**Introduction C program**

**What is c?**

* + C is a programming language developed at AT & T’s Bell Laboratories of USA in 1972 by Dennis Ritchie.
  + Any programming Language can be divided in to two categories.
    - Problem oriented (High level language)
    - Machine oriented (Low level language)
  + C is modular, portable, reusable

Feature of C Program:

* + Structured language
  + General purpose language
  + **Portability**
  + **Code Re-usability & Ability to customize**
  + **Limited Number of Key Word**

**C program Structure:**

**Documentation**-Consists of comments, some description of the program, programmer name and any other useful points that can be referenced later.

**Link**-Provides instruction to the compiler to link function from the library function.

**Definition**-Consists of symbolic constants.

**Global declaration**-Consists of function declaration and global variables.

**Main()-**Every C program must have a main() function which is the starting point of the program execution.

**Subprograms**-User defined functions**.**

**Example program:**

/\*sample\*/

#include<stdio.h>

#include<conio.h>

Void main()

{

Clrscr();

Printf(“Hello world”)

getch();

}

**Tokens in C**

C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol.

**Semicolons**

In a C program, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.

**Comments**

Comments are like helping text in your C program and they are ignored by the compiler. They start with /\*

**/\*sample program\*/**

**Identifiers**

C identifier is a name used to identify a variable, function, or any other userdefined item. An identifier starts with a letter A to Z, a to z, or an underscore ‘\_’ followed by zero or more letters, underscores, and digits (0 to 9). C does not allow punctuation characters such as @, $, and % within identifiers. C is a case-sensitive programming language. Thus, Manpower and manpower are two different identifiers in C.

**C Datatypes:**

Data types in c refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

Two types datatypes:

* Primary datatypes
* Secondary datatypes

**Primary datatypes:**

* + Int
  + Float
  + Double
  + Char
  + Void

**Secondary datatypes:**

* + Array
  + Pointer
  + Structure
  + Union
  + Enum

**Integer types:**

|  |  |  |
| --- | --- | --- |
| **Type** | **Storage size** | **Value range** |
| Char | 1 byte | -128 to 127 or 0 to 255 |
| unsigned char | 1 byte | 0 to 255 |
| signed char | 1 byte | -128 to 127 |
| Int | 2 or 4 bytes | -32,768 to 32,767 or -2,147,483,648 to 2,147,483,647 |
| unsigned int | 2 or 4 bytes | 0 to 65,535 or 0 to 4,294,967,295 |
| Short | 2 bytes | -32,768 to 32,767 |
| unsigned short | 2 bytes | 0 to 65,535 |
| Long | 8 bytes | -9223372036854775808 to 9223372036854775807 |
| unsigned long | 8 bytes | 0 to 18446744073709551615 |

**Floating point types:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Storage size** | **Value range** | **Precision** |
| Float | 4 byte | 1.2E-38 to 3.4E+38 | 6 decimal places |
| Double | 8 byte | 2.3E-308 to 1.7E+308 | 15 decimal places |
| long double | 10 byte | 3.4E-4932 to 1.1E+4932 | 19 decimal places |

**Variables:**

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

**Variable definition in C:**

A variable definition tells the compiler where and how much storage to create for the variable.

**Syntax:** Type variable\_list;

**Example:**

**int a,b,c;**

**char ch,str;**

**float d,e;**

**double n;**

**Example program:**

#include<stdio.h>

#include<conio.h>

void main()

{

int a;

float b;

char str[50];

clrscr();

printf("Enter the integer:");

scanf("%d",&a);

printf("Enter the float value:");

scanf("%f",&b);

printf("Enter the string:");

scanf("%s",&str);

printf("integer = %d\n",a);

printf("float value = %f\n",b);

printf("string value = %s\n",str);

getch();

}

**Defining Constant:**

There are two simple ways in C to define constants,

* Using **#define** preprocessor.
* Using **const** keyword.

**Syntax:** #define identifier value;

**Example program:**

#include <stdio.h>

#define LENGTH 10

#define WIDTH 5

void main()

{

int area;

clrscr();

area = LENGTH \* WIDTH;

printf("value of area : %d", area);

getch();

}

**Using Constant(const) keyword:**

You can use **const** prefix to declare constants with a specific type as follows,

**Syntax:** const type variable=value;

**Example program:**

#include <stdio.h>

#include<conio.h>

void main()

{

const int LENGTH=10

#define WIDTH=5

int area;

clrscr();

area = LENGTH \* WIDTH;

printf("value of area : %d", area);

getch();

}

**Storage class in c:**

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. They precede the type that they modify. We have four different storage classes in a C program −

* auto
* register
* static
* extern

**Auto:**

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

**Syntax:** auto datatype variable\_name;

**Register:**

The **register** storage class is used to define local variables that should be stored in a register instead of RAM.

