Computer Programming using C

Functions and Structured Programming

Instructor: HOU, Fen

2025

What is Structured Programming?

- **Structured programming** is a problem-solving strategy and a programming methodology, where we divide the whole program into small modules, so that program becomes easy to understand.
- The construction of a program should embody top-down design of the overall problem into finite subtasks.
- Each **subtask** can be coded directly as a **function**.
- These functions can be used in main function to solve the overall problem.

Why Do We Use Structured Programming?

- It enables the programmer to understand the program easily.
- If a program consists of thousands of instructions and an error occurs, then it is very difficult to find that error in the whole program, but in structured programming we can easily detect the error and then go to that location, and correct it.
- It saves a lot of time.

Function

- We divide the whole program into small blocks called functions
- Function is a block of statements that are executed to perform a task.
- Function plays an important role in structured programming.
- Using a function is like hiring a person to do a specific job.

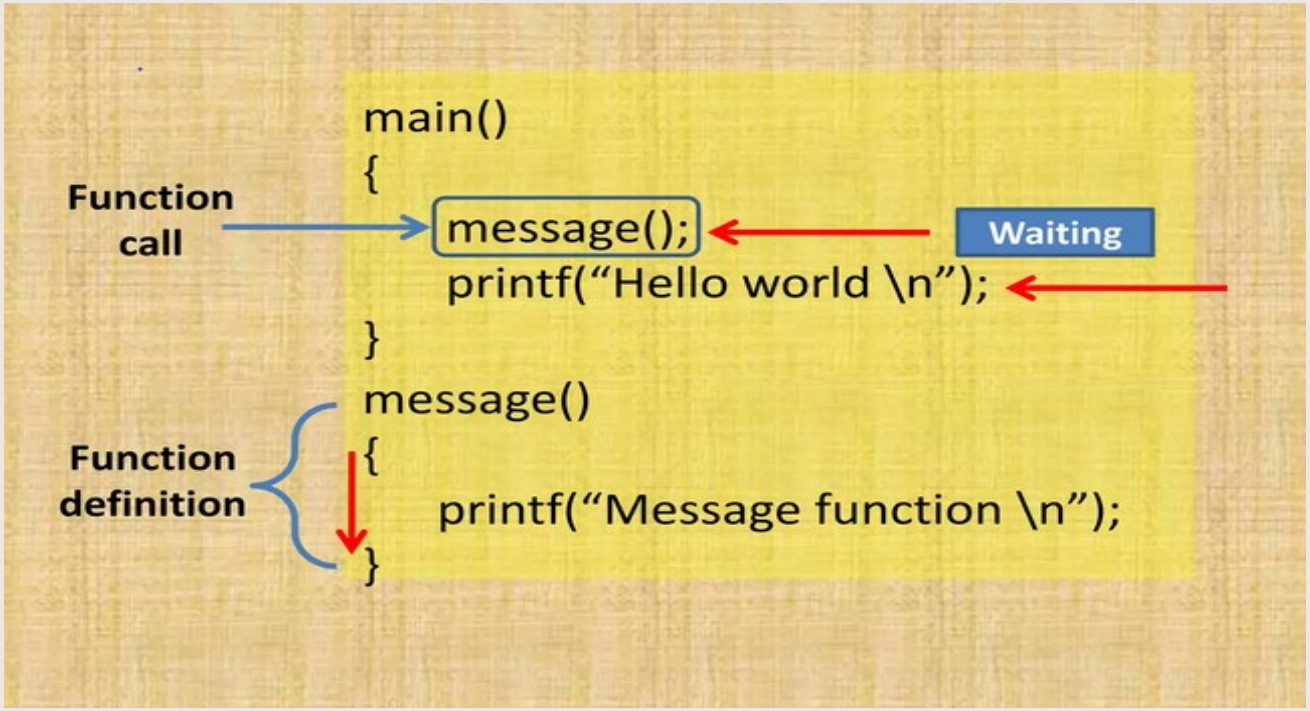
Function

- Function Call
  - When a function is called by its name the control moves to the function definition and executes all the statements written in it.
  - Semi colon is used after the name of function.

- Function Definition
  - Function definition contains the instructions that are executed when a function is called.
  - Semi colon is not used after the name of function in the function definition.

Function Call and Definition

- Function Call
  - When a function is called by its name the control moves to the function definition and executes all the statements written in it.
  - Semi colon is used after the name of function.
- Function Definition
  - Function definition contains the instructions that are executed when a function is called.
  - Semi colon is not used after the name of function in the function definition.



