Functions - Chapter Summary

1 C Program Structure

Every C program must contain a function called `main`, and execution of the program will always start with the first statement in `main`.

In addition to `main`, a C program usually consists of several small functions, each with a specific task to perform, and each called in turn from `main` or from another function. Some of the functions will be from a library, and others will be written by the programmer.

The use of functions helps to create modular programming, where each small step of the programming task is clearly identified and assigned to a particular function. This in itself encourages good programming habits, while the use of meaningful names for functions helps to ensure that programs are easy to read and understand. Modularity makes it easier to find and correct errors, and once a function is written, it may be called many times in a program, which saves having to write the same section of code over and over again.

A function from one program may be used in a completely different program, and this is made easier if the function was originally written in a different file from that containing the `main` function. This allows a programmer to build on what has already been achieved, rather than having to start from scratch each time a new program is written.

However, as will be shown later in this chapter, the use of functions in C programs also helps to preserve data integrity, because variables defined and used in one function cannot unintentionally be altered by the actions of any other functions, and this feature is especially useful in the design and creation of large and complex programs.

2 The Function Prototype

A function prototype is a statement included in a program before any calls to the function it describes. It defines the type returned by the function itself, the function's name, and the number and type of each of the function's arguments, thus:

```
unsigned long int grains_of_sand(int, double, long);
```

If the program containing the above definition includes a call to the function `grains_of_sand`, then the compiler will throw up an error message if the call specifies the wrong number or wrong type of arguments for this function. Additionally, the compiler will know that `grains_of_sand` returns a value of type `unsigned long int`, even if it has not yet encountered the definition of this function.

Like data, all functions have a type, and the type of a function depends on the type of value that it `returns` (see section 4). The above declaration causes memory to be

allocated for storing the value that the function will `return`, and determines how that value will be stored.

The safest position for function prototypes is at the top of `main`, after any variable declarations but before other programming statements, or within a header file included at the very top of the program.

In addition to the data types available for variables, a function can also be of type `void`, which means that it does not return a value. An example of a `void` function is found in Section 5.1.

WARNING: If no type is specified, the program will assume that the function is of type `int` and returns an integer value. Always use `void` to indicate the absence of a `return` value.

3 The Function Call

For one function to call another, a process referred to by some as summoning or invoking a function, the function's name must be stated, together with accompanying brackets enclosing any argument or arguments to the function. The arguments contain any values that the calling function needs to send to the function being invoked, and multiple arguments must be separated from each other by commas. Some functions require no arguments.

The following are all possible function calls:

```
menu();
area = rect_area(length, width);
vol = cube_vol(length, width, height);
printf("\nArea of rectangle is %f.\n",
          rect_area (length, width));
```

Function calls may be nested. In the last example, for instance, the `printf` function call used a call to `rect_area` as one of its arguments.

Arguments may be variables, or more complex expressions, but each expression in the argument list is evaluated, and the value is passed to the function being called.

4 The Function Definition

The function being invoked must be defined somewhere in the program; the existence of an appropriate prototype will ensure that the compiler will handle the call properly. Functions can be defined in any order, but function definitions, unlike function calls, cannot be nested one within another.

4.1 The Function Interface

The function definition comprises of a heading and body. The heading is similar to the prototype, starting with the return type, followed by the name of the function together with accompanying brackets that contain what are called the formal parameters taken by the function, if any. Again, any arguments must be separated from each other by commas.

The formal arguments must be defined within these parentheses, i.e. providing both type and name. The call provides the initial value, and the identifiers used are local to the function (see Section 6, below).

The body of the function contains the statements that perform the work of the function, and these may contain calls to other functions.

4.2 The `return` statement

The `return` statement signals an end to the function body and has the general form:

```
return expression;
```

A `return` statement explicitly terminates execution of the statements in a function and returns control to the calling function. This means that any statements in the function after the `return` statement may not be executed.

For functions that have no useful return value, the `return` statement does not include an expression and the function is of type `void`, e.g.:

```
void dart_score(int dart1, int dart2, int dart3)
{
     int score = dart1 + dart2 + dart3;
     if (score < 180)
     {
          printf("\nScore is: %d.", score);
          return;
     }
     printf("\nONE HUNDRED AND EIGHTY!");
}
```

There is no need for an `else` before the final `printf` statement in the above example, because if the score is below `180`, the `return` included in the if construction immediately sends control back to the calling function, and the final `printf` statement is therefore never encountered.

