**What is C function?**

     A large C program is divided into basic building blocks called C function. C function contains set of instructions enclosed by “{  }” which performs specific operation in a C program. Actually, Collection of these functions creates a C program.

**2. Uses of C functions:**

* + C functions are used to avoid rewriting same logic/code again and again in a program.
  + There is no limit in calling C functions to make use of same functionality wherever required.
  + We can call functions any number of times in a program and from any place in a program.
  + A large C program can easily be tracked when it is divided into functions.
  + The core concept of C functions are, re-usability, dividing a big task into small pieces to achieve the functionality and to improve understandability of very large C programs.

**3. C function declaration, function call and function definition:**

There are 3 aspects in each C function. They are,

* + Function declaration or prototype  - This informs compiler about the function name, function parameters and  return value’s data type.
  + Function call – This calls the actual function
  + Function definition – This contains all the statements to be executed.

| **S.no** | **C function aspects** | **syntax** |
| --- | --- | --- |
| 1 | function definition | return\_type function\_name ( arguments list ) { Body of function; } |
| 2 | function call | function\_name ( arguments list ); |
| 3 | function declaration | return\_type function\_name ( argument list ); |

**Simple example program for C function:**

* + As you know, functions should be declared and defined before calling in a C program.
  + In the below program, function “square” is called from main function.
  + The value of “m” is passed as argument to the function “square”. This value is multiplied by itself in this function and multiplied value “p” is returned to main function from function “square”.

|  |
| --- |
| #include<stdio.h>  // function prototype, also called function declaration  float square ( float x )    // main function, program starts from here  int main( )  {            float m, n ;          printf ( "\nEnter some number for finding square \n");          scanf ( "%f", &m ) ;          // function call          n = square ( m ) ;          printf ( "\nSquare of the given number %f is %f",m,n );    }    float square ( float x )   // function definition  {          float p ;          p = x \* x ;          return ( p ) ;  } |

**4. How to call C functions in a program?**

There are two ways that a C function can be called from a program. They are,

* 1. Call by value
  2. Call by reference

**1. Call by value:**

* + In call by value method, the value of the variable is passed to the function as parameter.
  + The value of the actual parameter can not be modified by formal parameter.
  + Different Memory is allocated for both actual and formal parameters. Because, value of actual parameter is copied to formal parameter.

Note:

* + Actual parameter – This is the argument which is used in function call.
  + Formal parameter – This is the argument which is used in function definition

**Example program for C function (using call by value):**

* + In this program, the values of the variables “m” and “n” are passed to the function “swap”.
  + These values are copied to formal parameters “a” and “b” in swap function and used.

|  |
| --- |
| #include<stdio.h>  // function prototype, also called function declaration  void swap(int a, int b);    int main()  {      int m = 22, n = 44;      // calling swap function by value      printf(" values before swap  m = %d \nand n = %d", m, n);      swap(m, n);  }    void swap(int a, int b)  {      int tmp;      tmp = a;      a = b;      b = tmp;      printf(" \nvalues after swap m = %d\n and n = %d", a, b);  } |

**2. Call by reference:**

* + In call by reference method, the address of the variable is passed to the function as parameter.
  + The value of the actual parameter can be modified by formal parameter.
  + Same memory is used for both actual and formal parameters since only address is used by both parameters.

**Example program for C function (using call by reference):**

* + In this program, the address of the variables “m” and “n” are passed to the function “swap”.
  + These values are not copied to formal parameters “a” and “b” in swap function.
  + Because, they are just holding the address of those variables.
  + This address is used to access and change the values of the variables.

|  |
| --- |
| #include<stdio.h>  // function prototype, also called function declaration  void swap(int \*a, int \*b);    int main()  {        int m = 22, n = 44;      //  calling swap function by reference      printf("values before swap m = %d \n and n = %d",m,n);      swap(&m, &n);    }    void swap(int \*a, int \*b)  {      int tmp;      tmp = \*a;      \*a = \*b;      \*b = tmp;      printf("\n values after swap a = %d \nand b = %d", \*a, \*b);  } |

