UNIT-4

Topics to be covered in UNIT-4

Functions

– User – defined functions

-Function definition, arguments, return value, prototype, arguments and parameters,

- inter-function communication.

-Standard functions

- – Math functions.

-Scope – local, global.

Parameter passing

– Call by value and call by reference.

Recursive functions

– Definition, examples, advantages and disadvantages.

Macros

– Definition, examples, comparison with functions.

Storage Classes

– auto, extern, static and Register

Objectives

- To design and implement programs with more than one function.
- To understand the purpose of the function declaration, call, and definition.
- To understand the four basic function designs.
- To understand how two functions communicate through parameters.
- To understand the differences between global and local scope.

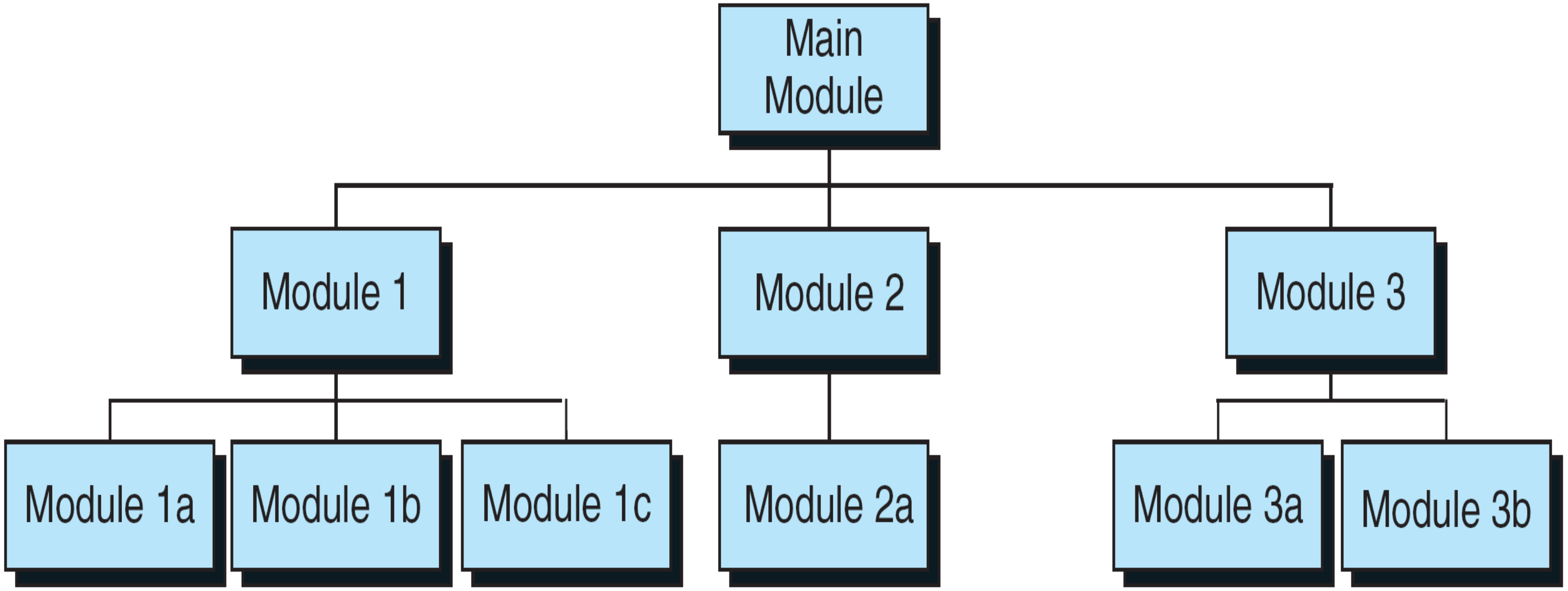
When the program grows into larger, may have many disadvantages.

1. Difficult to write a larger programs
2. Difficult to identify the logical errors and correct.
3. Difficult to read and understand.
4. Larger programs prone for more errors.

Designing Structured Programs

- The programs we have done so far have been very simple.
- They solved problems that could be understood without too much effort.
- The principles of **top–down** design and **structured** programming dictate that a program should be **divided** into a main module and its related modules.
- Each module is in turn divided into **sub-modules** until the resulting modules are intrinsic; that is, until they are implicitly **understood** without further division.

- Top-down design is usually done using a visual representation of the modules known as a structured chart.



Structure Chart

FUNCTIONS IN C

- If any module has sub modules that is called as calling module and sub modules are called modules.

- one module is communicate with other module through calling module

- A c program consists of one or more functions, one and only one of which must be named main

- the function which is calling other function is called calling function

- A function whom it called is called function

- A called function receives control from the a calling function, when it completes its task, it returns control to the calling function.

- the main function called by the operating system, in turn main function calls the other function to do some task.

.

FUNCTIONS IN C

- A **function** is a self-contained block of code that carries out some specific and well-defined task.

- In general, the **purpose** of a function is to **receive** zero or more pieces of data, operate on them, and **return** at most one piece of data.

- C functions are classified into two categories
    1. **Library** Functions
    2. **User Defined** Functions

FUNCTIONS IN C (contd…)

Library Functions:

- These are the built in functions available in standard library of C.

- The standard C library is collection of various types of functions which perform some standard and predefined tasks.

Example:

- **abs (a)** function gives the absolute value of a, available in <math.h>

- **pow (x, y)** function computes x power y. available in <math.h>

- **printf ()** and **scanf ()** performs I/O functions and etc..,

FUNCTIONS IN C (contd…)

User Defined Functions:

- These functions are written by the programmer to perform some specific tasks.
- Example: main (), sum (), fact () , show(), display() and etc.,
- The main distinction between these two categories is that library functions are not required to be written by us whereas a user defined function has to be developed by the user at the time of writing a program.

Note:

- In C, a program is made of one or more functions, one and only one of which must be called **main**.
- The **execution** of the program always **starts** with **main**, but it can call other functions to do some part of the job.

Advantages of User-defined Functions

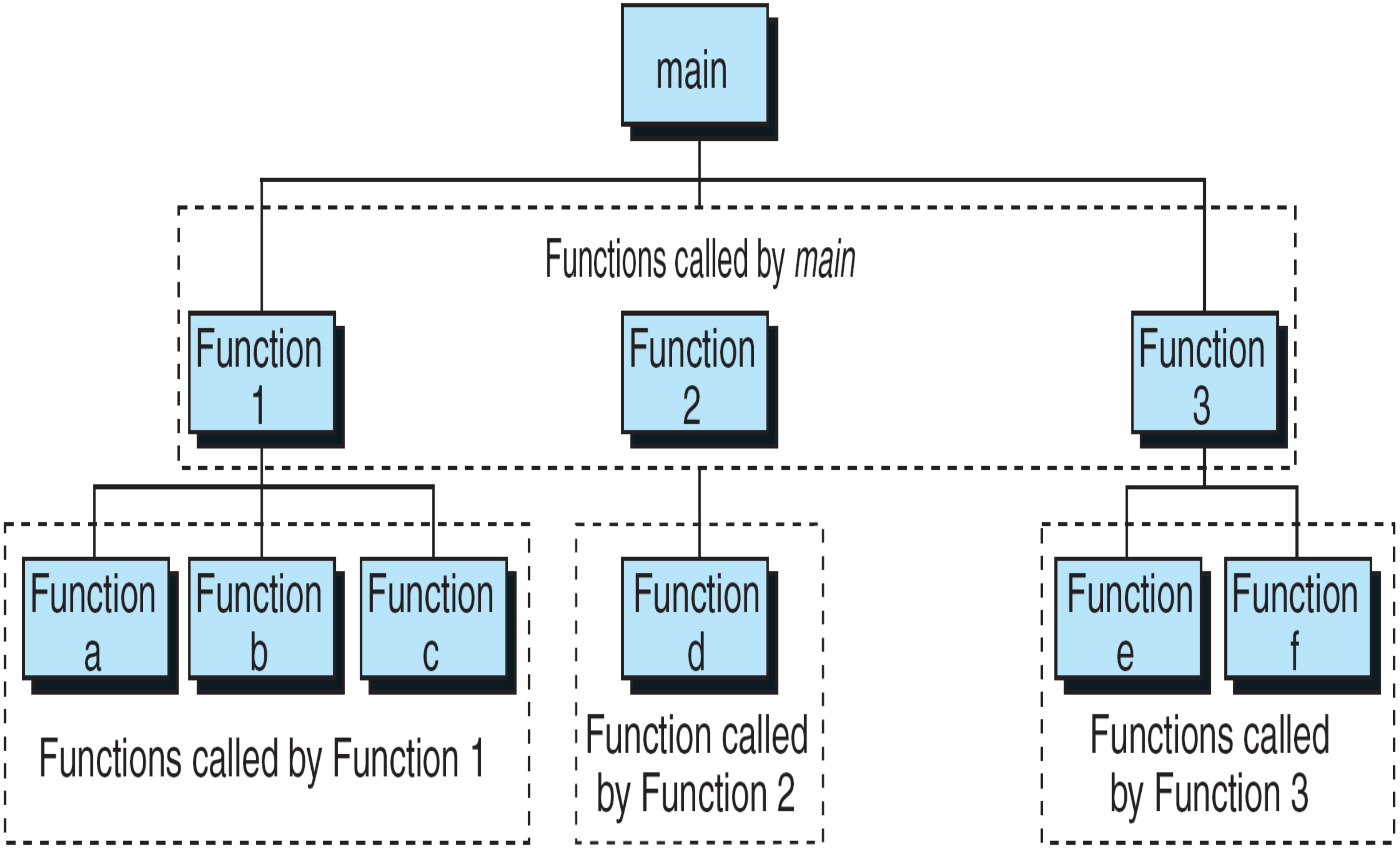
Modular Programming: It facilitates top down modular programming.

