Software and Embedded System Lab 2 (ELEE08022)

Writing Your Own Functions in C language

Dr Jiabin Jia

Email: jiabin.jia@ed.ac.uk

Writing Your Own Functions

Within **`main()`**, the program can call any number of other functions:

- pre-written/library functions e.g. **`printf()`, `scanf()`, `pow()`**
- *Your own* functions specific to a given problem's solution, which:
  - receive data as *arguments* via the *parameter list* (in parentheses)
  - operate on the data using ordinary *C* statements
  - return a result back to **`main()`**

A built-in function is used ***(called, invoked)*** as follows:

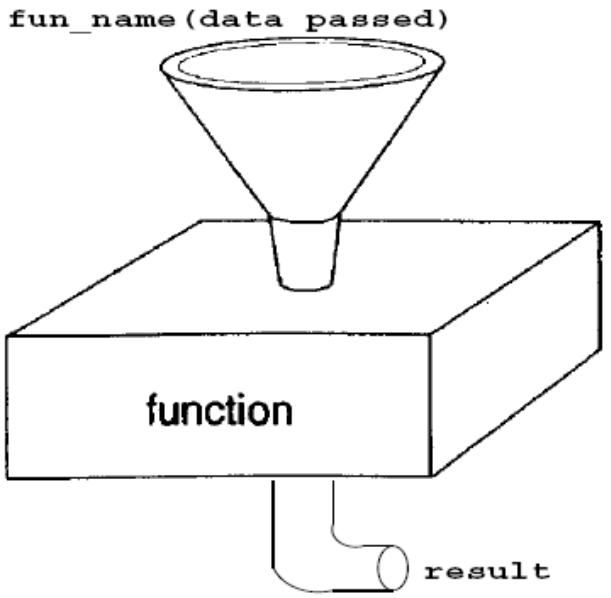
```
double root, length = 2.0;
root = sqrt(length); /* a function is called */
```

C functions can have *multiple arguments*, but have *only one* return value

Introduction to Functions

A function is just a small "program" which operates on some data to produce the result we require

- name of function (**`fun_name`**) identifies action called for
- argument(s) passed to it from calling function - often **`main()`**
- return value of function is result of actions performed in:
- body of function is sequence of normal *C* statements, and may include other calls to functions, all of which together carry out the required task



- we have already written many programs to carry out various tasks
- we should have no difficulty in writing our own functions!

Declaring and *calling* a function

A program to read in two numbers and print out the maximum of the two values. The `main()` program (or *function*) could be:

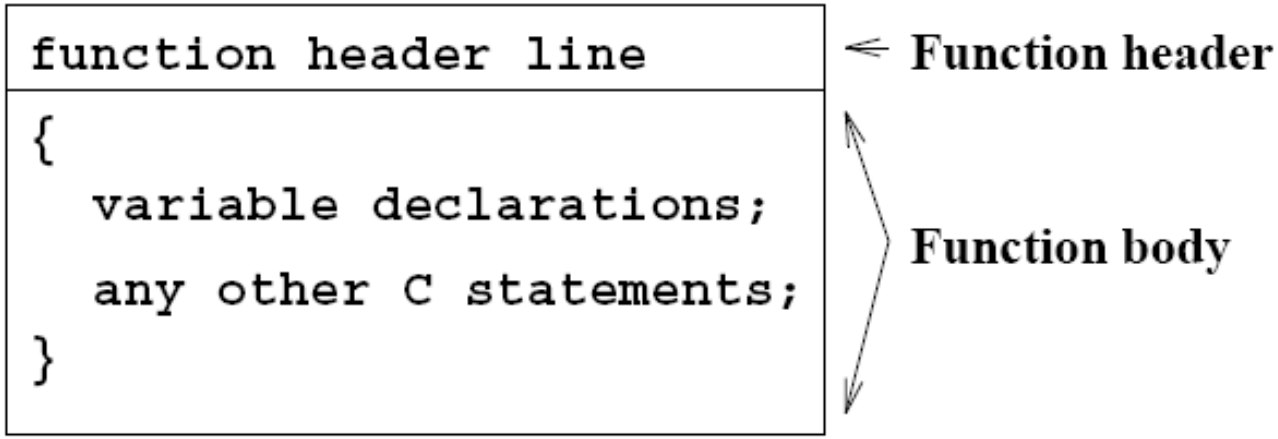
```
#include <stdio.h>
int main(void)
{
  float firstnum, secnum, maxnum;

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

  scanf("%f", &firstnum); /* read in the first number */
  scanf("%f", &secnum);  /* read in the second number */

  maxnum = find_max(firstnum,secnum); /* function call */

  printf("Maximum of numbers entered is %f\n", maxnum);
  return 0;
}
```