**Syntax:** register datatype variable\_name;

**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.

**Syntax:**static datatype variable\_name

**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.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions.

**Syntax:** extern void function();

**Operators in C:**

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators −

* Arithmetic Operators
* Relational Operators
* Logical Operators
* Bitwise Operators
* Assignment Operators
* Misc Operators

**Arithmetic Operators: A=10,B=20**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Adds two operands. | A + B = 30 |
| − | Subtracts second operand from the first. | A − B = -10 |
| \* | Multiplies both operands. | A \* B = 200 |
| / | Divides numerator by de-numerator. | B / A = 2 |
| % | Modulus Operator and remainder of after an integer division. | B % A = 0 |
| ++ | Increment operator increases the integer value by one. | A++ = 11 |
| -- | Decrement operator decreases the integer value by one. | A-- = 9 |

**Example program:**

#include<stdio.h>

#include<conio.h>

void main()

{

int a,b,c;

clrscr();

printf("Enter the value for a and b:");

scanf("%d,%d",&a,&b);

c=a+b;

printf("add is %d\n",c);

getch();

}

**Logical Operators**

Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Called Logical AND operator. If both the operands are non-zero, then the condition becomes true. | (A && B) is false. |
| || | Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true. | (A || B) is true. |
| ! | Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false. | !(A && B) is true. |

**Assignment operator:**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment operator. Assigns values from right side operands to left side operand | C = A + B will assign the value of A + B to C |
| += | Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand. | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand. | C -= A is equivalent to C = C - A |
| \*= | Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand. | C \*= A is equivalent to C = C \* A |
| /= | Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand. | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand. | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator. | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator. | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator. | C &= 2 is same as C = C & 2 |
| ^= | Bitwise exclusive OR and assignment operator. | C ^= 2 is same as C = C ^ 2 |
| |= | Bitwise inclusive OR and assignment operator. | C |= 2 is same as  C = C | 2 |

**Misc Operators ↦ sizeof & ternary**

Besides the operators discussed above, there are a few other important operators including **sizeof** and **? :** supported by the C Language.

[Show Examples](https://www.tutorialspoint.com/cprogramming/c_sizeof_operator.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| sizeof() | Returns the size of a variable. | sizeof(a), where a is integer, will return 4. |
| & | Returns the address of a variable. | &a; returns the actual address of the variable. |
| \* | Pointer to a variable. | \*a; |
| ? : | Conditional Expression. | If Condition is true ? then value X : otherwise value Y |

**Operators Precedence in C**

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has a higher precedence than the addition operator.

For example, x = 7 + 3 \* 2; here, x is assigned 13, not 20 because operator \* has a higher precedence than +, so it first gets multiplied with 3\*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

[Show Examples](https://www.tutorialspoint.com/cprogramming/c_operators_precedence.htm)

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | () [] -> . ++ - - | Left to right |
| Unary | + - ! ~ ++ - - (type)\* & sizeof | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Shift | << >> | Left to right |
| Relational | < <= > >= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Left to right |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %=>>= <<= &= ^= |= | Right to left |
| Comma | , | Left to right |

**Conditional Statements:**

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.



**Types of decision making statements:**

* If
* If else
* If else ladder
* Nested if else
* Switch statement
* Nested switch statement

**If condition:**

**Syntax:**

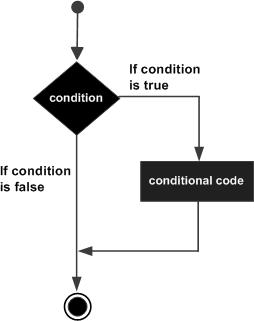
if(condition)

{

Statement;

}

**Flow chart:**



**Example program:**

#include<stdio.h>

#include<conio.h>

void main()

{

int n;

clrscr();

printf("Enter the number:");

scanf("%d",&n);

if(n>0)

{

printf("the number is positive");

}

getch();