Reduction of Source Code: The length of the source program can be reduced by using functions at appropriate places.

Easier Debugging: It is easy to locate and isolate a faulty function for further investigation.

Code Reusability: A program can be used to avoid rewriting the same sequence of code at two or more locations in a program.

Function Sharing: Programming teams does a large percentage of programming. If the program is divided into subprograms, each subprogram can be written by one or two team members of the team rather than having the whole team to work on the complex program.


main
Functions called by *main*
Function 1
Function 2
Function 3
Function a
Function b
Function c
Function d
Function e
Function f
Functions called by Function 1
Function called by Function 2
Functions called by Function 3

Structure Chart for a C Program

User-defined Functions

- Like every other object in C, functions **must be** both **declared** and **defined**.
- The function **declaration or prototype** gives the **whole picture of the function** that needs to be defined later.
- The function **definition** contains the code for a function.
- A function name is used **three times** in a C program: for **declaration**, in a **call**, and for **definition**.

The general form of a C user-defined function:

```
return_type function_name (argument declaration)
{
    //local declarations
      ......
     //statements
       ......
    return (expression);
 }
```

The general form of a C user-defined function (contd…)

return-type:

- Specifies the type of value that a function returns using the return statement.
- It can be any valid data type.
- If no data type is specified the function is assumed to return an integer result.

function-name:

- Must follow same rules of variable names in C.
- No two functions have the same name in a C program.

argument declaration:

- It is a comma-separated list of variables that receive the values of the argument when function is called.
- If there is no argument declaration the bracket consists of keyword void.

- A function name is used **three times** in a C program: for **declaration**, in a **call**, and for **definition**.

Function Declaration (or) Prototype:

- The ANSI C standard expands the concept of forward function declaration.
- This expanded declaration is called a function prototype.

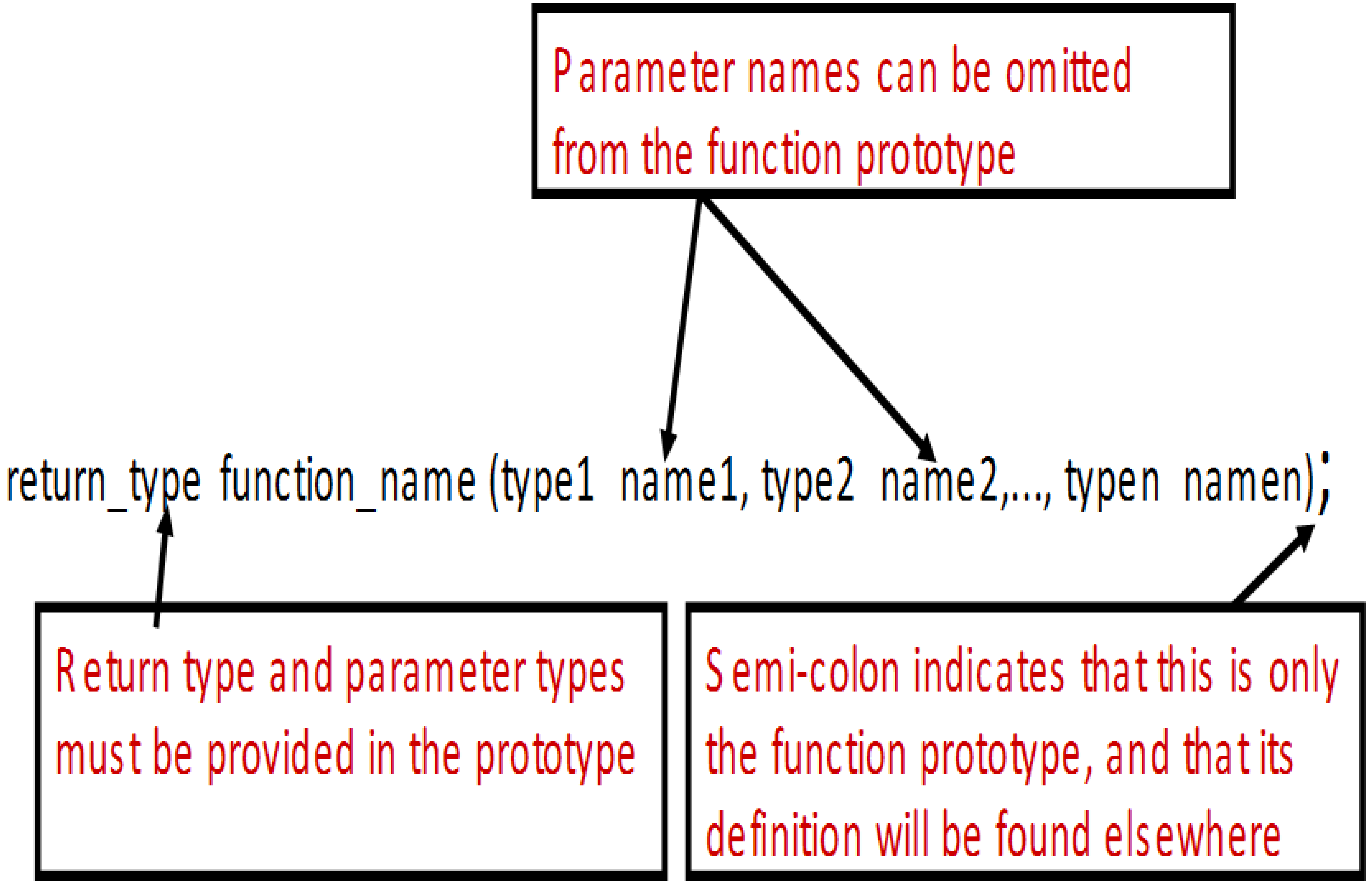
A function prototype performs two special tasks:

- First it identifies the **return type** of the function so that the compile can generate the correct code for the return data.

- Second, it specifies the **type** and number of arguments used by the function.

Note: The prototype normally goes near the top of the program and must appear before any call is made to the function.

The general form of the function prototype or declaration is:



Parameter names can be omitted from the function prototype

```
return_type function_name (type1 name1, type2 name2,..., typen namen);
```

Return type and parameter types must be provided in the prototype

Semi-colon indicates that this is only the function prototype, and that its definition will be found elsewhere

User-defined Functions (Contd…)

The Function Call:

- A function call is a postfix expression.
- The operand in a function is the function name.
- The operator is the parameter lists (…), which contain the actual parameters.

Example:

```
void main ()
{
        sum (a, b);
}
```

- When the compiler encounters the function call ,the control is transferred to the function **sum()**.

- The function is executed line by line and produces the output for the sum of two numbers and then control is transferred back to the main function.

Function Definition: The function definition contains the code for a function.

It is made up of two parts:

- The **function header** and the **function body**, the compound statement is must.