Function

- C program must contain at least one function.
- Execution of C program begins with main() function.
- If there are more than one function then one function must be main().
- There is no limit on number of functions in C program.
- After the execution of function, control returns to main()

Function Definition

General syntax:

```
ret-type  function-name (parameter list)
          {
              body of the function
              return  (expression)
          }
```

- A function may return any type of data **except** an array.
- The **parameters** receive the values of the **arguments** when the function is called.

Function

- Functions are the building blocks of C
- Function topics include:
  - Function type
  - Function variable
  - Function argument
  - Function scope
  - Return value
  - Function prototype
  - Recursion

Keyword - void

void ⇨ used to indicate:

A. - No Return Value.

B. - No Parameters or Arguments.

* This allows **Type compatibility** to be handled by the compiler.

```
void    Fname (void)
```

- A **VOID** function **can not be used as an operand in an expression.**

```
void func() {
printf("a good mark");
}
 void main (void)
    { int a=func();
....
}
```

- A **NON-VOID** function can be **used as an operand in an expression**.

```
x = power(y);
```

The return statement

- Returning from a function
- A function terminates execution and returns to the caller in **two ways**.

1. Occurs when the **last statement** in the function **has been executed**.

OR

2. Occurs when execute the return statement.

The return Statement

syntax:

return;

return expression;

Return (expression);

Example

return;

return 77;

return max;

return (a+b);

return ++a;

return (0);

The return Statement

- It causes an **immediate exit** from the function.
  - Returns **program control** to calling environment.
  - Allows Returning of a **value**.
- The value of ***expression*** will become the **return value of the function**.

The return Statement

Immediate exit:

```
/* Return the maximum value. */
int max(int a, int b)
{
  return (a>b) ? a : b ;
  printf("Function is performed \n");
  }
```

Return Rules

- You can use as many **return** statements as you like within a function. However, the function will stop executing as soon as it encounters the **first** return.

```
/* Return the maximum value. */
int max(int a, int b)
{
  if (a>b)
        return a;
    else
       return b;
  }
```

- The } that ends a function also causes the function to return. It is the same as a **return without** any specified value. If this occurs within a **non-void** function, then the return value of the function is **undefined**.

- **A function declared as void cannot contain a return statement** that **specifies a value.**

Placement of Functions

Before the compiler sees a call to a function it wants to know **the Number** and **Type** of its **Parameters** and the type of the Return Value of the Function.

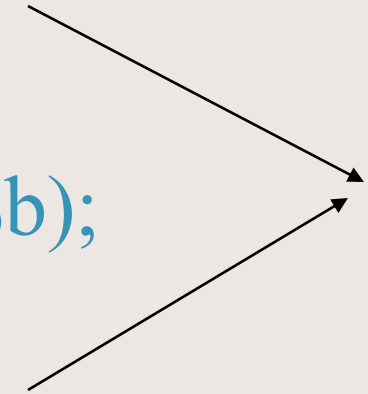
Two methods of doing so:

1. Place **Function Before main function**
2. Provide **Function Prototype （函数声明） Before the call.**

```
#include <stdio.h>

void f(int a, int b)
{
  printf("%d ", a%b);
}

int main (void)
{
  f(10,3);
  return 0;
}
```



Before Main

```
#include <stdio.h>

void f(int a, int b)
{
  printf("%d ", a%b);
}

int main (void)
{
  f(10,3);
  return 0;
}
```

Before Main

Placement of Functions

```
#include <stdio.h>

int   min(int a, int b);

int main(void)
{
   int   j, k, minimum;

   printf("Input two integers:  ");
   scanf("%d%d", &j, &k);
   minimum = min(j, k);
   printf("\nOf the two values %d and %d, "
          "the minumum is %d.\n\n", j, k, minimum);
   return 0;
}

int min(int a, int b)
{
   if (a < b)
      return a;
   else
      return b;
}
```

function prototype 函数声明

function invocation 函数调用

function definition 函数定义

Function Prototype 函数声明

A function call can appear before the function is defined (i.e., code of function) if the function prototype is provided before the call

* The called function can be **defined later in the same file**.

* The called function **can be in another file**.

Function Prototype is "Heading" of function.

FP's - **Can** be placed at the **top of the File** - **typically above any Function Definitions But below** :

#include

#define

This gives the FP's **File Visibility**. Know throughout the file.

