**UNIT 1**

**OVERVIEW OF C**

**INTRODUCTION**

The C programming language is a general-purpose, high-level language that was originally developed by Dennis M. Ritchie to develop the UNIX operating system at Bell Labs. C was originally first implemented in 1972. C is a successor of B language, which was introduced around 1970. The language was formalized in 1988 by the American National Standard Institute. (ANSI). In 1978, Brian Kernighan and Dennis Ritchie produced the first publicly available description of C, now known as the K&R standard. The C has now become a widely used professional language for various reasons.

* Easy to learn
* Structured language
* It produces efficient programs.
* It can handle low-level activities.
* It can be compiled on a variety of computer platforms.

**Languages**

Computer Languages are broadly classified into three categories, namely

1. **Machine Level Language:**

In machine level language computer only understand digital numbers i.e. in the form of 0 and 1. So, instruction given to the computer is in the form binary digit, which is difficult to implement instruction in binary code

1. **Assembly Level Language:**

The assembly language is on other hand modified version of machine level language. Where instructions are given in English like word as ADD, SUM, MOV etc. It is easy to write and understand but not understand by the machine.

1. **High level languages:**

High level language is machine independent, means it is portable. The language in this category is C, Pascal, Cobol, Fortran etc. High level languages are not understood by the machine. So, it needs to translate by the translator into machine level. A translator is software which is used to translate high level language as well as low level language in to machine level language

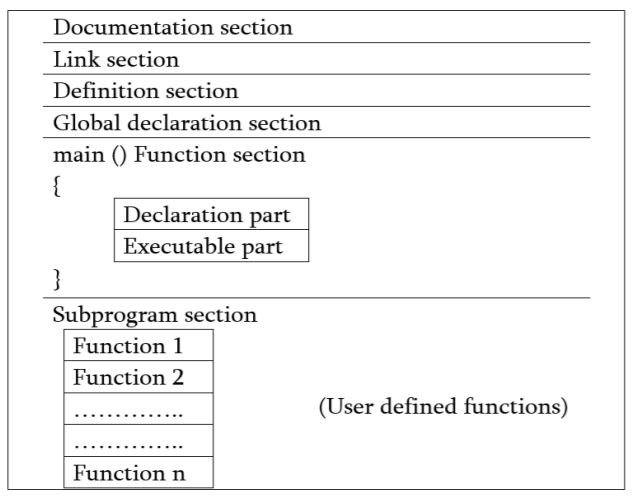
**Compiler and Interpreter**

Compiler and interpreter are used to convert the high-level language into machine level language. The program written in high level language is known as source program and the corresponding machine level language program is called as object program. Both compiler and interpreter perform the same task but there working is different. Compiler read the program at-a-time and searches the error and lists them. If the program is error free then it is converted into object program. When program size is large then compiler is preferred. Whereas interpreter read only one line of the source code and convert it to object code. It checks error, statement by statement and hence of take more time.

**IMPORTANCE OF C**

* C is a robust programming with an impressive set of built-in functions and a variety of operators which you can use to write any complex program.
* C programs are fast and efficient. This is because C uses a powerful set of data types and operators. C combines the power and capability of assembly language with the user-friendly features of a high-level language.
* C is the most widely used older programming language. It continues to go strong while older programming languages such as BASIC and COBOL have been virtually forgotten.
* C is very much portable, which means programs written on a machine using C can be used on other machines as well without any modification.
* C program consists of several functions that are supported by C library. In fact, you can create your own function, which can then be added to the C library.

**BASIC STRUCTURE OF C PROGRAMS**



1**. Documentation section**: The documentation section consists of a set of comment lines giving the name of the program, the author, and other details, which the programmer would like to use later.

2. **Link section**: The link section provides instructions to the compiler to link functions from the system library such as using the #include directive.

3. **Definition section**: The definition section defines all symbolic constants such using the #define directive.

4. **Global declaration section**: There are some variables that are used in more than one function. Such variables are called global variables and are declared in the global declaration section that is outside of all the functions. This section also declares all the user-defined functions.

5. **main () function section**: Every C program must have one main function section. This section contains two parts; declaration part and executable part

1. *Declaration part*: The declaration part declares all the variables used in the executable part.

2. *Executable part*: There is at least one statement in the executable part. These two parts must appear between the opening and closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function is the logical end of the program. All statements in the declaration and executable part end with a semicolon.