Function header consists of three parts:

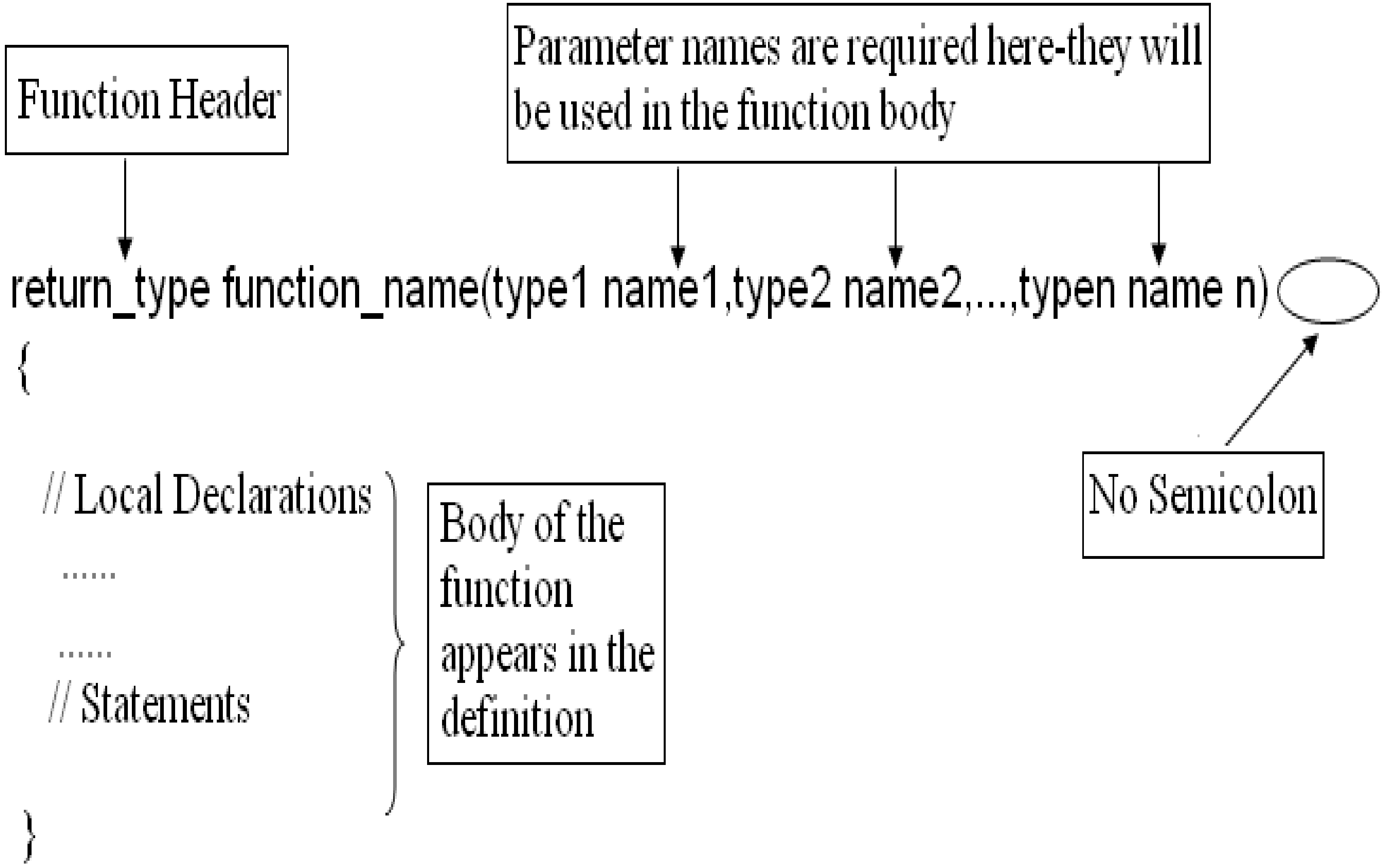
- the **return type**,
- the **function name**, and
- the **formal parameter list**.

- **Function body** contains local declarations and function statement.

- Variables can be declared inside function body.

- Function **can not be** defined inside another function.

Function Definition


Function Header
Parameter names are required here-they will be used in the function body
return_type function_name(type1 name1,type2 name2,...,typen name n)
{
// Local Declarations
......
......
// Statements
}
Body of the function appears in the definition
No Semicolon

➢A function gets called when the function name is followed by a semicolon.

```
void main()
    {
     fun1();
   }
```

➢When a function is followed by a pair of braces in which one or more statements may be included is called function definition.

```
void fun1()
{
  printf("we are in fun1");
}
```

- ➢Any function can be called from any other function. Even main can be called from other functions**.**

- ➢A function can be called any number of times.

- ➢The order in which functions are defined in program and the order in which they get called need not be same.

- ➢A function can be called from another function but a function can not be defined in another function.

- ➢ function can also called itself is called rec**ursion.**

Each function is called in the sequence we have specified in the main function.

```
void fun1(); void fun2(); void fun3();
void main()
   {
    printf("\n I am calling other functions");
   func1();
  func2();
  func3();
  printf("all functions are called\n");
   }//main
  func1()
  {
  printf("in func1\n");
  }//func1
  func2()
  {
  printf("in func2\n");
  }//func2
  func3()
 {
  printf("in func1\n");
  }//func3
```

Output:

```
I am calling other function
in func1
in func2
in func3
all functions are called.
```

After each function task is over the control passed to the calling function(main()).

7 One function can also call the other function which has already been called

```
void fun1(); void fun2(); void fun3();
void main()
 {
            printf(“I am in main\n”);
            func1();
            printf(“I back in main\n”);
 }//main
void func1()
 {
            printf(“in func1\n”);
             func2();
            printf(“I back in func1\n”);
}//func1
 void func2()
 {
            printf(“In func2\n”);
             func3();
            printf(“I back in func2\n”);    }//func2
 void func3()
 {
 printf(“in fac3\n”);              }//func3
```

- **a function cannot be defined inside the other function.**

```
 main()
{
  func1();
  func2();
 }//main
 func1()
 {
   printf(“\nthis is func1”);
   func2()
   {
      printf(“\nthis is func2”);
   }//func2()
}//func1()
```

✗

- called function may or may not send information (data) back to the calling function.
- If called function return some value to the calling function at the end of the function definition (before }) use return statement.

```
void func1();
 void main()
   {
     func1();
     printf("\nhai");
  }//main
  void func1()
 {
    printf("hello");
    return;
 }//func1()
```

return;   this statement says that does not return any value. This may or may not used in void functions.

return(expression);- it return some value to the calling function.

Note- a function return only single value.

Categories of functions

- Functions may belong to one of the following categories.

1. functions with no arguments and no return value.
2. functions with no arguments and return value.
3. functions with arguments and no return value.
4. functions with arguments and return value.

1. functions with no arguments and no return value.

- If the function has no arguments, it does not receive any arguments/ parameters from the calling function.
- It does not return any value to the calling function.
- In this case, return type must be specified as void and put empty parenthesis.

```
#include<stdio.h>

void msg();
int main()
{
msg();
return 0;
}//main
```

```
void msg()
{
   printf("learning c is easy");
}
```

```
#include <stdio.h>
void  add();   // function declaration or prototype
int main()
{
add();          // function call
return 0;
}
void  add()    // function definition
{
int a,b,c;
printf("enter a, b values");
scanf("%d%d", &a,&b);
c=a+b;
printf(" a+b =%d",c);
}
```

2.functions with no arguments and return value.

- In which called function does not receive in arguments from calling function. However, end of its task it return some value to the calling function.
- In this case, return value must be specified and parenthesis is empty

```
#include<stdio.h>

char msg();
int main()
{
char  x;
x=msg();
printf("%c",x);
return 0;
}//main
```

```
char msg()
{
   char ch;
printf("enter character");
scanf("%c",&ch);
 return ch;
}
```

```
#include <stdio.h>
int add();   // function declaration
int main()
{
int x;
x=add();          // function call
printf(“addition=%d”,x);
return 0;
}
int add()    // function definition
{
int a,b,c;
printf(“enter a, b values”);
scanf(“%d%d”, &a,&b);
c=a+b;
return c;
}
```

- Generally, there are two ways that a function terminates execution and returns to the caller.
- When the last statement in the function has executed and conceptually the function's ending '}' is encountered.
- Whenever it faces return statement.

The return statement: It is the mechanism for returning a value from the called function to its caller.

The general form of the return statement is:

return expression;

Functions with **no arguments** and **return values** (Contd…)

- The calling function is free to ignore the returned value.

- Expression after the return is not necessary.

The return statement has two important uses:

1. It causes an immediate exit of the control from the function. That is ,it causes program execution to return to the called function.
2. It returns the value present in the expression.

Example:

```
return(x + y);
return (6 * 8);
return (3);
return;
```

3. functions with arguments and no return value.

- Called function receives arguments from the calling function and called function doesn’t return any value to calling function.
- In function prototype, we have to specifies no of arguments in the parenthesis and specify return type as void.

```
#include<stdio.h>
void msg(float);
int main()
{
float  avg;
printf(“enter avg of student”);
scanf(“%f”,&avg);
msg(avg);
return 0;
}//main
```

```
void msg( float per)
{
  printf(“percentage of student=%f”,per);
}
```

The arguments in the calling functions are called actual arguments.(ex avg)
The arguments in the called function are called formal arguments.(ex per)
Actual and formal arguments need not be same

```
#include <stdio.h>
void  add(int ,int );   // function declaration or prototype
int main()
{
int a,b;
printf("enter a, b values");
scanf("%d%d", &a,&b);
 add(a,b);        // function call
return 0;
}
void  add(int x, int y)    // function definition
{
int  z;
z=x+y;
printf(" addition=%d",z);
}
```

3. Functions with **arguments** and **no return values**:

- In this category there is data transfer from the calling function to the called function using parameters.
- But, there is no data transfer from called function to the calling function.

