4 Control Statements in C

C provides two types of Control Statements

- Branching Structure
- Looping Structure

4.1 Branching Structure

Branching is deciding what actions to take and looping is deciding how many times to take a certain action.Decision making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false. Show below is the general form of a typical decision making structure found in most of the programming languages.

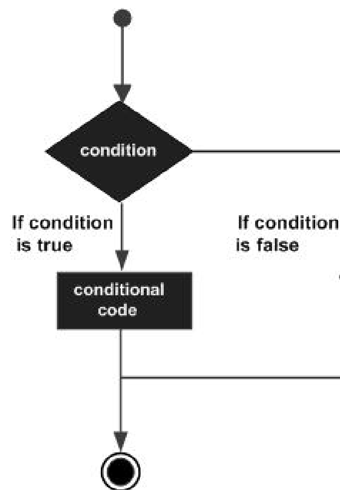

Figure 3: FlowChart Control Statement

4.1.1 if Statement

An if statement consists of a Boolean expression followed by one or more statements.If the Boolean expression evaluates to true, then the block of code inside the 'if' statement will be executed. If the Boolean expression evaluates to false, then the first set of code after the end of the 'if' statement (after the closing curly brace) will be executed. C programming language assumes any non-zero and non-null values as true and if it is either zero or null, then it is assumed as false value.

```
1 #include <stdio.h>
2 int main () {
3     int a = 10;
4     if( a < 20 ) {
5         printf("a is less than 20\n" );
6     }
7     printf("value of a is : %d\n", a);
8     return 0;
9 }
```


BCANOTESNEPAL.com

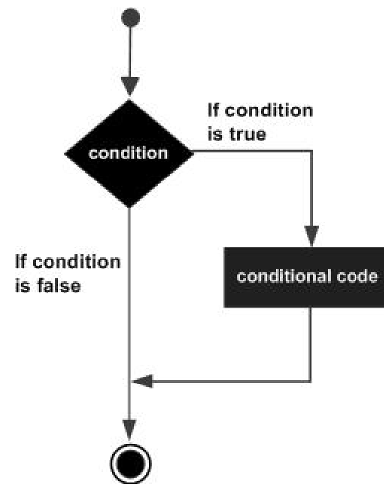

Figure 4: if statement flowchart

4.1.2 if..else Statement

An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.If the Boolean expression evaluates to true, then the if block will be executed,otherwise, the else block will be executed.C programming language assumes any non-zero and non-null values as true,and if it is either zero or null, then it is assumed as false value.

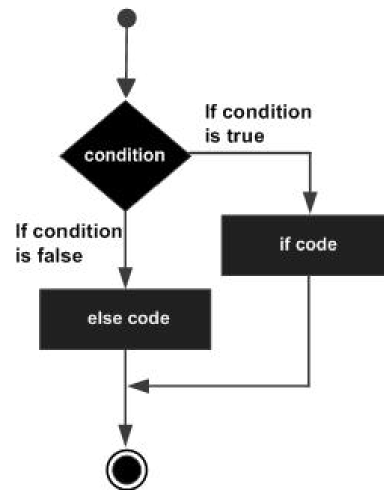

Figure 5: If else Statement

```
1 #include <stdio.h>
2 int main () {
3     int a = 100;
4     if( a == 10 ) {
5         printf("Value of a is 10\n" );
6     }
7     else if( a == 20 ) {
8         printf("Value of a is 20\n" );
9     }
10    else if( a == 30 ) {
11        printf("Value of a is 30\n" );
12    }
13    else {
```


BCANOTESNEPAL.com

```
14        printf("None of the values is matching\n" );
15    }
16    printf("Exact value of a is: %d\n", a );
17    return 0;
18 }
```

4.1.3 if..else if..else Statement

An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement.When using if...else if...else statements, there are few points to keep in mind:

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

4.1.4 Nested if Statement

It is always legal in C programming to nest if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

4.1.5 Switch Statement

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case.The following rules apply to a switch statement:

- The expression used in a switch statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

```
1 #include <stdio.h>
2 int main () {
3    /* local variable definition */
4    char grade = 'B';
5    switch(grade) {
```


BCANOTESNEPAL.com

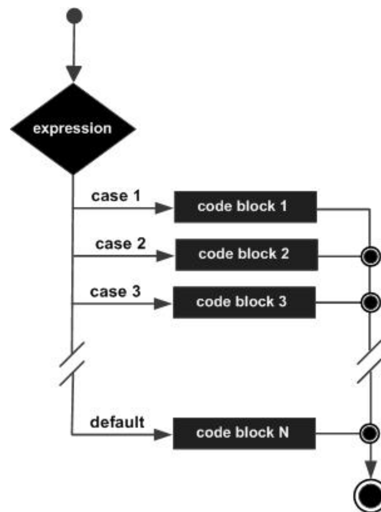

Figure 6: switch statement flowchart

```
6         case 'A' :
7             printf("Excellent!\n" );
8             break;
9         case 'B' :
10        case 'C' :
11            printf("Well done\n" );
12            break;
13        case 'D' :
14            printf("You passed\n" );
15            break;
16        case 'F' :
17            printf("Better try again\n" );
18            break;
19        default :
20            printf("Invalid grade\n" );
21    }
22    printf("Your grade is  %c\n", grade );
23    return 0;
24 }
```

4.1.6 The ? : Operator

Exp1? Exp2 : Exp3 is the general structure of the ? : operator.