The *Function Prototype* is a *declaration* of a function's properties: **name** (**`find_max`**), **two arguments** with **type** (**`float`** and **`float`**), and **return type** (float). This is all the information that the compiler needs to ***create a call to*** this function.

Defining a function

- Writing a function is referred to as the ***definition*** of a function.
- Every *C* function consists of two parts, header and body:

- ***header***
- ***body***



The ***header*** specifies:

- ***data type*** of the value **`return`**ed
- ***name*** of the function
- ***type, number*** and ***order*** of ***arguments*** expected

In the ***body***:

- operates the ***argument*** data using normal *C* statements
- **`return`**s ***one value*** (at most) to the calling function.

The function header

The function header for **`find_max()`** could be:

```
float find_max(float x, float y)/* NOTE - NO semicolon */
```

- shows **`return`** type of function (**`float`**)
- **`x`** and **`y`** are referred to as *arguments*
- *all arguments* must have *names* and *individual data types*. They are separated by commas

The function body

- Want to select the larger of two numbers passed to **`find_max()`** and **`return`** this value to the *calling* function
- Use a **`return`** statement to send a value back to the calling function

`return` *expression*`;`

- Execution of function statements ends when **`return`** is executed
- After the *called* function (**`find_max()`**) has finished, program control reverts back to the *calling* function, in this case **`main()`**

find_max()

```
float find_max(float x, float y)   /* function header */
{                                  /* start of function body */
   float max;                      /* local variable declaration */
   if (x >= y) {                   /* find most positive value */
      max = x;
   } else {
      max = y;
   }
   return max;                     /* function result value */
}                                  /* end function definition */
```

Actual *call* of **find_max()** in our **main()** function:

```
maxnum = find_max (firstnum, secnum);
```

return_value go_to_function with_values_for **x** **y**

- variables **firstnum** and **secnum** are the *actual arguments*
- *called* function *always* receives a ***copy*** of the ***argument value(s)***
- *control is transferred* to the called function

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

  scanf("%f", &firstnum);

  scanf("%f", &secnum);

  maxnum = find_max(firstnum, secnum);/* function call */

  printf("Maximum of numbers entered is %f\n", maxnum);
  return 0;
}

float find_max(float x, float y) /* function header */
{
   float max;                          /* function body */
   if (x >= y) {
      max = x;
    } else {
      max = y;
    }
   return max;                        /* function result value */
}
```

Conclusion

Type in two float numbers:

```
35.0
15.0
```

The output from the above program is

```
Maximum of numbers entered is 35.000000
```

- we can use **`find_max()`** any time we want
- using functions fits well to *top down program design*
- using functions increases modularity of your program

User-written functions can be placed:

- ***BEFORE*** or ***AFTER*** the **`main()`** function
- but ***NEVER INSIDE*** **`main()`** or any other function

```c
/* We can put find_max () before main () */
#include <stdio.h>
float find_max(float x, float y) /* definition . . .  */
{                                /* and declaration    */
   float max;
   if (x >= y)
      max = x;
   else
      max = y;
   return max;
}
int main(void)
{
  float firstnum, secnum, maxnum;
  float find_max(float, float); /* even not strictly needed! */