Local Variables:

- Variables that are defined within a function are called local variables.
- A local variable comes into existence when the function is entered and is destroyed upon exit.

Function Arguments: The arguments that are supplied to function are in two categories

1. Actual Arguments/Parameters

2. Formal Arguments/Parameters

Functions with **arguments** and **no return values** (Contd…)

Actual Arguments/Parameters:

- Actual parameters are the expressions in the calling functions.
- These are the parameters present in the calling statement (function call).

Formal Arguments/Parameters:

- Formal parameters are the variables that are declared in the header of the function definition.

Note: Actual and Formal parameters must match exactly in type, order, and number. Their names however, do not need to match.

4. functions with arguments and with return value.

- Called function receives arguments from the calling function and called function return any value to calling function.
- In function prototype, we have to specifies no of arguments in the parenthesis and specify return type .

```
#include<stdio.h>
double msg(double);
int main()
{
double  sum;
printf(“enter avg of student”);
scanf(“%lf”,&sum);
printf(“%lf”,msg(sum));
return 0;
}//main
```

```
double msg( double avg)
{
  double per;
per=avg/6;
return per;

}
```

```
#include <stdio.h>
int  add(int x,int y);   // function declaration or prototype
int main()
{
int a,b,x;
printf("enter a, b values");
scanf("%d%d", &a,&b);
x=add(a,b);       // function call
printf(" addition=%d",x);
return 0;
}
int add(int x, int y)    // function definition
{
int  z;
z=x+y;
return z;
}
```

4. functions with arguments and return value.

```
#include<stdio.h>
int add(int,int);
int main()
{
int a,b;

printf("\nenter a and b");
scanf("%d %d",&a,&b);
printf(“%d",add(a,b));
 return 0;
}//main
int add(int a,int b)
{
int c;
c=a+b;
return c;
}//add()
```

```
#include<stdio.h>
int add(int x,int y);
int main()
{
int a,b;

printf("\nenter a and b");
scanf("%d %d",&a,&b);
c=add(a,b);
printf(“c=%d",c);
return 0;
}//main
int add(int a,int b)
{
return a+b;
}//add()
```

```
#include<stdio.h>
int add(int,int);
int main()
{
int a,b;

printf("\nenter a and b");
scanf("%d %d",&a,&b);
printf("%d",add(a,b));
return 0;
}//main
int add(int a,int b)
{
return a+b;
}//add()
```

Nesting of functions

C permits nesting of functions, main function can call function1, function1 can call function2, which in turn calls function3 ..,
There is no limit as how deeply functions can be nested.

Consider the following example:

In the below example ,when the main() function finds the function call sum(), then the control is transferred from main() to the function sum(), there sum() is calling the function read(), then the control is transferred to read() function, then the body of the function read() executes, then the control is transferred back to sum(). Again read() function is invoked.
Observe the chain of control transfers between the nested functions.

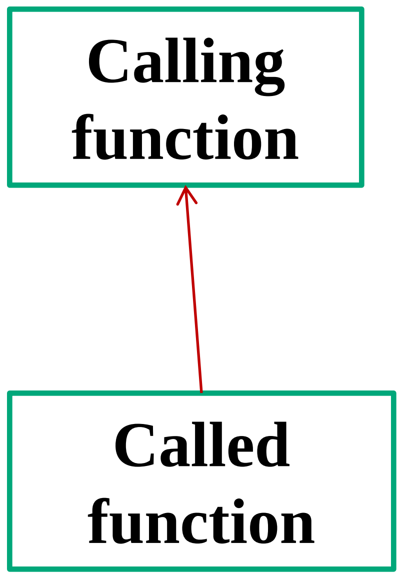
```
   #include<stdio.h>                    //Example program for nested-functions
int read ();
int sum (int, int);
int main ()
{
          printf ("%d", sum ());
   return 0;
}
int sum (int x, int y)
{
          x=read ();
          y=read ();
          return x + y;
}
int read ()
{
          int p;
          printf ("\n Enter any value: ");
          scanf ("%d", &p);
          return p;
                                                                          }
```

INTER-FUNCTION COMMUNICATION

The calling and called functions communicate with each other using three strategies.

- **Upward communication**
- **Downward communication**
- **Bidirectional communication**

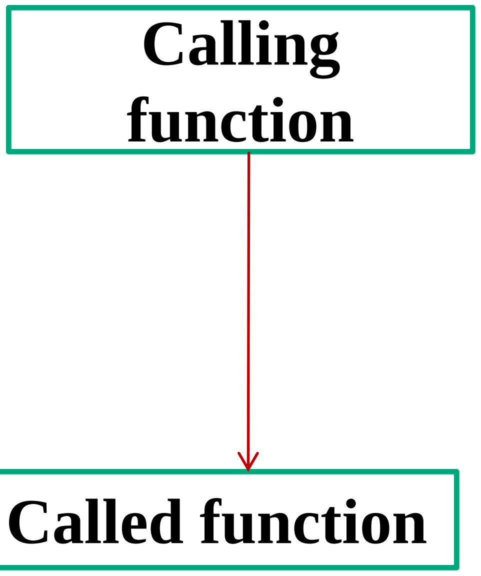
UPWARD COMMUNICATION



- In this strategy ,only called function can pass the data to the calling function, it will not receive anything from calling function. Upward communication in c can be implemented by using return statement.
- Both called and calling function must declare a data variable to save the address and receive the data.

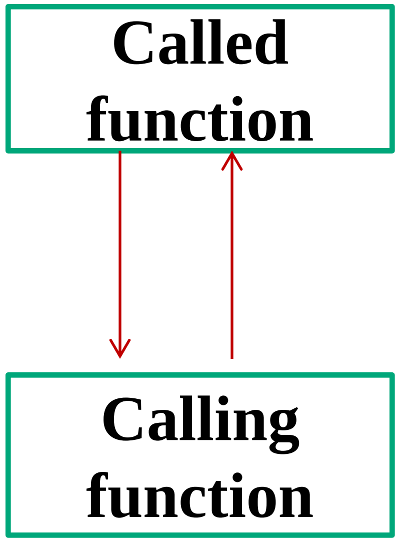
Downward Communication

- In this strategy, calling function is the active function i.e., the information can only be sent by the calling function to the called function, there will be no response from the called function(One way communication).
- Called function can make changes on the received information , but this will not effect on the original information of the calling function



Bidirectional Communication

- In this strategy, both called and calling function can send the data to each other.



Standard Functions

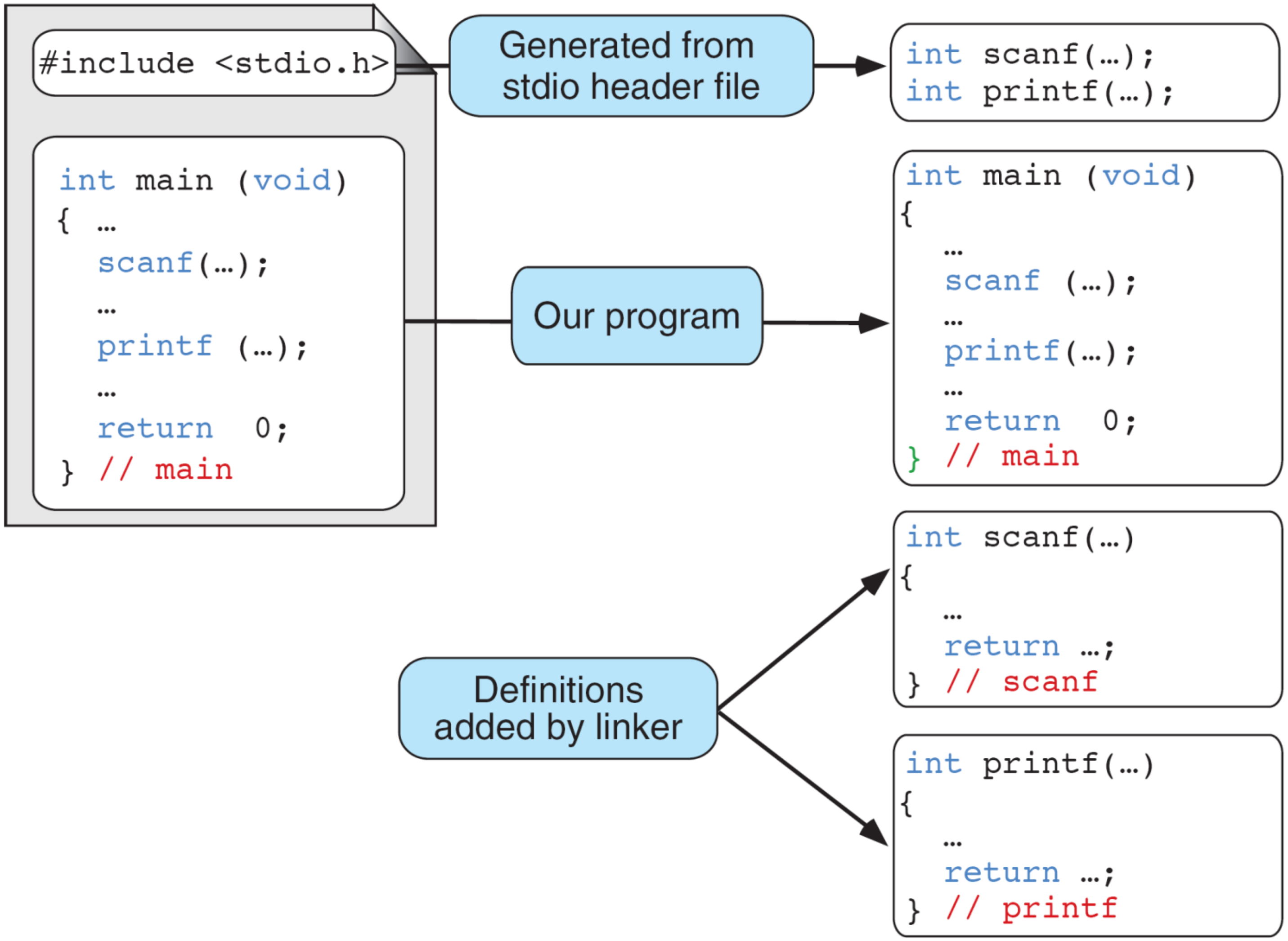
- C provides a rich collection of standard functions whose definitions have been written and are ready to be used in our programs.