6. *Subprogram section*: If the program is a multi-function program then the subprogram section contains all the user-defined functions that are called in the main () function. User-defined functions are generally placed immediately after the main () function, although they may appear in any order.

**SAMPLE C PROGRAM**

1. Simple C code that would print the words "Hello World":

*#include*

*int main()*

*{*

*/\* my first program in C \*/*

*printf("Hello, World! \n");*

*return 0;*

*}*

1. C code to add two numbers.

*#include*

*int main()*

*{*

*int a=5, b= 10;*

*int sum=0;*

*sum=a+b;*

*printf(“Sum= %d”, sum);*

*}*

**PROGRAMMING STYLE**

C is a free form language so it does not care, where on the line we begin typing. Although there is different style, we should use one style for total consistency.

1. Meaningful names for variables, constants and functions. Use underscores for multi-word variables. For example, use hot\_water\_temperature instead of hotWaterTemperature. Using a variable name with a single character is only appropriate if the variable is the iteration variable of a loop.
2. Good indentation (3 or 4 spaces). Use a proper editor that assists with indentation.
3. If variables have the same type, declare them on the same line if possible.
4. Leave one blank line between variable declarations and the first line of code in a function.
5. Consistent style (e.g., use of curly brackets). Use braces; avoid loops and conditionals without them.
6. Opening brace must be on the same line as conditional or function.
7. Define values as constants when needed (do not use variables for constants). Do not use numbers in your expressions if those numbers have a special meaning (e.g., 3.1415); instead define constants (e.g., using #define) for them.
8. #defined constants must be in uppercase (e.g., #define MAX\_LEN 10 and NOT #define max\_len 10).
9. You should leave one blank space between operators (e.g., x = 5 + 7), and leave one space after a comma.
10. Use parentheses for clarity, especially with bit operations.

**EXECUTING C PROGRAM**

The step by step processes of executing a C program are:

**1) Creating the program.**

You can use any C/C++ compiler such as Turbo C/C++ or Dev C++ and online compilers. Open the compiler or IDE (Integrated Development Environment) and then type all the source code for your program carefully and correctly.

**2) Compile the source code:**

You can click the Compile button or menu from main menu or you can also press Alt + F9 to compile in Turbo C/C++. If you got error message, try to fix all errors.

**3) Linking the program**

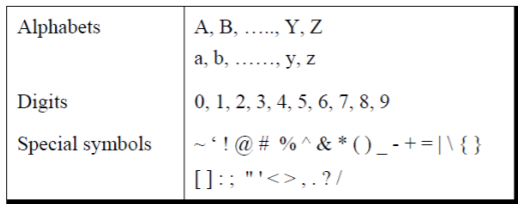
You need to link all the programs with functions that are needed from the C library. This is automatically done during compilation.

**4) Run the program:**

After fixing all the errors, you can run you program by click the run button or menu. You can also press Ctrl + F9 to run in Turbo C/C++.

**CHARACTER SET**

A character denotes any alphabet, digit or special symbol used to represent information in C programming. These are the programming elements which made up the C programming Language. Valid alphabets, numbers and special symbols allowed in C are 17 which are shown as below:



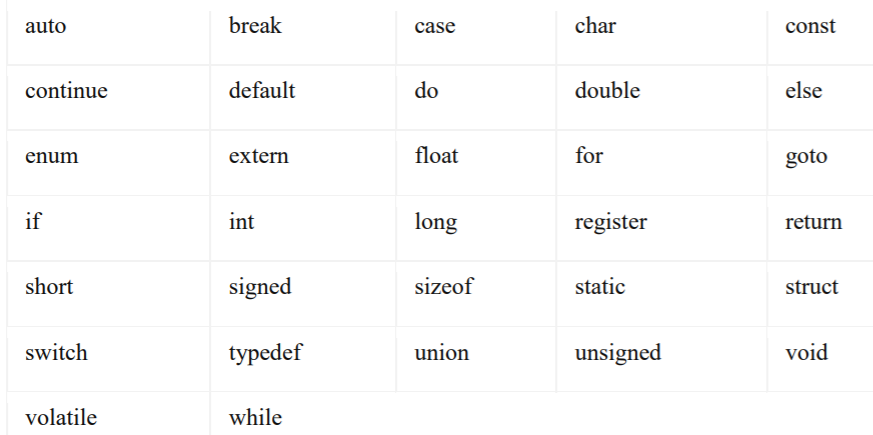
**C TOKENS**

C tokens are the basic buildings blocks in C language which are constructed together to write a C program. Each and every smallest individual unit in a C program is known as C tokens. C tokens are of six types. They are

