**Unit-2**: Conditional Control -Statements :Simple if, if...else - Conditional Statements: else if and nested if - Conditional Statements: Switch case - Un-conditional Control Statements: break, continue, goto - Looping Control Statements:for, while, do..while - Looping Control Statements: nested for, nested while - Introduction to Arrays -One Dimensional (1D) Array Declaration and initialization - Accessing, Indexing and operations with 1D Arrays - Array Programs - 1D - Initializing and Accessing 2D Array, Array Programs -2D-Pointer and address-of operators -Pointer Declaration and dereferencing, Void Pointers, Null pointers Pointer based Array manipulation

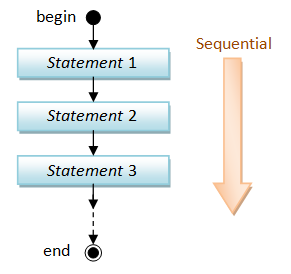
**CONTROL STATEMENTS**

Control Structures or control statements specifies the logical flow of control in programs. Types of control statements are:

* **Sequence Statements**
* **Selection or Conditional or Decision-Making Statements**
  + if, if-else, if-else if-else, switch-case
* **Iteration or Repetitive or Looping Statements** 
  + for loop, while loop, do-while loop
* **Unconditional Control Statements**
  + break, continue, goto

**Sequence Control Statements**

Sequence control structure refers to the sequential (line-by-line) execution of statements, in the same order in which they appear in the program.



**/\*Example: Sequential Control Construct\*/**

int main() {

int a, b, sum;

printf(“Enter the numbers”);

scanf(“%d%d”,&a, &b);

sum = a+b;

printf(“Sum of a and b = %d”, sum);

return 0;

}

**Conditional Control Statements**

* Simple if
* if … else
* else if
* nested if
* switch case

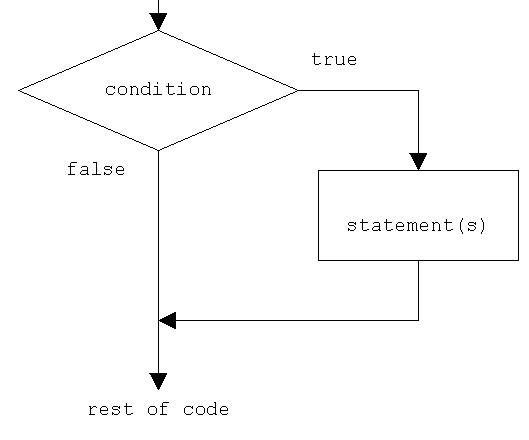
**The** if **Statement**

The C programming language provides a general decision-making capability in the formof a language construct known as the *if* statement. The general format of this statement is as follows:

**if (expression)**

***program statement***

The *if* statement is used to stipulate execution of a program statement (or statements if enclosed in braces) based upon specified conditions.



**/\* Absolute Number\*/**

#include <stdio.h>

int main() {

int number;

printf("Enter the number:");

scanf("%d",&number);

if(number < 0)

{

number = -number;

}

printf("\nAbsolute value is %d",number);

return 0;

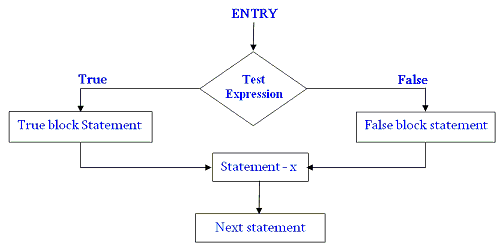
}

**Output:**

Enter the number:-75

Absolute value is 75

**The** if-else **Construct**



if (*expression*)

*program statement 1*

else

*program statement 2*

The **if-else** is actually just an extension of the general format of the **if** statement. Ifthe result of the evaluation of *expression* is TRUE, *program statement 1*, which immediately follows, is executed; otherwise, *program statement 2* is executed. In either case, either *program statement 1* or *program statement 2* is executed, but not both.

