SCHOOL OF ENGINEERING

Software and Embedded Systems Laboratory 2 (ELEE08022)
C language part

Writing Your Own Functions

As we have seen throughout this course, each C program must contain a *single* **`main()`** function, which is used by the operating system to start the program running. The process of using a function is usually referred to as *calling* or *invoking* the function. The effect is to cause the statements inside the function (the function body) to be executed, as we have seen so far with **`main()`**.

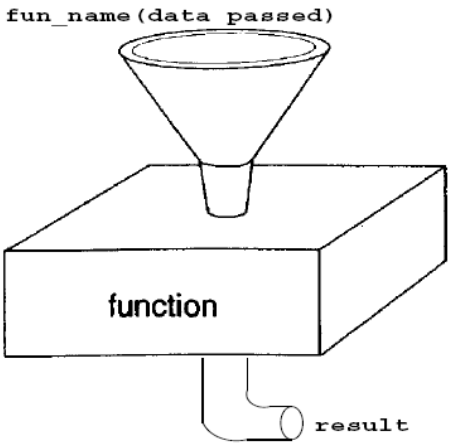
From the **`main()`** function, any number of other functions may be called, which may also themselves call other functions. It is legal in the C language for a function to call itself — *recursion*. In some languages, names such as *routine*, *subroutine*, *procedure*, or *definition* are used instead of, or in addition to, what is known in C as a function.

We have already seen how the standard-io library functions **`printf()`** and **`scanf()`**, as well as mathematical library functions such as **`pow()`** and **`sqrt()`**, are called from **`main()`**. If a library function already exists to perform a required action, or something close to it, it is usually best to make use of the library function. Not only should a library function be reasonably free of bugs, and be a quite computationally efficient implementation, but its behaviour will be well understood by other programmers (who may have to work on the program after you) and it will save you time in the long run (once you have learned how to make proper use of it), improving your productivity.

Of course, there will always be situations where a library function will not do, such as when a teaching exercise or assignment requires a home-made example, or where no suitable library function exists. In these cases, we write our own C functions, something we will do frequently in the rest of this course.

3.1 Introduction to Functions

The general process of calling a function requires that the function: receives a list of input data (known as *arguments*), operates on the data, and **`return`**s a result. Where the specific purpose requires it, any or all(!) of these three parts may be omitted from a function.



Note that C functions can accept *many* argument values but can only **`return`** *one* value as the function call's result. We shall see that this is not really as much of a limitation as it might seem. As we have already seen with library functions (e.g. **`printf()`** and **`sqrt()`**), a function is called by giving the function's name and passing any input data to it in the parentheses which follow the function's name.

The function name (**fun_name**) identifies the *called* function which must be able to accept the data passed to it by the function doing the *calling* (often **main()**). When the *called* function successfully receives the data, it then operates on it to produce the desired result.

It is important to realise that a *function* is itself simply a sequence of *instructions* (usually statements written in C) which carry out the required task, and which have been put into a convenient re-usable package. Since, by now, we have written a considerable number of programs to carry out various tasks, we should have no difficulty in writing our own functions. After all, a *function* is just a (small) "program" which operates on some data to produce the result we require.

As an example of writing a function, let us consider a program to read in two numbers and print out the *maximum* of the two values. The **main()** program (or function) is given below:

Program 3_1

```
/* A program to read-in two numbers and print out the maximum */
#include <stdio.h>
int main(void)
{
  float firstnum, secnum, maxnum;
  float find_max(float, float);       /* the function prototype */

  scanf("%f", &firstnum);
  scanf("%f", &secnum);

  maxnum = find_max(firstnum, secnum); /* the function is called */

  printf("The maximum of the two numbers entered is %f\n", maxnum);

  return 0;
}
```

This program assumes the existence of a *function* named **find_max()**, which we have not yet written!

3.2 Declaring a function

The first place **find_max()** is mentioned in this program is in the statement immediately below the declaration statement for the variables used:

```
 float find_max(float, float);      /* the function prototype */
```

This is actually the *declaration statement* for the function **find_max**. Just as we must declare a *variable* before we use it in a program we must declare a *function* before we call it, and we do this by means of a *function prototype*, as shown above. Like the declaration of a variable, the function prototype links a *name* (**find_max**) with a *data_type* (in this case a **float**). The name is followed by parentheses indicating that a *function* is being declared, in contrast to an array name [square brackets], or a scalar variable - nothing following the name. Thus this statement *declares* that **find_max** is a *function* which **return**s a **float**ing point *result*. As the comment attached to the line indicates, that

statement is also a **function prototype** since inside the parentheses following the function name there are *two* data type keywords separated by a comma. This tells the *C compiler* that the *function* **find_max** requires *two* **arguments** (input values), each of which must be a **float**ing point number. Therefore, this single C statement tells the compiler that **find_max** is the *name* of a function, which *accepts* two floating point values and returns a floating point result.