The value of a ? expression is determined like this:

- Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression.
- If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

BCANOTESNEPAL.com

4.2 Looping Structure

Programming languages provide various control structures that allow for more complicated execution paths. A loop statement allows us to execute a statement or group of statements multiple times.C programming language provides the following types of loops to handle looping requirements.

- while loop
- do while loop
- for loop

4.2.1 while loop

A while loop in C programming repeatedly executes a target statement as long as a given condition is true.

```
1 while(condition)
2 {
3           statement(s);
4 }
```

Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is any nonzero value. The loop iterates while the condition is true. When the condition becomes false, the program control passes to the line immediately following the loop.Here, the key point to note is that a while loop might not execute at all. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed. flow chart and program remaining

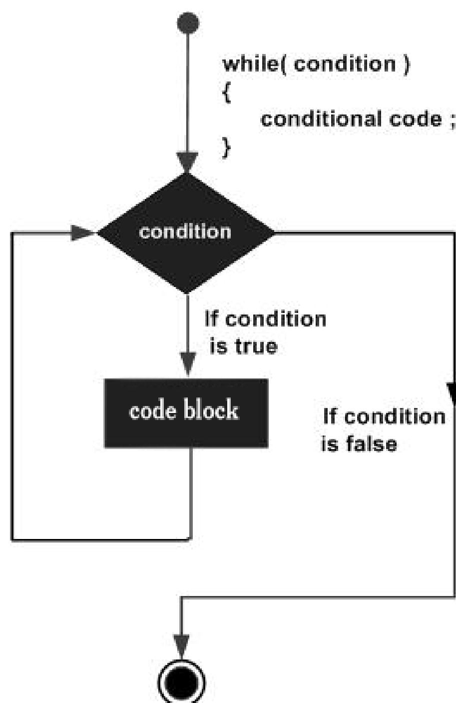

Figure 7: while loop flowchart

BCANOTESNEPAL.com

4.2.2 for loop

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

```
for ( init; condition; increment ) {
   statement(s);
}
```

1. The init step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
2. Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.
3. Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.
4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the 'for' loop terminates.

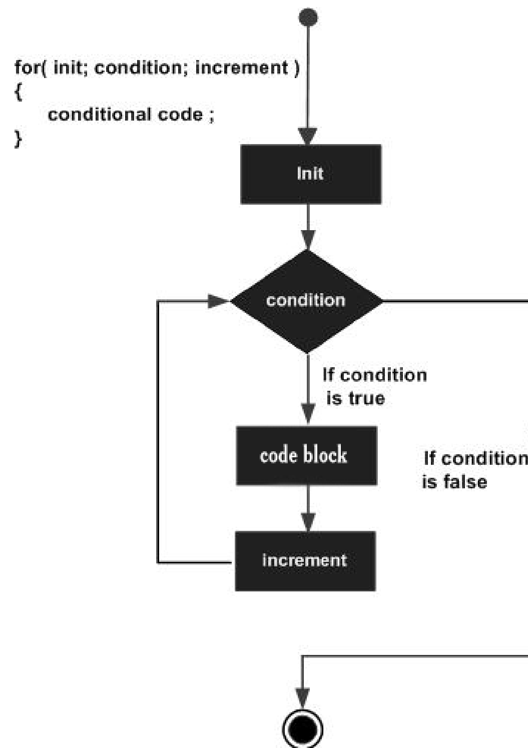

Figure 8: for loop flowchart

4.2.3 do while loop

Unlike for and while loops, which test the loop condition at the top of the loop, the do...while loop in C programming checks its condition at the bottom of the loop.A

BCANOTESNEPAL.com

do...while loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.

```
do {
        statement(s);
}while(condition);
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

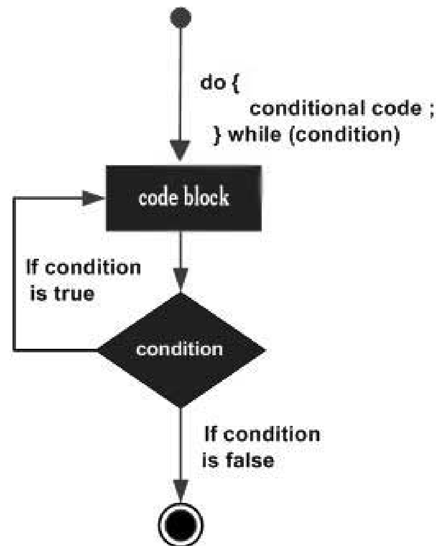

Figure 9: do..while loop flowchart

4.2.4 nested loop

C programming allows to use one loop inside another loop.

- Nested for loop

```
for ( init; condition; increment ) {
   for ( init; condition; increment ) {
      statement(s);
   }
   statement(s);
}
```

- Nested while loop

```
while(condition) {

   while(condition) {
      statement(s);
   }

   statement(s);
}
```

- Nested do while loop

BCANOTESNEPAL.com

```
do {

    statement(s);

    do {
        statement(s);
    }while( condition );