**TYPE OF C FUNCTION**

* 1. C function with arguments (parameters) and with return value
  2. C function with arguments (parameters) and without return value
  3. C function without arguments (parameters) and without return value
  4. C function without arguments (parameters) and with return value

| **S.no** | **C function** | **syntax** |
| --- | --- | --- |
| 1 | with arguments and with return values | int function ( int );         // function declaration function ( a );                // function call int function( int a )       // function definition {statements;  return a;} |
| 2 | with arguments and without return values | void function ( int );     // function declaration function( a );                // function call void function( int a )   // function definition {statements;} |
| 3 | without arguments and without return values | void function();             // function declaration function();                     // function call void function()              // function definition {statements;} |
| 4 | without arguments and with return values | int function ( );             // function declaration function ( );                  // function call int function( )               // function definition {statements;  return a;} |

**Note:**

* + If the return data type of a function is “void”, then, it can’t return any values to the calling function.
  + If the return data type of the function is other than void such as “int, float, double etc”, then, it can return values to the calling function.

**1. Example program for with arguments & with return value:**

      In this program, integer, array and string are passed as arguments to the function. The return type of this function is “int” and value of the variable “a” is returned from the function. The values for array and string are modified inside the function itself.

|  |
| --- |
| **#include<stdio.h>**  **#include<conio.h>**  **int add(int x, int y)**  **{**  **int result;**  **result = x+y;**  **return(result);**  **}**  **void main()**  **{**  **int z;**  **clrscr();**  **z = add(952,321);**  **printf("Result %d.\n\n",add(30,55));**  **printf("Result %d.\n\n",z);**  **getch();**  **}** |

**2. Example program for with arguments & without return value:**

      In this program, integer, array and string are passed as arguments to the function. The return type of this function is “void” and no values can be returned from the function. All the values of integer, array and string are manipulated and displayed inside the function itself.

|  |
| --- |
| #include<stdio.h>  #include<conio.h>  void add(int x, int y)  {  int result;  result = x+y;  printf("Sum of %d and %d is  %d.\n\n",x,y,result);  }  void main()  {  clrscr();  add(30,15);  add(63,49);  add(952,321);  getch();  } |

**3. Example program for without arguments & without return value:**

      In this program, no values are passed to the function “test” and no values are returned from this function to main function.

|  |
| --- |
| #include<stdio.h>    void test();    int main()  {      test();      return 0;  }    void test()  {         int a = 50, b = 80;         printf("\nvalues : a = %d and b = %d", a, b);  } |

**4. Example program for without arguments & with return value:**

      In this program, no arguments are passed to the function “sum”. But, values are returned from this function to main function. Values of the variable a and b are summed up in the function “sum” and the sum of these value is returned to the main function.

|  |
| --- |
| #include<stdio.h>    int sum();    int main()  {      int addition;      addition = sum();      printf("\nSum of two given values = %d", addition);      return 0;  }    int sum()  {         int a = 50, b = 80, sum;           sum = a + b;         return sum;  } |

**Do you know how many values can be return from C functions?**

* Always, only one value can be returned from a function.
* If you try to return more than one values from a function, only one value will be returned that appears at the right most place of the return statement.
* For example, if you use “return a,b,c” in your function, value for c only will be returned and values a, b won’t be returned to the program.
* In case, if you want to return more than one values, pointers can be used to directly change the values in address instead of returning those values to the function.

**ARRAY**

Array is a collection of variables belongings to the same data type. You can store group of data of same data type in an array.

* + Array might be belonging to any of the data types
  + Array size must be a constant value.
  + Always, Contiguous (adjacent) memory locations are used to store array elements in memory.
  + It is a best practice to initialize an array to zero or null while declaring, if we don’t assign any values to array.