The Function Prototype

syntax:

ret-type function-name (**parameter type list**)

- **The parameter type list is typically a comma-separated list of types.**
- **Identifiers are optional. They do not affect the prototype.**

int f(char c, int i);	is equivalent to	int f(char, int);
void f(int j, int i);	is equivalent to	void f(int, int);

Function Parameters (Formal Parameters)

- **General form:**

fun-name(**type varname1, type varname2**,….)

- All function parameters **must be declared individually**, each including **both** the type and name.

```
void f(int i, int k, int j) /*correct*/
void f(int i, k, float j)   /* wrong */
```

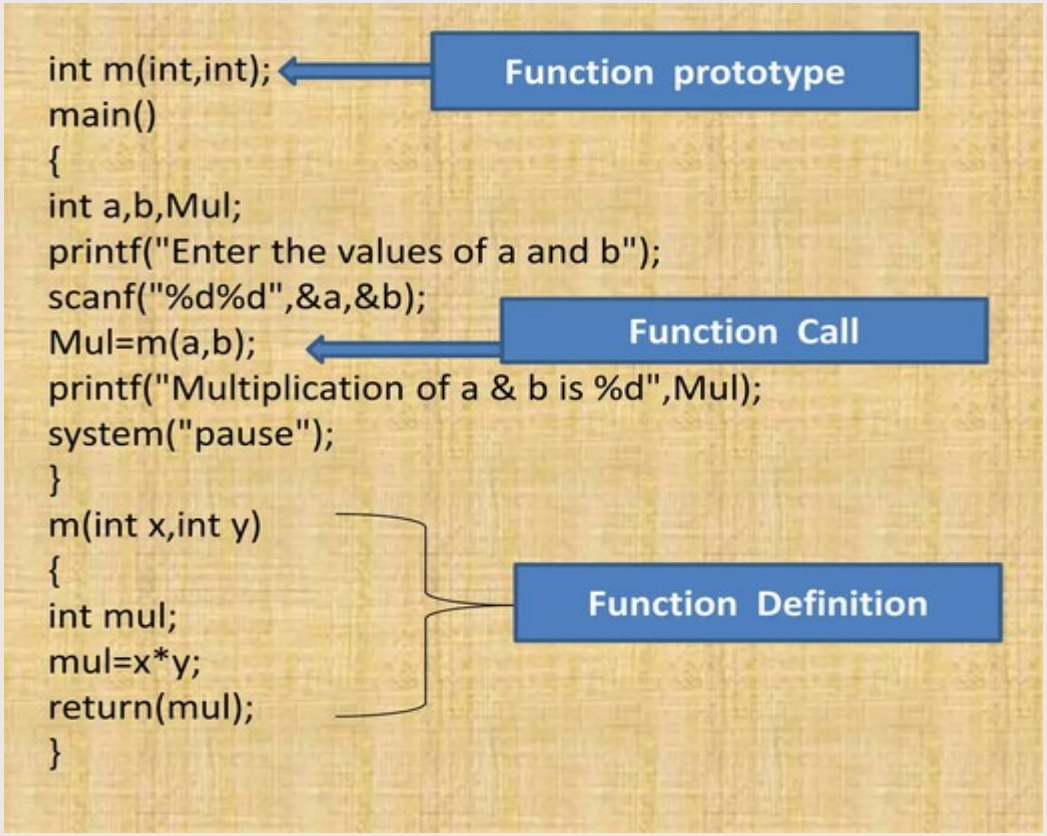
- **Parameters behave like any other local variable.**

Function Prototype

- Function prototype contains following things about the function:
  - The data type returned by the function
  - The number of parameters received
  - The data types of the parameters
  - The order of the parameters
  - If there is no datatype and return value then we use void at the place of datatype and parameters.