}while( condition );
```

- Find Prime numbers between 2 to 100 using prime number

```
#include <stdio.h>
int main () {
    /* local variable definition */
    int i, j;
    for(i = 2; i<100; i++) {
        for(j = 2; j <= (i/j); j++)
            if(!(i%j)) break; // if factor found, not prime
            if(j > (i/j)) printf("%d is prime\n", i);
    }
    return 0;
}
```

4.3 Break Continue and Goto Statement

4.3.1 Break Statement

The break statement terminates the loop (for, while and do...while loop) immediately when it is encountered. The break statement is used with decision making statement such as if...else. The syntax of the statement is
break;
Example of Goto Statement

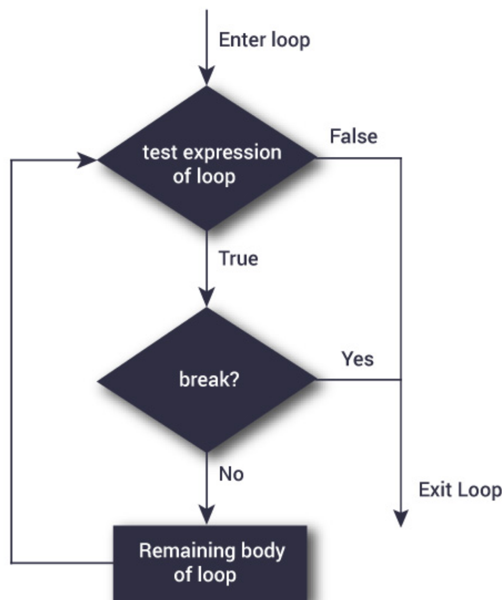

Figure 10: Flowchart Break Statement

BCANOTESNEPAL.com

```c
// Program to calculate the sum of maximum of 10 numbers
// Calculates sum until user enters positive number
# include <stdio.h>
int main()
{
int i;
double number, sum = 0.0;
for(i=1; i <= 10; ++i)
{
printf("Enter a n%d: ",i);
scanf("%lf",&number);
// If user enters negative number, loop is terminated
if(number < 0.0)
{
break;
}
sum += number; // sum = sum + number;
}
printf("Sum = %.2lf",sum);
return 0;
}
```

Working of Break Statement

```c
while (test Expression)
{
    // codes
    if (condition for break)
    {
        break;
    }
    // codes
}
```

```c
for (init, condition, update)
{
    // codes
    if (condition for break)
    {
        break;
    }
    // codes
}
```

Figure 11: Working of Break Statement

4.3.2 Continue Statement

The continue statement skips some statements inside the loop. The continue statement is used with decision making statement such as if...else. The syntax is
continue;

Example of Continue Statement

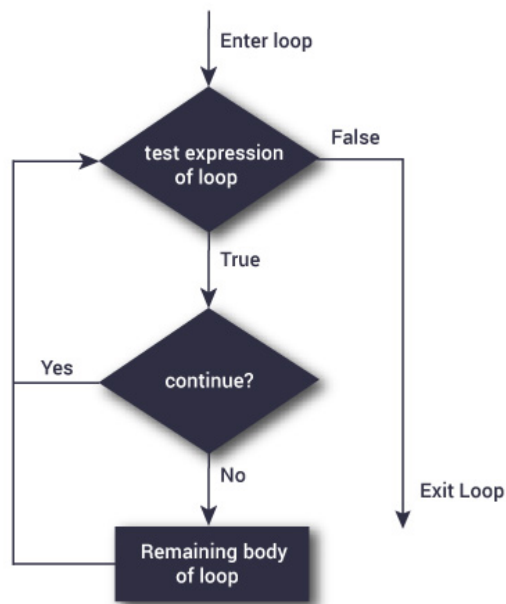

Figure 12: Flowchart Continue Statement

```
1 // Program to calculate sum of maximum of 10 numbers
2 // Negative numbers are skipped from calculation
3 # include <stdio.h>
4 int main()
5 {
6 int i;
7 double number, sum = 0.0;
8 for(i=1; i <= 10; ++i)
9 {
10 printf("Enter a n%d: ", i);
11 scanf("%lf",&number);
12 // If user enters negative number, loop is terminated
13 if(number < 0.0)
14 {
15 continue;
16 }
17 sum += number; // sum = sum + number;
18 }
19 printf("Sum = %.2lf",sum);
20 return 0;
21 }
```

Working of Continue Statement

BCANOTESNEPAL.com

```
while (test Expression)
{
    // codes
    if (condition for continue)
    {
        continue;
    }
    // codes
}
```

```
for (init, condition, update)
{
    // codes
    if (condition for continue)
    {
        continue;
    }
    // codes
}
```

Figure 13: Working of Continue Statement

4.3.3 Goto Statement

The goto statement is used to alter the normal sequence of a C program. The syntax of goto statement is

```
1 goto label;
2 .......
3 .......
4 .......
5 label:
6 statements;
```

Example of Goto Statement