**Example for C Arrays:**

* + int a[10];       // integer array
  + char b[10];   // character array   i.e. string

**Types of C arrays:**

There are 2 types of C arrays. They are,

* 1. One dimensional array
  2. Multi dimensional array
     + Two dimensional array
     + Three dimensional array, four dimensional array etc…

**1. One dimensional array in C:**

* + Syntax : data-type arr\_name[array\_size];

|  |  |  |
| --- | --- | --- |
| **Array declaration** | **Array initialization** | **Accessing array** |
| Syntax:  data\_type arr\_name [arr\_size]; | data\_type arr\_name [arr\_size]= (value1, value2, value3,….); | arr\_name[index]; |
| int age [5]; | int age[5]={0, 1, 2, 3, 4, 5}; | age[0];\_/\*0\_is\_accessed\*/ age[1];\_/\*1\_is\_accessed\*/ age[2];\_/\*2\_is\_accessed\*/ |
| char str[10]; | char str[10]={‘H’,‘a’,‘i’}; (or) char str[0] = ‘H’; char str[1] = ‘a’; char str[2] = ‘i; | str[0];\_/\*H is accessed\*/ str[1];  /\*a is accessed\*/ str[2];  /\* i is accessed\*/ |

**Example program for one dimensional array in C:**

|  |
| --- |
| #include<stdio.h>    int main()  {      int i;      int arr[5] = {10,20,30,40,50};      // declaring and Initializing array in C      //To initialize all array elements to 0, use int arr[5]={0};      /\* Above array can be initialized as below also         arr[0] = 10;         arr[1] = 20;         arr[2] = 30;         arr[3] = 40;         arr[4] = 50;      \*/      for (i=0;i<5;i++)      {          // Accessing each variable          printf("value of arr[%d] is %d \n", i, arr[i]);      }  } |

**2. Two dimensional array in C:**

* + Two dimensional array is nothing but array of array.
  + syntax : data\_type array\_name[num\_of\_rows][num\_of\_column]

|  |  |  |  |
| --- | --- | --- | --- |
| **S.no** | **Array declaration** | **Array initialization** | **Accessing array** |
| 1 | Syntax:  data\_type arr\_name [num\_of\_rows][num\_of\_column]; | data\_type arr\_name[2][2] = {{0,0},{0,1},{1,0},{1,1}}; | arr\_name[index]; |
| 2 | Example: int arr[2][2]; | int arr[2][2] = {1,2, 3, 4}; | arr [0] [0] = 1;  arr [0] ]1] = 2; arr [1][0]  = 3; arr [1] [1] = 4; |

**Example program for two dimensional array in C:**

|  |
| --- |
| #include<stdio.h>  int main()  {      int i,j;      // declaring and Initializing array      int arr[2][2] = {10,20,30,40};      /\* Above array can be initialized as below also         arr[0][0] = 10;   // Initializing array         arr[0][1] = 20;         arr[1][0] = 30;         arr[1][1] = 40;      \*/      for (i=0;i<2;i++)      {         for (j=0;j<2;j++)         {            // Accessing variables            printf("value of arr[%d] [%d] : %d\n",i,j,arr[i][j]);         }      }  } |

# C – Pointer

* C Pointer is a variable that stores/points the address of the another variable.
* C Pointer is used to allocate memory dynamically i.e. at run time.
* The variable might be any of the data type such as int, float, char, double, short etc.

Syntax

data\_type\*var\_name;  
Example : int \*p;  char \*p;

* Where, \* is used to denote that “p” is pointer variable and not a normal variable.

#### ****Key points to remember about pointers in C:****

* Normal variable stores the value whereas pointer variable stores the address of the variable.
* The content of the C pointer always be a whole number i.e. address.
* Always C pointer is initialized to null, i.e. int \*p = null.
* The value of null pointer is 0.
* & symbol is used to get the address of the variable.
* \* symbol is used to get the value of the variable that the pointer is pointing to.
* If pointer is assigned to NULL, it means it is pointing to nothing.
* Two pointers can be subtracted to know how many elements are available between these two pointers.
* But, Pointer addition, multiplication, division are not allowed.
* The size of any pointer is 2 byte (for 16 bit compiler).

