SIES: BASIC C PROGRAMMING

L #11: FUNCTIONS

Seung Beop Lee
School of International Engineering and Science
CHONBUK NATIONAL UNIVERSITY

Outline

- Function definition
- Function declarations
- Function call
- Function arguments

Function

Function

- What is the function in C?
  - A **function** is a group of statements that together perform a task.
  - Every C program has at least one function, which is **main(),** and all the most trivial programs can define additional functions.
  - **Each function performs a specific task**.
  - The C standard library provides numerous built-in functions that your program can call.
  - A **function** is known with various names like a **method** or a **sub-routine** or a **procedure**, **etc**.

Function definition

- Syntax
  - The general form of a **function definition** in C programming language is as follows:

```
return_type function_name( parameter list )
{
   body of the function
}
```

✓ **Return Type**: A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

✓ **Function Name**: This is the actual name of the function. The **function name** and the **parameter list** together **constitute** the function signature.

✓ **Parameters**: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

✓ **Function Body**: The function body contains a collection of statements that define **what the function does**.

Function definition

- Syntax
  - The general form of a **function definition** in C programming language is as follows:

```
return_type function_name( parameter list )
{
   body of the function
}
```

✓ **Return Type**: A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

✓ **Function Name**: This is the actual name of the function. The **function name** and the **parameter list** together **constitute** the function signature.

✓ **Parameters**: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

✓ **Function Body**: The function body contains a collection of statements that define **what the function does**.

Function definition

- Syntax
  - The general form of a **function definition** in C programming language is as follows:

```
return_type function_name( parameter list )
{
   body of the function
}
```

✓ **Return Type**: A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

✓ **Function Name**: This is the actual name of the function. The **function name** and the **parameter list** together **constitute** the function signature.

✓ **Parameters**: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

✓ **Function Body**: The function body contains a collection of statements that define **what the function does**.

Function declarations

- Function declarations

```c
#include <stdio.h>

/* function declaration */
int max(int num1, int num2);

int main()
{
    /* local variable definition */
    int a = 100;
    int b = 200;
    int ret;

    /* calling a function to get max value */
    ret = max(a, b);

    printf("Max value is : %d\n", ret);

    return 0;
}

//Function definition
/* function returning the max between two numbers */
int max(int num1, int num2)
{
    /* local variable declaration */
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
```

Function definition

- Syntax
  - The general form of a **function definition** in C programming language is as follows:

```
return_type function_name( parameter list )
{
   body of the function
}
```

✓ **Return Type**: A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

✓ **Function Name**: This is the actual name of the function. The **function name** and the **parameter list** together **constitute** the function signature.

✓ **Parameters**: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

✓ **Function Body**: The function body contains a collection of statements that define **what the function does**.

Function definition

- Example
  - Following is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum between the two:

```
/* function returning the max between two numbers */
int max(int num1, int num2)
{
   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Function declarations

- Function declarations
  - A function declaration has the following parts:

```
return_type function_name( parameter list );
```

  - A **function declaration** tells the compiler about a function name and how to call the function (i.e. call by value and call by reference). The actual **body of the function (i.e. function definition)** can be defined separately.

```c
#include <stdio.h>

/* function declaration */
int max(int num1, int num2);

int main()
{
    /* local variable definition */
    int a = 100;
    int b = 200;
    int ret;

    /* calling a function to get max value */
    ret = max(a, b);

    printf("Max value is : %d\n", ret);

    return 0;
}

//Function definition
/* function returning the max between two numbers */
int max(int num1, int num2)
{
    /* local variable declaration */
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
```

```c
/* function declaration */
int max(int num1, int num2);
```

```c
//Function definition
/* function returning the max betwee
int max(int num1, int num2)
{
    /* local variable declaration */
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
```

Function declarations

- Function declarations
  - A function declaration has the following parts:

```
return_type function_name( parameter list );
```

  - A **function declaration** tells the compiler about a function name and how to call the function (i.e. call by value and call by reference). The actual **body of the function (i.e. function definition)** can be defined separately.
  - For the above defined function max(), following is the function declaration:

```
int max(int num1, int num2);
```

  - **Parameter names** are **not important** in function declaration **only their type is required**, so following is also valid declaration:

```
int max(int, int);
```

  - The function declaration is required when <u>you define a function in one source file</u> and <u>you call that function in another file</u>. In such case you should declare the function <u>at the **top** of the file</u> calling the function.

Function Call

- Calling a function
  - While creating a C function, you give **a definition of what the function has to do**. To use a function, you will have to **call that function** to perform the defined task.
  - When a program **calls a function**, program control (i.e. compiler) is transferred to the called function. A called function performs defined task, and when its return statement is executed or when its function-ending closing brace is reached, it returns program control (i.e. compiler) back to the main program.
  - To call a function, you simply need to **pass the required parameters along with function name**, and if function returns a value, then you can store returned value.