The next reference to **find_max()** in Program 3_1 is when it is brought into action or *called*:

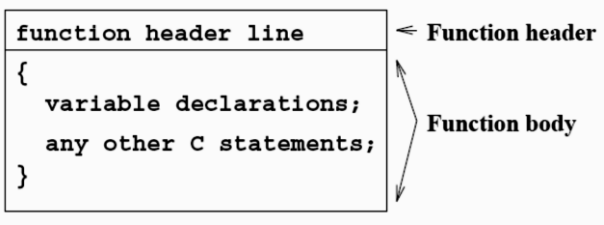
```
maxnum = find_max(firstnum, secnum); /* the function is called */
```

The function **find_max()** is referred to as the *called* function, since it is called *from* the **main()** function. The function that does the calling, in this case **main()**, is always referred to as the *calling* function. We therefore say that **main()** *calls* **find_max()**.

When **find_max()** is *called* by the above statement the values stored in **firstnum** and **secnum** are *passed* to the function, which operates on these values and then **return**s the maximum value, which is then *assigned* to the variable **maxnum**. If **find_max** is a standard *system* or *library* function (i.e. written by someone else), we could simply compile and run our program with no further thought, but it isn't the case. We need to wirte a **find_max()** function.

3.3 Defining a function

Writing a function is referred to as the *definition* of the function. We have already had a great deal of practice in writing functions (all our programs have consisted of **main()** functions!). Like the **main()** function, every C function consists of two parts, a *function header* and a *function body*, as shown below:



The purpose of the *function header* is to identify the *data type* of the value **return**ed by the function, to provide the function with a *name*, and to specify the *number*, *order*, and *type* of *arguments* expected. The purpose of the *function body* is to *operate* on the *input data* and to give the function a **return** value back in the place from which it was called, just as if the function name were a simple variable name.

If the return value type is omitted, the function is defined to return an integer value, by default. Note that the omission of a return type does *not* indicate that the function does not return a value; there is a special type *void* for use in such cases.

The arguments that will be supplied to the function are represented in the function header by names known as *parameters* or *formal arguments*. These are just like variables, and are used like variables

when we *write* the body of a function, which we must do *before* we know what the *real argument values* will be (at *execution*).

At this point, it is worth emphasising that *every* argument to a C function is a value. Some languages have more complex means of transferring data into functions, but C always reduces anything (constant value, variable name, complex expression, ...) supplied as a function argument into its plain value. It is that value, which replaces instances of the parameter name used inside the function body, when the function is executed.

Thus, the function header for **`find_max()`** could be:

```
float find_max(float x, float y)
```

NOTE THAT THE FUNCTION HEADER IS NOT TERMINATED BY A SEMICOLON, just as **`main()`** never has a semicolon, because the statement is not yet finished! The function name and all parameter names in the header line (**`find_max`**, **`x`** and **`y`**) are chosen by the programmer. All parameters listed in the function header must be separated by commas and must have their *individual data types specified separately*.

Having written the function header for the **`find_max()`** function we now need to write the function *body*. As shown above, a function body begins with an opening brace, **`{`**, followed by any necessary *variable declarations* followed by any valid C statements, and ends with a closing brace, **`}`**. There is nothing new here, since this is exactly the same structure used in all the **`main()`** functions we have written. Since the purpose of the **`find_max()`** function is to select the larger of two numbers passed to it we will need to use an **`if-else`** statement to find the maximum of the two numbers. If we wish to store the maximum of the two numbers passed to it we will need to declare *one more* variable to hold this value. Once we have determined the maximum value all that remains is to include a statement within our function **`find_max()`** which will **`return`** this value to the *calling* function (in this case **`main()`**).