}

**If else..condtion statement:**

**Syntax:**

if (condition)

{

Statement1

}

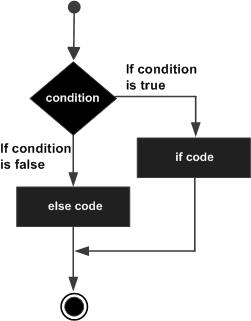
else

{

Statement2

}

**Flow chart:**



**Example program:**

#include<stdio.h>

#include<conio.h>

void main()

{

int a,b;

clrscr();

printf("enter the value for a,b:");

scanf("%d,%d",&a,&b);

if(a>b)

{

printf("a is big");

}

else

{

printf("b is big");

}

getch();

}

**If else ladder:**

**Syntax:**

if(condition1)

{

Statement1;

}

else if(condition2)

{

Statement2;

}

else if(condition3)

{

Statement3;

}

else

{

Statement4;

}

**Example program:**

#include<stdio.h>

#include<conio.h>

void main()

{

int a,b,c;

clrscr();

printf("Enter the value for a,b,c:");

scanf("%d,%d,%d",&a,&b,&c);

if((a==b)&&(a==c))

{

printf("three numbers are equal");

}

else if((a>b)&&(a>c))

{

printf("a is big:");

}

else if((b>a)&&(b>c))

{

printf("b is big");

}

else

{

printf("c is big");

}

getch();

}

**Nested if:**

**Syntax:**

if(condition)

{

if(condition)

{

Statement;

}

}

**Example program:**

#include<stdio.h>

#include<conio.h>

void main() {

int a = 100;

int b = 200;

if( a == 100 ) {

if( b == 200 ) {

printf("Value of a is 100 and b is 200\n" );

}

}

printf("Exact value of a is : %d\n", a );

printf("Exact value of b is : %d\n", b );

getch();

}

**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.

**Syntax:**

switch(expression) {

case constant-expression :

statement(s);

break;

case constant-expression :

statement(s);

break;

default : /\* Optional \*/

statement(s);

}

**Flow Chart:**



**Example program:**

#include<stdio.h>

#include<conio.h>

void main() {

char op;

int a,b;

printf("Enter an operator (+, -, \*,/): ");

scanf("%c", &op);

printf("Enter two operands: ");

scanf("%d %d",&a, &b);

switch(operator)

{

case '+':

printf("%d + %d = %d",a,b,a+b);

break;

case '-':

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

break;

case '\*':

printf(%d \* %d = %d",a,b,a\*b);

break;

case '/':

printf(%d / %d = %d",a,b,a/b);

break;

default:

printf("Error! operator is not correct");

}

getch();

}

**Nested switch:**

It is possible to have a switch as a part of the statement sequence of an outer switch. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise.

**Syntax:**

switch(ch1) {

case 'A':

printf("This A is part of outer switch" );

switch(ch2) {

case 'A':

printf("This A is part of inner switch" );

break;

case 'B':

}

break;

case 'B':

}

**Example program:**

#include <stdio.h>

#include<conio.h>

void main () {

int a = 100;

int b = 200;

switch(a) {

case 100:

printf("This is part of outer switch\n", a );

switch(b) {

case 200:

printf("This is part of inner switch\n", a );

}

}

printf("Exact value of a is : %d\n", a );

printf("Exact value of b is : %d\n", b );

getch();

}

**Looping statements:**

when a block of code needs to be executed several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

**Flow chart:**



**Types of looping statements:**

* + - For
    - While
    - Do while
    - Nested loops

**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.

**Syntax:**

for ( init; condition; increment ) {

statement(s);

}

**Flow chart:**



**Example program:**

#include<stdio.h>

#include<conio.h>

void main()

{

int n,i;

clrscr();

printf("Enter the number:");

scanf("%d",&n);

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

{

printf("values are %d",i);

}

getch();

}

**While loop:**

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

**Flow chart:**

****

Syntax:

while(condition) {

statement(s);

}

**Example Program:**

#include<stdio.h>

#include<conio.h>

void main()

{

int count=0,n;

clrscr();

printf("Enter the number:")

while(count>=n)

{

count++;

}

getch();

}

**Do While:**

 do...while loop in C programming checks its condition at the bottom of the loop

**Syntax:**

do{

statement;

}while(condition);

**Flow chart:**



**Example program:**

#include<stdio.h>

#include<conio.h>

void main()

