x and y**

------

my\_funct(x, y);

------

------

}

**x and y are local to my\_funct(). They are not the same variables as x and y in main()**

void my\_funct(int horiz, int vert)

{

int x, y, z;

------

x = horiz - 32;

y = vert + 32;

**These statements only affect my\_funct ()’s x and y, not main()’s.**

------

------

}

Although both functions in the above program declare variables called **x** and **y**, they are quite separate entities. No matter what **my\_funct()** does to its **x** and **y**, it will have no effect on the **x** and **y** in **main()***,* and vice versa**.**

The fact that variables are local to functions means that you don't have to worry when choosing variable names that you might already have another variable of the same name elsewhere - it doesn't matter as long as you don't try to have two variables of the same name in the same function (i.e. with the same scope).

What's more, in a real development situation, it's more likely that several programmers will share the task of developing the collection of functions which are needed for a program. They don't need to worry about what names they choose within their functions and whether they match what their colleagues are using. It's a step towards making large programs easier to create by splitting them up into independent functions which can be independently developed.

However you may find it confusing to have the same data names occurring in two (or more) of your own functions. If you do then you should avoid the problem by always using unique names. You may be able to devise a simple system of coding into your data names whether the item is local or an argument. This could be achieved using a prefix or a suffix to the name.

Formal argument names

The above program illustrates another feature of functions. The arguments declared in the header of **my\_funct()** are called **horiz** and **vert***.*  These are called ***formal******arguments***. They are local to the function since they are declared in the function, and as such are not visible outside the function - in particular in **main()***.*

Note that the names of the formal arguments need not be the same as the names of the variables in the call to the function (**x** and **y** in this case). In fact, since a particular function might be called from many places in a program (or from several different programs if we put the function into a library file), it is unlikely that this would be the case.

The things that **must match** between a call to a function and the function header are:-

- the **number** of arguments

- their **type**

- their **order**.

The compiler will check the first two of these - you may need to check the third.

When a programming team is developing a large program, members have to consult with each other about the interface between functions. The person who is developing a function must let others who will be calling that function know what arguments will be required - how many, their type and their order.

This extra task of co-ordinating the links between functions is one of the few disadvantages to the **modular approach** of breaking big problems down into smaller ones.

Another way of returning data from a function

We have already seen how we can make a function send back some data to the caller via an output (or an input-output) argument. We can have as many output arguments as we need to return whatever data is necessary - it might be a single integer, a couple of floats, a whole array of strings, or something even bigger.

If a function has to return only one piece of data we can use a different method of returning it -and this other method has advantages in certain situations.

Let's take our old example of **num\_doubler()***.* First we must specify in the function header what type of data item is to be returned, like this

int num\_doubler(int aNum)

Note that where we've always had **void** before we now have **int**. This specifies that the function will return a single integer - but not through an argument. We can have functions returning **float**s, **char**s, and most other types of data. When we put **void** as the return type it means there is no returned data by this method, though there may well be output arguments. The ***return*** mechanism is the **return** statement as you can see in the new body for this function:-

**int num\_doubler(int aNum)** *// int type function returns an int*

**{**

**int doubledNum;**

**doubledNum = aNum \* 2;**

**return (doubledNum);**

**}**

The **return** statement sends the value of **result** back to the caller via the integer return value of the function. Notice, by the way, that the function could be further simplified so that it contains no actual statements other than the header and **return**:-

int num\_doubler(int aNum) *// int type function returns an int*

{

return (aNum \* 2);

}

Note that we put whatever is to be returned in brackets after the **return** keyword. (Chap0806.cpp)

The next question of course is how does the calling function receive the returned value from the called function? Here's an example of a **main** function which calls **num\_doubler**:-

void main()

{

int inNum;

cout << "Enter a number (0 to quit):- ";

cin >> inNum;

cout << inNum << " doubled is " << num\_doubler(inNum)

<< endl << endl;

}

Note that we've included the function call in the **cout** statement as though it were a variable. Indeed, whenever you use the **return** method of returning data from a function, you can treat the function call as though it were a variable. Here's another example of this sort of reference:-

if (num\_doubler(inNum) > 10)

cout << ". . . .

In the examples above the value which comes back is used - but lost. We can retain it by assigning it to a variable during the function call, like this:-

answerNum = num\_doubler(inNum);

The returned value is then bound into **answerNum**, which can then be used again later:-

cout << "That number doubled is " << answerNum;

or

answerNum = num\_doubler(answerNum);

You can even do things like:-

answerNum = num\_doubler(num\_doubler(inNum));

You need to be careful about getting the brackets right. What would the above statement do?

Function exercises

Create functions to do the following. You'll need to write a **main()** to test each one:-

8.7 Return an integer value which is the cube of the input argument (**int**).

8.8 Return the Fahrenheit equivalent (**float**) of the Centigrade input argument (**float**). To convert °C to °F, multiply by 1.8 then add 32.

8.9 Return an encrypted form (**char**) of the input argument (**char**). Whatever character is passed to the function should have a coding factor of 2 added to it. E.g. if you pass the function a 'c' it will return an 'e':- if you pass a 'H' it will return a 'J' etc.

8.10 Modify the above function so that the coding factor to be added to the character is passed in a second argument (which will be of type **int**).

8.11 Return (**float**) the smaller of two passed arguments (**float**s).

8.12 Return (**char**) from two passed arguments (**char**s) the one which is nearest the beginning of the alphabet.

8.13 Return (**int**) the number of days in a given month, passed as an integer, e.g. if the function is given 3 (i.e. March) it will return 31. Use a **switch** statement in the function.

8.14 Return an integer obtained from the keyboard which is within the range specified by two input (**int**) arguments. The function will only terminate and return the integer when the user enters a value within the range specified (input validation!).

*If you have time ...*

8.15 Return (**int**) the number of pence in a cash amount passed as a float. E.g. if £16.43 is passed, the function will return 1643.

8.16. Return (**float**) a value which is the fractional part of a **float** value passed in. E.g. if 6.27 is passed, the function returns 0.27. (Chap0810.cpp)

8.17 Write a function which draws a row of characters centred on the next line. The character required is passed as a **char** argument and the number of characters to be displayed is passed as an **int**eger argument. The function should return the indent (**int**) - the number of spaces in from the left that the characters start.

Note two useful aspects of functions which illustrate why they are used so extensively in programming:-

First, they help make the workings of the program clearer because:-

a) functions hide the detail of the code from immediate view so they simplify the higher level functions such as **main()**

b) well-chosen function names make the code easy to read and understand.

Second, they reduce the amount of code in a program which repeatedly uses the same few lines of code. A function **display\_items()** might be called called several times - if we didn't use a function we would have to write the lines of code to display the item list repeatedly – and if we needed to change that code we would need to hunt through the whole program changing every occurrence.

Structure diagrams again

The structure diagram for a program might look like this:-

**main**

**display\_items**

**add\_vat**

Note that although **main()** calls **display\_items** twice, this doesn't show in this hierarchy chart, since a hierarchy chart is only meant to show calling ***relationships*** between functions.

It is possible to draw more sophisticated structure charts with extra symbols in the top right corner to convey more information:-

**\** a bar across the corner of a box means it appears more than once on the diagram - i.e. it is called from more than one other function

**\*** an asterisk indicates the box is carried out more than once - i.e. it is called from inside a loop

**o**a circle indicates a box which is called conditionally - i.e. it is called from inside an IF.

Structure charts can also be marked with arrows to show the flow of parameters between functions and their type. Such diagrams quickly become cluttered however - so the parameter information is better recorded on an ***interface table***.