To **`return`** a valid result value a function must use a **`return`** statement, which has the form:

```
return expression;
```

When the **`return`** statement is executed, the *expression* is evaluated first. The value of the expression is then automatically converted to the *data type of the function name*, as declared in the function header line, before being sent back to the calling function. Execution of a function terminates when a **`return`** statement is executed. After the *called* function (**`find_max()`**) has finished program control reverts back to the calling function (in this case **`main()`**). Thus the complete function definition for the **`find_max()`** function is:

```
/*Here is one possible implementation of the function find_max( )*/
float find_max(float x, float y) /* function header */
{                                /* start of function body */
  float max;                     /* variable declaration */
    if (x >= y) {                /* find the MOST POSITIVE value */
      max = x;
```

```
   } else {
      max = y;
  }
  return max;                          /* result of function call */
}                                      /* end of function definition */
```

An inspection of the C code in this function shows that it correctly implements the requirements we described in the paragraph above, so it will **return** the maximum value of the two floating point values in the variables **x** and **y**, which are *declared* in the function header line. Remember that **x** and **y** are called the *parameters* of **find_max()**, but where and how do **x** and **y** get assigned values? We shall consider this in the next section.

3.4 Calling a function

As we seen before, *calling* a function is very straightforward (e.g. **printf()**, **sqrt()** etc). We simply need to use the name of the function and include any data to be passed to the function within the parentheses following the function name. This is illustrated here for the *call* of **find_max()** in our **main()** function.

```
maxnum = find_max (firstnum, secnum); /*the function is called here*/
       <- |____|   |_________________|
          |    |             |
          | This calls    This causes two
          | find_max()    values to be passed
          |               to find_max()
          |
       return value from find_max is assigned to maxnum
```

The items enclosed within the parentheses are called *actual arguments* of the called function - in this case the variables **firstnum** and **secnum**. In C, a *called* function *always* receives a ***copy*** of the *value stored in the variable passed as an argument*. Thus the above statement causes the *current values* of **firstnum** and **secnum** to be *passed* to **find_max()**. After the values are passed, *control is transferred* to the called function - i.e. the computer stops executing the statements in **main()** and begins executing the statements in **find_max()**. When the **return** statement in **find_max()** is executed *control returns to the calling function* (**main()**) On return from the function call the value **return**ed from **find_max()** is assigned to **maxnum**.

When we run the entire program, type in two float numbers:

```
35.0
15.0
```

The output from the program is

```
The maximum of the two numbers entered is 35.000000
```

Obviously, going to the effort of writing a function for such a simple task seems like an unnecessary complication. However, now that we have written a function called **`find_max()`**, we can use it any time we want to determine the maximum of two numbers. We can call our function as many times as we want within a program and in this case it would make our **`main()`** program easier to write and to understand. Using functions (usually more complicated than **`find_max()`**) fits in well to our desired approach of *top-down* program design - since each component of an algorithm to carry out our desired task can be implemented as a function (as we proposed in Session 1).

User-written functions can be placed *after* the **`main()`** function, as we have done in the above example or they can be put at the top of a program *above* the **`main()`** function. We will always place **`main()`** first because this usually gives a clearer picture of what the program does before we see the details of each function. This is very much a matter of personal preference and you can use either style. However, a user-written function must **NEVER** be placed *inside* **`main()`** or another function. Each C function is a separate and independent piece of code with its own parameters and variables and cannot be included within another function.

3.5 Scope of Variables

C functions are constructed to be independent *modules*. As we have seen, input values are passed to a function using the function's argument list, and the function's value is set inside the function body, at a point where execution is complete, using a **`return`** statement. Thus, a function can be thought of as a closed box with aperture(s) at the top to receive *argument* values and a single output "pipe" at the bottom of the box to directly **`return`** a value (as illustrated on page 1). This is a useful picture since it emphasises the fact that what goes on *inside* the function, including all *variable declarations* within the function's *body*, are hidden from the view of all other functions (i.e. private).

Since the variables created inside a function, including the parameter names, are available only to the function itself, they are said to be *local* to the function, or *local variables*. This term refers to the *scope* of a variable, where scope is defined as the region of the program where the variable is valid or "known".

A variable can have either *local scope* or a *global scope*. Local variables are only meaningful when used in expressions or statements inside the function where they have been declared. This means that the *same name* could be declared and used for *different variables* in more than one function. For each function in which the variable with the common name is declared, a separate and distinct variable is created, with a well-defined scope, which does not overlap with the scope of any other variable sharing the common name.

As we have seen, C functions are independent *modules*, which have input data *passed* to them using the function's *argument list*, and which **`return`** a result value to the calling function.