#### ****Example program for pointer in C:****

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| #include <stdio.h>  int main()  {        int \*ptr, q;          q = 50;          /\* address of q is assigned to ptr       \*/          ptr = &q;          /\* display q's value using ptr variable \*/          printf("%d", \*ptr);          return 0;  } Storage Class Specifiers Storage class specifiers in C language tells the compiler where to store a variable, how to store the variable, what is the initial value of the variable and life time of the variable.  **Syntax:** storage\_specifier data\_type variable \_name  **Types of Storage Class Specifiers in C:**  There are 4 storage class specifiers available in C language. They are,   * 1. auto   2. extern   3. static   4. register  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **S.No.** | **Storage Specifier** | **Storage place** | **Initial / default value** | **Scope** | **Life** | | **1** | **auto** | CPU Memory | Garbage value | local | Within the function only. | | **2** | **extern** | CPU memory | Zero | Global | Till the end of the main program. Variable definition might be anywhere in the C program | | **3** | **static** | CPU memory | Zero | local | Retains the value of the variable between different function calls. | | **4** | **register** | Register memory | Garbage value | local | Within the function |   **Note:**   * + For faster access of a variable, it is better to go for register specifiers rather than auto specifiers.   + Because, register variables are stored in register memory whereas auto variables are stored in main CPU memory.   + Only few variables can be stored in register memory. So, we can use variables as register that are used very often in a C program.   **Example program for auto variable in C:**  The scope of this auto variable is within the function only. It is equivalent to local variable. All local variables are auto variables by default.   |  | | --- | | #include<stdio.h>  void increment(void);    int main()  {     increment();     increment();     increment();     increment();     return 0;  }    void increment(void)  {     auto int i = 0 ;     printf ( "%d ", i ) ;     i++;  } |   **Example program for static variable in C:**        Static variables retain the value of the variable between different function calls.   |  | | --- | | //C static example  #include<stdio.h>  void increment(void);    int main()  {     increment();     increment();     increment();     increment();     return 0;  }    void increment(void)  {     static int i = 0 ;     printf ( "%d ", i ) ;     i++;  } |   **Example program for extern variable in C:**  The scope of this extern variable is throughout the main program. It is equivalent to global variable. Definition for extern variable might be anywhere in the C program.   |  | | --- | | #include<stdio.h>  int x = 10 ;  int main( )  {     extern int y ;     printf ( "The value of x is %d \n", x ) ;     printf ( "The value of y is %d",y ) ;     return 0;  }  int y = 50 ; |   **Example program for register variable in C:**   * + Register variables are also local variables, but stored in register memory. Whereas, auto variables are stored in main CPU memory.   + Register variables will be accessed very faster than the normal variables since they are stored in register memory rather than main memory.   + But, only limited variables can be used as register since register size is very low. (16 bits, 32 bits or 64 bits)  |  | | --- | | #include <stdio.h>    int main()  {     register int i;     int arr[5];          // declaring array     arr[0] = 10;         // Initializing array     arr[1] = 20;     arr[2] = 30;     arr[3] = 40;     arr[4] = 50;     for (i=0;i<5;i++)     {        // Accessing each variable        printf("value of arr[%d] is %d \n", i, arr[i]);     }     return 0;  } |   **Preprocessor directives:**   * Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing. * Commands used in preprocessor are called preprocessor directives and they begin with “#” symbol. * Below is the list of preprocessor directives that C language offers.  |  |  |  |  | | --- | --- | --- | --- | | **S.no** | **Preprocessor** | **Syntax** | **Description** | | 1 | Macro | #define | This macro defines constant value and can be any of the basic data types. | | 2 | Header file inclusion | #include <file\_name> | The source code of the file “file\_name” is included in the main program at the specified place | | 3 | Conditional compilation | #ifdef, #endif, #if, #else,  #ifndef | Set of commands are included or excluded in source program before compilation with respect to the condition | | 4 | Other directives | #undef, #pragma | #undef is used to undefine a defined macro variable. #Pragma is used to call a function before and after main function in a C program |   A program in C language involves into different processes.  