* Keywords (eg: int, while),
* Identifiers (eg: main, total),
* Constants (eg: 10, 20),
* Strings (eg: ―total‖, ―hello‖),
* Special symbols (eg: (), {}),
* Operators (eg: +, /,-,\*)

**KEYWORDS**

C keywords are the words that convey a special meaning to the c compiler. The keywords cannot be used as variable names. There are 32 keywords in C programming. The list of C keywords is given below:



**IDENTIFIERS**

Identifiers are user defined word used to named entities like variables, arrays, functions, structures etc. In simple word, identifier in C is the name we have given to a variable, array, function, structure etc. Rules for naming identifiers are:

1) name should only consist of alphabets (both upper and lower case), digits and underscore (\_) sign.

2) first characters should be alphabet or underscore.

3) name should not be a keyword.

4) since C is a case sensitive, the upper case and lower case considered differently, for example code, Code, CODE etc. are different identifiers.

5) identifiers are generally given in some meaningful name such as value, net\_salary, age, data etc. An identifier name may be long, some implementation recognizes only first eight characters, most recognize 31 characters. ANSI standard compiler recognize 31 characters. Some invalid identifiers are 5cb, int, res#, avg no etc.

**CONSTANTS**

Constants or literals are those values that do not change over a period of time. There are different types of constants in C are:

1. **Integer Constants**

Integer constants refer to a sequence of digits without decimal points. It may include decimal integer, octal integer and hexadecimal integers proceeded by an optional sign of + or – symbol. For example, 123, 043, 0x9F, etc.

1. **Real Constants**

Quantities represented by numbers containing fractional parts like 17.548 are called real or floating-point constants. It can be represented in dotted decimal notation -0.73, 3.14, etc. and using exponential form such as 0.65e4, 12e-5, 3.18E3, etc. The constant before E is called mantissa and after E is called exponents.

1. **Character Constants**

A character constant contains a single character enclosed within a pair of single quote marks. For example, ‘A, ‘5’, ‘;’, etc. Remember, a digit or number enclosed in a single quote becomes a character constant.

1. **String Constants**

A string constant is a sequence of character enclosed in double quotes. The characters may be letters, numbers, special characters, or blank spaces. For example, “Alvin”, “2015”, “7+12”, etc.

**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.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive.

**DATATYPES**

Data types refer to the type of data that we are going to use e.g integer, character etc. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted. The value of a variable can be changed any time.

**BASIC DATATYPE**

There are four (4) basic datatypes namely: char, int, float, double. A variable declared to be of type int can be used to declare numbers. A variable declared to be of type float can be used for storing floating- point numbers (values containing decimal places). The double type is the same as type float, only with roughly twice the precision. The char data type can be used to store a single character, such as the letter a, the digit character 6, or a semicolon.

There are two types of type qualifier in C: Size qualifier (short, long), Sign qualifier (signed, unsigned). **When the qualifier unsigned is used the number is always positive, and when signed is used number may be positive or negative.** If the sign qualifier is not mentioned, then by default sign qualifier is assumed. The range of values for signed data types is less than that of unsigned data type. Because in signed type, the left most bit is used to represent sign, while in unsigned type this bit is also used to represent the value. The size and range of the different data types on a 16-bit machine is given below:

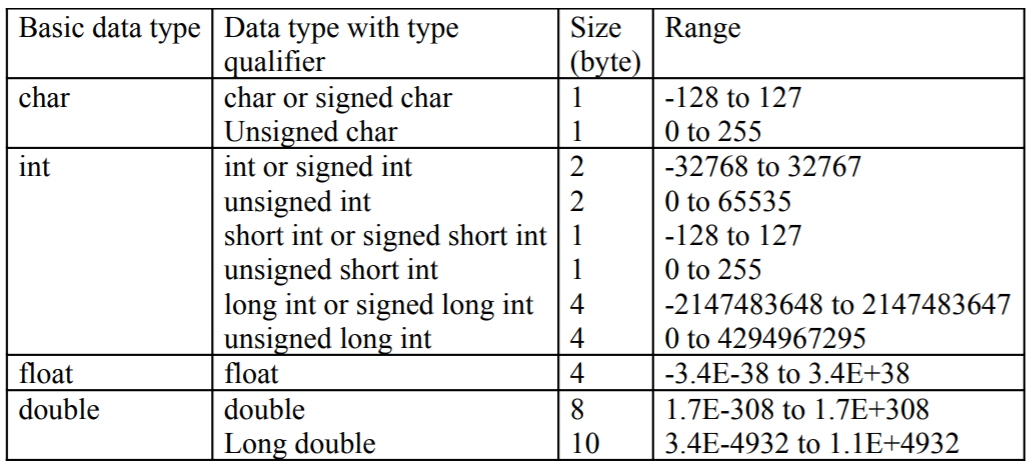


Table: Basic Data Type

**DECLARATION OF VARIABLES**

A variable declaration provides assurance to the compiler that there exists a variable with the given type and name so that the compiler can proceed for further compilation without requiring the complete detail about the variable. A variable definition has its meaning at the time of compilation only; the compiler needs actual variable definition at the time of linking the program. A variable declaration is useful when multiple files are used.

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows –

*type variable\_list;*

Here, type must be a valid C data type including char, w\_char, int, float, double, bool, or any user-defined object; and variable\_list may consist of one or more identifier names separated by commas. Some valid declarations are shown here –

*int i, j, k;*

*char c, ch;*

*float f, salary;*

*double d;*

**ASSIGN VALUE TO VARIABLE**

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows –

*type variable\_name = value;*

Some examples are –

*int d = 3, f = 5; // definition and initializing d and f.*

*byte z = 22; // definition and initializes z.*

*char x = 'x'; // the variable x has the value 'x'.*

Variable can also be assigned elsewhere after its declaration. It means it is not compulsory to assign a value at the time of declaration.

*Eg. int a,b;*

*a=12;*

*b=3;*

**DEFINE SYMBOLIC CONSTANTS**

Symbolic constant is a name that substitute for a sequence of characters and, characters may be numeric, character or string constant. These constants are generally defined at the beginning of the program as #define name value, here name generally written in upper case for example

*#define MAX 10*

*#define CH ‘b’*

The first directive tells the compiler to replace instances of MAX with 10. Use #define for constants to increase readability.

Example:

*#include <stdio.h>*

*#define PI 3.14*

*int main(void) {*

*float radius = 5.0;*

*// Using the constant PI*

*float area = PI \* radius \* radius;*

*printf("The area of circle is %f", area);*

*return 0;*

*}*

**OPERATORS**

In C language, operators are symbols that represent operations to be performed on one or more operands. C language offers many types of operators. They are,

1. Arithmetic operators

2. Assignment operators

3. Relational operators

4. Logical operators

5. Bit wise operators

6. Conditional operators (ternary operators)

7. Increment/decrement operators

8. Special operators

**ARITHMETIC OPERATORS**

An arithmetic operator in C is a symbol used to perform mathematical operations on operands, such as addition, subtraction, multiplication, division, and modulo. These operators take one or more numerical values (operands) and return a computed result. Following table shows all the arithmetic operators supported by C language.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Name of the Operator** | **Arithmetic Operation** | **Syntax** |
| **+** | Addition | Add two operands. | x **+**y |
| **–** | Subtraction | Subtract the second operand from the first operand. | x **–** y |
| **\*** | Multiplication | Multiply two operands. | x **\*** y |
| **/** | Division | Divide the first operand by the second operand. | x **/**y |
| **%** | [Modulus](https://www.geeksforgeeks.org/modulo-operator-in-c-cpp-with-examples/) | Calculate the remainder when the first operand is divided by the second operand. | x **%** y |

*// C program to demonstrate syntax of binary arithmetic*

*// operators*

*#include <stdio.h>*

*int main()*

*{*

*int a = 10, b = 4, res;*

*// printing a and b*

*printf("a is %d and b is %d\n", a, b);*

*res = a + b; // addition*

*printf("a + b is %d\n", res);*

*res = a - b; // subtraction*

*printf("a - b is %d\n", res);*

*res = a \* b; // multiplication*

*printf("a \* b is %d\n", res);*

*res = a / b; // division*

*printf("a / b is %d\n", res);*

*res = a % b; // modulus*

*printf("a %% b is %d\n", res);*

*return 0;*

*}*

*Output:*

*a is 10 and b is 4*

*a + b is 14*

*a - b is 6*

*a \* b is 40*

*a / b is 2*

*a % b is 2*

**RELATIONAL OPERATORS**

Relational operators in the C language are symbols that are used to compare two values and determine the relationship between them. Also known as comparison operators, they are binary operators, i.e., they compare two operands and always return the output in Boolean type, i.e., true or false. Following table shows all the relational operators supported by C language.