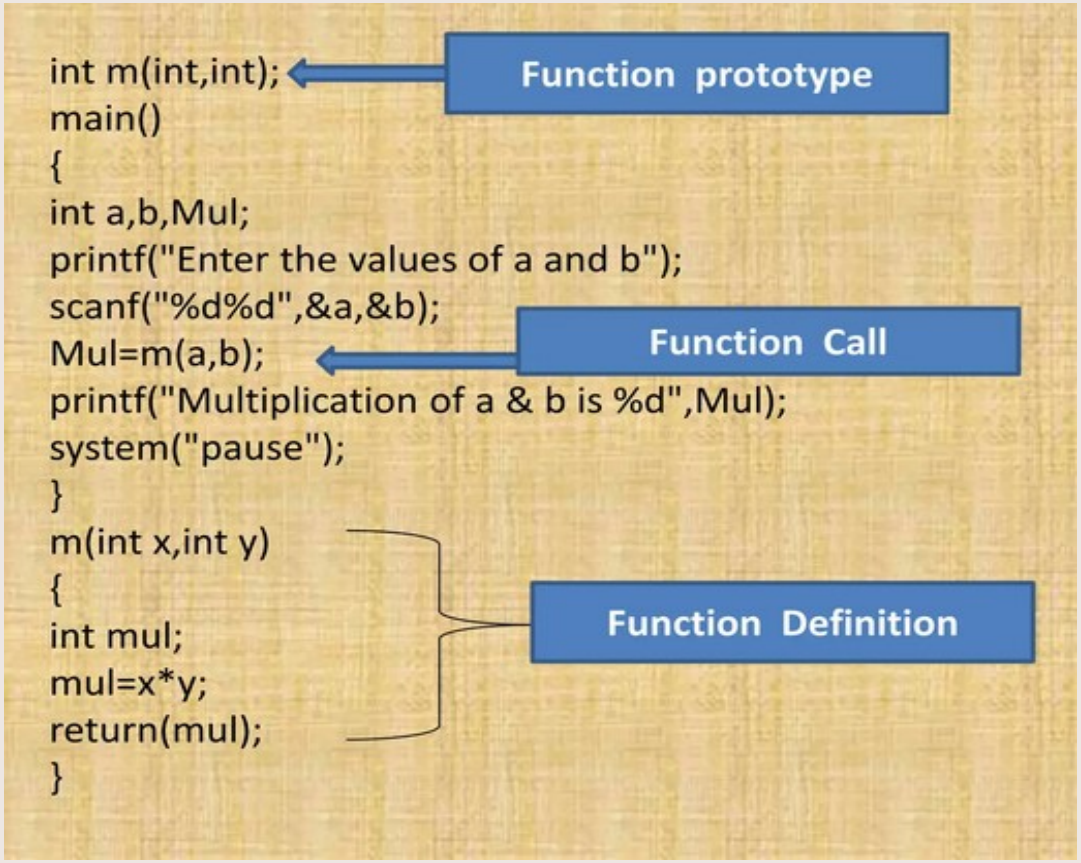
Function Prototype

- The data type returned by the function
- The number of parameters received
- The data types of the parameters
- The order of the parameters



Function Prototype

- A function can return only one value at a time.



Variable Scope

- A variable is a **named location** in memory that is used to hold a value that can be modified by the program.
- All variables **must be declared** before they can be used.

Variable Scope

- Variable scope determines the area in a program where variable can be accessed.
- When a variable loses its scope, it means its value is lost.
- Common types of variables in C include
  - Local variable
  - Global variable
- Global variable can be accessed anywhere in a program.
- Local variable can be accessed only in the function in which it is declared.

- **Where variables are declared**:

  - **Inside functions**:

    local variables （局部变量）

  - **Outside of all functions**:

    global variables （全局变量）

  - **In the function heading**:

    formal parameters （形参）

The Types of Variables

```
#include<stdio.h>
int a,b;                    <-- Global Variable
main()
{
printf("Enter the values of a & b");
scanf("%d%d",&a,&b);
sum(a,b);
system("pause");
  }
sum(inta,intb)
{
int c;                      <-- Local Variable
c=a+b;
printf("Sum is %d",c);
  }
```

Local Variable

- Declared **inside a function**.
- Can be used only by statements that are **inside** the block in which the variables are declared.
- A local variable is **created** upon entry into its block and **destroyed** upon exit.

```
void func1(void)
{
  int x;
  x = 10;
}

void func2(void)
{
  int x;
  x = -199;
}
```

```
void f(void)
{
  int t;
  scanf("%d", &t);

  if(t==1) {
    char s; /*  created only upon
                          entry into this block */
  }

   /* s is not known here */
}
```