```
1 // Program to calculate the sum and average of maximum of 5 numbers
2 # include <stdio.h>
3 int main()
4 {
5 const int maxInput = 5;
6 int i;
7 double number, average, sum=0.0;
8 for(i=1; i<=maxInput; ++i)
9 {
10 printf("%d. Enter a number: ", i);
11 scanf("%lf",&number);
12 // If user enters negative number, flow of program moves to label jump
13 if(number < 0.0)
14 goto jump;
15 sum += number; // sum = sum+number;
16 }
17 jump:
18 average=sum/(i-1);
19 printf("Sum = %.2f\n", sum);
```


BCANOTESNEPAL.com

```
20 printf("Average = %.2f", average);
21 return 0;
22 }
```

Reasons to avoid using goto statement

1. The use of goto statement may lead to code that is buggy and hard to follow.
2. Goto make jumps that are out of scope so it is difficult to follow.

BCANOTESNEPAL.com

5 Arrays and Strings

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

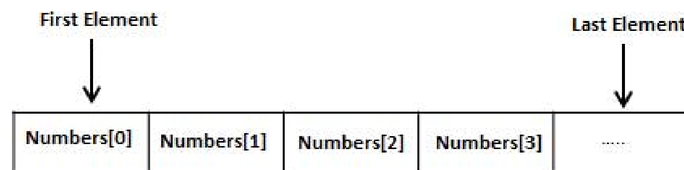

Figure 14: Arrays Representation

5.1 One Dimensional Array

5.1.1 Declaring and Initialising

```
1 data_type array_name[array_size];
2 double balance[10];
```

balance is a variable array which is sufficient to hold up to 10 double numbers.size of array defines the number of elements in an array. Each element of an array can be accessed and used as per the need of the program.

```
1 int age[5]={2,4,34,3,4};
2 int age[]={4,6,8,9,10};
```

In the second case the compiler determines the size of an array by calculating the number of elements in an array.

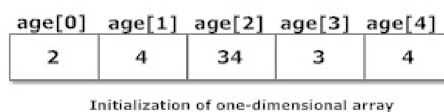

Figure 15: Arrays Initialisation

Program to find sum of marks of n students

```
1 #include <stdio.h>
2 int main(){
3     int marks[10],i,n,sum=0;
4     printf("Enter number of students: ");
5     scanf("%d",&n);
6     for(i=0;i<n;++i){
7         printf("Enter marks of student%d: ",i+1);
8         scanf("%d",&marks[i]);
9         sum+=marks[i];
10    }
11    printf("Sum= %d",sum);
12 return 0;
13 }
```


BCANOTESNEPAL.com

5.2 MultiDimensional Array

C programming language allows programmer to create arrays of arrays known as multidimensional arrays.

```
float a[2][6];
```

	col 1	col 2	col 3	col 4	col 5	col 6
row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]	a[0][5]
row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]	a[1][5]

Figure: Multidimensional Arrays

Figure 16: Arrays Initialisation

5.2.1 Initialisation of Multidimensional Arrays

```
int c[2][3]={{1,3,0},{-1,5,9}};
int c[2][3]={1,3,0,-1,5,9};
```

In the second case the compiler creates two rows from the given array.
Program to add two matrices

```
#include <stdio.h>
int main(){
    float a[2][2], b[2][2], c[2][2];
    int i,j;
    printf("Enter the elements of 1st matrix\n");

    for(i=0;i<2;++i)
        for(j=0;j<2;++j){
        printf("Enter a%d%d: ",i+1,j+1);
        scanf("%f",&a[i][j]);
        }

    printf("Enter the elements of 2nd matrix\n");
    for(i=0;i<2;++i)
        for(j=0;j<2;++j){
        printf("Enter b%d%d: ",i+1,j+1);
        scanf("%f",&b[i][j]);
        }

    for(i=0;i<2;++i)
        for(j=0;j<2;++j){
        c[i][j]=a[i][j]+b[i][j];
        }
    printf("\nSum Of Matrix:");
    for(i=0;i<2;++i)
        for(j=0;j<2;++j){
        printf("%.1f\t",c[i][j]);
            if(j==1)
                printf("\n");
       }
return 0;
}
```

Program to Multiply Two matrices

```
#include <stdio.h>
```

```
2 int main()
3 {
4   int m, n, p, q, c, d, k, sum = 0;
5   int first[10][10], second[10][10], multiply[10][10];
6 
7   printf("Enter rows and columns of first matrix\n");
8   scanf("%d%d", &m, &n);
9   printf("Enter the elements of first matrix\n");
10 
11   for (c = 0; c < m; c++)
12     for (d = 0; d < n; d++)
13       scanf("%d", &first[c][d]);
14 
15   printf("Enter rows and columns of second matrix\n");
16   scanf("%d%d", &p, &q);
17 
18   if (n != p)
19     printf("can't be multiplied with each other.\n");
20   else
21   {
22     printf("Enter the elements of second matrix\n");
23 
24     for (c = 0; c < p; c++)
25       for (d = 0; d < q; d++)
26         scanf("%d", &second[c][d]);
27 
28     for (c = 0; c < m; c++) {
29       for (d = 0; d < q; d++) {
30         for (k = 0; k < p; k++) {
31           sum = sum + first[c][k]*second[k][d];
32         }
33 
34         multiply[c][d] = sum;
35         sum = 0;
36       }
37     }
38 
39     printf("Product of entered matrices:-\n");
40 
41     for (c = 0; c < m; c++) {
42       for (d = 0; d < q; d++)
43         printf("%d\t", multiply[c][d]);
44 
45       printf("\n");
46     }
47   }
48 
49   return 0;
50 }
```

5.3 String

Strings are actually one-dimensional array of characters terminated by a null character. Thus a null-terminated string contains the characters that comprise the string followed by a null.The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
1 char greeting[6]="Hello";
2 char greeting[6]={'H','e','l','l','o','\0'};
```