{

int j=0,n;

clrscr();

printf("Enter the number:");

scanf("%d",&n);

do{

printf("value of variable j is:%d\n",j);

j++;

}while(j<=n);

getch();

}

**Nested loops:**

**Nested for loop:**

**Syntax:**

for ( init; condition; increment ) {

for ( init; condition; increment ) {

statement(s);

}

statement(s);

}

**Nested while loop:**

**Syntax:**

while(condition) {

while(condition) {

statement(s);

}

statement(s);

}

**Nested do..While:**

**Syntax:**

do {

statement(s);

do {

statement(s);

}while( condition );

}while( condition );

**Example program:**

#include <stdio.h>

#include<conio.h>

void main ()

{

int i, j;

for(i = 2; i<100; i++)

{

for(j = 2; j <= (i/j); j++)

if(!(i%j))

break;

if(j > (i/j)) printf("%d is prime\n", i);

}

getch();

}

**Loop condition statements:**

* + **Break**
  + **Continue**
  + **Goto**

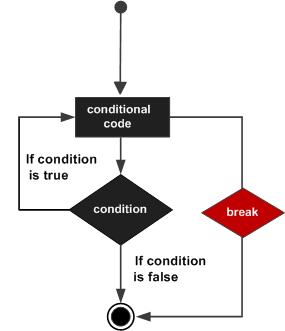
**Break:**

The **break** statement in C programming has the following two usages −

* When a **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
* It can be used to terminate a case in the **switch** statement.

**Syntax:** break;

**Flow chart:**



**Example program:**

#include <stdio.h>

#include<conio.h>

void main () {

int a = 10;

while( a < 20 )

{

printf("value of a: %d\n", a);

a++;

if( a > 15) {

break;

}

}

getch();

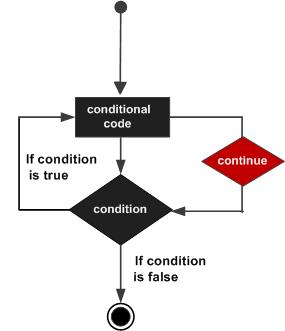
}

**Continue statement:**

The continue statement in C programming works somewhat like the breakstatement. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between.

**Syntax:** continue;

**Flowchart:**



**Example program:**

#include <stdio.h>

#include<conio.h>

void main () {

int a = 10;

do {

if( a == 15) {

a = a + 1;

continue;

}

printf("value of a: %d\n", a);

a++;

} while( a < 20 );

getch();

}

**Go to statement:**

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

**Syntax:**

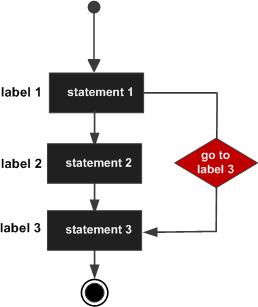
goto label;

..

.

label: statement;

**flow chart:**



**Example program:**

#include <stdio.h>

#include<conio.h>

void main () {

int a = 10;

LOOP:do {

if( a == 15) {

a = a + 1;

goto LOOP;

}

printf("value of a: %d\n", a);

a++;

} while( a < 20 );

getch();

}

**Functions in c program:**

**Functions:**

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.

**Syntax:**

return\_type function\_name( parameter list ) {

body of the function

}

**Example program:**

#include<stdio.h>

#include<conio.h>

int add();

void main()

{

int answer;

answer = add();

printf("The addition of the two numbers is: %d\n",answer);

getch();

}

int add()

{

int num1 = 10, num2 = 5;

return num1+num2;

}

Function call can be divided into two type:

* + **Call by value**
  + **Call by reference**

**Call by value:**

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

**Example:**

#include<stdio.h>

#include<conio.h>

int swap( int x, int y );

void main()

{

int m, n ;

clrscr();

printf ( "\nEnter the number:\n");

scanf ( "%d %d",&m,&n) ;

add( m,n ) ;

printf ( "\nSwapping of the given number %d is %d",m,n );

}

int swap(int x, int y )

{

int temp;

temp=x;

x=y;

y=temp;

return ;

}

**Call by reference:**

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

**Example program:**

#include<stdio.h>

#include<conio.h>

int swap( int \*x, int \*y );

void main()

{

int m, n ;

clrscr();

printf ( "\nEnter the number:\n");

scanf ( "%d %d",&m,&n) ;

add( &m,&n ) ;

printf ( "\nSwapping of the given number %d is %d",m,n );

}

int swap(int \*x, int \*y )

{

int temp;

temp=\*x;

\*x=\*y;

\*y=temp;

return ;

}

**Array concept**

**Array:**

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.