```c
#include <stdio.h>

/* function declaration */
int max(int num1, int num2);

int main()
{
    /* local variable definition */
    int a = 100;
    int b = 200;
    int ret;

    /* calling a function to get max value */
    ret = max(a, b);

    printf("Max value is : %d\n", ret);

    return 0;
}

//Function definition
/* function returning the max between two numbers */
int max(int num1, int num2)
{
    /* local variable declaration */
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
```

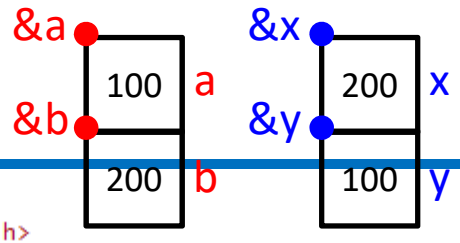
```
C:\Users\SBLEE\source\repos\Proj
Max value is : 200
```

Function Arguments

- If a function is to use **arguments**, it **must declare** variables that **accept the values of the arguments**. These variables are called the **formal parameters of the function**.
- The formal parameters behave like **other local variables inside the function** and are **created upon entry into the function** and **destroyed upon exit**.
- While calling a function, there are **two ways** that arguments can be passed to a function:

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by reference	This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Function Arguments



- Function call by value
  - The **call by value** method of passing arguments to a function copies the actual value of an argument (e.g. a and b) into the formal parameter (e.g. x and y) of the function. In this case, changes made to the parameter inside the function have no effect on the argument (i.e. a and b).
  - By default, C programming language uses **call by value method** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function.

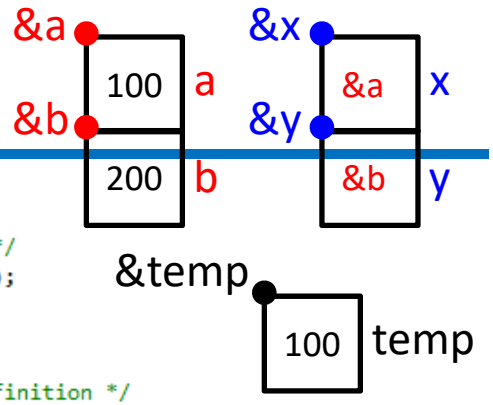
```c
#include <stdio.h>

/* function declaration */
void swap(int x, int y);

int main()
{
    /* local variable definition */
    int a = 100;
    int b = 200;

    printf("Before swap, value of a : %d\n", a);
    printf("Before swap, value of b : %d\n", b);

    /* calling a function to swap the values */
    swap(a, b);

    printf("After swap, value of a : %d\n", a);
    printf("After swap, value of b : %d\n", b);
    return 0;
}

// call by value
/* function definition to swap the values */
void swap(int x, int y)
{
    int temp;
    temp = x; /* save the value of x */
    x = y; /* put y into x */
    y = temp; /* put temp into y */
    return;
}

/*
//
```

C:\Users\SBLEE\source\repos\Project1\Debug\Project1.

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 100
After swap, value of b : 200
```

Function Argum

*x means the value saved at memory indicated by the address saved in x.



- Function call by reference
  - The **call by reference** method of passing arguments to a function copies the address of an argument (e.g. a and b) into the formal parameter (e.g. x and y). Inside the function, the address is used to access the actual argument (e.g. a and b) used in the call. This means that changes made to the parameter (e.g. x and y) **affect** the passed argument (e.g. a and b).
  - To pass the **value by reference**, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