- To use these functions, we must include their function declarations.

- We discuss the following:
  - **Math Functions**

Some of the **header files** includes the following functions are**:**

<**stdio.h**>	Standard I/O functions
<**stdlib.h**>	Utility functions such as string conversion routines, memory allocation routines, etc..,
<**string.h**>	String manipulation functions
<**math.h**>	Mathematical functions
<**ctype.h**>	Character testing and conversion functions



```
#include <stdio.h>

int main (void)
{ …
  scanf(…);
  …
  printf (…);
  …
  return  0;
} // main
```

```
int scanf(…);
int printf(…);
```

```
int main (void)
{
  …
  scanf (…);
  …
  printf(…);
  …
  return  0;
} // main
```

```
int scanf(…)
{
  …
  return …;
} // scanf
```

```
int printf(…)
{
  …
  return …;
} // printf
```

Library Functions and the Linker

Math Functions

The following are some of the math functions:

- Absolute
- Ceil
- Floor
- Truncate
- Round
- Power
- Square Root

All the above functions are available in **math.h** header file.

Absolute Functions:

- An absolute value is the positive rendering of the values regardless of its sign.
- The integer functions are **abs, labs, llabs**.

Syntax:

```
int abs(int number);
long labs(long number);
long long llabs(long long number);
```

- The float function is **fabs**:

```
float fabs(float number);
```

Example:	abs(3)	gives	3
	fabs(3.4)	gives	3.4

Ceil Function:

- A ceiling is the smallest integer value greater than or equal to a number.
- Format of **ceil** function:

  double ceil(double number);
  float ceilf(float number);
  long double ceill(long double number);

- **Example:**

ceil(-1.9)	gives	-1.0
ceil(1.1)	gives	2.0

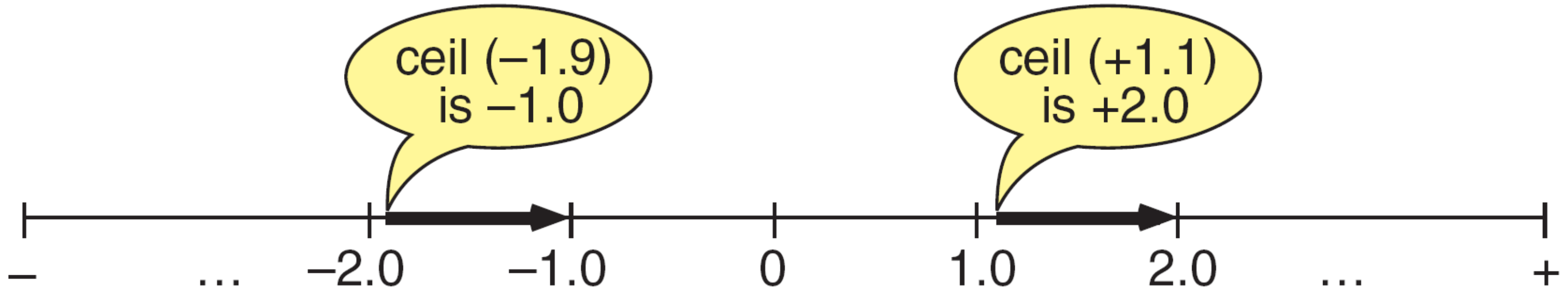
Floor Function:

- A floor is the largest integer value that is equal to less than a number.
- Format of **floor** function:

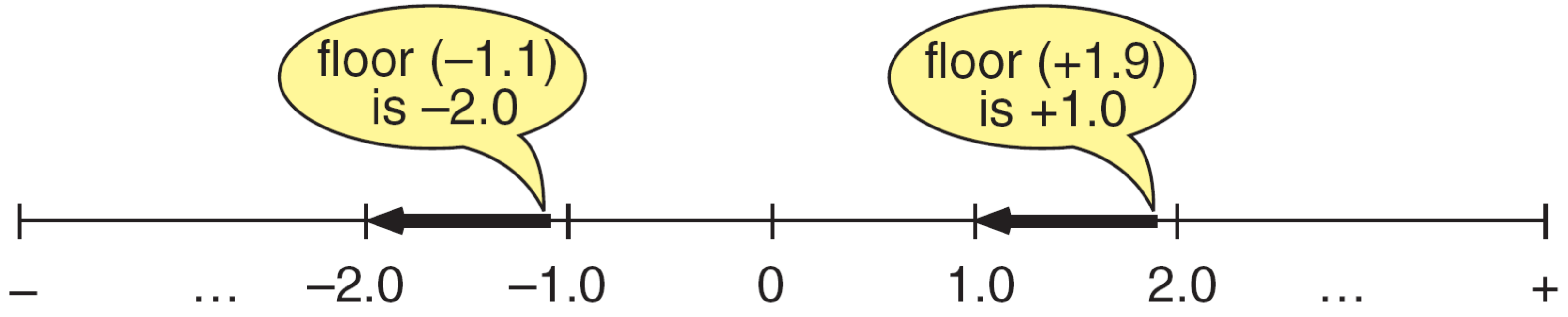
  double floor(double number);
  float floorf(float number);
  long double floorl(long double number);

- **Example:**

floor(-1.1)	gives	-2.0
floor(1.9)	gives	1.0



Ceiling Function



Floor Function

Truncate Function:

- The truncate function return the integral in the direction of 0.
- Format of **trunc** function:

  double trunc(double number);
  float truncf(float number);
  long double truncl(long double number);

- **Example:**

```
trunc(-1.1)      gives   -1.0
trunc(1.9)       gives    1.0
```

Round Function:

- The round function returns the nearest integral value.
- Format of **round** function:

  double round(double number);
  float roundf(float number);
  long double roundl(long double number);

- **Example:**

```
round(-1.1)      gives   -1.0
round(1.9)       gives    2.0
round(-1.5)      gives   -2.0
```

Power Function:

- The power function returns the value of the x raised to the power y that is $\mathbf{x}^{y}$.

- Format of **pow** function:

  double pow(double number1, double number2);

- **Example:**

  pow(2,5)   gives 32

Square Root Function:

- The square root function returns the non negative square root of a number.

