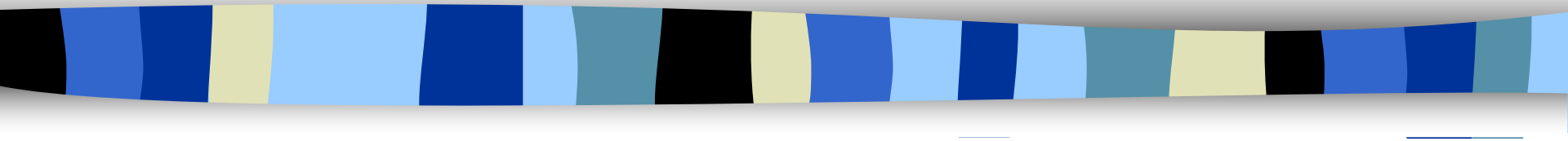
Functions



Introduction

- Man is an intelligent species, but still cannot perform all of life's tasks all alone.
- You may
  - call a mechanic to fix up your bike,
  - hire a gardener to mow your lawn,
  - rely on a store to supply you groceries every month.

- A computer program (except for the simplest one) finds itself in a similar situation. It cannot handle all the tasks by itself.
- Instead, it requests other program like entities called 'functions' in C to get its tasks done.

Contd.

- A function is a sub-program i.e. subset of the program.
- A function is a group of statements that together perform a task.
- It is written only once and re-used whenever required, as many times as required.
- Every C program has at least one function, which is main().

  – **Every C program consists of one or more functions.**
  – **A function will carry out its intended action whenever it is *called or invoked.***

Contd.

- **Function**

  self-contained block of statements that perform some specific, well-defined task.

  using a function is something like hiring a person to do a specific job for you. Sometimes the interaction with this person is very simple; sometimes it's complex.

- **A function can also be referred as a method or a sub-routine or a procedure, etc.**

Why Functions?

- **Benefits**
  - **Modularity: Break big problems into smaller ones.**
    - **Manageable program development.**
    - **Construct a program from small pieces or components.**
  - **Re-Use: Build using what others have already built.**
    - **Use existing functions as building blocks for new programs.**
  - **Abstraction: hide internal details**
    - **E.g., library functions.**
  - **In large programs, debugging and editing tasks is easy with the use of functions.**

Why Functions?

- **Example – Roti**

- **Modularity-**

Process broken down to three parts - making the dough, rolling and roasting.

- **Re-Use-**

Flour, Rolling-pin and gas stove are not made by us but re-used. Any brand flour can be used.

- **Abstraction-**

While writing recipe for roti, we are not concerned with details of each small process - as long as the roti is edible, nutritious and soft enough.

Type of Function

- There are two type of function in C Language.
- Library function or pre-defined or built-in function.
  - These functions are provided by the system and stored in library, therefore it is also called ‘Library Functions‘. e.g. scanf(), printf(), sqrt(), pow(), strcmp(), strlen() etc.
  - To use these functions, you just need to include the appropriate C header files.

- User defined function.
  - These functions are defined by the user at the time of writing the program according to their requirement.
  - For example suppose user want to create a function to add two number then he can create a function with name sum().

Using Functions - Example

```
int main()
{
int a,k,b;
b = pow(a,k);
printf("%d",b);
return 0;
}
```

Note- For every input to function, some output is defined. It either returns the output to the calling sub-program, or performs some well defined procedure on the input.

- Function pow() - Inputs are values of `a' and `k‘.
- Output value of `a^k ', which is assigned to `b'
- Function printf() – Inputs are format string and values of respective variables.
- Prints the values on the screen in the specified format.

Functions

- For every input to it, some output is defined. It either returns the output to the calling sub-program, or performs some well defined procedure on the input.

- In general, a function will process information that is passed to it from the calling portion of the program, and returns a single value.
  - Information is passed to the function via special identifiers called *arguments or parameters.*
  - The value is returned by the "*return" statement*.

- Some functions may not return anything.
  - Return data type specified as "*void".*

Parts of Function

- Function Prototype (function declaration)
- Function Definition
- Function Call

Function Prototype/ Declaration

- A function **declaration** tells the compiler about a function name and how to call the function (how many input parameters it will take and of what type. It also tells what will be the return type).

- The actual body of the function can be defined separately.
- Syntax:

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

- Example:

```
int sum(int a, int b);
void display(float c, int d, int e);
```

- ***Note:*** *At the time of function declaration function must be terminated with* ***;***
- *Function prototypes are usually written at the beginning of a program, ahead of any functions (including main()).*

Function Prototype/ Declaration

- **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**. (default is **int**).

- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature. Can be any valid identifier.

- **Parameters/ Arguments** − This is a value which is pass in function at the time of calling of function. 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. Parameter names are not important in function declaration only their type is required. int sum(int, int);

Declaring Functions: Points to Remember

- Every function should have a **unique name**.
- There is no restriction on what data type you can pass as parameter or receive as return value.
- Specifying the **return type is optional** (assumed as int if not specified).
- **Parameters are optional**; that is, a function may contain no parameters. int sum();

- **Declaration is not Definition!**
- Prototype only needed if function definition comes after use in program.
- The argument names can be different; but it is a good practice to use the same names as in the function definition

Function Definition

- **Syntax:**

```
return_type function_name (parameter list)
{
body of the function // declarations and statements
}
```

- **Function-name**: any valid identifier
- **Return-type**: data type of the result (default int)
  - void – indicates that the function returns nothing
- **Parameter-list**: comma separated list, declares parameters
  - A type must be listed explicitly for each parameter unless, the parameter is of type int
- **Body of the function**: contains a collection of statements that define what the function does.

Function Definition

- The first line contains the return-value-type, the function name, and optionally a set of comma-separated arguments enclosed in parentheses.
  - Each argument has an associated type declaration.
  - The arguments are called *formal arguments or formal parameters.*
- Example:

```
int add (int A, int B)
```

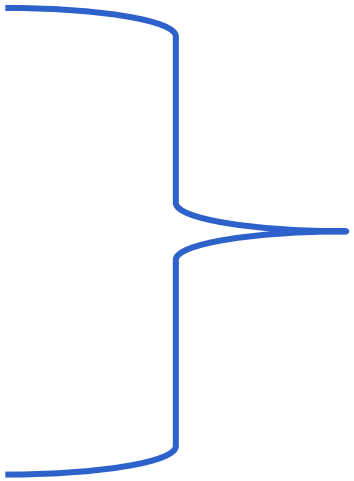
- The argument data types can also be declared on the next line:

```
int add (A, B)
        {
                int A, B; -----
        }
```