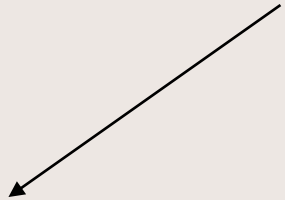
```
#include <stdio.h>
int main(void)
{
  int  x;
  x = 10;

  if(x == 10) {
    int  x;        /*  this masks the outer x */
    x = 99;
    printf("Inner x: %d\n", x);
  }

   printf("Outer x: %d\n", x);
   return 0;
}
```

```
#include <stdio.h>
void f(void);
int main(void)
{
  int i;
  for(i=0; i<10; i++)  f();
  return 0;
}

void f(void)
{
  int j = 10;
  printf("%d\n", j);
  j++;
}
```



Prints What?

```
#include <stdio.h>
void f(void);
int main(void)
{
  int i;
  for(i=0; i<10; i++)  f();
  return 0;
}

void f(void)
{
  int j = 10;
  printf("%d\n", j);
  j++;
}
```

Prints What?

```
10
10
10
10
10
10
10
10
10
10
```

Global Variables

- They are **known throughout the program** in which they are defined.
- They are **declared outside of any function**
- Storage for global variables is in a **fixed region (Global Storage Area)**.

The Scope of Global Variables

```
01. int a, b;  /*global variable */
02. void func1 () {
03. .....
04. }
05. float x, y;  /*  global variable*/
06. int func2() {
07. ....
08.  }
09. int main() {
10. ....
11. return 0;
12. }
```

- The global variables a and b are valid in functions func1, func2, and main;
- The global variables x and y are valid in functions func2, and main;

Global Variable and Local Variable

When global variable and local variable are the same identifier, the global variable is invalid within the scope of local variable. That is, the local variable masks the global variable.

```
int a=3,b=5;
max(int  a, int b)
{   int c;
    c=a>b?a:b;
    return(c);
}
main()
{   int a=8;
    printf(max=%d,max(a,b));
}
```

Global Variable and Local Variable

When global variable and local variable are the same identifier, the global variable is invalid within the scope of local variable. That is, the local variable masks the global variable.

```
int a=3,b=5;
max(int  a, int b)
{   int c;
    c=a>b?a:b;
    return(c);
}
main()
{   int a=8;
    printf(max=%d,max(a,b));
}
```

Result:

max=8

```c
#include <stdio.h>

int count;  /*  GLOBAL  */

void func1(void);
void func2(void);

int main(void)
{
  count = 100;
  func1();
  printf("count is %d", count);
  return 0;
}
```

```c
void func1(void)
{
  int temp;
  temp = count;
  func2();
  printf("count is %d", count);
}

void func2(void)
{
 int count;   /* LOCAL */
 for(count=1; count<10; count++)
    putchar('.');
 printf("count is %d", count);
}
```

Global Variable and Local Variable

```
01. #include <stdio.h>
02.
03. int n = 10;      /*global Variable */
04.
05. void func1(){
06.     int n = 20;     /* local variable */
07.     printf("func1 n: %d\n", n);
08. }
09.
10. void func2(int n){
11.     printf("func2 n: %d\n", n);
12. }
13.
14. void func3(){
15.     printf("func3 n: %d\n", n);
16. }
17.
18. int main(){
19.     int n = 30;     /* local variable */
20.     func1();
21.     func2(n);
22.     func3();
23.
24.     {
25.         int n = 40;     /* local variable */
26.         printf("block n: %d\n", n);
27.     }
28.     printf("main n: %d\n", n);
29.
30.     return 0;
31. }
```

Global Variable and Local Variable

```c
#include <stdio.h>

int n = 10; /*global Variable */

void func1(){
    int n = 20; /* local variable */
    printf("func1 n: %d\n", n);
}

void func2(int n){
    printf("func2 n: %d\n", n);
}

void func3(){
    printf("func3 n: %d\n", n);
}

int main(){
    int n = 30; /* local variable */
    func1();
    func2(n);
    func3();

    {
        int n = 40; /* local variable */
        printf("block n: %d\n", n);
    }
    printf("main n: %d\n", n);

    return 0;
}
```

- Result

  func1 n:20

  func2 n:30

  func3 n:10

  block n:40

  main n:30

Calculate the area and volume of a cuboid

```
01. #include <stdio.h>
02.
03. int s1, s2, s3;  /*area*/
04.
05. int vs(int a, int b, int c){
06.     int v;  /* volume */
07.     v = a * b * c;
08.     s1 = a * b;
09.     s2 = b * c;
10.     s3 = a * c;
11.     return v;
12. }
13.
14. int main(){
15.     int v, length, width, height;
16.     printf("Input length, width and height: ");
17.     scanf("%d %d %d", &length, &width, &height);
18.     v = vs(length, width, height);
19.     printf("v=%d, s1=%d, s2=%d, s3=%d\n", v, s1, s2, s3);
20.
21.     return 0;
22. }
```