The `return` statement is also used to pass a single value back to the calling function, and this value is the value of the expression in the `return` statement.

When a `return` statement that includes an expression is encountered, the expression is evaluated, and the value of the expression is then, if necessary, converted to the type specified by the function type, before being passed back to the calling function.

If control is always returned at the end of the body, the `return` statement is redundant as the final } heralds the end of the function.

4.3 Control after a Function call

When the body of a function definition has been executed and a `return` statement or the final brace `}` is encountered, the flow of control is immediately returned to the calling function. If the original function call was part of a larger statement or was nested inside another function call, the rest of the statement containing the function call is duly executed in order according to precedence. If the original function call stood alone (see first example below) as an independent statement, then control is returned to the first statement after the function call in the calling function, i.e. when a function has been executed (cf. `rect_area_print` below), control returns to `main` and the final `printf` statement there is executed, which simply prints **OK**. to indicate that the program has successfully finished.

5 Example Programs

5.1 A function of type `void`

The first example is a program that uses a function to calculate the area of a rectangle and display the result on the screen.

```
/* First, the prototype */
void rect_area_print(double, double);

int main(void)
{
     double side1, side2;
     printf("This program prints out the area"
            "of a rectangle\n\n");
```

```
     printf("Please type in the long side\n");
     printf("followed by the short side\n");
     scanf("%lf%lf", &side1, &side2);

     /* then the call */
     rect_area_print(side1, side2);

     printf("\nOK. \n");

     return 0;
}
/* and finally, the definition */

void rect_area_print(double length, double width)
{
     double area = (length * width);
     printf("\nThe rectangle has an area of %f.", area);
}
```

5.2 A Function of type `float`

```
/* The prototype indicates a return type of double */
double rect_area(double, double);

int main(void)
{
     double side1, side2, result;
     printf("The program prints the area of a rectangle \n");

     printf("\nPlease type in the long side");
     scanf("%lf", &side1);

     printf("\n...and now the short side");
     scanf("%lf", &side2);

     result = rect_area(side1, side2);
```

```
     printf("The Area is %.2f \n", result);
     printf("\nOK.\n");
     return 0;
}

/* The definition */

double rect_area(double length, double width)
{
     double area = length * width;
     return area;
}
```

6 Actual Arguments and Call by Value

In the examples above, `main` passed the values of the two variables `side1` and `side2` as the arguments to the functions `rect_area_print` and `rect_area`. The values of `side1` and `side2` are referred to as the actual arguments of the functions.

A function call assigns the value of each actual argument to its corresponding formal parameter at the beginning of the function body. When this has been done, the body of the function is then executed.

Note that the variables `length` and `width` are quite distinct from the variables `side1` and `side2`. Even if `side1` and `side2` were to be renamed `length` and `width`, the variables in `rect_area` would have a separate identity and be stored at different locations in memory to their namesakes in `main`. Only the values of the arguments are passed from one function to another, and this feature of C is referred to as call by value.

This means that whatever changes may be effected to the values of the variables in one function, they cannot affect the values of any variables in any other functions.

There are times when it is useful, however, for one function to change the variables in another function, and for this to happen, the addresses of the variables must be passed from one function to another, to achieve what is known as call by reference. The chapter on Pointers and Functions explains how this can achieved through the use of pointers.

7 Scope of Data and Program Blocks

The scope of data defines where in a program that data can be referred to by name.

Descriptions of scope often refer to program blocks. A block is any section of a program that is, or may be, included in opening and closing braces `{` and `}`. In the context of a data item's scope, the most commonly-used example of a block is a function body, but the bodies of `if` statements and loop bodies, for instance, also count as program blocks, even if they are simple statements, which may be enclosed in braces, rather than compound statements, which must be.

Local Data

Data declared inside a program block can only be used within that block, and is therefore described as local data.

Global Data

Data that is declared outside program blocks may be used anywhere in the program after it has been declared, and is described as global data.

8 Storage Classes

In declaration, data is assigned differing amounts of memory, and values are stored in this memory in different ways, according to the type of the data - whether, for instance, it is of type `int`, `char` or `float`. Additionally, the way that memory is assigned, for example the position of a data's storage in memory, the initial value assigned to it, and the length of time it is allocated to a particular type of data, all depend on where the data item is declared, and on its storage class.