**// Program to determine if a number is even or odd**

#include <stdio.h>

int main ()

{

int number\_to\_test, remainder;

printf ("Enter your number to be tested: ");

scanf ("%d", &number\_to\_test);

remainder = number\_to\_test % 2;

if (remainder == 0)

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

else

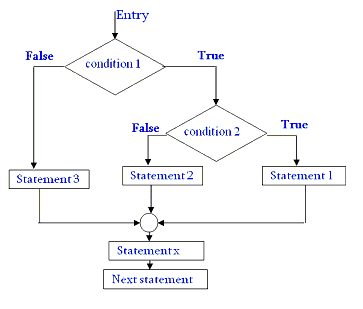
printf ("The number is odd.\n");

return 0;

}

**Nested** if **Statements**

It is usage of one if statement within another if statement.

****

**//Using nested if else statement**

#include <stdio.h>

#include <string.h>

int main()

{

char username[10];

int password;

printf("Username:");

scanf("%s",username);

printf("Password:");

scanf("%d",&password);

if(strcmp(username, "rahul")== 0)

{

if(password==12345)

printf("Login successful");

else

printf("Password is incorrect, Try again.");

}

else

printf("Username is incorrect, Try again.");

}

**The** else if **Construct**

The *else if* offers the most efficient implementation for processing mutually exclusive clauses. When one clause evaluates to TRUE, all subsequent clauses are ignored.

if (*expression 1*)

*program statement 1*

else if (*expression 2*)

*program statement 2*

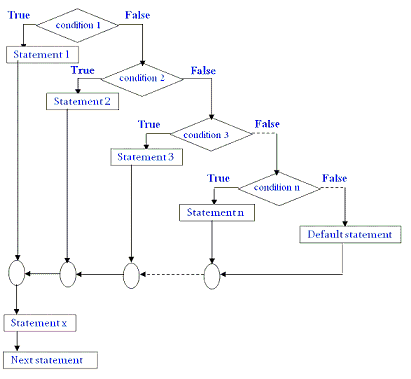
…

else if (expression n)

program statement n

else

*default program statement*

****

**// Program to display the grade of a student**

#include <stdio.h>

int main ()

{

int marks;

printf("Enter your marks between 0-100\n");

scanf("%d", &marks);

/\* Using if else ladder statement to print Grade of a Student \*/

if(marks >= 90){

/\* Marks between 90-100 \*/

printf("YOUR GRADE : A\n");

} else if (marks >= 70){

/\* Marks between 70-89 \*/

printf("YOUR GRADE : B\n");

} else if (marks >= 50){

/\* Marks between 50-69 \*/

printf("YOUR GRADE : C\n");

} else {

/\* Marks less than 50 \*/

printf("YOUR GRADE : Failed\n");

}

return 0;

}

**The** switch **Statement**

switch (*expression*)

{

case *value1*:

*program statements*

...

break;

case *value2*:

*program statements*

...

break;

...

case *valuen*:

*program statements*

...

break;

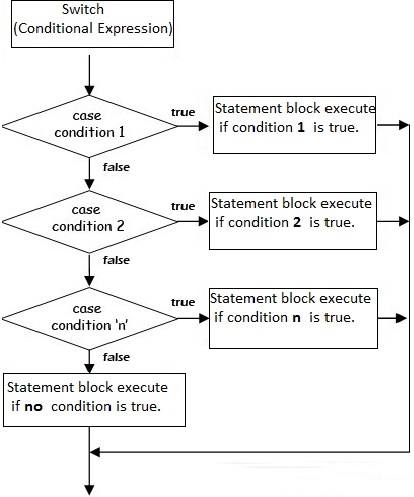
default:

*program statements*...

break;

}

* The *expression* enclosed within parentheses is successively compared against the values *value1*, *value2, ..., valuen*, which must be simple constants or constant expressions. It must be an integer or character.
* If a case is found whose value is equal to the value of *expression*, the program statements that follow the case are executed. Note that when more than one such program statement is included, they do *not* have to be enclosed within braces.
* The *break* statement signals the end of a particular case and causes execution of the switch statement to be terminated. Remember to include the *break* statement at the end of every case. Forgetting to do so for a particular case causes program execution to continue into the next case whenever that case gets executed.
* The special optional case called *default* is executed if the value of *expression* does not match any of the case values.