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

  maxnum = find_max(firstnum,secnum);

  printf("Maximum of numbers entered is %f\n", maxnum);
  return 0;
}
```

Passing Arrays to Functions

A *C* function is passed a *copy* of the *values* of the *arguments* given in the *calling* function. Passing argument values stored in *individual* array *elements* is done in exactly *the same way* as for simple *scalar variables*:

```
. . . = find_max(volts[0], volts[1]);
```

```
float find_max(float x, float y)
{
    float max;
    if (x >= y) {
        max = x;
    } else {
        max = y;
    }
    return max;
}
```

Passing a *whole array* of values to a function is treated differently:

- *whole arrays* are **NOT** copied
- Compiler gives the *called* function *access to the original array*

```c
/* Find Biggest element in an Array  */
#include <stdio.h>
int main(void)
{
  int scores[5] = {5,15,42,9,28}; /* init'd declaration */
  int i, big;

  big = scores[0];    /* init big to value of first element*/

  for (i = 1; i < 5; ++i) { /* cycle through others */
     if (big < scores[i]) {   /* check if other is bigger */
        big = scores[i];      /* if so, remember value */
     }
  }
  printf("Biggest value: %d\n", big);
  return 0;
}
```

Running this program gives: **Biggest value: 42**

```c
#include <stdio.h> /* Find Biggest element in Array*/
int main(void)
{
  int scores[5] = {5,15,42,9,28};       /* decl & init */
  int find_biggest(int [5]);    /* function prototype */

  printf("Biggest value: %d\n", find_biggest(scores));
  return 0;
}

int find_biggest(int array[5])          /* fn definition */
{
  int i, big = array[0]; /* first element is initial candidate */

  for (i = 1; i < 5; ++i) {     /* cycle through others */
     if (big < array[i]) {    /* identify new candidate */
        big = array[i];    /* store new found candidate */
     }
  }
  return big;                  /* ultimate candidate value */
}
```

Running this program gives: **`Biggest value: 42`**

Scope of Variables

C functions are *independent modules* which 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 **`return`** a result

- *scope* is the region of the program where the variable name is valid

- a variable can have either a *local scope* or a *global scope*

- generally, what is *inside* the function, *including all variable names*, is *hidden from view* (scope) of all other functions

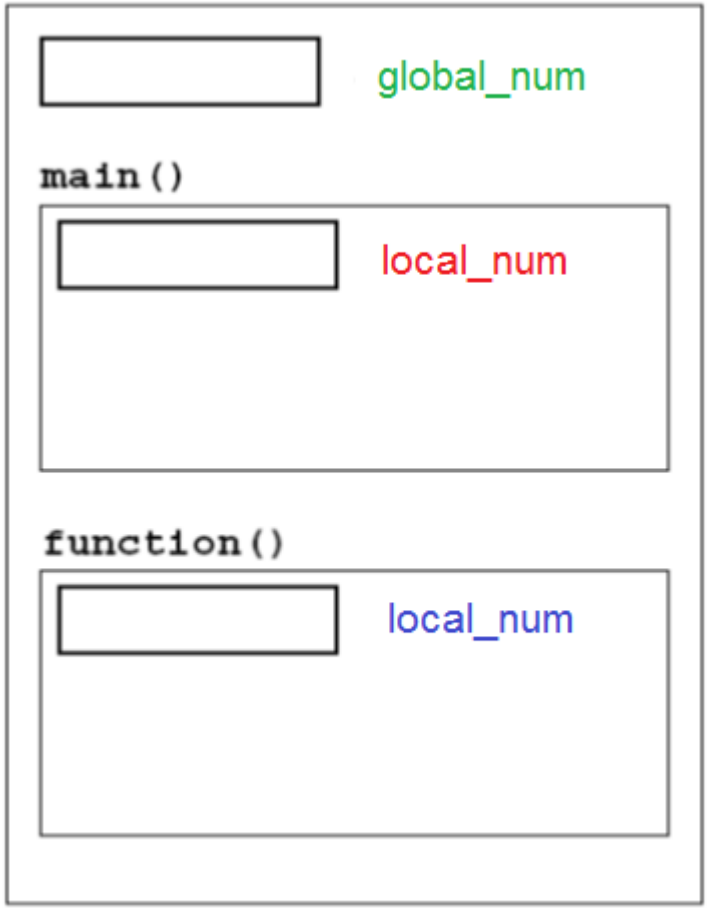
- variables declared inside a function are valid and available for use only inside the function itself, they are said to be *local variables*

- two functions can declare and use the same variable name:— which creates ***two*** *separate* and *distinct* variables, one inside *each* function