Function Definition

- The body of the function is actually a compound statement that defines the action to be taken by the function.

```
int add (int A, int B)
{
        int C;
        C=A+B;
        return C;
}
```



Body of function

Function Declaration and Definition Example

```
int max(int num1, int num2); // function declaration
main() {
------
}
int max(int num1, int num2) // function definition starts here
{
   int result;
   if (num1 > num2)
      result = num1;
   else
      result = num2;
   return result;
}
```

Function Definition: Points to Remember

- Note that the first line has to be consistent with the declaration.
- Declarations and statements are written in the function body which is enclosed in braces { }.
- Functions can not be defined inside other functions.
- For all functions which return some value, the return statement is mandatory. The type of the value being returned should be consistent with the declaration.
- If type is not consistent, typecasting has to be used to explicitly convert the return value.
- Any statements and expressions can be put into the function body.
- Return type and return statement are optional.
- *The data types are assumed to be of type int if they are not shown explicitly. However, the omission of the data types is considered poor programming practice, even if the data items are integers.*

Function Call

- To use a function, you will have to call that function to perform the defined task.
- When a sub-program calls a function, the program control is transferred to the called function.
- A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the calling sub-program (calling function).
- For calling any function just write name of function and if any parameter is required then pass parameter.
- **A function call has following syntax;**

  function-name (parameter-list);

- **Example:** max(12,5);

Function Call Example

```
#include<stdio.h>
void message();
void main( )
{
     message( ) ;                    // function call
     printf ( "\nCry, and Cry with yourself!" ) ;
}
void message( )
{
    printf ( "\nSmile, and the world smiles with you..." ) ;
    return;                          //optional
}
```

Smile, and the world smiles with you...
Cry, and Cry with yourself

- main( ) becomes the ‘calling’ function, whereas message( ) becomes the ‘called’ function.

Function Call: Points to Remember

- If the function returns a value, the function access is often written as **an** assignment statement; e.g.,
- **y = polynomial(x);**
- This function call causes the value returned by the function to be assigned to the variable **y.**
- A function cannot be defined within another function.
  - All function definitions must be disjoint.
- It is possible to call another function from one function.
- Nested function calls are allowed.
  - A calls B, B calls C, C calls D, etc.
  - The function called last will be the first to return.
- A function can also call itself, either directly or in a cycle.
  - A calls B, B calls C, C calls back A.
  - Called *recursive call* or *recursion*.

Example:

```
#include<stdio.h>
void italy( ) ;
void brazil( ) ;
void argentina( ) ;
void main( )
{
printf ( "\nI am in main" ) ;
    italy( ) ;
    brazil( ) ;
    argentina( ) ;
}
```

```
void italy( )
{
    printf ( "\nI am in italy" ) ;
}
void brazil( )
{
    printf ( "\nI am in brazil" ) ;
}
void argentina( )
{
   printf ( "\nI am in argentina" );
}
```

Output

I am in main

I am in italy

I am in brazil

I am in argentina

Structure of C program with functions

- Include statements
- Function prototypes/Declarations
- Main function
- Body of main function & call to the functions
- Function definitions

```
#include<stdio.h>
void add();
void main()
{
      add();
 }
 void add()
 {
    Body of add function
 }
```

Example 1 – how values are passed & how program will execute

```
void calsum(int, int, int);
void main( )
{
   int a, b, c;
   printf ( "\n Enter any three numbers " ) ;
   scanf ( "%d %d %d", &a, &b, &c ) ;
    calsum ( a, b, c ) ;
   }
void calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
   printf ( "\nSum = %d", d) ;
}
```

Example 1

```
void calsum(int, int, int);
void main( )
{
   int a, b, c ;
   printf ( "\n Enter any three numbers " ) ;
   scanf ( "%d %d %d", &a, &b, &c ) ;
    calsum ( a, b, c ) ;
   }
void calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
   printf ( "\nSum = %d", d) ;
}
```

Example 1

```
void calsum(int, int, int);
void main( )
{
   int a, b, c;
   printf ( "\n Enter any three numbers " ) ;
   scanf ( "%d %d %d", &a, &b, &c ) ;
    calsum ( a, b, c ) ;
   }
void calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
   printf ( "\nSum = %d", d) ;
}
```

Example 1

```
void calsum(int, int, int);
void main( )
{
   int a, b, c;
   printf ( "\n Enter any three numbers " ) ;
   scanf ( "%d %d %d", &a, &b, &c ) ;
    calsum ( a, b, c ) ;
   }
void calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
   printf ( "\nSum = %d", d) ;
}
```

Example 1

```
void calsum(int, int, int);
void main( )
{
   int a, b, c;
   printf ( "\n Enter any three numbers " ) ;
   scanf ( "%d %d %d", &a, &b, &c ) ;
    calsum ( a, b, c ) ;
   }
void calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
   printf ( "\nSum = %d", d) ;
}
```

Enter any three numbers

Example 1

```
void calsum(int, int, int);
void main( )
{
   int a, b, c;
   printf ( "\n Enter any three numbers " ) ;
   scanf ( "%d %d %d", &a, &b, &c ) ;
    calsum ( a, b, c ) ;
   }
void calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
   printf ( "\nSum = %d", d) ;
}
```

Enter any three numbers 10 20 30

Example 1

```
void calsum(int, int, int);
void main( )
{
   int a, b, c;
   printf ( "\n Enter any three numbers " ) ;
   scanf ( "%d %d %d", &a, &b, &c ) ;
    calsum ( a, b, c ) ;
   }
void calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
   printf ( "\nSum = %d", d) ;
}
```

Example 1

```
void calsum(int, int, int);
void main( )
{
   int a, b, c;
   printf ( "\n Enter any three numbers " ) ;
   scanf ( "%d %d %d", &a, &b, &c ) ;
    calsum ( a, b, c ) ;
   }
void calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
   printf ( "\nSum = %d", d) ;
}
```

Example 1

```
void calsum(int, int, int);
```

```
void calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
   printf ( "\nSum = %d", d) ;
}
```

Example 1

```
void calsum(int, int, int);
```

```
void calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
   printf ( "\nSum = %d", d) ;
}
```

Example 1

```
void calsum(int, int, int);
```

```
void calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
   printf ( "\nSum = %d", d) ;
}
```

Example 1

```
void calsum(int, int, int);

void calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
   printf ( "\nSum = %d", d) ;
}
```

Example 1

```
void calsum(int, int, int);
```

```
void calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
   printf ( "\nSum = %d", d) ;
}
```

Enter any three numbers 10 20 30

Sum = 60

Example 1

```
void calsum(int, int, int);
```

```
void calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
   printf ( "\nSum = %d", d) ;
    return;
}
```