**First element Last element**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Number[0]** | **Number[1]** | **Number[2]** | **Number[3]** | **……..** |

**Array declaration:**

**Syntax**: datatype arrayname[array\_size];

**Array initialization:**

**Syntax:**datatype array\_name[array\_size]={array\_element1,array\_element2,…..};

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Element1 | Element2 | Element3 | Element4 | ………. |

[0] [1] [2] [3] [n]

**Example:(one dimensional array)**

#include<stdio.h>

#include<conio.h>

void main()

{

int i,num[5],sum;

clrscr();

for(i=0;i<=5;i++)

{

printf("Enter the array elements:");

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

}

for(i=0;i<=5;i++)

{

sum=sum+num[i];

}

printf("sum of the array elements are:%d",sum);

getch();

}

**Two dimensional array:**

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y].

**Syntax:** datatype arrayname[size1][size2];



**Example:**

#include<stdio.h>

#include<conio.h>

void main(){

int a[2][3];

int i, j;

clrscr();

for(i=0; i<2; i++) {

for(j=0;j<3;j++) {

printf("Enter value for a[%d][%d]:", i, j);

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

}

}

printf("Two Dimensional array elements:\n");

for(i=0; i<2; i++) {

for(j=0;j<3;j++) {

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

if(j==2){

printf("\n");

}

}

}

getch();

}

**Passing array as function arguments:**

passing a array as function arguments in 3 ways,

* Formal parameters as a pointer
* Formal parameters as a sized array
* Formal parameters as an unsized array

**Formal parameters as a pointer:**

**Syntax:** void myFunction(int \*variable\_name) {

.

.

}

**Formal parameter as size array:**

**Syntax:** void myFunction(int variable\_name[size]){

.

.

}

**Formal parameter as unsized array:**

**Syntax:** void myFunction(int variable\_name[]){

.

.

}

**Pointer to array:**

**Syntax:** datatype \*variable\_name;

datatypevariable\_name[size];

**Example:**

#include<stdio.h>

#include<conio.h>

void main() {

double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};

double \*p;

int i;

p = balance;

printf( "Array values using pointer\n");

for ( i = 0; i < 5; i++ ) {

printf("\*(p + %d) : %f\n", i, \*(p + i) );

}

printf( "Array values using balance as address\n");

for ( i = 0; i < 5; i++ ) {

printf("\*(balance + %d) : %f\n", i, \*(balance + i) );

}

getch();

}

**String**

Strings are actually one-dimensional array of characters terminated by a **null**character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

**String declaration:**

**Example: “c string”**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **C** |  | **s** | **T** | **r** | **i** | **n** | **g** | **\0** |

**Syntax:** char variable\_name[size];

S[0] s[1] s[2] s[3] s[4]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

**String initialization:**

**Syntax:**char variable\_name[]=”string\_name”;

char c[] = "abcd";

char c[50] = "abcd";

char c[] = {'a', 'b', 'c', 'd', '\0'};

char c[5] = {'a', 'b', 'c', 'd', '\0'};

**Example program:**

#include<stdio.h>

#include<conio.h>

void \_main()

{

int str[50];

clrscr();

printf("Enter the name:")

scanf("%s",&str);

printf("string:%s",str);

getch();

}

**Passing string to function:**

Example program:

#include <stdio.h>

#include<conio.h>

void displayString(char str[]);

void main()

{

char str[50];

clrscr();

printf("Enter string: ");

scanf("%s",&str)

displayString(str);

getch();

}

void displayString(char str[])

{

printf("String Output: ");

printf("String:%s",str);

}

**Functions of string:**

* **String copy**
* **String concardination**
* **String length**
* **String compare**
* **String character**

|  |  |
| --- | --- |
| **S.no** | **Functions of string** |
| 1 | **strcpy(s1, s2);**  Copies string s2 into string s1. |
| 2 | **strcat(s1, s2);**  Concatenates string s2 onto the end of string s1. |
| 3 | **strlen(s1);**  Returns the length of string s1. |
| 4 | **strcmp(s1, s2);**  Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2. |