**/\* Program to evaluate simple expressions of the form value operator value \*/**

#include <stdio.h>

int main ()

{

float value1, value2;

char operator;

printf ("Type in your expression.\n");

scanf ("%f %c %f", &value1, &operator, &value2);

switch (operator)

{

case '+':

printf ("%.2f\n", value1 + value2);

break;

case '-':

printf ("%.2f\n", value1 - value2);

break;

case '\*':

printf ("%.2f\n", value1 \* value2);

break;

case '/':

if (value2 == 0)

printf ("Division by zero.\n");

else

printf ("%.2f\n", value1 / value2);

break;

default:

printf ("Unknown operator.\n");

break;

}

return 0;

}

* **Note:***else if* vs *switch*

| **else if** | **switch** |
| --- | --- |
| Both are used for selecting or deciding among mutually exclusive options. | |
| It tests expressions based on range of values or conditions. | It tests expressions which have discrete integer or character constant values. |
| It is more suitable when number of options are less. | It is more suitable when number of options are more. |

**Looping Statements**

The looping statements enable you to develop concise programs containing repetitive processes that could otherwise require thousands or even millions of program statements to perform. The C programming language contains three different program statements for program looping. They are known as the *for* statement, the *while* statement, and the *do while* statement.

**The *for* loop**

The general format of the for statement is as follows:

**for (*init\_expression; loop\_condition; loop\_expression* )**

***program statement***

* The first component of the for statement, labeled *init\_expression*, is used to set the initial values *before* the loop begins.
* The second component of the for statement the condition or conditions that are necessary *for* the loop to continue. In other words, looping continues *as long as* this condition is satisfied. When the *loop\_condition* is no longer satisfied, execution of the program continues with the program statement immediately following the for loop.
* The final component of the for statement contains an expression that is evaluated each time *after* the body of the loop is executed.

**/\* Factorial of a given number\*/**

#include <stdio.h>

int main() {

unsigned long long int fact = 1;

int num,i;

printf("Enter the number:");

scanf("%d",&num);

if(num <= 0) {

printf("\nFactorial of Negative Numbers cannot be found");

return 0;

}

for(i=1;i<num;i++) {

fact = fact\*i;

}

printf("\nFactorial of %d is %llu",num,fact);

return 0;

}

**The *while* statement**

while (*expression*)

*program statement*

The *expression* specified inside the parentheses is evaluated. If the result of the expression evaluation is TRUE, the *program statement* that immediately follows is executed. After execution of this statement (or statements if enclosed in braces), the *expression* is once again evaluated. If the result of the evaluation is TRUE, the program statement is once again executed. This process continues until the *expression* finally evaluates as FALSE, at which point the loop is terminated. Execution of the program then continues with the statement that follows the *program statement*.

**/\*Check whether the given number is palindrome or not\*/**

#include <stdio.h>

int main() {

int num, number, remainder, reverseOfNumber=0;

printf("Enter the number to check:");

scanf("%d",&num);

number = num;

while(number > 0) {

remainder = number % 10;

reverseOfNumber = reverseOfNumber\*10 + remainder;

number = number/10;

}

if(num == reverseOfNumber)

printf("The given number %d is a palindrome",num);

else

printf("The given number %d is not a palindrome",num);

return 0;

}

for (*init\_expression*; *loop\_condition*; *loop\_expression*)