Example 1

```
void calsum(int, int, int);
void main( )
{
   int a, b, c;
   printf ( "\n Enter any three numbers " ) ;
   scanf ( "%d %d %d", &a, &b, &c ) ;
    calsum ( a, b, c ) ;
   }
void calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
   printf ( "\nSum = %d", d) ;
    return;
}
```

Example 2

```
int calsum(int, int, int);
void main( )
{
   int a, b, c, sum; //new variable to catch value returned by calsum
   printf ( "\n Enter any three numbers " ) ;
   scanf ( "%d %d %d", &a, &b, &c ) ;
    sum=calsum ( a, b, c ) ;
   printf ( "\nSum = %d", sum) ;
   }
int calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
   return d;
}
```

Example 2

```
int calsum(int, int, int);
void main( )
{
   int a, b, c, sum ;
   printf ( "\n Enter any three numbers " ) ;
   scanf ( "%d %d %d", &a, &b, &c ) ;
    sum=calsum ( a, b, c ) ;
   printf ( "\nSum = %d", sum) ;
   }
int calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
return d;
}
```

Example 2

```
int calsum(int, int, int);
void main( )
{
   int a, b, c, sum;
   printf ( "\n Enter any three numbers " ) ;
   scanf ( "%d %d %d", &a, &b, &c ) ;
    sum=calsum ( a, b, c ) ;
   printf ( "\nSum = %d", sum) ;
   }
int calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
return d;
}
```

Example 2

```
int calsum(int, int, int);
void main( )
{
   int a, b, c, sum;
   printf ( "\n Enter any three numbers " ) ;
   scanf ( "%d %d %d", &a, &b, &c ) ;
    sum=calsum ( a, b, c ) ;
   printf ( "\nSum = %d", sum) ;
   }
int calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
return d;
}
```

Enter any three numbers

Example 2

```
int calsum(int, int, int);
void main( )
{
   int a, b, c, sum;
   printf ( "\n Enter any three numbers " ) ;
   scanf ( "%d %d %d", &a, &b, &c ) ;
    sum=calsum ( a, b, c ) ;
   printf ( "\nSum = %d", sum) ;
   }
int calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
return d;
}
```

Enter any three numbers 10 20 30

Example 2

```
int calsum(int, int, int);
void main( )
{
   int a, b, c, sum;
   printf ( "\n Enter any three numbers " ) ;
   scanf ( "%d %d %d", &a, &b, &c ) ;
    sum=calsum ( a, b, c ) ;
   printf ( "\nSum = %d", sum) ;
   }
int calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
return d;
}
```

Example 2

```
int calsum(int, int, int);
void main( )
{
   int a, b, c, sum;
   printf ( "\n Enter any three numbers " ) ;
   scanf ( "%d %d %d", &a, &b, &c ) ;
    sum=calsum ( a, b, c ) ;
   printf ( "\nSum = %d", sum) ;
   }
int calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
return d;
}
```

Example 2

```
int calsum(int, int, int);
```

```
int calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
return d;
}
```

Example 2

```
int calsum(int, int, int);
```

```
int calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
return d;
}
```

Example 2

```
int calsum(int, int, int);
```

```
int calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
return d;
}
```

Example 2

```
int calsum(int, int, int);
```

```
int calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
return d;
}
```

Example 2

```
int calsum(int, int, int);
void main( )
{
   int a, b, c, sum;
   printf ( "\n Enter any three numbers " ) ;
   scanf ( "%d %d %d", &a, &b, &c ) ;
    sum=calsum ( a, b, c ) ;
   printf ( "\nSum = %d", sum) ;
   }
int calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
return d;
}
```

Example 2

```
int calsum(int, int, int);
void main( )
{
   int a, b, c, sum;
   printf ( "\n Enter any three numbers " ) ;
   scanf ( "%d %d %d", &a, &b, &c ) ;
    sum=calsum ( a, b, c ) ;
   printf ( "\nSum = %d", sum) ;
   }
int calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
return d;
}
```

Example 2

```
int calsum(int, int, int);
void main( )
{
   int a, b, c , sum;
   printf ( "\n Enter any three numbers " ) ;
   scanf ( "%d %d %d", &a, &b, &c ) ;
    sum=calsum ( a, b, c ) ;
   printf ( "\nSum = %d", sum) ;
   }
int calsum ( int x, int y, int z )
{
   int d ;
   d = x + y + z ;
return d;
}
```

Enter any three numbers 10 20 30

Sum = 60

Function Return Value

- Information is returned from the function to the calling portion of the program via the return statement.

- The return statement also causes the program logic to return to the point from which the function was accessed.
- In general terms, the return statement is written as

  return ***expression,***

- The value of the ***expression*** is returned to the calling portion of the program.

- The ***expression*** is optional. If the ***expression*** is omitted, the return statement simply causes control to revert back to the calling portion of the program, without any transfer of information.

Function Return Value

- Only one expression can be included in the return statement. Thus, a function can return only one value to the calling portion of the program via return
- A function definition can include multiple return statements, each containing a different expression. But only one return statement gets executed.
- Functions that include multiple branches often require multiple returns. For example, as shown on next slide.
- Also, the **return statement** need not always be present at the end of the called function.

Function Return Value

```
char lower-to-upper(char c1)
{
if (c1 >= 'a' && c1 <= 'z')
return('A' + C1 - 'a');
else
return(c1);
}
```

- The return statement can be absent altogether from a function definition, though this is generally regarded as poor programming practice.

Function Return Value

- **Returning control**
  - **If nothing returned**
    - **return;**
    - **or, until reaches right brace }**
  - **If something returned**
    - **return *expression;***
- If the data type specified in the first line of definition is inconsistent with the expression appearing in the **return** statement, the compiler will attempt to convert the quantity represented by the expression to the data type specified in the first line.
- This could result in a compilation error, or it may involve a partial loss of data (e.g., due to truncation).
- In any event, inconsistencies of this type should be avoided.

Function Calling Types

- There are different types of function calling. Depending on the number of parameters it can accept , function can be classified into following 4 types –

Function Type	Parameter	Return Value
Type 1	Accepting Parameter	Returning Value
Type 2	Accepting Parameter	Not Returning Value
Type 3	Not Accepting Parameter	Returning Value
Type 4	Not Accepting Parameter	Not Returning Value

Type 1 : Accepting Parameter and Returning Value

- Above type of method is written like this –

```
int add(int i, int j)
{
  return i + j;
}
```

and Called like this –

```
int answer = sum(2,3);
```

- We need to assign the function call to any of the variable since we need to capture returned value.