```c
#include <stdio.h>  /* LOCAL & GLOBAL variables */
int global_num;         /* external (global) declaration */
int main(void)
{
  int  local_num;           /* variable local to main() */
  void function(void);      /* function prototype */
  global_num = 25;  /* affects value in both functions */
  local_num = 30;       /* affects value in main() only */
  printf("From main() 1: global_num = %d\n", global_num);
  printf("From main() 1: local_num = %d\n\n", local_num);
  function();               /* call 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;
}

void function(void)      /* NO values passed or returned*/
{
  int local_num;        /* variable local to function() */
  local_num = 50;  /* only affects value in function() */
  printf("In function(): global_num = %d\n", global_num);
  printf("In function(): local_num = %d\n\n", local_num);
  global_num = 15;  /* affects value in both functions */
  return;           /* void function: no value returned */
}
```

Running program gives the result:

```
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
```



- `global_num` ***same*** variable in both functions

- `local_num` : ***different*** variable in each function
  - only *one* accessible at any time

Local Variable Storage Classes

- Local variables (declared *inside* a function *body*) may be given a storage class: **`auto`** or **`static`**.

`auto` - Automatic

This class enables the compiler to work most effectively at making best use of memory and instruction execution of hardware

If no explicit storage class given, the **`auto`** class is assumed as default
Function *parameters* are always **`auto`** variables

As a program executes:

The function is called:
storage is *automatically allocated* by the computer from a general pool
The function body is executed:
`auto` variables in this function are accessible and hold their values
The function **`returns`** (execution is completed):
`auto` variable storage is returned to the general pool for recycling, such variables are *no longer accessible* and their *values are lost*

```
/* Non-persistent nature of auto variables */
#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();                  /* function is called here */
  return 0;
}
/* no value passed to function and no value returned   */
void print_val(void)
{
  int val = 0;        /* allocate val as an auto variable */
  printf("Value of automatic variable val is %d\n", val);
  ++val;
  return;           /* DE-allocate the auto variable val */
}
```

The output produced by this program is:

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

Local Variable Storage Classes
`static` – Persistent Memory

A local variable which should retain its value between calls to the function must be declared with the **`static`** storage class

- storage for a **`static`** variable is NOT *allocated and recycled* each time the function is executed
- storage for **`static`** variables is allocated on a *permanent* basis
  — for the duration of executing the *whole program*
- any value stored in a **`static`** variable *retains its value* between calls to the function in which it is declared
- the *initialisation* of **`static`** variables is done at *compile time*
  — NOT during the execution of the program
- a **`static`** local variable is initialised *only once with zero*
  — effectively *before* the *program starts execution*
  — NOT each time the function in which it is declared is entered
- a static variable may make it difficult for the compiler to produce efficient instruction sequences.

```c
#include <stdio.h>
int main(void)  /* Permanent nature of static variables*/
{
  int count;          /* allocate the auto variable count */
  void print_val(void);           /* function prototype */
  for(count = 0; count < 3; ++count)
    print_val();                  /* function is called here */
  return 0;
}

          /* no argument values and no value returned   */
void print_val(void)
{
  static int val = 0;  /* value persists between calls */
  printf("Value of static variable val is %d\n", val);
  ++val;
  return;
}
```

The output produced by this program is:

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