1. Write a C program to search a given number in the array using 1D array.

#include <stdio.h>

int main()

{

int arr[100], n, i, num, found = 0;

printf("Enter the size of the array: ");

scanf("%d", &n);

printf("Enter the array elements: ");

for(i=0; i<n; i++)

{

scanf("%d", &arr[i]);

}

printf("Enter the number to be searched: ");

scanf("%d", &num);

for(i=0; i<n; i++)

{

if(arr[i] == num)

{

printf("Number found at index %d.\n", i);

found = 1;

break;

}

}

if(found == 0)

{

printf("Number not found in the array.\n");

}

return 0;

}

1. Write a program, which reads your name from the keyboard

and outputs a list of ASCII codes, which represent your name.

#include <stdio.h>

int main()

{

char name[100];

int i;

printf("Enter your name: ");

scanf("%s", name);

printf("The ASCII codes for your name are: ");

for(i=0; name[i]!='\0'; i++)

{

printf("%d ", name[i]);

}

return 0;

}

1. Write a C program to illustrate function with arguments and no return value.

#include <stdio.h>

void printMessage(char message[])

{

printf("The message is: %s\n", message);

}

int main()

{

char myMessage[] = "Hello, world!";

printMessage(myMessage);

return 0;

}

1. Differentiate actual parameters and formal parameters. Write an example program

Formal parameters refer to the parameters that are declared in the function definition, and are used to define the inputs that the function expects to receive. Formal parameters are also known as function parameters or arguments.

Actual parameters refer to the values that are passed to a function when it is called. Actual parameters are also known as function arguments.

#include <stdio.h>

void printSum(int a, int b)

{

int sum = a + b;

printf("The sum of %d and %d is %d\n", a, b, sum);

}

int main()

{

int x = 5, y = 7;

printSum(x, y); // x and y are actual parameters

return 0;

}

1. Write a C program to calculate the root of a Quadratic Equation using Switch case statement.

# include<stdio.h>

# include<conio.h>

# include<math.h>

main (){

   float a,b,c,r1,r2,d;

   printf (“enter the values of a b c”);

   scanf (“ %f %f %f”, &a, &b, &c);

   d= b\*b – 4\*a\*c;

   if (d>0){

      r1 = -b+sqrt (d) / (2\*a);

      r2 = -b-sqrt (d) / (2\*a);

      printf (“The real roots = %f %f”, r1, r2);

   }

   else if (d= =0){

      r1 = -b/(2\*a);

      r2 = -b/(2\*a);

      printf (“roots are equal =%f %f”, r1, r2);

   }

   else

      printf(“Roots are imaginary”);

   getch ();

}

1. Given a number, write a program using while loop to reverse the digits of the number.

For example, the number

12345 should be written as 54321

#include <stdio.h>

int main()

{

int num, reversed\_num = 0, remainder;

printf("Enter a number: ");

scanf("%d", &num);

while(num != 0)

{

remainder = num % 10;

reversed\_num = reversed\_num \* 10 + remainder;

num = num / 10;

}

printf("The reversed number is: %d\n", reversed\_num);

return 0;

}

1. What are the differences between NULL pointer and void pointer?

NULL pointer: A NULL pointer is a pointer that does not point to any memory address. It is often used as a marker or sentinel value to indicate the end of a list or data structure. It is defined as **(void \*)0** or simply **0** and can be assigned to any pointer type. A NULL pointer cannot be dereferenced as it points to nothing.

Void pointer: A void pointer is a pointer that points to a memory location of any data type. It is declared as **void \*** and can be used to store the address of any data type. Since a void pointer does not have any associated data type, it cannot be directly dereferenced. However, it can be cast to any other pointer type before dereferencing.

The null pointer is basically used in a program to assign the value 0 to a pointer variable of any data type. The void pointer, on the other hand, has no value assigned to it and we use it to store the addresses of other variables in the program- irrespective of their data types.

1. Write a program which will read a text and count all occurrences of a particular word.

#include <stdio.h>

#include <string.h>

int main()

{

char text[1000], word[50];

int i, count = 0;

printf("Enter a text: ");

fgets(text, sizeof(text), stdin);

printf("Enter a word to count: ");

scanf("%s", word);

for(i=0; i<strlen(text); i++)

{

if(strncmp(&text[i], word, strlen(word)) == 0)

{

count++;

}

}

printf("The word \"%s\" occurs %d times in the text.\n", word, count);

return 0;

}

1. What are the various types of conditional statements? Explain each with an example.

There are three types of conditional statements in C programming: if, if-else, and switch. Here's an explanation and an example of each:

1. If statement: The **if** statement is used to execute a block of code if a condition is true. The syntax of the **if** statement is as follows:

if (condition)

{

// code to execute if the condition is true

}

Here's an example that uses the **if** statement to check if a number is positive:

#include <stdio.h>

int main()

{

int num;

printf("Enter a number: ");

scanf("%d", &num);

if (num > 0)

{

printf("The number is positive.\n");

}

return 0;

}

2.If-else statement: The **if-else** statement is used to execute one block of code if a condition is true, and another block of code if the condition is false. The syntax of the **if-else** statement is as follows:

if (condition)

{

// code to execute if the condition is true

}

else

{

// code to execute if the condition is false

}

Here's an example that uses the **if-else** statement to check if a number is even or odd:

#include <stdio.h>

int main()

{

int num;

printf("Enter a number: ");

scanf("%d", &num);

if (num % 2 == 0)

{

printf("The number is even.\n");

}

else

{

printf("The number is odd.\n");

}

return 0;

}

3.Switch statement: The **switch** statement is used to execute different blocks of code depending on the value of a variable. The syntax of the **switch** statement is as follows

switch (variable)

{

case value1:

// code to execute if variable equals value1

break;

case value2:

// code to execute if variable equals value2

break;

// more cases

default:

// code to execute if variable doesn't match any case

break;

}

Here's an example that uses the **switch** statement to print the name of a day of the week based on its number:

#include <stdio.h>

int main()

{

int day;

printf("Enter a day number (1-7): ");

scanf("%d", &day);

switch (day)

{

case 1:

printf("Sunday\n");

break;

case 2:

printf("Monday\n");

break;

case 3:

printf("Tuesday\n");

break;

case 4:

printf("Wednesday\n");

break;

case 5:

printf("Thursday\n");

break;

case 6:

printf("Friday\n");

break;

case 7:

printf("Saturday\n");

break;

default:

printf("Invalid day number\n");

break;

}

return 0;

}

1. Explain the various categories of user defined functions in C with examples?

**Function Categories**

Functions are defined using parameters, arguments, and return values. All these elements can be used in different ways in a program to define the functionality of a function as per the need of the user. When we talk about types of functions in C, user-defined functions can be broadly classified in four different ways:

**a) Functions With Arguments and Return Values**

This type of function has arguments and always returns a value. In this method, the arguments are passed to the function while calling it. The function will return some value when it is called from main() or any subfunction in the program. In it, data types are a must to define. If the return is of 'int' type, then the return value will also be 'int' type. This type of user-defined function is also called a fully dynamic function because the total control is in the hand of the end-user.

Lets see an example:

#include <stdio.h>

void main()

{

int sub(int,int); *// return value and arguments of function*

int a=15,b=7;

int result = sub(a,b);

printf("a-b = %d",result);

}

int sub(int x,int y) *// return value and arguments of function*

{

return(x-y); *// this is return value,'int type'*

}

**Output**:

a-b = 8

**b) Functions With Arguments and Without Return Values**

In this type of function, the arguments are also passed to the function while calling it. But it will not return any value when the function is called from the main function or any submethod. This function contains arguments but does not return any value. In this method, we allow the user to enter the values as input rather than fixed values. Since we are allowing the user to enter the values as input, we do not expect any return type value. This type of function can be used in real-life problems. Let us take an example where the user will enter two values as input and these values will be passed to the user-defined function, which will do addition.

#include<stdio.h>

void Add(int, int);

void main() *// main function*

{

int x, y;

printf("\n Enter two integer values to add: \n");

scanf("%d %d",&x, &y);

*// dynamic values are called*

Add(x, y);

}

void Add(int x, int y)

{

int Sum;

Sum = x + y;

printf("\n Total sum of %d and %d is = %d \n", x, y, Sum);

}

**Output**:

Enter two integer values to add:

10

20

Total sum of 10 and 20 is = 30

**Explanation**:

Integer variables x and y are declared inside the main function. Then we called the "Add" with contains user-entered values. In the "Add(x,y)", there are integer variables of 'Sum' also integer (x,y) is present as arguments in this function. So, it will allow the user to pass two values as an 'integer'. Then, x and y are added using the + operator and 'printf' is used to print the result.

**c) Functions Without Arguments and With Return Values**

In this type of user-defined function, we do not pass any arguments when we define, declare or call a function. But, this type of function returns some value when it is called from the main() function or any submethod in the program. In it, the data type of return value is dependent on the return type of the declaration function. Suppose, if the return type is of 'int' then the return value will be also of int type. Let us see an example where we will write a program to multiply two integers without arguments and a return keyword.

#include<stdio.h>

int Multiply();

int main()

{

int Mul;

Mul = Multiply(); *// no argument passed*

printf("\n The multiplication of x and y is = %d \n", Mul );

return 0;

}

int Multiply()

{

int Mul, x = 5, y = 10;

Mul = x \* y;

return Mul;

}

**Output**:

The multiplication of x and y is = 50

In the above example, we can see there are no arguments passed in "Multiply". But it is returning the value as int type as "Mul".