Type 2 : Accepting Parameter and Not Returning Value

- Above type of method is written like this –

```
void add(int i, int j)
{
  printf("%d",i+j);
}
```

above method can be called using following syntax –

```
sum(2,3);
```

Type 3 : Not Accepting Parameter but Returning Value

- Above type of method is written like this –

```
int add()
{
  int i, int j;
  i = 10;
  j = 20;
  return i + j;
}
```

called using following syntax –

```
result = add();
```

Type 4 : Not Accepting Parameter and Not Returning Value

- Above type of method is written like this –

```
void add()
{
  int i, int j;
  i = 10;
  j = 20;
  printf("Result : %d",i+j);;
}
```

and called like this –

```
add();
```

Rules of Writing Function in C Programming

- Rule 1. C program is a collection of one or more functions

```
main()
{
pune();
mumbai();
maharashtra();
}
```

- Rule 2. A function gets called when the function name is followed by a semicolon.

```
main( )
 {
 display( ) ;
 }
```

- Rule 3 : A function is defined when function name is followed by a pair of braces in which one or more statements may be present.

Rules of Writing Function in C Programming

```
display( )
    {
    statement 1 ;
    statement 2 ;
    statement 3 ;
    }
```

- Rule 4. Any function can be called from any other function.(main also)

```
main( )
{
 message( ) ;
}
 message( )
   {
   printf ( "\nWe are going to call main" ) ;
   main( ) ;
   }
```

Rules of Writing Function in C Programming

- Rule 5. A function can be called any number of times.

```
main()
{
message( );
message( );
}

message()
{
printf("\nLearning C is very Easy");
}
```

- Rule 6. The order in which the functions are defined in a program and the order in which they get called need not necessarily be same

Rules of Writing Function in C Programming

```
main( )
    {
    message1( ) ;
    message2( ) ;
    }
    message2( )
    {
    printf ( "\n I am First Defined But Called Later " ) ;
    }
    message1( )
    {
    printf ( "\n I am Defined Later But Called First ") ;
    }
```

- Rule 7. A function can call itself (Process of Recursion)

Rules of Writing Function in C Programming

```
int fact(int n)
{
 if(n==0)
    return(1);
 return(n*fact(n-1));
}
```

- Rule 8. A function can be called from other function, but a function cannot be defined in another function.

Function Parameters/Arguments

- When a function is called from some other function, the corresponding arguments in the function call are called *actual arguments* or *actual parameters*.
- The arguments in the function definition are called *formal arguments or formal parameters.*
- In a normal function call, there will be one actual argument for each formal argument.
- The actual arguments may be expressed as constants, single variables, or more complex expressions.
- However, each actual argument must be of the same data type as its corresponding formal argument.

Function Parameters/Arguments

- **Parameter Written In Function Definition is Called "Formal Parameter".**
- **Parameter Written In Function Call is Called "Actual Parameter".**

```
void main()
{
int num1;
display(num1);
}
void display(int para1)
{
----------
----------
}
```

Para1 is "**Formal Parameter**"

num1 is "**Actual Parameter**"

Function Parameters/Arguments

- Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.
- **Points to note:**

– The formal and actual arguments must match in their data types.

– The notion of positional parameters is important

– The identifiers used as formal arguments are “local”.

  - Not recognized outside the function.
  - Names of formal and actual arguments may differ.

Parameter Passing in C

Myth

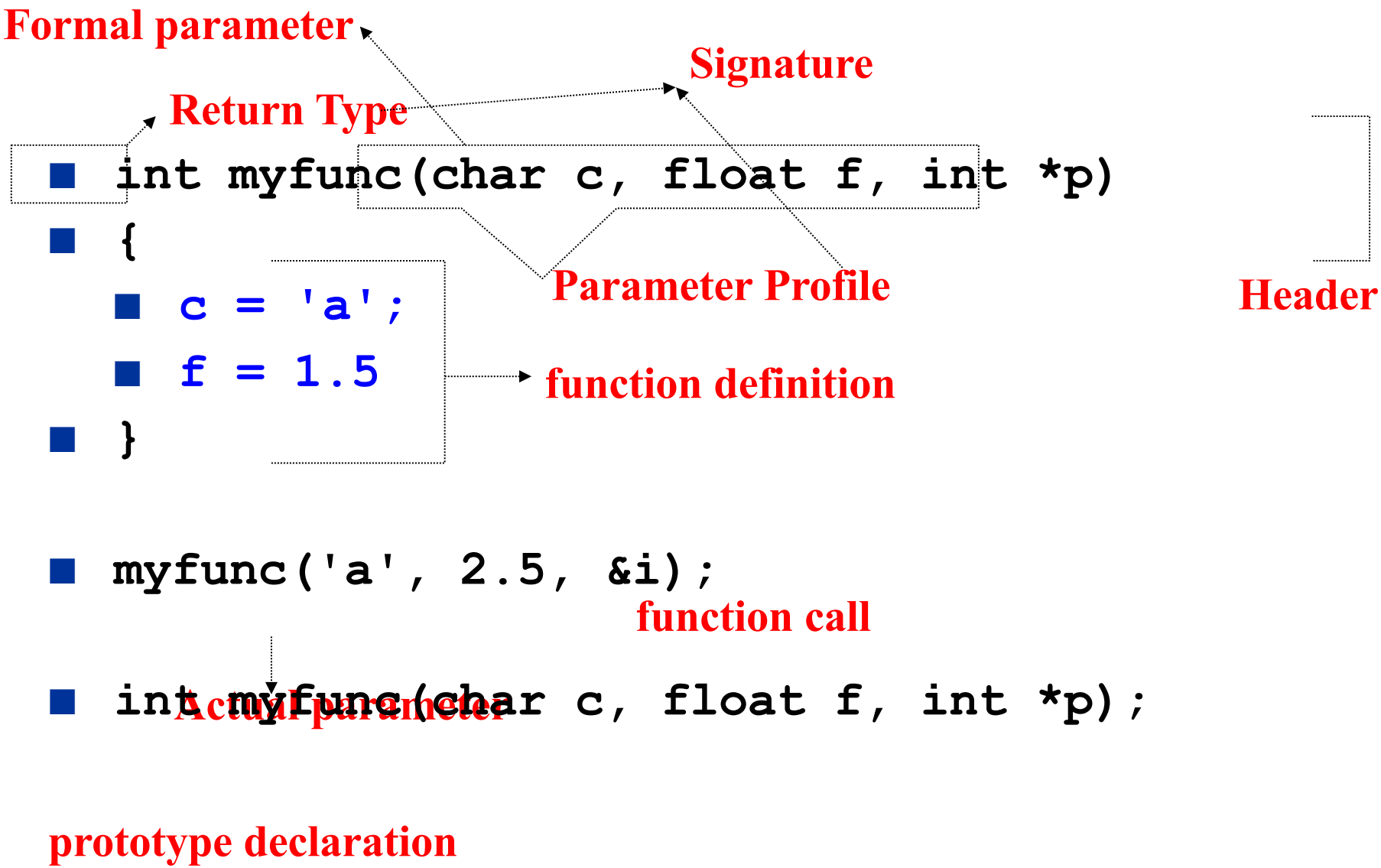
- C supports “Pass by Value” and “Pass by Reference” mechanisms of parameter passing
- Passing variables is *pass by value* and passing address is *pass by reference*