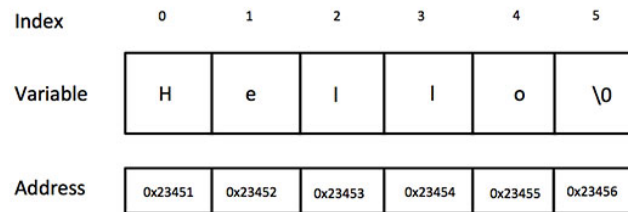

Figure 17: String Representation

5.3.1 Accessing Strings

```
1 //no need to use address & in scanf for strings
2 scanf("%s",greeting);
3 gets(greeting);
4
5 printf("%s",greeting);
6 puts(greeting);
```

5.3.2 String related fuctions

- strcpy(s1,s2) copies string s2 into the string s1.
- strcat(s1,s2) concatenates string s2 onto the end of string s1
- strlen(s1) returns the length of string s1
- strcmp(s1,s2) returns 0 if s1 and s2 are the same, less than 0 if s1<s2 and greater than 0 if s1>s2
- strrev(s1) reverses the string s1 and places it in s1

BCANOTESNEPAL.com

6 Function

6.1 Function Definition

A function is a group of statements that together perform a task. Every C program has at least one function, which is main(), and all the most trivial programs can define additional functions.A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

6.2 Types of C function

There are two types of C functions

- Library Function
- User Defined Function

6.2.1 Library Function

Library functions are the in-built function in C programming system.

```
main()
printf()
scanf()
strcpy()
strcat()
strcmp()
```

are the examples of Library functions available in C.

6.3 User Defined Function

C allows programmer to define their own function according to their requirement. These types of functions are known as user-defined functions.
Function Prototype(Declaration)
Every function in C programming should be declared before they are used. These type of declaration are also called function prototype. Function prototype gives compiler information about function name, type of arguments to be passed and return type.

```
int add(int a,int b);
```

6.4 Defining a Function

The general form of a function definition in C programming language is as follows

```
return_type function_name( parameter list ) {
   body of the function
}
```

A function definition in C Programming consists of a function header and a function body.Here are all parts of a function.

- `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.
- `Function Name` This is the actual name of the function. The function name and the parameter list together constitute the function signature.

- `Parameters` 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. Parameters are optional; that is, a function may contain no parameters.

- `Function Body` The function body contains a collection of statements that define what the function does.

6.5 Calling a function

Function with return and without return can be called to using the different syntax.

```
add(a,b); //calling function without return
c=add(a,b); //calling function with return
```

6.6 Call by Value

If data is passed by value, the data is copied from the variable used in for example main() to a variable used by the function. So if the data passed (that is stored in the function variable) is modified inside the function, the value is only changed in the variable used inside the function.

```
#include <stdio.h>
void call_by_value(int x) {
        printf("Inside function x = %d before adding 10.\n", x);
        x += 10;
        printf("Inside function x = %d after adding 10.\n", x);
}
int main() {
        int a=10;

        printf("a = %d before function.\n", a);
        call_by_value(a);
        printf("a = %d after function.\n", a);
        return 0;
}
```

In the main() we create a integer that has the value of 10. We print some information at every stage, beginning by printing our variable a. Then function call_by_value is called and we input the variable a. This variable (a) is then copied to the function variable x. In the function we add 10 to x (and also call some print statements). Then when the next statement is called in main() the value of variable a is printed. We can see that the value of variable a isn't changed by the call of the function call_by_value().

6.7 Call by Reference

If data is passed by reference, a pointer to the data is copied instead of the actual variable as is done in a call by value. Because a pointer is copied, if the value at that pointers address is changed in the function, the value is also changed in main().

```
#include <stdio.h>
void call_by_reference(int *y) {
        printf("Inside function y = %d before adding 10.\n", *y);
        (*y) += 10;
        printf("Inside function y = %d after adding 10.\n", *y);
}
```


BCANOTESNEPAL.com

```
7 int main() {
8         int b=10;
9         printf("b = %d before function.\n", b);
10        call_by_reference(&b);
11        printf("b = %d after function.\n", b);
12        return 0;
13 }
```

We start with an integer b that has the value 10. The function call_by_reference() is called and the address of the variable b is passed to this function. Inside the function there is some before and after print statement done and there is 10 added to the value at the memory pointed by y. Therefore at the end of the function the value is 20. Then in main() we again print the variable b and as you can see the value is changed (as expected) to 20.