- Format of **sqrt** function:

  double sqrt(double number);

- **Example:**

  sqrt(4.0)   gives 2.0

Parameter Passing Techniques

- There are two ways of passing parameters to a function

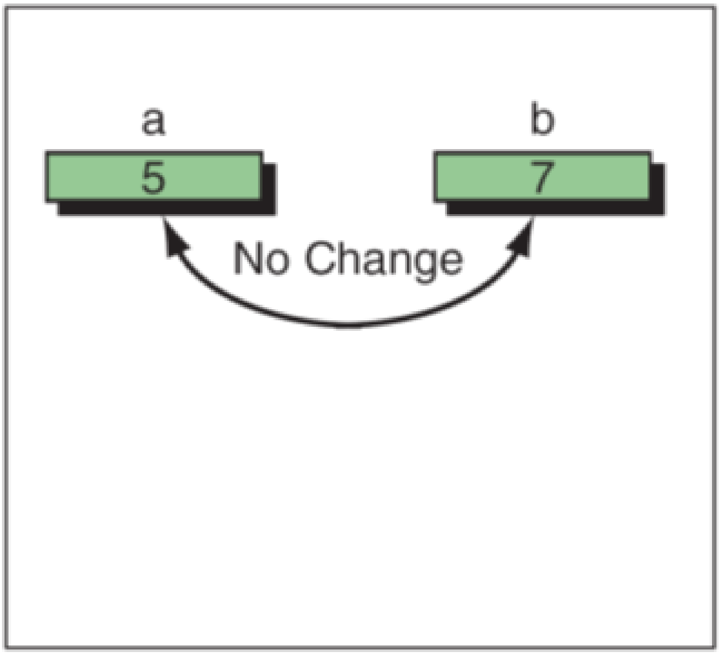
  - **Call-by-value**
  - **Call-by-reference**

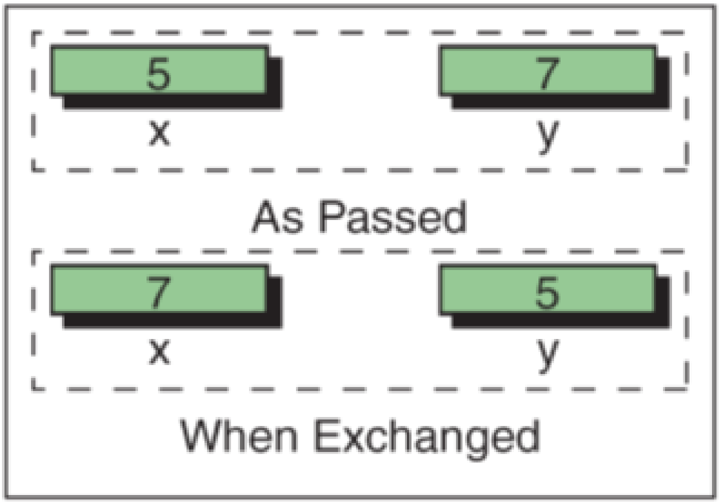
Call-by-value:

- When a function is called with actual parameters, the values of actual parameters are copied into the formal parameters.
- If the values of the formal parameters changes in the function, the values of the actual parameters are not changed.
- This way of passing parameters is called call by value (pass by value).

Call by value

```
#include <stdio.h>
void  swap(int x, int y);
int main()
{
int a,b;
printf("enter a, b values");
scanf("%d%d", &a,&b);
printf("before swapping a,b values:%d%d",a,b);
swap(a,b);      // function call
printf("after swapping a,b values:%d%d",a,b);
return 0;
}
void  swap(int x, int y)
{
int temp;
temp=x;
x=y;
y=temp;
}
```



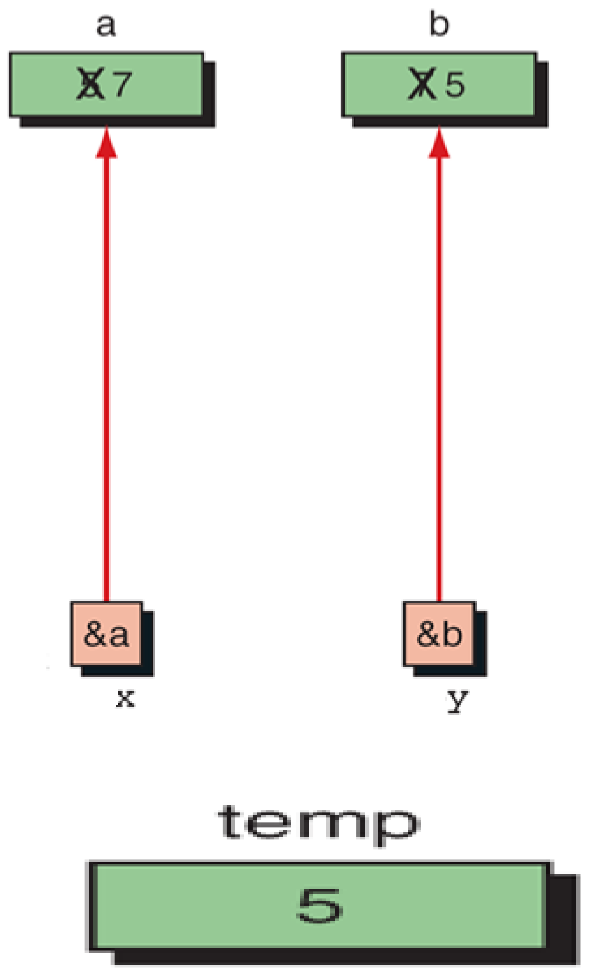


Call-by-Reference:

- In this method , the addresses of the actual arguments are copied to the corresponding parameters in the “called” function.
- Any modifications done to the formal parameters in the ”called function” causes the actual parameters to change.

Call by reference

```
#include <stdio.h>
void  swap(int *x, int *y);
int main()
{
int a,b;
printf("enter a, b values");
scanf("%d%d", &a,&b);
printf("before swapping a,b values:%d%d",a,b);
swap(&a,&b);
printf("after swapping a,b values:%d%d",a,b);
return 0;
}
void  swap(int *x, int *y)
{
int temp;
temp=*x;
*x=*y;
*y=temp;
}
```



Call by Value	Call by Reference
When Function is called the values of variables are passed.	When a function is called address of variables is passed.
Formal parameters contain the value of actual parameters.	Formal parameters contain the address of actual parameters.
Change of formal parameters in the function will not affect the actual parameters in the calling function	The actual parameters are changed since the formal parameters indirectly manipulate the actual parameters

Scope

- **Scope** determines the region of the program in which a defined object is **visible**.

- Scope pertains to any object that can be declared, such as a variable or a function declaration.

There are three types:

- **Global Scope (File Scope)**
- **Local Scope (Block Scope)**
- **Function Scope**

Rules for Local Scope(Block Scope):

- Block is zero or more statements enclosed in a set of braces.
- Variables are in scope from their point of declaration until the end of the block.
- Variables defined within a block have a local scope.
- They are visible in that block scope only. Outside the block they are not visible.
- For example, a variable declared in the formal parameter list of a function has block scope, and active only in the body only.
- Variable declared in the initialization section of a for loop has also block scope, but only within the for statement.

//Example for local scope or block scope

```
{
            int a = 2;
            printf (“%d\n”, a);                          /* outer block a 2 is printed */
            {
                        int a = 5;
                        printf (“%d\n”, a);             /* inner block a 5 is printed */
            }
            printf (“%d\n”, a);                         /* 2 is printed */
}
 /* a no longer defined */                              Outer “a” Masked.
```

- In the above block, A variable that is declared in an outer block is available in the inner block unless it is re-declared.
- In this case the outer block declaration is temporarily “masked”.

Rules for Global Scope (File Scope):

- File scope generally includes all declarations outside function and all function headers.
- An object with file scope sometimes referred to as a global object.
- File scope includes the entire source file for a program, including any files included in it.
- An object with file scope has visibility through the whole source file in which it is declared.
- For Example, a variable declared in the global section can be accessed from anywhere in the source program.

```
//Example program for global scope
void fun1();
void fun2();
int g=10;
void main()
{
        g++;
        printf("main=%d",g);
        fun1();
        printf("back in main=%d",g);
}
void fun1()
{
        g=30;
        printf("fun1 = %d\n",g);
        fun2();
}
void fun2()
{
    g++;
 printf("fun2=%d",g);
        }
```

```
//Example program for global scope
void fun1();
void fun2();
int count;
void main()
{
        count = 100;
        fun1();
}
void fun1()
{
        printf("Count is = %d\n",count);
        fun2();
}
void fun2()
{
        int count;
        for(count = 0;count < 5;count++)
                printf("*");
}
```

Output:

```
Count is = 100
*****
```

Local Variables:

- Variables that are declared in a block known as local variables.
- They are known in the block where they are created and active in the block only.
- They hold their values during the execution of the block.
- After completing the execution of the block they are undefined.

Global Variables:

- Global variables are known throughout the entire program and may be used in any piece of code.
- Also, they will hold their values during the entire execution of the program.
- Global variables are created by declaring them outside of the function and they can be accessed by any expression.
- In general we are declaring global variable at the top of the program.

Rules for Function Scope:

- Variables defined within a function (including main) are local to this function and no other function has direct access to them!
- The only way is by passing variables to a function as parameters.
- The only way to pass (a single) variable back to the calling function is via the return statement.