Truth

C supports only “Pass by value” mechanism of parameter passing

Passing variables or addresses are both *pass by value* in C

Some definitions

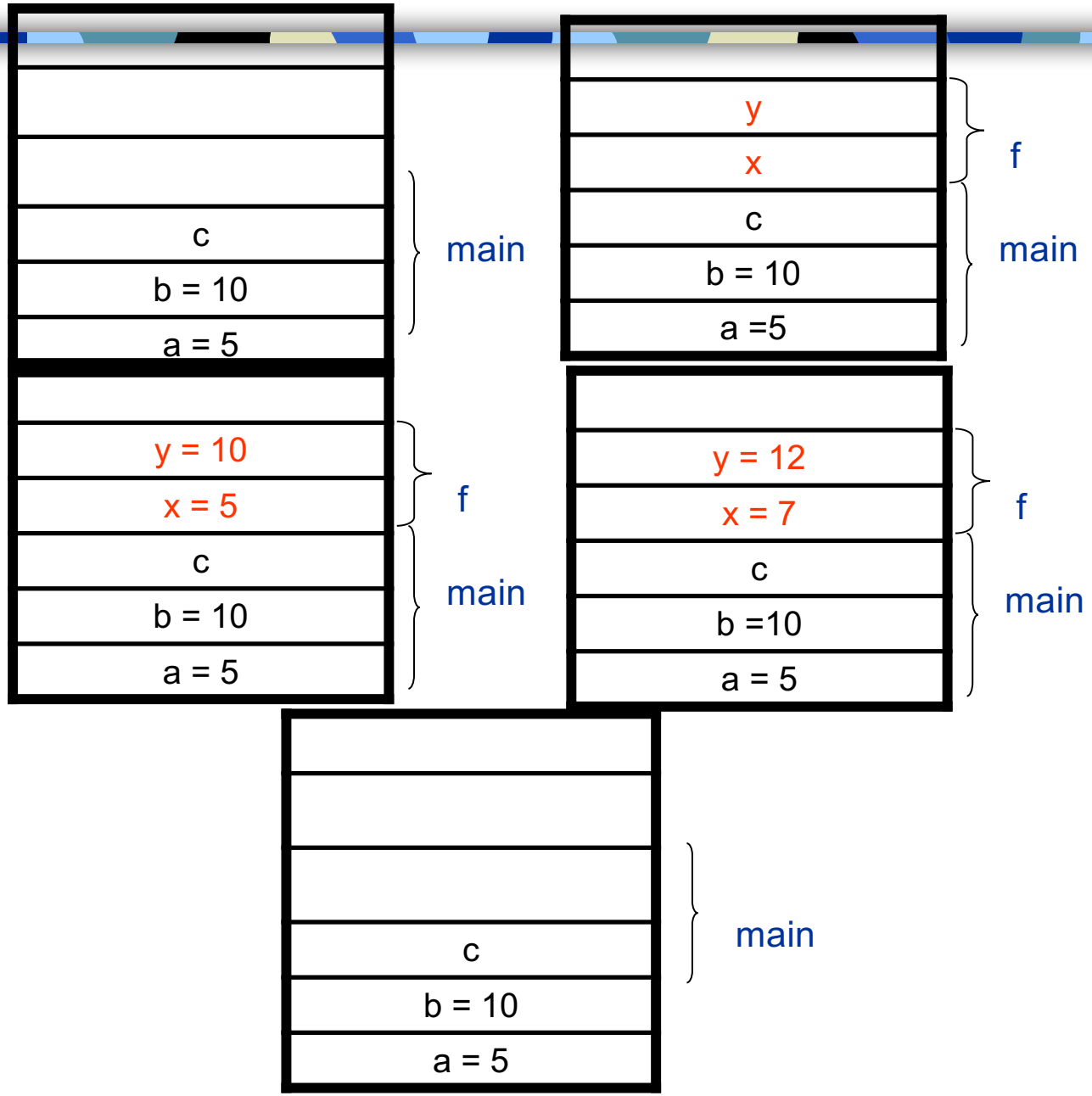


The system "stack"

- Your hardware, operating system and compiler together introduce a "stack" into your program.
  - The stack can be imagined to be an array in which we insert elements at one end and remove from the same end in reverse order
- The programmers need NOT know about this stack
- The function arguments and local variables are stored on this stack

Pass by value

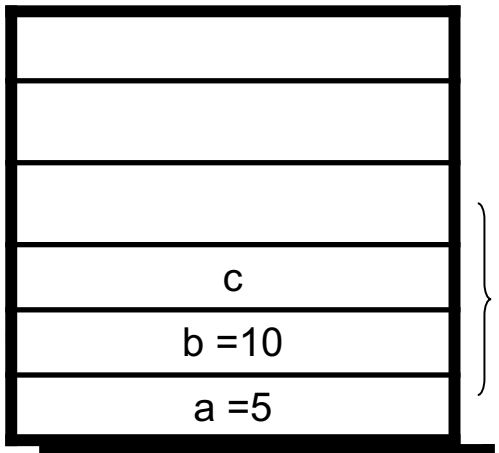
```
int f(int x, int y)
{
    x= 7;
    y = 12;
}
int main() {
    int a, b, c;
    a = 5; b = 10;
    f(a, b);
}
```

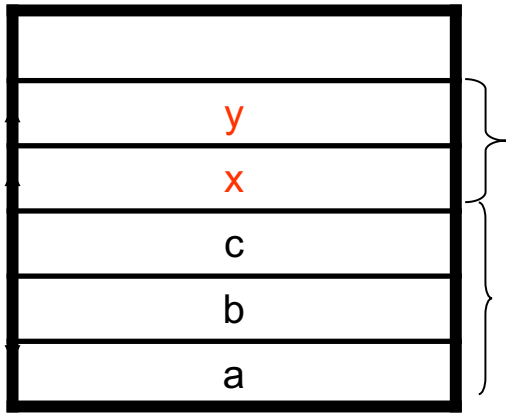


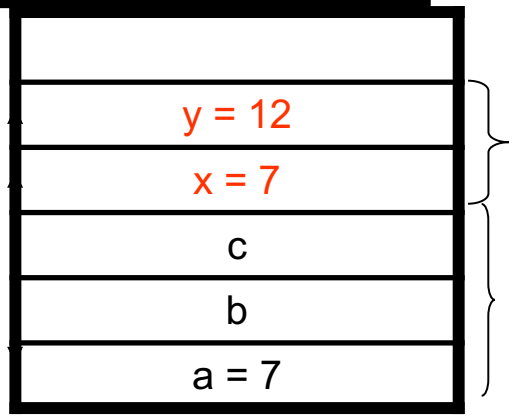
On a function call, the value of actual parameters are copied into formal parameters.