|  |  |  |
| --- | --- | --- |
| Operator | Meaning of Operator | Example |
| == | Equal to | 5 == 3 is evaluated to 0 |
| > | Greater than | 5 > 3 is evaluated to 1 |
| < | Less than | 5 < 3 is evaluated to 0 |
| != | Not equal to | 5 != 3 is evaluated to 1 |
| >= | Greater than or equal to | 5 >= 3 is evaluated to 1 |
| <= | Less than or equal to | 5 <= 3 is evaluated to 0 |

*#include <stdio.h>*

*int main()*

*{*

*int a = 5, b = 5, c = 10;*

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

*printf("%d == %d is %d \n", a, c, a == c);*

*printf("%d > %d is %d \n", a, b, a > b);*

*printf("%d > %d is %d \n", a, c, a > c);*

*printf("%d < %d is %d \n", a, b, a < b);*

*printf("%d < %d is %d \n", a, c, a < c);*

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

*printf("%d != %d is %d \n", a, c, a != c);*

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

*printf("%d >= %d is %d \n", a, c, a >= c);*

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

*printf("%d <= %d is %d \n", a, c, a <= c);*

*return 0;*

*}*

*Output:*

*5 == 5 is 1*

*5 == 10 is 0*

*5 > 5 is 0*

*5 > 10 is 0*

*5 < 5 is 0*

*5 < 10 is 1*

*5 != 5 is 0*

*5 != 10 is 1*

*5 >= 5 is 1*

*5 >= 10 is 0*

*5 <= 5 is 1*

*5 <= 10 is 1*

**LOGICAL OPERATORS**

Logical operators in C are symbols used to combine conditions and evaluate their truth values. The three main types are: Logical AND (&&): It evaluates to true only if both conditions are true. Logical OR (||): It evaluates to true if at least one of the conditions is true. Following table shows all the logical operators supported by C language.

|  |  |  |
| --- | --- | --- |
| Operator | Meaning | Example |
| && | Logical AND. True only if all operands are true | If c = 5 and d = 2 then, expression ((c==5) && (d>5)) equals to 0. |
| || | Logical OR. True only if either one operand is true | If c = 5 and d = 2 then, expression ((c==5) || (d>5)) equals to 1. |
| ! | Logical NOT. True only if the operand is 0 | If c = 5 then, expression !(c==5) equals to 0. |

*#include <stdio.h>*

*int main()*

*{*

*int a = 5, b = 5, c = 10, result;*

*result = (a == b) && (c > b);*

*printf("(a == b) && (c > b) is %d \n", result);*

*result = (a == b) && (c < b);*

*printf("(a == b) && (c < b) is %d \n", result);*

*result = (a == b) || (c < b);*

*printf("(a == b) || (c < b) is %d \n", result);*

*result = (a != b) || (c < b);*

*printf("(a != b) || (c < b) is %d \n", result);*

*result = !(a != b);*

*printf("!(a != b) is %d \n", result);*

*result = !(a == b);*

*printf("!(a == b) is %d \n", result);*

*return 0;*

*}*

*Output:*

*(a == b) && (c > b) is 1*

*(a == b) && (c < b) is 0*

*(a == b) || (c < b) is 1*

*(a != b) || (c < b) is 0*

*!(a != b) is 1*

*!(a == b) is 0*

**ASSIGNMENT OPERATORS**

Assignment operators are used for assigning value to a variable. The left side operand of the assignment operator is a variable and right side operand of the assignment operator is a value. The value on the right side must be of the same data-type of the variable on the left side otherwise the compiler will raise an error. There are following assignment operators supported by C language:

|  |  |  |
| --- | --- | --- |
| Operator | Example | Same as |
| = | a = b | a = b |
| += | a += b | a = a+b |
| -= | a -= b | a = a-b |
| \*= | a \*= b | a = a\*b |
| /= | a /= b | a = a/b |
| %= | a %= b | a = a%b |