Recursion in C

- In C, functions can call themselves .
- **A function calling itself is called recursive function.**
- Recursion is a repetitive process, where the function calls itself.
- Generally **normal function** only knows how to solve the simplest case of the problem.
- When the simplest case is given as an input, the function will immediately return with an answer.
- However, if a more **complex inpu**t is given, a recursive function will divide the problem into 2 pieces:
    - A part that it knows how to solve and
    - Another part that it does not know how to solve.

Concept of recursive function:

- A recursive function is invoked or called to solve a problem.
- The statement that solves a problem is known as the **base case**.
- Every recursive function must have a **base case**.
- The rest of the function is known as the **general case**.
- The recursion step is done until the problem converges to become the simplest case.
- This simplest case will be solved by the function which will then return the answer to the previous copy of the function.
- The sequence of returns will then go all the way up until the original call of the function finally return the result.

Example: Recursive Factorial Function

Iteration Definition:

fact (n) = 1	if n=0
= n*(n-1)*(n-2)…….3*2*1	if n>0

Recursion Definition:

fact (n) = 1	if n=0	(Base Case)
= n*fact (n-1)	if n>0	(General Case)

```
#include<stdio.h>
long factorial (long);                                /* function prototype */
void main (void)
{
        long int n;
        printf("enter n");
        scanf("%ld",&n);
        printf ("%ld! = %1d\n",n, factorial (n));          /* function call */
}
long factorial (long n)                          /* function definition */
{
        if (n = =0)
                return 1;
        else
               return (n * factorial (n-1));
}
```

OUTPUT:

4! = 24

Designing a Recursive Function:

In the above program, once the base condition has reached, the solution begins.

The program has found one part of the answer and can return that part to the next more general statement.

Thus, after calculating factorial (0) is 1, and then it returns 1.That leads to solve the next general case,

factorial (1) → 1*factorial (0) → 1*1 → 1

The program now returns the value of factorial (1) to next general case, factorial (2),

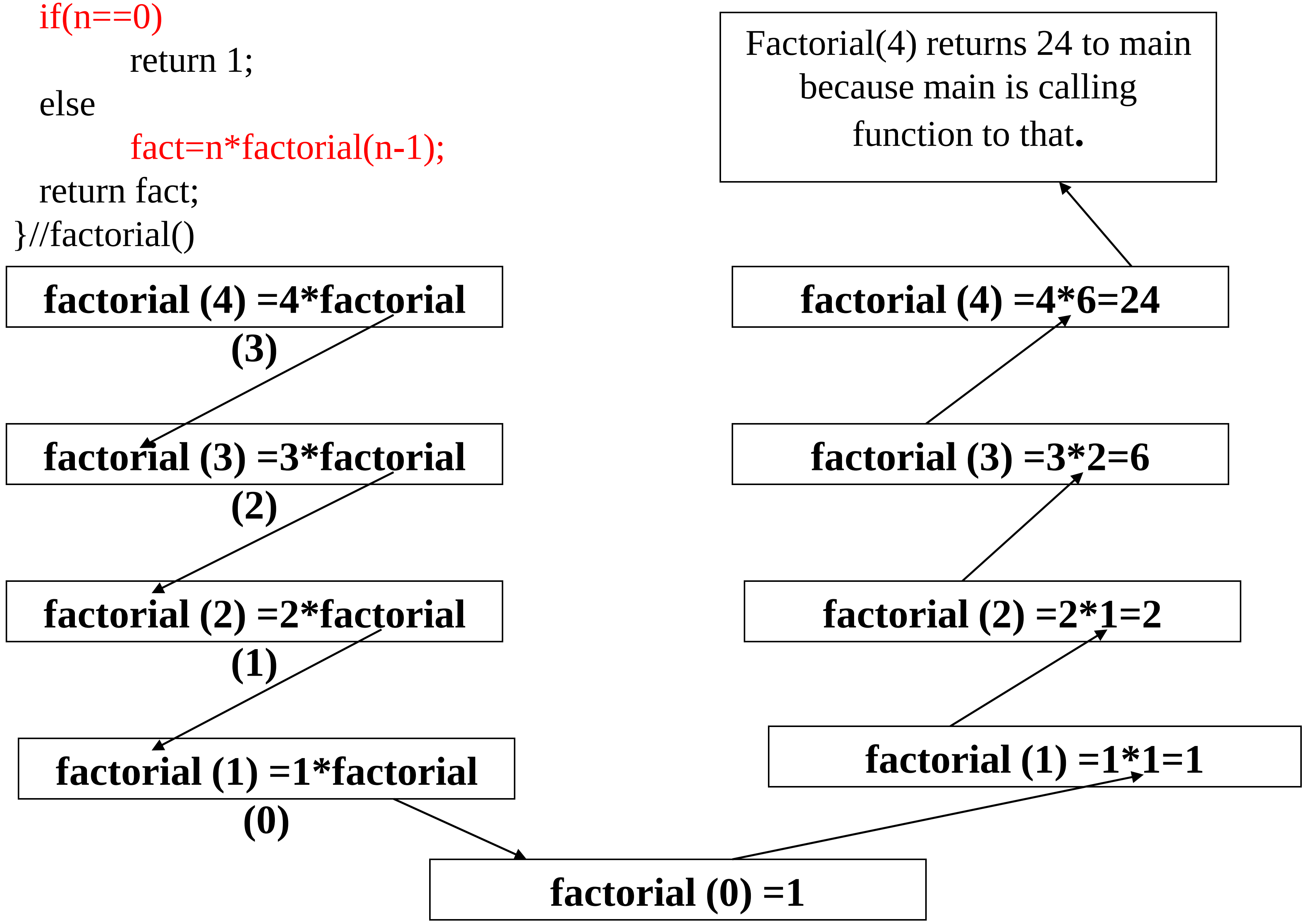
factorial (2) → 2*factorial (1) → 2*1 → 2

As the program solves each general case in turn, the program can solve the next higher general case, until it finally solves the most general case, the original problem.

The following are the rules for designing a recursive function:

1. First, determine the base case.
2. Then, determine the general case.
3. Finally, combine the base case and general case in to a function.

```
factorial(int n)
  {                                number=4
    int fact;
     if(n==0)
          return 1;
     else
          fact=n*factorial(n-1);
     return fact;
  }//factorial()
```


Factorial(4) returns 24 to main because main is calling function to that.

factorial (4) =4*factorial (3)
factorial (3) =3*factorial (2)
factorial (2) =2*factorial (1)
factorial (1) =1*factorial (0)
factorial (0) =1
factorial (1) =1*1=1
factorial (2) =2*1=2
factorial (3) =3*2=6
factorial (4) =4*6=24

Recursion

Advantages:

1. Complex case analysis and nested loops can be avoided.
2. Recursion can lead to more readable and efficient algorithm descriptions.
3. Through recursion one can solve problems in easy way while its iterative solution is very big and complex

Disadvantages:

1. Recursive solution is always logical and it is very difficult to debug and understand
2. In recursive we must have an if statement somewhere to force the function to return without the recursive call being executed, otherwise the function will never return
3. Recursion takes a lot of stack space
4. Recursion uses more processor time

Fibonacci using recursion

```
#include<stdio.h>
int fib(int n);
void main()
{
int  n,res;
printf(“enter n”);
scanf(“%d”,&n);
res=fib(n);
printf(“Nth Fibonacci term=%d”, res);
}
int fib(int  n)
{
if (n==1)
return 0;
else if(n==2)
return 1;
else
return(fib(n-1)+fib(n-2));
}
```

Ackerman using recursion

```
#include<stdio.h>
int ack(int m,int n);
void main()
{
int   m,n,res;
printf(“enter  m and n”);
scanf(“%d%d”, &m, &n);
res=ack(m,n);
printf(“value=%d”,res);
}
int  ack(int m,int  n)
{
if (m==0)
return (n+1);
else if(n==0)
return (ack(m-1,1));
else if(m>0 && n>0)
return(ack(m-1,ack(m,n-1)));
}
```

Difference between Iteration and Recursion

ITERATION	RECURSION
Iteration explicitly uses repetition structure.	Recursion achieves repetition by calling the same function repeatedly.
Iteration is terminated when the loop condition fails	Recursion is terminated when base case is satisfied.
May have infinite loop if the loop condition never fails	Recursion is infinite if there is no base case or if base case never reaches.
Iterative functions execute much faster and occupy less memory space.	Recursive functions are slow and takes a lot of memory space compared to iterative functions

Macros

Preprocessor Commands:

- The C compiler is made of two functional parts: **a preprocessor** and **a translator.**

- The preprocessor is a program which processes the source code before it passes through the compiler.

- The translator is a program which converts the program into machine language and gives the object module.

- There are three major tasks of **a preprocessor directive**:
  - Definition of symbolic constants and macros**(macro definition)**
  - Inclusion of other files **(file inclusion**
  - **Conditional** compilation of program code/Conditional execution of preprocessor directives

Symbolic Constants:

- Macro definition without arguments is referred as a **constant**.
- The body of the macro definition can be **any constant** value including integer, float, double, character, or string.
- However, character constants must be enclosed in **single quotes** and string constants in **double quotes**.