Below diagram will help you to understand all the processes that a C program comes across.  http://fresh2refresh.com/wp-content/uploads/object%20and%20exe.png  **Example program for #define, #include preprocessors in C:**   * + #define **-**This macro defines constant value and can be any of the basic data types.   + #include <file\_name> **-**The source code of the file “file\_name” is included in the main C program where “#include <file\_name>” is  mentioned.  |  | | --- | | #include <stdio.h>    #define height 100  #define number 3.14  #define letter 'A'  #define letter\_sequence "ABC"  #define backslash\_char '\?'    void main()  {     printf("value of height    : %d \n", height );     printf("value of number : %f \n", number );     printf("value of letter : %c \n", letter );     printf("value of letter\_sequence : %s \n", letter\_sequence);     printf("value of backslash\_char  : %c \n", backslash\_char);    } |   **Example program for conditional compilation directives:**  **a)   Example program for #ifdef, #else and #endif in C:**   * + “#ifdef” directive checks whether particular macro is defined or not. If it is defined, “If” clause statements are included in source file.   + Otherwise, “else” clause statements are included in source file for compilation and execution.  |  | | --- | | #include <stdio.h>  #define RAJU 100    int main()  {     #ifdef RAJU     printf("RAJU is defined. So, this line will be added in " \            "this C file\n");     #else     printf("RAJU is not defined\n");     #endif     return 0;  } |   **b)  Example program for #ifndef and #endif in C:**   * + #ifndef exactly acts as reverse as #ifdef directive. If particular macro is not defined, “If” clause statements are included in source file.   + Otherwise, else clause statements are included in source file for compilation and execution.  |  | | --- | | #include <stdio.h>  #define RAJU 100  int main()  {     #ifndef SELVA     {        printf("SELVA is not defined. So, now we are going to " \               "define here\n");        #define SELVA 300     }     #else     printf("SELVA is already defined in the program”);       #endif     return 0;    } |   **c)   Example program for #if, #else and #endif in C:**   * + “If” clause statement is included in source file if given condition is true.   + Otherwise, else clause statement is included in source file for compilation and execution.  |  | | --- | | #include <stdio.h>  #define a 100  int main()  {     #if (a==100)     printf("This line will be added in this C file since " \            "a \= 100\n");     #else     printf("This line will be added in this C file since " \            "a is not equal to 100\n");     #endif     return 0;  } |   **Example program for undef in C:**  This directive undefines existing macro in the program.   |  | | --- | | #include <stdio.h>    #define height 100  void main()  {     printf("First defined value for height    : %d\n",height);     #undef height          // undefining variable     #define height 600     // redefining the same for new value     printf("value of height after undef \& redefine:%d",height);  } |   **Example program for pragma in C:**  Pragma is used to call a function before and after main function in a C program.   |  | | --- | | #include <stdio.h>    void function1( );  void function2( );    #pragma startup function1  #pragma exit function2    int main( )  {     printf ( "\n Now we are in main function" ) ;     return 0;  }    void function1( )  {     printf("\nFunction1 is called before main function call");  }    void function2( )  {     printf ( "\nFunction2 is called just before end of " \              "main function" ) ;"  } |   **More on pragma directive in C:**   |  |  |  | | --- | --- | --- | | **S.no** | **Pragma command** | **description** | | 1 | #Pragma startup <function\_name\_1> | This directive executes function named “function\_name\_1” before | | 2 | #Pragma exit <function\_name\_2> | This directive executes function named “function\_name\_2” just before termination of the program. | | 3 | #pragma warn – rvl | If function doesn’t return a value, then warnings are suppressed by this directive while compiling. | | 4 | #pragma warn – par | If function doesn’t use passed function parameter , then warnings are suppressed | | 5 | #pragma warn – rch | If a non reachable code is written inside a program, such warnings are suppressed by this directive. | |