**String copy:**

#include<stdio.h>

#include<conio.h>

#include<string.h>

void main()

{

int i;

char str1[10],str2[10];

clrscr();

printf(" Enter string 1");

gets(str1);

strcpy(str2,str1)

str2[i] = '\0';

printf("\n The Copied string2 is %s ",str2);

getch();

}

**String length:**

The function takes a single argument, i.e, the string variable whose length is to be found, and returns the length of the string passed

**Example:**

#include<stdio.h>

#include<conio.h>

#include<string.h>

void main()

{

char str[20],length;

clrscr();

printf("Enter the string:");

gets(str);

length=strlen(str);

printf("string length:%d",length);

getch();

}

**String comparision:**

The strcmp() compares two strings character by character. If the first character of two strings are equal, next character of two strings are compared. This continues until the corresponding characters of two strings are different or a null character '\0' is reached.

**Example:**

#include <stdio.h>

#include<conio.h>

#include <string.h>

void main()

{

char str1[],str2[];

int result;

clrscr();

printf("Enter the string1:")

gets(str1)

printf("Enter the string2:");

gets(str2)

result = strcmp(str1, str2);

printf("strcmp(str1, str2) = %d\n", result);

getch();

}

**String concatenation:**

Strcat() function used to concatenate the two string.

**Example:**

#include <stdio.h>

#include <string.h>

#include<conio.h>

void main()

{

char str1[] , str2[],c;

printf("Enter the first string:");

gets(str);

printf("Enter the second string:");

gets(str);

c=strcat(str1,str2);

printf("%s",c);

getch();

}

**Pointers in c:**

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers.

**Example:**

#include<stdio.h>

#include<conio.h>

void main();

{

int a;

char b[10];

clrscr();

printf("address of first variable: %x\n",a);

printf("address of second variable:%x\n",b);

getch();

}

What are Pointers?

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location.

Syntax: datatype \*variable\_name;

**Example:**

#include <stdio.h>

#include<conio.h>

void main ()

{

int var = 20;

int \*ip;

clrscr();

ip = &var;

printf("Address of var variable: %x\n", &var );

printf("Address stored in ip variable: %x\n", ip );

printf("Value of \*ip variable: %d\n", \*ip );

getch();

}

**Pointer to pointer:**

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



Syntax: datatype \*\*var\_name;

**Example:**

#include<stdio.h>

#include<conio.h>

void main ()

{

int a = 10;

int \*p;

int \*\*pp;

clrscr();

p = &a;

pp = &p;

printf("address of a: %x\n",p);

printf("address of p: %x\n",pp);

printf("value stored at p: %d\n",\*p);

printf("value stored at pp: %d\n",\*\*pp);

getch();

}

**Pointer to array:**

**Syntax:** datatype \*var\_name[size];

**Example:**

#include<stdio.h>

#include<conio.h>

void main()

{

int \*p;

int (\*ptr)[5];

int arr[5];

clrscr();

p = arr;

ptr = &arr;

printf("p = %p, ptr = %p\n", p, ptr);

p++;

ptr++;

printf("p = %p, ptr = %p\n", p, ptr);

getch();

}

**Structures:**

**structure** is another user defined data type available in C that allows to combine data items of different kinds.Structures are used to represent a record.

**Defining a structure:**

struct [structure tag] {

member definition;

member definition;

...

member definition;

} [one or more structure variables];

**Example:**

struct Student {

char name[50];

char dept[50];

char subject[100];

int rollno;

} student;

**Acessing structure:**

To access any member of a structure, we use the **member access operator (.)**.

Example

#include <stdio.h>

#include<conio.h>

struct studentdetails{

char name[50];

int id;

int age;

}student;

void main()

{

printf("Enter the student name:");

scanf("%s",&student.name);

printf("Enter the student id:");

scanf("%d",&student.id);

printf("Enter the student age:");

scanf("%d",&student.age);

printf("Student Name is: %s", student.name);

printf("\nStudent Id is: %d", student.id);

printf("\nStudent Age is: %d", student.age);

getch();

}

**Structure as a function arguments:**

**Example**

#include <stdio.h>

#include<conio.h?

struct student

{

char name[50];

int age;

}s1;

void display(struct student s);

void main()

{

printf("Enter name:");

scanf ("%[^\n]%\*c", s1.name);

printf("Enter age:");

scanf("%d", &s1.age);

display(s1);

getch();