Example:

```
#define PI 3.142
 #define T ‘h’
 #define AND &&
 #define LESSTHAN <
 #define MES “Welcome to C”
```

```
#include<stdio.h>
#define  T ‘h’
int main()
{
  printf(“ %c", T);
  return 0;
}
```

```
#include <stdio.h>
 #define L “hello”
 int main()
{
printf(L);
return 0 ;
}
```

```
#include <stdio.h>
 #define PI 3.1415
 int main()
{
 float r, a;
printf("Enter the radius: ");
scanf(“%f", &r);
a=PI*r*r;
printf("Area=%f",a);
 return 0;
 }
```

```
#include<stdio.h>
#define L  <
int main()
{
 int a=10,b=5;
if( a L b)
  printf(“ a is small”);
else
printf(“b is small”);
  return 0;
}
```

```
#include <stdio.h>
 #define P  printf(“welcome”);\
                  printf(“Snist”);
 int main()
{
 printf(“start”);
P
 return 0;
 }
```

Macro Definition:

- A macro definition command associates a name with a sequence of tokens.
- The name is called the macro name and the tokens are referred to as the macro body.

The following syntax is used to define a macro:

#define macro_name(<arg_list>) macro_body

Here **#define** is a define directive, **macro_name** is a valid C identifier.

macro_body is used to specify how the name is replaced in the program before it is compiled.

Macro Definition (Cont…):

- Macro must be coded on a single line.
- We can use **backslash(\ )** followed **immediately** by a new line to code macro in multiple lines.
- Performs a text substitution – no data type checking.
- We need to carefully code the macro body.
- Whenever a macro call is encounter, the preprocessor replaces the call with the macro body.
- If body is not created carefully it may create an error or undesired result.

1.We can also define macros that works like a function call

```
#include<stdio.h>
#define square(x) (x*x)
void main()
{
int a=10;
printf("\nThe square of %d=%d", a, square(a));
}
```

2.The macros can take function like arguments, the arguments are not checked for data type.

```
#include<stdio.h>
#define   max(x,y) x>y?x:y
void main()
{
int a,b;
float x,y;
printf("enter a,b,x,y");
scanf("%d%d%x%y",&a,&b,&x,&y);
printf("\nmaxvalue=%d",  max(a,b));
printf("\n max=%f",max(x,y));
}
```

Nested Macros:

- C handles nested macros by simply rescanning a line after macro expansion.

- Therefore, if an expansion results in a new statement with a macro ,the second macro will be properly expanded.

For Example:

```
#define sqre(a)  (a*a)
#define cube(a)  (sqre(a)*a)
```

The expansion of	**x=cube(4);**
results in the following expansion:	**x=(sqre(4)*4);**
After rescanning becomes	**x=((4*4)*4);**

File Inclusion:

- ➢ The first job of a preprocessor is file inclusion that is copying of one or more files into programs.

- ➢ The files are usually header files and external files containing functions and data declarations.

General form is:

#include filename

It has two different forms:

1. #include <filename>

- ➢ It is used to direct the preprocessor to include header files from the system library.

2. #include “filename”

- ➢ It is used to direct the preprocessor look for the files in the current working directory and standard library.

```
//Example program to include user defined header file

#include<stdio.h>
 #include "MyFile.h"
 void main()
{
        int a=10,b=20;
        printf("\n The sum=%d", sum(a,b));
}
```

```
MyFile.h
        #include<stdio.h>
        #define sum(x, y)  (x+y)
```

OUTPUT:

```
        The sum=30
```

Macro	Function
Macro is **Preprocessed**	Function is **Compiled**
No Type Checking	**Type Checking** is Done
Code Length **Increases**	**Code** Length remains **Same**
Speed of Execution is **Faster**	Speed of Execution is **Slower**
Before Compilation macro name is replaced by macro value	During function call , Transfer of Control takes place
Useful where small code appears many time	Useful where large code appears many time
Generally Macros do not extend beyond one line	Function can be of any number of lines
Macro does not Check **Compile Errors**	Function Checks **Compile Errors**

Difference between Macro and Function

Storage classes in c

- The storage class determines the part of member storage is allocated for an object and how long the storage allocation continues to exit.
- A scope specifies the part of the program which a variable name is visible, that is the accessibility of the variable by its name.
- **Storage class tells us:**

  1) Where the variable is stored.

  2) Initial value of the variable.

  3) Scope of the variable. Scope specifies the part of the program which a variable is accessed.

  4) Life of the variable. How long variable exist in the program.
- **There are four types of storage classes:**

- 1) Automatic storage class
- 2) Register storage class
- 3) Static storage class
- 4) External storage class

1.Automatic storage class

1. the variable is declared with keyword auto. (auto keyword is optional while declaring variables.)
2. they must be declared at the start of a block.
3. Memory is allocated automatically upon entry to a block and freed automatically upon exit from the block.
4. Storage- automatic variable stored in the memory.
5. Default initial value- garbage value.
6. Scope- local to the block in which they are declared, including any blocks nested within that block. For this reasons automatic variable are also called local variable.
7. Life – within the block in which the variable is defined.

Examples

```
 main()
  {
    auto int x;
   printf("%d",x);
}
```

Output – gives garbage value.

```
 void main()
{
  int x=10;
  {
   int x=20;
   {
     int x=30;
     printf("x=%d",x);
   }
   printf("x=%d",x);
  }
printf("x=%d",x);
}
```

Output- 30 20 10

2. Register storage class

1. Automatic variables are allocated storage in the main memory of the computer; however, for most computers, accessing data in memory is considerably slower than processing directly in the CPU.
2. Registers are memory located within the CPU itself where data can be stored and accessed quickly. Normally, the **compiler determines** what data is to be stored in the registers of the CPU at what times.
3. Thus, register variables provide a certain control over efficiency of program execution.
4. Variables which are used repeatedly or whose access times are critical may be declared to be of storage class register.
5. Variables can be declared with keyword register as register int x;
6. Storage- variable stored in cpu registers rather than memory.
7. Default initial value- garbage value.
8. Scope- local to the block in which they are declared, including any blocks nested within that block. For this reasons automatic variable are also called local variable.
9. Life – within the block in which the variable is defined.

```
void  main()
 {
   register int x;
```

3.Static storage class

1. Variables must be declared with the key word static. static int x;
2. Storage- variable stored in computer memory.
3. Default initial value- zero.
4. Scope- local to the block in which they are declared.
5. Life –value of the variable persists between the different function calls.
6. Static automatic variables continue to exist even after the block in which they are defined terminates. Thus, the value of a static variable in a function is retained between repeated function calls to the same function.
7. In the case of recursive function calls we use s**tatic** storage class.
8. Static variables may be initialized in their declarations; however, the initializes must be **constant expressions**, and initialization is done only once at compile time when memory is allocated for the static variable.
9. Static and auto variables are differ in their life and initial value.

Difference between static and auto

```
 void main()
{
 add();
 add();
 add();

}//main
 add()
{
 auto int i=1;
 printf(“%d”,i);
 i++;
}//add
```

Out put 1 1 1

```
 void main()
{
 add();
 add();
 add();

}//main
 add()
{
 static  int i=1;
 printf(“%d”,i);
 i++;
}//add
```

Out put 1 2 3

```
 void main()
{
 static int i;
printf(“%d”,i);
 }
```

Out put 0

4. External storage class

1. All variables we have seen so far have had limited scope (the block in which they are declared) and limited lifetimes (as for automatic variables).
2. These variables are declared with keyword extern as

   extern int x; (extern keyword may be omitted).
3. Storage- Memory.
4. Default initial value- zero.
5. Scope- as long as the programs execution.
6. Life- doesn’t come to an end.
7. External variables are declared outside the all the functions. So that they are available to all the functions in a program.

```
 int i;
 void main()
{
   printf(“%d”,i);->0
   add();--------→1
   add();------→2
   sub();------→1
   sub();----→0
    }//main()
 add()
{
   i++;
  printf(“i=%d”i);
}//add()
Sub()
 {
    i--;
    printf(“i=%d”,i);
 }//sub()
```

```
 extern int x=100;
 void main()
{
  int x=200;
   printf(“x=%d”,x);
   display();
}//
 display()
{
          printf(“%d”,x);
}
```

Note- local variable has highest preference than global variable so that local variable value gets printed.

```
 int x=20;
void main()
 {
   extern int y;
   printf(“%d %d”,x,y);
}
 int y=20;
```

- If any variable declared out side of the main is treated as extern variable/ global variable.
-If any variable declared as global variable inside of the main that must be declared with key word extern and defined outside of the main.