*program statement*

can be equivalently expressed in the form of a while statement as

*init\_expression*;

while (*loop\_condition*) {

*program statement*

*loop\_expression;*

}

**The *do while* Construct**

When developing programs, it sometimesbecomes desirable to have the test made at the end of the loop rather than at the beginning. Naturally, the C language provides a special language construct to handle such a situation. This looping statement is known as the do statement. The syntax of this statement is as follows:

**do {**

**program statements**

**} while (loop\_expression);**

Execution of the do statement proceeds as follows: the *program statement* is executed first. Next, the *loop\_expression* inside the parentheses is evaluated. If the result of evaluating the *loop\_expression* is TRUE, the loop continues and the *program statement* is once again executed. As long as evaluation of the *loop\_expression* continues to be TRUE, the *program statement* is repeatedly executed. When evaluation of the expression proves FALSE, the loop is terminated, and the next statement in the program is executed in the normal sequential manner.

The do statement is simply a transposition of the while statement, with the looping conditions placed at the end of the loop rather than at the beginning.

Remember that, unlike the for and while loops, the do statement guarantees that the body of the loop is executed at least once.

**/\*Summation of 12+32+52+…+n2\* using Do-While\*/**

#include<stdio.h>

int main()

{

int i=1,sum=0,n;

printf("\nEnter the value of n:");

scanf("%d",&n);

do

{

sum=sum+(i\*i);

i+=2;

}

while(i<=n);

printf("Sum of the series=%d\n",sum);

return(0);

}

* **Note:**
* Avoid *do…while* and prefer *while* or *for* statements for looping capabilities. The do…while construct is avoided due to readability issues and to prevent errors.
* But there are cases when it is useful. For example, let’s say you want to write a loop where you are prompting the user for input and depending on input execute some code. You would want this to execute at least once and then ask the user whether he wants continue. In such cases, do…while loop results in lesser and cleaner code compared to a while loop.

**Unconditional Control Statements**

* The *break* statement
* The *continue* statement
* The *goto* statement

**The *break* Statement**

A *break* statement causes the innermost enclosing loop or switch to be exited immediately. It skips whatever follows it in the loop body. *breaks* can be useful because they are sometimes the simplest and best way to end a loop. But you might want to avoid using too many, because they can also make the code a little harder to read.

**/\*Example: Print from 0 to 5 using break\*/**

#include <stdio.h>

int main() {

int i;

for (i = 0; i < 10; i++) {

if (i == 6) {

break;

}

printf("%d\n", i);

}

return 0;

}

* **Note:** *breaks don’t break if statements*

**The *continue* Statement**

The continue statement causes the next iteration of the enclosing for, while, or do loop to begin. The continue statement in while and do loops takes the control to the loop's test-condition immediately, whereas in the for loop it takes the control to the increment step of the loop. The continue statement applies only to loops, not to switch.

***/\* Sum of all numbers except numbers divisible by 5 till 20\*/***

#include <stdio.h>

int main()

{

int i, sum = 0;

for (i = 1; i <= 20; i++)

{

if (i % 5 == 0)

continue;

sum += i;

}

printf("Sum = %d", sum);

}

**The *goto* Statement**

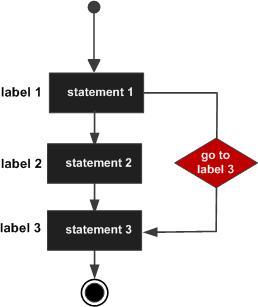
A *goto* statement provides an unconditional jump from the *goto* to a labeled statement in the same function.

goto label;

.. .

label: statement;

* **Note:** The use of *goto* statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify.

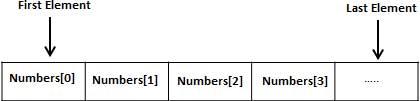


**ARRAYS**

An array is a collection of data items, all of the same type, accessed using a common name. A one-dimensional array is like a list. A two dimensional array is like a table. The C language places no limits on the number of dimensions in an array, though specific implementations may.

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.