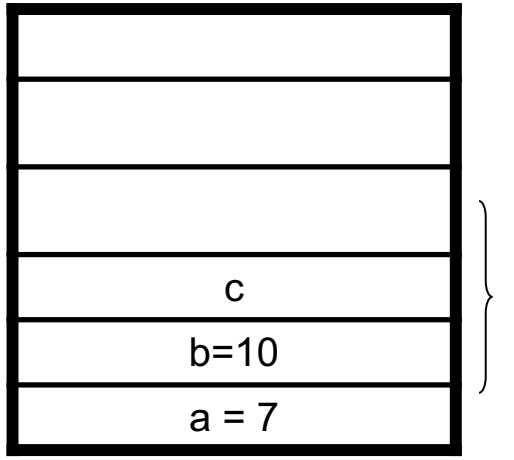
Pass by reference: Example

```
f int ref x, int y
begin
    x = 7; y = 12
    return;
end
main begin
    int a, b, c;
    a = 5; b = 10;
    f (a, b);
end
```









The variable x (or y) acts as if it was an alias for the variable a (or b)

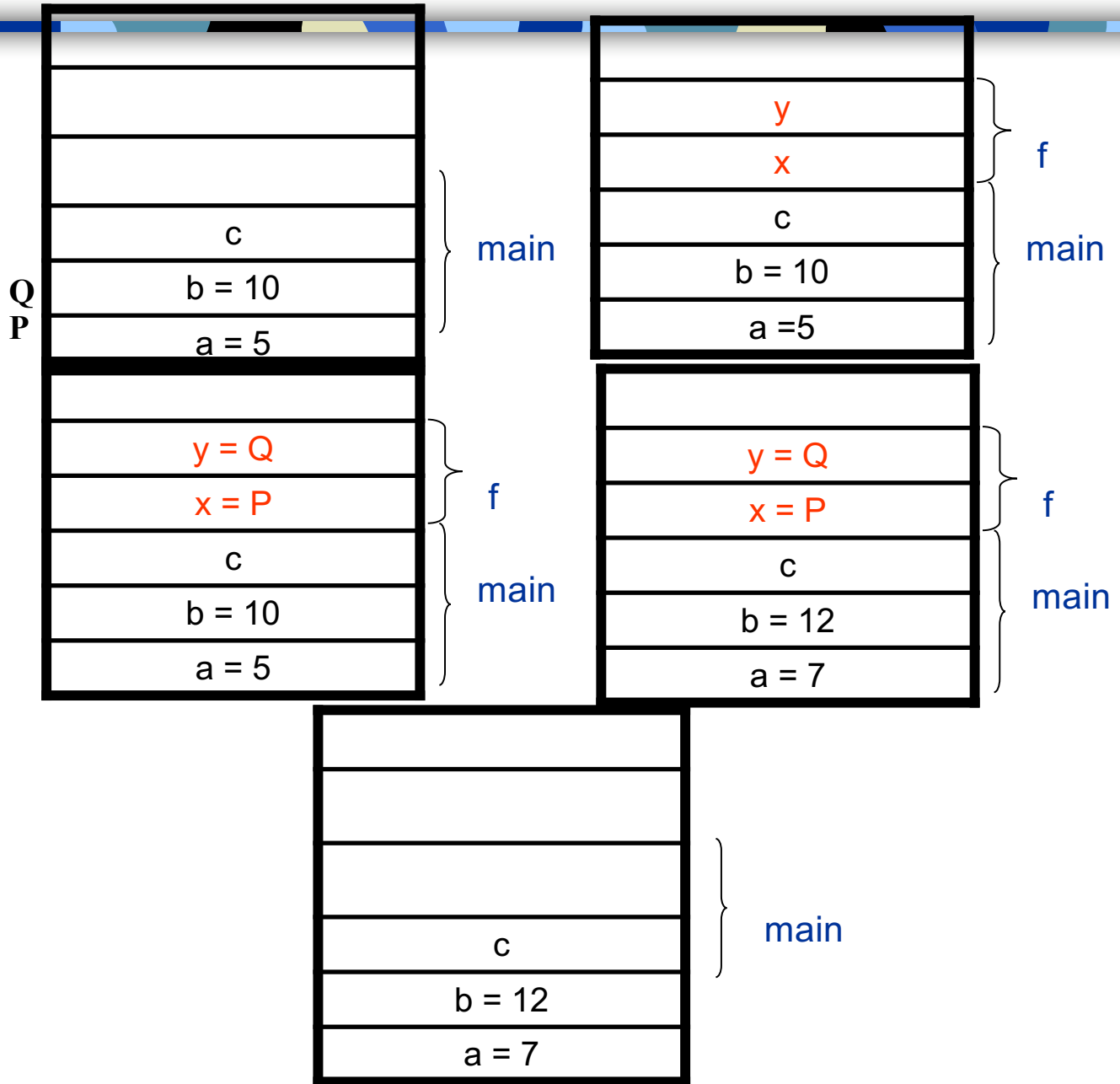
Pass by Reference: C++ example

- In Languages like C++ pass by reference is possible like this

```
void swap(int &a, int &b ) {
  temp = a; /* No * here */
}
int main() {
    int m = 20, n = 30;
    swap(m, n);/* No & here */
}
```

Passing address in C

```
int f(int *x, int
*y)
{
    *x= 7;
    *y = 12;
}
int main() {
    int a, b, c;
    a = 5; b = 10;
    f(&a, &b);
}
```



More about the earlier example

- & fetches the address
  - It's an “operator”
  - Compiler generates machine code to “fetch address of”
- Pass by value !
  - Actual argument passed is of type “address of integer”
    - This is done by programmer by writing &
  - Formal argument is of type “address of integer” that is integer pointer
    - This is done by programmer by specifying “int *”
  - On a function call “copy of actual argument is made into formal argument”
    - That's pass by value

Suppose ...

```
int f(int *x, int
*y)
{
    *x= 7;
    *y = 12;
}
int main() {
    int a, b, c;
    a = 5; b = 10;
    f(a, b);
}
```

- If you don't write '&', the code still gets compiled !
  - Actual argument will be integer, formal argument will be “int *”, and compiler will only issue a warning.