**d) Functions Without Arguments and Without Return Values**

In this type of user-defined function, we do not pass any arguments when we define, declare or call a function. Also, this function does not return any values when it is called from the main() function or any submethod inside the program. This type of function is used when we do not expect any return value but we need some statement to print the result as output. Since this function has no arguments and returns a value, it does not receive any data when the function is called.

Let us see an example :

#include<stdio.h>

*// Declaration*

void Add();

void main()

{

Add();

}

void Add()

{

int Sum, x = 15, y = 50;

Sum = x + y;

printf("\n Total sum of x = %d and y = %d is = %d", x, y, Sum);

}

**Output**:

Total sum of x = 15 and y = 50 is = 65

11.Write a C program for Matrix Multiplication using 2D array.

#include<stdio.h>

#include<stdlib.h>

**int** main(){

**int** a[10][10],b[10][10],mul[10][10],r,c,i,j,k;

system("cls");

printf("enter the number of row=");

scanf("%d",&r);

printf("enter the number of column=");

scanf("%d",&c);

printf("enter the first matrix element=\n");

**for**(i=0;i<r;i++)

{

**for**(j=0;j<c;j++)

{

scanf("%d",&a[i][j]);

}

}

printf("enter the second matrix element=\n");

**for**(i=0;i<r;i++)

{

**for**(j=0;j<c;j++)

{

scanf("%d",&b[i][j]);

}

}

printf("multiply of the matrix=\n");

**for**(i=0;i<r;i++)

{

**for**(j=0;j<c;j++)

{

mul[i][j]=0;

**for**(k=0;k<c;k++)

{

mul[i][j]+=a[i][k]\*b[k][j];

}

}

}

//for printing result

**for**(i=0;i<r;i++)

{

**for**(j=0;j<c;j++)

{

printf("%d\t",mul[i][j]);

}

printf("\n");

}

**return** 0;

}

**Output:**

enter the number of row=3

enter the number of column=3

enter the first matrix element=

1 1 1

2 2 2

3 3 3

enter the second matrix element=

1 1 1

2 2 2

3 3 3

multiply of the matrix=

6 6 6

12 12 12

18 18 18

12.i) Differentiate between call by value and call by reference with examples.

| **Call By Value** | **Call By Reference** |
| --- | --- |
| While calling a function, we pass values of variables to it. Such functions are known as “Call By Values”. | While calling a function, instead of passing the values of variables, we pass address of variables(location of variables) to the function known as “Call By References. |
| In this method, the value of each variable in calling function is copied into corresponding dummy variables of the called function. | In this method, the address of actual variables in the calling function are copied into the dummy variables of the called function. |
| With this method, the changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function. | With this method, using addresses we would have an access to the actual variables and hence we would be able to manipulate them. |
| // C program to illustrate  // call by value  #include <stdio.h>  // Function Prototype  void swapx(int x, int y);  // Main function  int main()  {  int a = 10, b = 20;  // Pass by Values  swapx(a, b);  printf("a=%d b=%d\n", a, b);  return 0;  }  // Swap functions that swaps  // two values  void swapx(int x, int y)  {  int t;  t = x;  x = y;  y = t;  printf("x=%d y=%d\n", x, y);  }  **Output:**  **x=20 y=10**  **a=10 b=20** | // C program to illustrate  // Call by Reference  #include <stdio.h>  // Function Prototype  void swapx(int\*, int\*);  // Main function  int main()  {  int a = 10, b = 20;  // Pass reference  swapx(&a, &b);  printf("a=%d b=%d\n", a, b);  return 0;  }  // Function to swap two variables  // by references  void swapx(int\* x, int\* y)  {  int t;  t = \*x;  \*x = \*y;  \*y = t;  printf("x=%d y=%d\n", \*x, \*y);  }  **Output:**  **x=20 y=10**  **a=20 b=10** |
| Thus actual values of a and b remain unchanged even after exchanging the values of x and y. | Thus actual values of a and b get changed after exchanging values of x and y. |
| In call-by-values, we cannot alter the values of actual variables through function calls. | In call by reference we can alter the values of variables through function calls. |
| Values of variables are passed by the Simple technique. | Pointer variables are necessary to define to store the address values of variables. |

ii) Write a program using any 3 inbuilt string functions.

#include <stdio.h>

#include <string.h>

int main() {

char str1[20] = "Hello";

char str2[20] = "World";

// Get the length of a string

printf("Length of str1 = %d\n", strlen(str1));

// Copy a string

strcpy(str1, str2);

printf("After copying, str1 = %s\n", str1);

// Concatenate two strings

strcat(str1, "!");

printf("After concatenation, str1 = %s\n", str1);

return 0;

}

Output:

Length of str1 = 5

After copying, str1 = World

After concatenation, str1 = World!