* **Declaring Arrays**

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

**type arrayName [ arraySize ];**

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement:

**double balance[10];**

Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

* **Initializing Arrays**

One can assign initial values to the elements of an array. Values in the list are separated by commas and the entire list is enclosed in a pair of braces. The statement

**int integers[5] = { 0, 1, 2, 3, 4 };**

sets the value of integers[0] to 0, integers[1] to 1, integers[2] to 2, and so on.

It is not necessary to completely initialize an entire array. If fewer initial values are specified, only an equal number of elements are initialized. The remaining values in the array are set to zero. So the declaration

**float sample\_data[500] = { 100.0, 300.0, 500.5 };**

initializes the first three values of sample\_data to 100.0, 300.0, and 500.5, and sets the remaining 497 elements to zero.

* By enclosing an element number in a pair of brackets, specific array elements can be initialized in any order. For example,

**float sample\_data[500] = { [2] = 500.5, [1] = 300.0, [0] = 100.0 };**

initializes the sample\_data array to the same values as shown in the previous example.

* **Accessing Array Elements**

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

**double salary = balance[9];**

The above statement will take the 10th element from the array and assign the value to salary variable.

int main(){

int a[10] = {47, 22, 19, 29, 85, 72, 56, 67, 32, 25};

int sum, i;

for(i=0; i<10; i++) {

sum = sum + a[i];

}

printf(“Sum = %d”, sum);

}

* **Multidimensional Arrays**
* Multi-dimensional arrays are declared by providing more than one set of square [ ] brackets after the variable name in the declaration statement.
* One dimensional arrays do not require the dimension to be given if the array is to be completely initialized.  By analogy, multi-dimensional arrays do not require **the first** dimension to be given if the array is to be completely initialized.  All dimensions after the first must be given in any case.
* For two dimensional arrays, the first dimension is commonly considered to be the number of rows, and the second dimension the number of columns.

**/\*MATRIX MULTIPLICATION\*/**

#include <stdio.h>

int main()

{

int a[10][10],b[10][10],result[10][10];

int i,j,k,row1,col1,row2,col2,sum;

printf("\nEnter the size of the row and column of Matrix 1:\n");

scanf("%d%d",&row1,&col1);

printf("\nEnter the size of the row and column of Matrix 2:\n");

scanf("%d%d",&row2,&col2);

/\*Checking whether Matrix Multiplication can be performed or not \*/

if(col1 != row2) {

printf("Matrices with entered orders can't be multiplied with each other.\n");

return 0;

}

printf("\n\tREADING FIRST MATRIX\n");

for(i=0;i<row1;i++) {

printf("\n");

for(j=0;j<col1;j++)

{

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

}

}

printf("\n\tREADING SECOND MATRIX\n");

for(i=0;i<row2;i++) {

printf("\n");

for(j=0;j<col2;j++)

{

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

}

}

for(i=0;i<row1;i++) //row of first matrix

for(j=0;j<col2;j++) //column of second matrix

{

sum=0;

for(k=0;k<row2;k++)

{

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

result[i][j]=sum;

}

}

printf("\n\tRESULT MATRIX\n");

for(i=0;i<row1;i++,printf("\n"))

for(j=0;j<col2;j++)

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

return(0);

}

**POINTERS**

* A pointer is a variable which contains the address in memory of another variable.
* A variable is declared by giving it a type and a name (e.g. **int k;**)
* A pointer variable is declared by giving it a type and a name (e.g. **int \*ptr**) where the asterisk tells the compiler that the variable named **ptr** is a pointer variable and the type tells the compiler what type the pointer is to point to (integer in this case).
* **Address of Operator ‘&’:** Once a variable is declared, we can get its address by preceding its name with the unary**&** operator, as in **&k**.
* **Dereference Operator ‘\*’:** We can "dereference" a pointer, i.e. refer to the value of that which it points to, by using the unary '\*' operator as in **\*ptr**.
* A "lvalue" of a variable is the value of its address, i.e. where it is stored in memory. The "rvalue" of a variable is the value stored in that variable (at that address).
* **Example:**