We have also seen that a function's *internal variables*, those variables declared within the function's *body* and *parameter list*, are hidden from all other functions, i.e. they have *local scope*. Use of a variable with local scope is only valid in expressions or statements *inside the function that declared it*.

It is possible to place declaration statements *outside* the function(s) in a file. Variables declared in this way, *external* to a function, have *global scope*. They are called *external variables* or *global variables*.

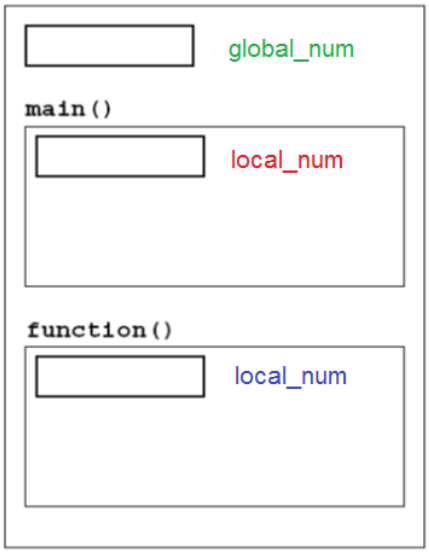
Once a variable has been declared with *global scope* it can be used in any function which is placed in the same file *after* the declaration. The program below demonstrates the use of a *global* variable **global_num** and two identically named, but distinct, *local* variables called **local_num**. The purpose of this program is only to provide a simple demonstration of the significance of *local and global scope*.

Program 3_2

```
#include <stdio.h>
int global_num; /* external declaration: global_num has global scope
*/
int main(void)
{
 int local_num;   /* local_num has local scope, limited to main() */
 void function(void);    /* function prototype */

 global_num = 25;        /* global_num value changes everywhere */
 local_num = 30;         /* affects value only inside main() */

 printf("From main() 1: global_num = %d\n", global_num);
 printf("From main() 1: local_num = %d\n\n", local_num);

 function();             /* execute the function function() */

 printf("From main() 2: global_num = %d\n", global_num);
 printf("From main() 2: local_num = %d\n", local_num);

 return 0;
}

/* definition of function() */
void function(void)   /* type is void as no value is returned */
                      /* void parameter list: no values passed in */
{
 int local_num; /*different local_num, scope limited to function()*/

 local_num = 50;          /* value changes only within function() */

 printf("From function(): global_num = %d\n", global_num);
 printf("From function(): local_num = %d\n\n", local_num);

 global_num = 15;   /* the value of global_num changes everywhere */

 return;            /* no value returned */
}
```

Running this program gives the following results:

```
From main() 1: global_num = 25
From main() 1: local_num = 30
From function(): global_num = 25
From function(): local_num = 50
From main() 2: global_num = 15
From main() 2: local_num = 30
```



This shows that the *global* variable **global_num** is common and accessible to both functions, so that changing its value in one function affects the value seen in the other. The name **local_num** describes *two distinct* variables, each *local* to one function. Even though they have identical names, changing the value of one local variable does not affect the other. The reason for this may be explained by the description below of "auto" variables. The diagram shows how the various storage areas are separated by the computer.

3.6 Global Variables — A Health Warning

At first sight, the use of *global* variables may seem to offer many advantages. Since any function can change the value of a global variable, a function can effectively "return" more than one value by making use of globals. Also, input values can be passed into functions without the tiresome business of making parameter/argument lists and having to get things in the right order. However, such considerations are very short-sighted, and tend to indicate laziness on the part of the programmer, rather than any benefit to the likely correctness and efficiency of the program.

The use of global variables *by-passes the few mechanisms provided by C which help the programmer* to ensure that functions are independent of each other, that they perform their task in a clear and correct fashion, and that they interact through mechanisms which can be automatically checked (by the compiler or more complex correctness proof checkers). Furthermore, the limited imagination of programmers in their choice of names for variables soon causes problems if more than a few global names must be chosen. If two variables, which are *supposed* to be distinct, are *accidentally* given the *same global name*, then no automatic checking will ever be able to spot the error, as the error is in the mind of the programmer, not in the C program.

There are a *VERY FEW* situations where it is *NECESSARY* to use global variables (such as where certain unusual types of function, where neither can ever call the other, must exchange data). In those situations, specialised coding styles and techniques can and should be adopted, to forestall *some* of the risks involved. Other risks of using global variables cannot be avoided and will haunt the dark recesses of any program which contains them.