There are four storage classes: `auto`, `static`, `register` and `extern`.

The storage class declaration `extern` is used with global data to indicate that data is defined in another file, and is thus an external data item.

External data is covered in more detail in the chapter Working with Larger Programs.

8.1 `auto`

Data declared within a program block, such as the body of a function, may be declared to be of storage class `auto`, although this is not strictly necessary, since the compiler will assume such data to be of class `auto` if no other storage class is specified.

A data item of type `auto` is so called because it is automatically created when execution begins of the block in which they are contained. Such items are usually stored in the memory stack, and therefore have no useful initial values, and have to be explicitly assigned values within the block in which they occur. They may be initialised with any valid expression.

The formal parameters of a function have a storage class of `auto`.

When the execution of a block is finished, all data of class `auto` that it contains disappear, along with all the values they have been assigned.

This means that an automatic data item is local in scope, since it only exists during the execution of the block in which it is defined.

`auto`s make very efficient use of memory, and offer protection for the data they are assigned, since they cannot be accidentally affected by statements in any function other than the one in which they are declared. They also contribute to the program portability of functions, since their declarations are contained within the function in which they are used.

8.2 `static`

Data items declared outside the body of a function are assumed by default to be of storage class `static` if no other storage class is specified.

Data items of this class get their name from the fact that they are created only once, at the start of program execution, and then remain statically allocated throughout program execution.

This means that a data item of class `static` retains its value between references.

Data items of class `static` have an initial default value of zero, and may be assigned constant expressions only.

Local `static`s

A data item that is declared within a block to be of class `static` is only local in scope, and may not be referenced outside the block in which it is declared.

Unlike `auto` data items, however, `static` items exist before the block in which they are defined is first executed, and does not disappear when the execution of that function is finished. Thus, a `static` data item declared within a function retains its value, even when the function in which it is defined is not being executed, and this value may therefore be referenced through successive function calls.

The function in the following example will return a new value of `time` that is one greater than the previous value of `time`, every time it is called. It effectively remembers the current value of `time`, because it is a `static` local variable.

```
int tic_toc(void)
{
     static int time;

     return ++time;
}
```

A function using `static` local data is more portable than one using global data, because all the information required by the function is contained within the function itself.

Global `statics`

A `static` data item that is declared outside the body of a function is referred to as static global data. It may be referenced anywhere within the file in which it is declared. `static` global data therefore has a different scope from both `auto` data, which is local to the block in which it is declared, and global data, which may be used anywhere in the program after it has been declared.

8.3 `register`

A declaration that a variable (and it must be a variable, not a `const`) is of class `register` requests that the variable be stored in one of the registers of the central processing unit (CPU), rather than, as normal, in random-access memory (RAM). The variable will be assigned to a register only if space is available, and it is not possible to refer to the address of a `register` variable.

Storing data in a register means that the operations that use that variable will be executed faster, because the value of the variable is already in the CPU and therefore there is no time spent on memory access when the variable is used.

There is only limited space available in the CPU registers at runtime, however, so there are several restrictions on the use of the `register` storage class for variables.

Only local variables, including the formal parameters for functions, may be declared to be `register` variables. A Global data item may not be classified as `register`.

A variable of class `register` has no useful initial value and must be assigned a value within the block in which it is declared. It may be assigned any valid expression.

The ISO standard for C specifies that the type of variables that may be declared to be of class `register` is implementation dependent. Many implementations restrict the use of `register` variables to types `int` and `char`, however.

The number of `register` variables that are allowed to exist at any one time depends both on the type of the processor being used and on the particular implementation of C. Typically, 8-bit systems allow one `register` variable at a time, and 16-bit systems allow two or more variables to be of storage class `register`. No error is generated if the compiler encounters more `register` variables than the maximum number allowed for the system, however. The additional variables are automatically converted to `auto` variables, and this helps to ensure that code is portable across different systems.

Because it is limited by the above constraints, use of the `register` storage class should be restricted to those variables that are referred to most often in a program, in order to make the most of the speed advantage that `register` variables provide.

`register` class variables are ideal for loop control, for instance, where the variables are employed every time the loop is iterated.