Calculate the area and volume of a cuboid

```
01.  #include <stdio.h>
02.
03.  int s1, s2, s3;  /*area*/
04.
05.  int vs(int a, int b, int c){
06.      int v;  /* volume */
07.      v = a * b * c;
08.      s1 = a * b;
09.      s2 = b * c;
10.      s3 = a * c;
11.      return v;
12.  }
13.
14.  int main(){
15.      int v, length, width, height;
16.      printf("Input length, width and height: ");
17.      scanf("%d %d %d", &length, &width, &height);
18.      v = vs(length, width, height);
19.      printf("v=%d, s1=%d, s2=%d, s3=%d\n", v, s1, s2, s3);
20.
21.      return 0;
22.  }
```

- Result

Input length, width and height: 10 20 30

V=6000, s1=200, s2=600, s3=300

- **Avoid using unnecessary global variables**.
  - They take up memory the **entire time** your program is executing.
  - Using a global variable where a local variable will do makes a function **less general**.
  - Using a large number of global variables can **lead to program errors**.

The Scope of a Function

- Each function is a discrete **block of code**.
- A function's code is **private** to that function and cannot be **accessed** by any statement in any other function except **through a call to that function**.

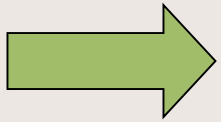
- Variables that are **defined within a Function** are **local variables**. Such variables are freed from memory when their function completes execution.

- Therefore, a **local variable** cannot hold its value between function calls.

- The **formal parameters** to a function also fall **within the function's scope**.
  - A **parameter is known** throughout the **entire** function.
  - A parameter comes into existence when the function is called and is destroyed when the function is exited,(i.e., the **Extent** of the variable).
- A function **cannot** be defined within a function (**no nested-functions**).

Invocation and Call-by-Value

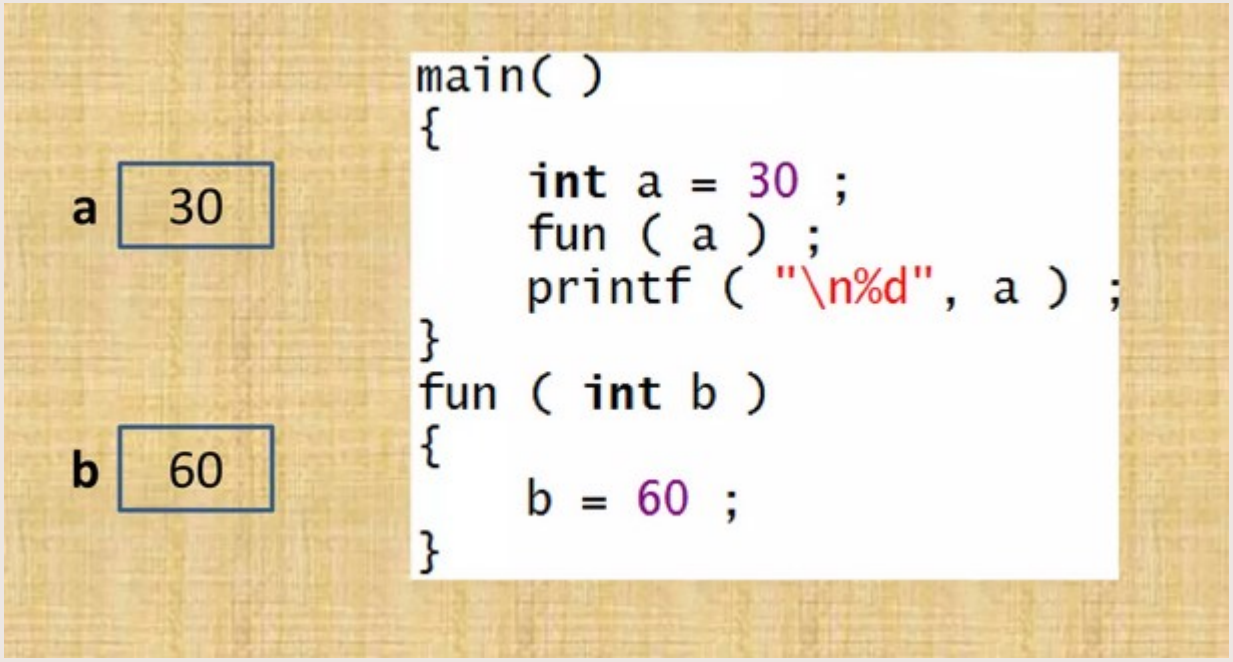
* When an Argument (variable, constant, or expression) is passed to a matching parameter (declared in Function Heading), **A Copy of the Argument Value** is made and is passed to the parameter.