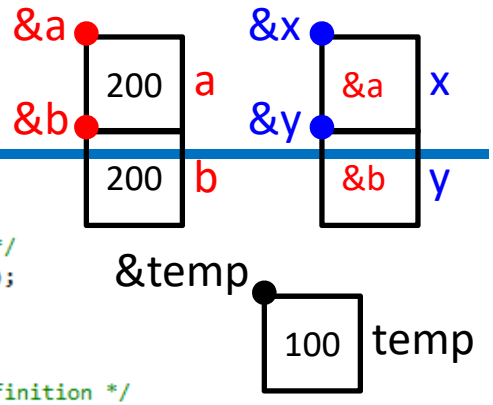
```c
#include <stdio.h>

/* function declaration */
void swap(int *x, int *y);

int main()
{
    /* local variable definition */
    int a = 100;
    int b = 200;

    printf("Before swap, value of a : %d\n", a);
    printf("Before swap, value of b : %d\n", b);

    /* calling a function to swap the values.
    * &a indicates pointer to a i.e. address of variable a and
    * &b indicates pointer to b i.e. address of variable b.
    */
    swap(&a, &b);

    printf("After swap, value of a : %d\n", a);
    printf("After swap, value of b : %d\n", b);

    return 0;
}

// call by reference
/* function definition to swap the values */
void swap(int *x, int *y)
{
    int temp;
    temp = *x; /* save the value at address x */
    *x = *y; /* put y into x */
    *y = temp; /* put temp into y */
    return;
}
/*

//
```

C:\Users\SBLEE\source\repos\Project1\Debug\Project1.exe

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 200
After swap, value of b : 100
```

Function Argum

*x means the value saved at memory indicated by the address saved in x.



- Function call by reference
  - The **call by reference** method of passing arguments to a function copies the address of an argument (e.g. a and b) into the formal parameter (e.g. x and y). Inside the function, the address is used to access the actual argument (e.g. a and b) used in the call. This means that changes made to the parameter (e.g. x and y) **affect** the passed argument (e.g. a and b).
  - To pass the **value by reference**, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

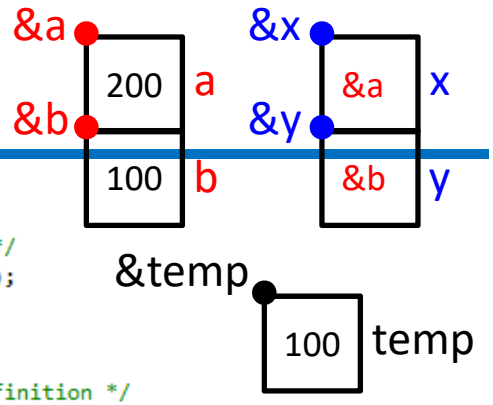
```c
#include <stdio.h>

/* function declaration */
void swap(int *x, int *y);

int main()
{
    /* local variable definition */
    int a = 100;
    int b = 200;

    printf("Before swap, value of a : %d\n", a);
    printf("Before swap, value of b : %d\n", b);

    /* calling a function to swap the values.
    * &a indicates pointer to a i.e. address of variable a and
    * &b indicates pointer to b i.e. address of variable b.
    */
    swap(&a, &b);

    printf("After swap, value of a : %d\n", a);
    printf("After swap, value of b : %d\n", b);

    return 0;
}

// call by reference
/* function definition to swap the values */
void swap(int *x, int *y)
{
    int temp;
    temp = *x; /* save the value at address x */
    *x = *y; /* put y into x */
    *y = temp; /* put temp into y */
    return;
}
/*

//
```

C:\Users\SBLEE\source\repos\Project1\Debug\Project1.exe

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 200
After swap, value of b : 100
```

Function Argum

*x means the value saved at memory indicated by the address saved in x.



- Function call by reference
  - The **call by reference** method of passing arguments to a function copies the address of an argument (e.g. a and b) into the formal parameter (e.g. x and y). Inside the function, the address is used to access the actual argument (e.g. a and b) used in the call. This means that changes made to the parameter (e.g. x and y) **affect** the passed argument (e.g. a and b).
  - To pass the **value by reference**, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

```c
#include <stdio.h>

/* function declaration */
void swap(int *x, int *y);

int main()
{
    /* local variable definition */
    int a = 100;
    int b = 200;

    printf("Before swap, value of a : %d\n", a);
    printf("Before swap, value of b : %d\n", b);

    /* calling a function to swap the values.
    * &a indicates pointer to a i.e. address of variable a and
    * &b indicates pointer to b i.e. address of variable b.
    */
    swap(&a, &b);

    printf("After swap, value of a : %d\n", a);
    printf("After swap, value of b : %d\n", b);

    return 0;
}

// call by reference
/* function definition to swap the values */
void swap(int *x, int *y)
{
    int temp;
    temp = *x; /* save the value at address x */
    *x = *y; /* put y into x */
    *y = temp; /* put temp into y */
    return;
}
/*

//
```

C:\Users\SBLEE\source\repos\Project1\Debug\Project1.exe

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 200
After swap, value of b : 100
```

Summary

Summary

- ✓ We considered the **function definition, function declaration, and function argurments.**

Thank You

(seungbeop.lee@gmail.com)