Address is NOT reference. References (in languages like C++ or Ada) are managed by languages (their compilers), Addresses (in languages like C) are managed by programmers.

Class Exercise

- Write a function which receives a **float and an integer value from main( ), find the product of these two values and return the product** which is printed through **main( ).**

- Write a function that receives marks of a student in 3 subjects and returns the average and percentage of these marks. Call this function from **main( ) and print the results in main( ).**

2. Solution

```
#include<stdio.h>
void avg_per(float*,float*);
void main()
{
  float avg, per;
  printf("Enter marks of 3 subject by any student: ");
  avg_per(&avg,&per);
  printf("Average = %.2f\nPercentage = %.2f%",avg,per);

}
void avg_per(float*avg,float*per)
{
  float m1,m2,m3,sum;
  scanf("%f%f%f",&m1,&m2,&m3);
  sum=m1+m2+m3;
  *avg=sum/3;
  *per=(sum*100)/150;
}
```

Call by Value- Class Exercise

Write a program to find out the maximum value between two integer numbers using function.

```
#include <stdio.h>
int max(int num1, int num2);       //function declaration
int main ()
{
   /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;
   ret = max(a, b);              //calling a function to get max value
   printf( "Max value is : %d\n", ret );
   return 0;
}
```

Call by Value- Class Exercise

```
/* 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 Example : Max amoung 3 Numbers

```
void max(int,int,int);
void main()
{
        int a,b,c;
        printf("Enter Three Numers");
        scanf("%d%d%d",&a,&b,&c);
        max(a,b,c);
}
void max(int x, int y, int z)
{
        int max = x;
        if(y>max)
            max=y;
        if(z>max)
             max=z;
        printf("Largest Number is %d", max);
}
```

What is the evaluation order of function parameters in C?

- Function parameters are not evaluated in a defined order.
- According to the draft C99 standard in C the order of evaluation of function arguments and the order in which side effects take place are both unspecified. *(from C99 §6.5.2.2p10:)*
- Compiler of C as it is traditionally build to maximize the speed and optimization can evaluate the function arguments in any way.
- Argument evaluation and argument passing are related but different problems.
- Consider the following function call:

  fun (a, b, c, d) ;

- In this call it doesn't matter whether the arguments are evaluated from left to right or from right to left.

What is the evaluation order of function parameters in C?

- However, in some function call the order of evaluation of arguments becomes an important consideration.
- For example:

```
int a = 1 ;
printf ( "%d %d %d", a, ++a, a++ ) ;
```

- It appears that this **printf( )** would output 1 2 3.

- This however is not the case.
- **It is compiler dependent in C. It is never safe to depend on the order of evaluation of side effects.**
- **For example, a function call like above may very well behave differently from one compiler to another**

C - Scope Rules

- A scope is a region of the program, and the scope of variables refers to the area of the program where the variables can be accessed after its declaration.

- A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable cannot be accessed.

- There are three places where variables can be declared in C programming language −

- Inside a function or a block which is called **local** variables.
- Outside of all functions which is called **global** variables.
- In the definition of function which are called **formal** parameters.

Local Variables

- **Variables that are declared inside a function or block of code are called local variables.**

- They can be used only by statements that are inside that function or block of code.
- These variables only exist inside the specific function that creates them.
- They are unknown to other functions and to the main program.
- Local variables cease to exist once the function that created them is completed.
- **They are recreated each time a function is executed or called.**
- The following example shows how local variables are used. Here all the variables a, b, and c are local to main() function.

Local Variables

```
#include <stdio.h>
int main ()
{
  /* local variable declaration */
  int a, b;
  int c;
  /* actual initialization */
  a = 10;
  b = 20;
  c = a + b;
  printf ("value of a = %d  b = %d and c = %d\n", a, b, c);
  return 0;
}
```

Global Variables

- Global variables are defined outside a function, usually on top of the program.
- Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.
- A global variable can be accessed by any function.
- That is, a global variable is available for use throughout your entire program after its declaration.
- **They do not get recreated if the function is recalled.**

Global Variables

```
#include <stdio.h>
int g;                    //global variable declaration
int main ()
{
  int a, b;               // local variable declaration
  /* actual initialization */
  a = 10;
  b = 20;
  g = a + b;
  printf ("value of a = %d b = %d and g = %d\n", a, b, g);
  return 0;
}
```

Global Variables

A program can have same name for local and global variables but it is considered bad programming practice and the value of local variable inside a function will take preference. Here is an example −

```
#include <stdio.h>
int g = 20;                    //global variable definition

int main ()
{
  int g = 10; // local variable declaration and initialization
  printf ("value of g = %d\n",  g);
  return 0;
}
```

Output- value of g = 10

Formal Parameters

- Formal parameters, are treated as local variables with-in a function and they take precedence over global variables.

```
#include <stdio.h>
int a = 20;                    //global variable definition
int sum(int, int);
int main ()
{
  /* local variable definition  in main function */
  int a = 10;
  int b = 20;
  int c = 0;
  printf ("value of a in main() = %d\n",  a);
  c = sum( a, b);
  printf ("value of c in main() = %d\n",  c);
  return 0;
}
```

Formal Parameters

```
/* function to add two integers */
int sum(int a, int b)          // a & b are formal parameters of sum()
{
   printf ("value of a in sum() = %d\n",  a);
   printf ("value of b in sum() = %d\n",  b);
   return a + b;
}
```

Output-

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

Initializing Local and Global Variables

- When a local variable is declared, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you declare them as follows −

Data Type	Initial Default Value
int	0
char	'\0'
float	0
double	0
Pointer	NULL

- ***Note- It is a good programming practice to initialize variables properly, otherwise your program may produce unexpected results, because uninitialized variables will take some garbage value already available at their memory location.***

C passing array elements to function

- Passing array, '**element by element**' to function.
- Individual element is passed to function using **Pass By Value** parameter passing scheme.
- Original Array elements remains same as Actual Element is never Passed to Function. Thus function body cannot modify **Original Value.**
- Suppose we have declared an array 'arr[5]' then its individual elements are arr[0],arr[1]…arr[4]. Thus we need 5 function calls to pass complete array to a function.

C passing single array element to function

```
#include <stdio.h>
void display(int age)
{
        printf("%d", age);
}
int main()
{
        int ageArray[ ] = { 2, 3, 4 };
        display(ageArray[2]); //Passing array element ageArray[2] only.
         return 0;
}
```

Class Exercise- C passing array elements to function and display