Arguments in calling Module.  **Parameters in called Module.**

* **Changes made to the parameter have no effect on the argument**. i.e., the parameter is implemented as a local variable which is initialized to the value passed from the Argument.

Invocation and Call-by-Value

- A Copy of the Argument Value is made and is passed to the matching parameter.
- Main () is the calling module/function while fun() is the called module/function.
- Changes made to the parameter b have no effect on the argument a.



```
#include <stdio.h>
int sqr(int x);

int main(void)
{
  int t=10;
  printf("%d and %d", sqr(t) , t);
  return 0;
 }

int sqr(int t)
{
  t = t*t;
  return t;
}
```

What is output?

```
#include <stdio.h>
int sqr(int x);

int main(void)
{
  int t=10;
  printf("%d and %d", sqr(t) , t);
  return 0;
 }

int sqr(int t)
{
  t = t*t;
  return t;
}
```

What is output?

```
100 and 10
```

Call-by-Value

```
01. #include <stdio.h>
02.
03. /*get the summation from m to n*/
04. int sum(int m, int n) {
05.     int i;
06.     for (i = m+1; i <= n; ++i) {
07.         m += i;
08.     }
09.     return m;
10. }
11.
12. int main() {
13.     int a, b, total;
14.     printf("Input two numbers: ");
15.     scanf("%d %d", &a, &b);
16.     total = sum(a, b);
17.     printf("a=%d, b=%d\n", a, b);
18.     printf("total=%d\n", total);
19.
20.     return 0;
21. }
```

Call-by-Value

```c
#include <stdio.h>

/*get the summation from m to n*/
int sum(int m, int n) {
    int i;
    for (i = m+1; i <= n; ++i) {
        m += i;
    }
    return m;
}

int main() {
    int a, b, total;
    printf("Input two numbers: ");
    scanf("%d %d", &a, &b);
    total = sum(a, b);
    printf("a=%d, b=%d\n", a, b);
    printf("total=%d\n", total);

    return 0;
}
```

- Result

  Input two numbers: 1 100

  a=1, b=100

  total=5050

Call by Value

- **variable, constant, or expression** is passed to a matching parameter.

```
01.  total = sum(10, 98);          /*constant*/
02.  total = sum(a+10, b-3);       /*expression*/
03.  total = sum( pow(2,2), abs(-100) );   /*return value of a function*/
```

```
/*  This function returns the average of 3 exam
    scores which are passed call-by-value.  */
#include  <stdio.h>
double average(int, int, int);

int  main(void)  {
    int  test1, test2, test3;
    double    avg;
    scanf("%d%d%d", &test1, &test2, &test3);
    avg = average(test1, test2, test3);
    printf(" The average is: %f", avg);
}
```

```
double average(int exam1, int exam2, int exam3)
{   double  testavg;
     testavg = (float)(exam1 + exam2 + exam3)/ 3;
     return   testavg;
}
```

Call by Value

```
01. #include <stdio.h>
02.
03. int sum(int m, int n){
04.     int i, sum=0;
05.  /*m,n,i,and sum are local Variable, valid only in function sum */
06.     for(i=m; i<=n; i++){
07.         sum+=i;
08.     }
09.     return sum;
10. }
11.
12. int main(){
13.     int begin = 5, end = 86;
14.     int result = sum(begin, end);
15.  /* begin, end, result are local variable, valid only in function main */
16.     printf("The sum from %d to %d is %d\n", begin, end, result);
17.
18.     return 0;
19. }
```

The exit( ) Function

- General form

  void exit(int *return_code*);

- This function causes **immediate termination of the entire program**, forcing a return to the operating system.

- The value of *return_code* is returned to the calling process, which is usually the operating system.

•Programmers frequently use **exit** when a **mandatory condition for program execution is not satisfied.**

```
#include <stdlib.h>

int main(void)
{
  if(!virtual_graphics()) exit(1);
  play();
    . . .
}
```