6.8 Passing array as an argument to a function

Likewise int and floats an array can be passed as an argument to the function .

```
1 float largest(float a[],int size);
2 main(){
3  float value[4]={2.5,-1.6,3.4,6.8};
4  printf("%f\n",largest(value,4));
5 }
6 float largest(float a[],int size){
7         int i;
8         float max;
9         max=a[0];
10        for(i=1;i<size;i++){
11                if(max<a[i])
12                max=a[i];
13                return(max);
14        }
15 }
```

6.9 Storage Class in C

6.9.1 auto

The auto storage class is the default storage class for all local variables.

```
1 {
2    int mount;
3    auto int month;
4 }
```

The example above defines two variables with in the same storage class. 'auto' can only be used within functions, i.e., local variables.

6.9.2 register

The register storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
1 {
2    register int  miles;
3 }
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

6.9.3 static

The static storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.In C programming, when static is used on a class data member, it causes only one copy of that member to be shared by all the objects of its class.

```
1 #include <stdio.h>
2 void func(void);
3 static int count = 5; /* global variable */
4 main() {
5     while(count--) {
6         func();
7     }
8     return 0;
9 }
10 /* function definition */
11 void func( void ) {
12     static int i = 5; /* local static variable */
13     i++;
14     printf("i is %d and count is %d\n", i, count);
15 }
```

6.9.4 extern

The extern storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.When you have multiple files and you define a global variable or function, which will also be used in other files, then extern will be used in another file to provide the reference of defined variable or function. Just for understanding, extern is used to declare a global variable or function in another file.

6.10 Recursion

A function that calls itself is known as recursive function and this technique is known as recursion in C programming.Recursion is more elegant and requires few variables which make program clean. Recursion can be used to replace complex nesting code by dividing the problem into same problem of its sub-type.In other hand, it is hard to think the logic of a recursive function. It is also difficult to debug the code containing recursion.

6.10.1 Factorial Using Recursion

```
1 #include<stdio.h>
2 int factorial(int n);
3 int main()
4 {
```


BCANOTESNEPAL.com

```
5     int n;
6     printf("Enter an positive integer: ");
7     scanf("%d",&n);
8     printf("Factorial of %d = %ld", n, factorial(n));
9     return 0;
10 }
11 int factorial(int n)
12 {
13     if(n!=1)
14      return n*factorial(n-1);
15 }
```

6.10.2 Fibonacci Series Using Recursion

```
1 #include<stdio.h>
2 int Fibonacci(int);
3 main()
4 {
5    int n, i = 0, c;
6    scanf("%d",&n);
7    printf("Fibonacci series\n");
8    for ( c = 1 ; c <= n ; c++ )
9    {
10      printf("%d\n", Fibonacci(i));
11      i++;
12   }
13   return 0;
14 }
15 int Fibonacci(int n)
16 {
17   if ( n == 0 )
18      return 0;
19   else if ( n == 1 )
20      return 1;
21   else
22      return ( Fibonacci(n-1) + Fibonacci(n-2) );
23 }
```

6.11 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.

```
1 #include<stdio.h>
2 #define height 100
3 #define number 3.14
4 #define letter 'A'
5 #define letter_sequence "ABC"
6 #define backslash_char '\?'
7 void main()
8 {
9          printf("value of height: %d\n",height);
10         printf("value of number : %f\n",number);
11         printf("value of letter : %c\n",letter);
12         printf("value of letter_sequence : %s\n",letter_sequence);
13         printf("value of backslah_char : %c\n",backslash_char);
14         getch();
15
16 }
```

Preprocessor	Syntax	Description
Macro	#define	macro defines constant value and can be any of the basic data types
Header Inclusion Files	#include<file_name>	source code of the file file_name is included in the program at the specified place
Conditional Compilation	#ifedf, #endif,#if,#else	set of commands are included or excluded in source program before compilation with respect to the condition
Other directives	#undef,#pragma	#undef is used to undefine a defined macro variable.#pragma is used to call a function before and after function in C program.

Table 2: Preprocessor Directives

6.12 Macro Substitution

One of the major use of the preprocessor directives is the creation of macros.Macro Substitution is a process where an identifier in a program is replaced by a predefined string composed of one or more tokens.The general form of macro definition is

```
#define identifier_string
#define count 100
#define area 5*12.46
#define cube(x) (x*x*x)
```