3.7 Storage Classes for Variables Declared Inside Functions

The concept of *storage class* in C is related to the *duration* of the existence of a variable during the execution of a program. This, in turn, is closely related to the way in which the variable is stored in physical computer memory. A variable declared *inside* a function can be declared as **auto** or **static**

storage class. If the storage class of a variable declared inside a function is not specified (as has been the case so far) then it defaults to be of the **auto** class.

3.7.1 Class auto (the default if no class specified)

The term **auto** is short for "automatic". The physical storage for such variables is *automatically allocated* by the computer *each time* the function is *called*, but when the function **return**s, that physical storage is *taken back* (to be used for something else).

For as long as the function body is being executed, the local variables can be used and they retain their values. However, as soon as execution of the function is completed, the storage for **auto** variables effectively disappears. This give and take process is done to make efficient use of computer memory, and it causes us no programming problems, as long as we understand its consequences.

The program below illustrates the temporary nature of the storage allocated for **auto** variables.

Program 3_3

```
#include <stdio.h>
int main(void)
{
 int count;              /* allocate the auto variable count */
 void print_val(void);   /* function prototype */

 for(count = 0; count < 3; ++count) {
   print_val();
  }
 return 0;
}
/* no value is passed to this function and no value is returned */
void print_val(void)
{
 int val = 0;            /* allocate the auto variable val */

 printf("Value of automatic variable val is %d\n", val);
 ++val;

 return ;
}
```

Each time the function **print_val()** is called from **main()**, the *local* variable **val** (which has **auto** storage class, by default) is allocated some memory and initialised to contain the value **0**. Therefore the **printf()** function call in **print_val()** is always passed the value **0** in **val**. The fact that **val** is incremented by **1** before the function returns has no real effect (some compilers will not even bother to make that happen) because as soon as the **return** statement is executed the storage space for **val** is scrapped and sent for re-cycling. On the next call to **print_val()**, fresh storage is allocated to **val** and initialised. The output produced by this program is therefore:

```
 Value of automatic variable val is 0
 Value of automatic variable val is 0
 Value of automatic variable val is 0
```

3.7.2 Class `static`

If we need a local variable to retain its value between calls to the function in which it is declared, we must declare it to have **`static`** storage class. The storage for a local **`static`** variable is *not* allocated and re-cycled at each call to the function. Rather, it is physically allocated and initialised just before **`main()`** is called (as the whole program is being *loaded* by the operating system).

Thus, storage for *static* variables is allocated on a permanent basis, i.e. for the duration of execution of the whole program. Hence any value stored in a *static* variable persists between function calls. This means that the storage for a **`static`** local variable is initialised only *once*, *before* the *program* begins execution, not every time the function in which it is declared is called.

The program below is very similar to Program 3_3 (above) but uses a **`static`** variable inside the function rather than an automatic one.

Program 3_4

```
#include <stdio.h>
int main(void)
{
 int count;             /* allocate the auto variable count */
 void print_val(void); /* function prototype */

 for(count = 0; count < 3; ++count) {
   print_val();
  }
 return 0;
}

/* no value is passed to this function and no value is returned */
void print_val(void)
{
  static int val = 0;  /* allocate the static variable val */

  printf("Value of static variable val is %d\n", val);
  ++val;

  return; /* nothing is returned */
}
```

Note again that the initialisation of the variable **`val`** (to **`0`**) is done at load time, i.e. just before the **`main()`** function is called. *It is NOT initialised on each entry to the* **`print_val()`** *function*! Since the value stored in the **`static`** variable **`val`** is retained on exit from the function **`print_val()`** and

is accessible again on subsequent entry the value passed to **printf()** increases by one each time the function is called. The output below illustrates this permanence of **static** variable storage:

```
Value of static variable val is 0
Value of static variable val is 1
Value of static variable val is 2
```

Unlike automatic variables, all **static** variables are initialised with (integer) zero if no specific initialisation is given in the declaration statement. We could therefore have omitted the initialisation of **val** in the above function. However, it is generally better programming practice to explicitly make such initialisations.

Because **static** variables must have their physical storage managed in a particular manner, they may give rather slower data access than automatics. As with global variables (which are stored in a similar fashion to static class local variables) the peculiar properties of the static class should not be used without careful examination of the need for it, which is why **auto** is the default class.