int a = 108;

int \*ptr = &a; **//Pointer Declaration**

a ptr

6400 5400

printf(“%d”, a); // a=108

printf(“%p”, &a); // Address of a: &a=6400

printf(“%p”, ptr); // ptr=6400

printf(“%p”, &ptr); // Address of ptr: &ptr=5400

printf(“%d”, \*ptr); **// Dereferencing: \*ptr=108**

* *lvalue of a=6400 and rvalue of a=108*
* *lvalue of ptr=5400 and rvalue of ptr=6400*
* **Advantages of using Pointers**
* It saves memory space
* Execution time is made faster
* **Pointer Arithmetic**

Note: Here the size of *int* is assumed to be 2 bytes. But almost all the modern compliers support *int* of size 4 bytes.

| **Data Type** | **Initial Address** | **Address after** | | **Required Bytes** |
| --- | --- | --- | --- | --- |
| **Increment Operation (++)** | **Decrement Operation (--)** |
| int i=2 | 4000 | 4002 | 3998 | 2 |
| char c=’x’ | 5000 | 5001 | 4999 | 1 |
| float f=2.2 | 6000 | 6004 | 5996 | 4 |
| double d=2.2 | 7000 | 7008 | 6992 | 8 |

* **Pointers & Arrays**

| 11 | 22 | 33 | 44 | 55 | 66 | 77 | 88 | 99 | 110 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]

6000 6002 6004 6006 6008 6010 6012 6014 6016 6018

***Examples:***

int a[10] = {11,22,33,44,55,66,77,88,99,110};

int \*ptr = &a[3]; //6006

ptr++; //6008 or &a[4]

printf(“Current value is %u”,\*ptr); //55

ptr = a; //Equivalent to ptr = &a[0];

printf("\nValue1:%d",\*ptr+1); //12 or content of a[0]+1

printf("\nValue2:%d",\*(ptr+1));

//content of next location or 22 or content of a[1]

printf("\nValue3:%d", ptr[3]); //content of a[3] or 44

printf("\nValue4:%d",\*(ptr+3)); // content of a[3] or 44

* **Null Pointer**

A pointer can be assigned the value 0 to explicitly represent that it does not currently have a pointee. Having a standard representation for "no current pointee" turns out to be very handy when using pointers. The constant NULL is defined to be 0 and is typically used when setting a pointer to NULL. Since it is just 0, a NULL pointer will behave like a boolean false when used in a boolean context. Dereferencing a NULL pointer is an error which, if you are lucky, the computer will detect at runtime -- whether the computer detects this depends on the operating system.

**Syntax**

int \*ptr = NULL;

int \*ptr = 0

**Applications of Null Pointer in C**

* Initialize pointers.
* Represent conditions such as the end of a list of unknown length.
* Indicate errors in returning a pointer from a function.
* **Void Pointer**

A void pointer is a pointer that has no associated data type with it. A void pointer can hold an address of any type and can be typecasted to any type.

int a = 10;

char b = 'x';

void \*p = &a; // void pointer holds address of int 'a'

p = &b; // void pointer holds address of char 'b'

* **Advantage of void pointers:** malloc() and calloc() return void \* type and this allows these functions to be used to allocate memory of any data type (just because of void \*).

int main(void)

{

// Note that malloc() returns void \* which can be

// typecasted to any type like int \*, char \*, ..

int \*x = malloc(sizeof(int) \* n);

}

* void pointers cannot be dereferenced. For example, the following program doesn’t compile.

int main()

{

int a = 10;

void \*ptr = &a;

printf("%d", \*ptr);

return 0;

}

Output:

Compiler Error: 'void\*' is not a pointer-to-object type

The following program compiles and runs fine.

int main()

{

int a = 10;

void \*ptr = &a;

printf("%d", \*(int \*)ptr);

return 0;

}

Output:

10

* The C standard doesn’t allow pointer arithmetic with void pointers.