BCANOTESNEPAL.com

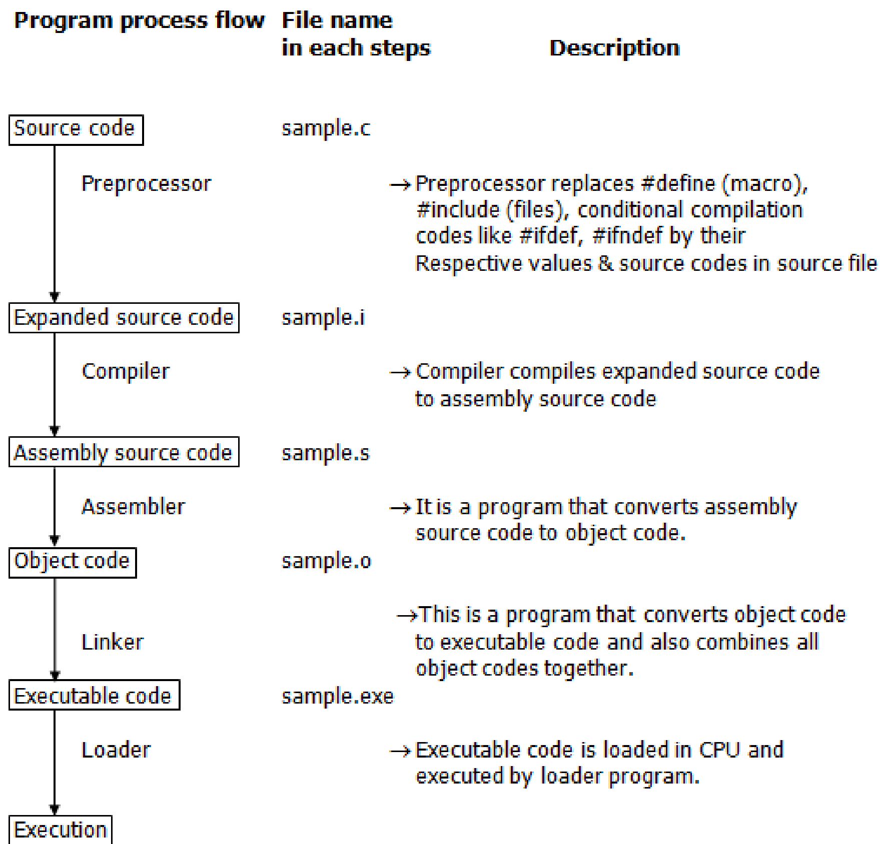

Figure 18: C program Compilation

7 Pointer

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is

```
1 type *var-name;
```

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations

```
1 int    *ip;    /* pointer to an integer */
2 double *dp;    /* pointer to a double */
3 float  *fp;    /* pointer to a float */
4 char   *ch     /* pointer to a character */
```

7.1 Using Pointers

```
1 #include <stdio.h>
2 int main () {
3
4    int  var = 20;   /* actual variable declaration */
5    int  *ip;        /* pointer variable declaration */
6    ip = &var;  /* store address of var in pointer variable*/
```

```
7
8     printf("Address of var variable: %x\n", &var  );
9     /* address stored in pointer variable */
10    printf("Address stored in ip variable: %x\n", ip );
11    /* access the value using the pointer */
12    printf("Value of *ip variable: %d\n", *ip );
13
14    return 0;
15 }
```

7.2 Null Pointer

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer.

```
1 #include <stdio.h>
2 int main () {
3
4     int  *ptr = NULL;
5     printf("The value of ptr is : %x\n", ptr  );
6     return 0;
7 }
```

7.3 Pointer Arithmetic

A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, –, +, and -.

```
1 ptr++
```

After the above operation, the ptr will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 2 bytes next to the current location in 16 bit system. This operation will move the pointer to the next memory location without impacting the actual value at the memory location. If ptr points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character will be available at 1001.

7.3.1 Incrementing Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer.

```
1 #include <stdio.h>
2 int main () {
3
4     int  var[3] = {10, 100, 200};
5     int  i, *ptr;
6
7     /* let us have array address in pointer */
8     ptr = var;
9
10    for ( i = 0; i < 3; i++) {
11        printf("Address of var[%d] = %x\n", i, ptr );
12        printf("Value of var[%d] = %d\n", i, *ptr );
13        /* move to the next location */
14        ptr++;
```


BCANOTESNEPAL.com

```
15     }
16     return 0;
17 }
```

7.3.2 Decrementing a Pointer

Likewise the increment of pointer the pointer variable can also be decremented which is shown as:

```
1 include <stdio.h>
2 int main () {
3
4     int  var[3] = {10, 100, 200};
5     int  i, *ptr;
6     /* let us have array address in pointer */
7     ptr = &var[2];
8     for ( i = 2; i >= 0; i--) {
9         printf("Address of var[%d] = %x\n", i, ptr );
10        printf("Value of var[%d] = %d\n", i, *ptr );
11        /* move to the previous location */
12        ptr--;
13    }
14    return 0;
15 }
```

7.4 Definition Pointer to Arrays

An array name is a constant pointer to the first element of the array. Therefore, in the declaration

```
1 double balance[50];
```

balance is a pointer to &balance[0], which is the address of the first element of the array balance. Thus, the following program fragment assigns p as the address of the first element of balance.

```
1 double *p;
2 double balance[10];
3 p = balance; //p=&balance[0];
```