```
#include<stdio.h>
void fun(int num)
{
printf("\nElement : %d \n",num);
}
void main()
{
int arr[5],i;
printf("\nEnter the array elements : ");
for(i=0;i< 5;i++)
    scanf("%d",&arr[i]);
printf("\nPassing array element by element.....");
for(i=0;i< 5;i++)
     fun(arr[i]);
}
```

C passing array elements to function

```
#include<stdio.h>
void show(int x)
{
printf("%d ",x);
}
void main()
{
int arr[3] = {1,2,3};
int i;
for(i=0;i<3;i++)
  show(arr[i]);
}
```

C passing entire array to function

- If you want to pass a single-dimension array as an argument in a function, you would have to declare a formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received.
- Similarly, you can pass multi-dimensional arrays as formal parameters.
- **Way-1**
- Formal parameters as a pointer −

```
void myFunction(int *param)
 {
   ..........
  }
```

C passing entire array to function

- **Way-2**
- Formal parameters as a sized array −

```
void myFunction(int param[10])
{
   .
   .
   .
}
```

C passing entire array to function

- **Way-3**
- Formal parameters as an unsized array −

```
void myFunction(int param[ ])
{
   .
   .
   .
}
```

C passing entire array to function

- **Passing entire array to function :**
- Parameter Passing Scheme : **Pass by Reference**
- Pass **name of array** as function parameter.
- Name contains the base address i.e. (Address of 0th element)
- Function Body Can Modify **Original Value (actual argument).**
- Array values are updated in function.
- Values are reflected inside main (calling) function also.

Example: Display elements of array

```
void Display(int [ ]);
int main()
{
        int A[5]={0,0,0,0,0},i;
        for(i=0;i<5;i++)
                scanf("%d", &A[i]);
        Display(A);
        return 0;
}
void Display(int B[ ])
{
        int i;
        for(i=0;i<5;i++)
                printf("%d ",B[i]);
}
```

Class Exercise- Display elements of array after x index

```
void Display(int [ ],int);
int main()
{
        int A[5]={0},i,x;
        for(i=0;i<5;i++)
                scanf("%d", &A[i]);
        printf("Enter index of array to print elements after it : ");
        scanf("%d",&x);
        Display(A,x);
        return 0;
}
void Display(int B[ ],int y)
{
        int i;
        for(i=y;i<5;i++)
                printf("%d  ",B[i]);
}
```

Class Exercise- Multiply value of each element of array by 3 in a function and display modified array in main function

```
#include<stdio.h>
void modify(int b[3]);
void main()
{
int arr[3] = {1,2,3},i;
modify(arr);
for(i=0;i<3;i++)
  printf("%d",arr[i]);
}
void modify(int a[3])
{
int i;
for(i=0;i<3;i++)
  a[i] = a[i]*3;
}
```

Output :

3 6 9

C Passing Multi-Dimensional Array

- Multi-dimensional arrays are passed to a function in the same way as a single dimensional array. The only difference is that the number of parentheses while declaring a function will increase according to the dimension of the array.

- To pass two-dimensional array to a function as an argument, starting address of memory area reserved is passed as in one dimensional array

C Passing Multi-Dimensional Array

```
#include <stdio.h>
void display(int a[ ][2]); /*declaring a function which takes a two dimensional integer array as an argument*/
int main()
{
    int num[2][2],i,j;
    printf("Enter 2x2 numbers:\n");
    for(i=0 ;i<2 ;i++)
      {
        for(j=0 ;j<2 ;j++){
            scanf("%d",&num[i][j]);
        }
    }
    display(num); //the whole 2D array gets passed to the function display()
    return 0;
}
```

C Passing Multi-Dimensional Array

```
void display(int a[ ][2])  //the formal parameter takes an array as argument.
{
    int i,j;
    printf("The matrix is:\n");  //the matrix is displayed using two for loops
    for(i=0 ;i<2 ;i++)
    {
        for(j=0 ;j<2 ;j++)
            printf("%d ",a[i][j]);
        printf("\n");
    }
}
```

Adding Functions to the Library

- Most of the times we either use the functions present in the standard library or we define our own functions and use them.
- Can we not add our functions to the standard library? And would it make any sense in doing so?
- We can add user-defined functions to the library.

#define: Macro definition

- Preprocessor directive in the following form:

  #define string1 string2

- Replaces string1 by string2 wherever it occurs before compilation. For example,

```
#include <stdio.h>
#define PI 3.1415926
main()
{
  float r=4.0,area;
  area=PI*r*r;
}
```

```
#include <stdio.h>


main()
{
  float r=4.0,area;
  area=3.1415926*r*r;
}
```

#define with arguments

- **#define statement may be used with arguments.**
  - **Example:** #define sqr(x) x*x
  - **How will macro substitution be carried out?**

    r = sqr(a) + sqr(30); → r = a*a + 30*30;

    r = sqr(a+b); → r = a+b*a+b;

    WRONG?
  - **The macro definition should have been written as:**

    #define sqr(x) (x)*(x)

    r = (a+b)*(a+b);

Core Dump (Segmentation fault) in C

- Core Dump/Segmentation fault is a specific kind of error caused by accessing memory that “does not belong to you.”

- When you run your program and the system reports a "segmentation violation," it means your program has attempted to access an area of memory that it is not allowed to access.

- When a piece of code tries to do read and write operation in a read only location in memory or freed block of memory, it is known as core dump.

- It is an error indicating memory corruption.

Common causes of this problem:

- Creating a very large array.
- Declaring the array as int arr [n]; // size undefined
- Accessing out of array index bounds

Common causes of this problem:

- Improper format control string in printf or scanf statements
- Forgetting to use "&" on the arguments to scanf
- Accessing beyond the bounds of an array
- Failure to initialize a pointer before accessing it
- Incorrect use of the "&" (address of) and "*" (dereferencing) operators
- Modifying a string literal
- Accessing an address that is freed

Warning: the `gets' function is dangerous and should not be used.

- This means that if the string read from the user is longer than the memory allocated to hold that string, it would try to write past that memory which would most likely lead to your program crashing because of segmentation fault (since you are using storage not allocated to the process).
- It doesn't have an array length/count parameter that it takes.

- If you have code like this:

```
char s[10];
gets( s );
```

- and you type in more than 10 characters when the program is run, you will overflow the buffer(memory) because gets() will continue to store characters past the end of the buffer, causing undefined behavior. The gets() function has no means of preventing you typing the characters and so should be avoided.

Warning: the `gets' function is dangerous and should not be used.

- In order to use ’gets’ safely, you have to know exactly how many characters you will be reading, so that you can make your buffer large enough.

- Instead you should use fgets(), which allows you to limit the number of characters read, so that the buffer does not overflow.:

```
char s[10];
fgets( s, 10, stdin );
```

- The puts() function is perfectly safe, *provided* the string that you are outputting is null-terminated.