}

void display(struct student s)

{

printf("\nDisplaying information\n");

printf("Name: %s", s.name);

printf("\nRoll: %d", s.age);

}

**Pointer to structure:**

**Example:**

#include <stdio.h>

#include<conio.h>

struct person

{

int age;

float weight;

}\*personPtr, person1;

void main()

{

personPtr = &person1;

printf("Enter age:");

scanf("%d", &personPtr->age);

printf("Enter weight:");

scanf("%f", &personPtr->weight);

printf("Displaying:\n");

printf("Age: %d\n", personPtr->age);

printf("weight: %f", personPtr->weight);

getch();

}

**Unions**

A union is a special data type available in C that allows to store different data types in the same memory location. Unions provide an efficient way of using the same memory location for multiple-purpose.

**Syntax:**

union [union tag] {

member definition;

member definition;

...

member definition;

} [one or more union variables];

**Example:**

#include <stdio.h>

#include<conio.h>

union unionjob

{

char name[32];

float salary;

int workerNo;

} ujob;

struct structJob

{

char name[32];

float salary;

int workerNo;

} sjob;

void main()

{

printf("size of union = %d bytes", sizeof(ujob));

printf("\nsize of structure = %d bytes", sizeof(sjob));

getch();

}

**File input/output operations:**

A file represents a sequence of bytes, regardless of it being a text file or a binary file. C programming language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices.

**File open:**

**fopen( )** function to create a new file or to open an existing file. This call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream.

**Syntax:**

FILE \*fopen( const char \* filename, const char \* mode );

|  |  |
| --- | --- |
| **sno.** | **Description** |
| 1 | **r**  Opens an existing text file for reading purpose. |
| 2 | **w**  Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file. |
| 3 | **a**  Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content. |
| 4 | **r+**  Opens a text file for both reading and writing. |
| 5 | **w+**  Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist. |
| 6 | **a+**  Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended. |

**Closing a file:**

int fclose( FILE \*fp );

**Writing a file:**

**i**nt fputc( int c, FILE \*fp );

**Reading a file:**

int fgetc( FILE \* fp );

Example(open,write,close):

# include <stdio.h>

# include <string.h>

#include<conio.h>

void main( )

{

FILE \*fp ;

char data[50];

printf( "Opening the file test.c in write mode" ) ;

fp = fopen("test.c", "w") ;

if ( fp == NULL )

{

printf( "Could not open file test.c" ) ;

}

printf( "\n Enter some text from keyboard” \

“ to write in the file test.c" ) ;

while ( strlen ( gets( data ) ) > 0 )

{

fputs(data, fp) ;

fputs("\n", fp) ;

}

printf("Closing the file test.c") ;

fclose(fp) ;

getch();

}

**Command line Arguments:**

It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program especially when you want to control

your program from outside instead of hard coding those values inside the code.

**Example:**

#include <stdio.h>

#include<conio.h>

int main( int argc, char \*argv[] ) {

if( argc == 2 ) {

printf("The argument supplied is %s\n", argv[1]);

}

else if( argc > 2 ) {

printf("Too many arguments supplied.\n");

}

else {

printf("One argument expected.\n");

}

getch();

}

**Type casting:**

Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a 'long' value into a simple integer then you can type cast 'long' to 'int'.

**Inbuilt typecasting:**

|  |  |
| --- | --- |
| **Typecast function** | **Description** |
| atof() | Converts string to float |
| [atoi()](http://fresh2refresh.com/c/c-type-casting/c-atoi-function/) | Converts string to int |
| [atol()](http://fresh2refresh.com/c/c-type-casting/c-atol-function/) | Converts string to long |
| [itoa()](http://fresh2refresh.com/c/c-type-casting/c-itoa-function/) | Converts int to string |
| [ltoa()](http://fresh2refresh.com/c/c-type-casting/c-ltoa-function/) | Converts long to string |

**Example(implicit typecasting):**

#include<stdio.h>

#include<conio.h>

int main()

{

int x = 10;

char y = 'a';

// value of 'a' is 97

x = x + y;

float z = x + 1.0;

printf("x = %d, z = %f", x, z);

getch();

}

**Example(explicit typecasting):**

#include<stdio.h>

#include<conio.h>

void main()

{

double x = 1.2;

int sum = (int)x + 1;

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

getch();

}