It is legal to use array names as constant pointers, and vice versa. Therefore, *(balance + 4) is a legitimate way of accessing the data at balance[4].Once you store the address of the first element in 'p', you can access the array elements using *p, *(p+1), *(p+2) and so on.

```
1 #include <stdio.h>
2 int main () {
3
4     /* an array with 5 elements */
5     double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
6     double *p;
7     int i;
8     p = balance; //p=&balance[0]
9
10    /* output each array element's value */
11    printf( "Array values using pointer\n");
12    for ( i = 0; i < 5; i++ ) {
13        printf("*(p + %d) : %f\n",  i, *(p + i) );
14    }
```

```
15     printf( "Array values using balance as address\n");
16
17     for ( i = 0; i < 5; i++ ) {
18        printf("*(balance + %d) : %f\n",  i, *(balance + i) );
19     }
20     return 0;
21 }
```

```
1 main(){
2 int *p, sum,i;
3 int x[5]={5,9,6,7,3};
4 i=0;
5 p=x;    //p=&x[0];
6
7            while(i<5){
8                        printf("x[%d] %d %u",i,*p,p);
9                        sum=sum+*p;
10                       i++; p++;
11           }
12           printf("\n sum=%d\n",sum);
13 }
```

Here the base address of array x is assigned to the pointer p. sum adds the value of the int pointed by the pointer p using (*p) indirection operator and the increment i++ increases the value of the variable i by i and the increment p++ increase the value of the address pointed by p which is an integer and infact increased by 2 bytes.Ultimately the sum is calculated and printed.

7.5 Returning Multiple Values from a function

Return statement can return a single value. however multiple values can be returned from functions using arguments that we pass to a function.The arguments that are used to send out information are called output parameters.The mechanism of sending back information through arguments is achieved using what are known as the `address operator(&)` and `indirection operator (*)`.

```
1 void mathoperation(int x,int y,int *s,int *d);
2 main(){
3            int x=20,y=20,s,d;
4            mathoperation(x,y,&s,&d);
5            printf("sum=%d\n diff=%d",s,d);
6            }
7            void mathoperation(int a,int b,int *sum, int *diff){
8                        *sum=a+b;
9                        *diff=a-b;
10           }
11 }
```

The variables *sum and *diff are known as pointers and sum and diff as pointer variables.Since they are declared as the int, they can point to locations of int type data.

7.6 Pointer to String

As string can be considered as a character array. C supports an alternative way to create strings using pointer variables of type char.

```
1 char *str ="good";
```


BCANOTESNEPAL.com

This creates a string literal and then stores its address in the pointer variables str. The pointer now pointer to the first character of the string good.we can print the content of the str using printf or puts function

```
printf("%s",str);
puts(str);
```

Remember although str is a pointer to the string, it is also the name of the string. Therefore we do not need an indirection operator *.

```
main(){
 char *name;
 int length;
 char *cptr=name;
 name="DELHI"
 printf("%s\n",name);
 while(*cptr!='\0'){
         printf("%c is stored at address at %u\n",*cptr,cptr);
         cptr++;
 }
 length=cptr-name;
 printf("\n Length of String is =%d\n",name);
}
```

7.7 Double pointer

Stores the address of a pointer variable. Generally declared as

```
**ptr;
```

```
#include<stdio.h>
int main()
{

int num = 45 , *ptr , **ptr2ptr ;
ptr     = &num;
ptr2ptr = &ptr;
printf("%d",**ptr2ptr);
return(0);
}
```

7.8 Dynamic Memory Allocation

The exact size of array is unknown untill the compile time,i.e., time when a compier compiles code written in a programming language into a executable form. The size of array you have declared initially can be sometimes insufficient and sometimes more than required. Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

7.8.1 Malloc

The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form

```
ptr=(cast-type*)malloc(byte-size)
```


BCANOTESNEPAL.com

Function	Use of Function
malloc	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free	dellocate the previously allocated space
realloc	Change the size of previously allocated space

Table 3: Function Dynamic Memory Allocation

Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

```
1 ptr=(int*)malloc(100*sizeof(int));
```

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

7.8.2 Calloc

The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

```
1 ptr=(cast-type*)calloc(n,element-size);
```

This statement will allocate contiguous space in memory for an array of n elements.

```
1 ptr=(float*)calloc(25,sizeof(float));
```

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

7.8.3 free

Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

```
1 free(ptr);
```

This statement cause the space in memory pointer by ptr to be deallocated.
Find sum of n elements entered using malloc

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int n,i,*ptr,sum=0;
5     printf("Enter number of elements: ");
6     scanf("%d",&n);
7     ptr=(int*)malloc(n*sizeof(int));   //malloc allocation
8     if(ptr==NULL)
9     {
10        printf("Error! memory not allocated.");
11        exit(0);
12    }
13    printf("Enter elements of array: ");
14    for(i=0;i<n;++i)
```


BCANOTESNEPAL.com

```
15      {
16          scanf("%d",ptr+i);
17          sum+=*(ptr+i);
18      }
19      printf("Sum=%d",sum);
20      free(ptr);
21      return 0;
22  }
```

Find sum of n elements entered using malloc

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(){
4      int n,i,*ptr,sum=0;
5      printf("Enter number of elements: ");
6      scanf("%d",&n);
7      ptr=(int*)calloc(n,sizeof(int));
8      if(ptr==NULL)
9      {
10         printf("Error! memory not allocated.");
11         exit(0);
12     }
13     printf("Enter elements of array: ");
14     for(i=0;i<n;++i)
15     {
16         scanf("%d",ptr+i);
17         sum+=*(ptr+i);
18     }
19     printf("Sum=%d",sum);
20     free(ptr);
21     return 0;
22 }
```


BCANOTESNEPAL.com