*// Working of assignment operators*

*#include <stdio.h>*

*int main()*

*{*

*int a = 5, c;*

*c = a; // c is 5*

*printf("c = %d\n", c);*

*c += a; // c is 10*

*printf("c = %d\n", c);*

*c -= a; // c is 5*

*printf("c = %d\n", c);*

*c \*= a; // c is 25*

*printf("c = %d\n", c);*

*c /= a; // c is 5*

*printf("c = %d\n", c);*

*c %= a; // c = 0*

*printf("c = %d\n", c);*

*return 0;*

*}*

*Output:*

*c = 5*

*c = 10*

*c = 5*

*c = 25*

*c = 5*

*c = 0*

**INCREMENT DECREMENT OPERATORS**

The increment ( ++ ) and decrement ( --) operators in C are unary operators for incrementing and decrementing the numeric values by 1 respectively. The incrementation and decrementation are one of the most frequently used operations in programming for looping, array traversal, pointer arithmetic, and many more. C has two special unary operators called increment (++) and decrement (--) operators. These operators increment and decrement value of a variable by 1.

++x is same as x = x + 1 or x += 1  
--x is same as x = x - 1 or x - = 1

Increment and decrement operators can be used only with variables. They can't be used with constants or expressions.

Increment/Decrement operators are of two types:

1. Prefix increment/decrement operator.
2. Postfix increment/decrement operator.

**Prefix increment/decrement operator**

The prefix increment/decrement operator immediately increases or decreases the current value of the variable. This value is then used in the expression. Let's take an example:

y = ++x;

Here first, the current value of x is incremented by 1. The new value of x is then assigned to y. Similarly, in the statement:

y = --x;

the current value of x is decremented by 1. The new value of x is then assigned to y.

**Postfix Increment/Decrement operator**

The postfix increment/decrement operator causes the current value of the variable to be used in the expression, then the value is incremented or decremented. For example:

y = x++;

Here first, the current value of x is assigned to y then x is incremented.

Similarly, in the statement:

y = x--;

the current value of x is assigned to y then x is decremented.

*// Working of increment and decrement operators*

*#include <stdio.h>*

*int main()*

*{*

*int a = 10, b = 100;*

*float c = 10.5, d = 100.5;*

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

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

*printf("++c = %f \n", ++c);*

*printf("--d = %f \n", --d);*

*return 0;*

*}*

*Output:*

*++a = 11*

*--b = 99*

*++c = 11.500000*

*--d = 99.500000*

**CONDITIONAL OPERATORS**

The conditional operator in C is kind of like the if-else statement as it follows the same algorithm as of if-else statement but the conditional operator takes less space and helps to write the if-else statements in the shortest way possible. It is also known as the ternary operator in C as it operates on three operands. Conditional operators return one value if condition is true and returns another value is condition is false. This operator is also called as ternary operator.

*Syntax :*

*(Condition)? true\_value: false\_value;*

*Example :*

*(A > 100 )? 0 : 1;*

In above example, if A is greater than 100, 0 is returned else 1 is returned. This is equal to if else conditional statements.

**BIT-WISE OPERATORS**

Bitwise operators in C play a crucial role in manipulating data at the bit level, enabling programmers to effectively handle low-level operations. These binary-based operators facilitate various tasks, such as setting, clearing, or toggling specific bits in a number or performing Boolean logic without relying on any additional libraries. The Bitwise operators supported by C language are listed in the following table.

|  |  |
| --- | --- |
| Operators | Meaning of operators |
| & | Bitwise AND |
| | | Bitwise OR |
| ^ | Bitwise exclusive OR |
| ~ | Bitwise complement |
| << | Shift left |
| >> | Shift right |

*// C Program to demonstrate use of bitwise operators*

*#include <stdio.h>*

*int main()*

*{*

*// a = 5(00000101), b = 9(00001001)*

*unsigned int a = 5, b = 9;*

*// The result is 00000001*

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

*printf("a&b = %u\n", a & b);*

*// The result is 00001101*

*printf("a|b = %u\n", a | b);*

*// The result is 00001100*

*printf("a^b = %u\n", a ^ b);*

*// The result is 11111010*

*printf("~a = %u\n", a = ~a);*

*// The result is 00010010*

*printf("b<<1 = %u\n", b << 1);*

*// The result is 00000100*

*printf("b>>1 = %u\n", b >> 1);*

*return 0;*

*}*

*Output:*

*a = 5, b = 9*

*a&b = 1*

*a|b = 13*

*a^b = 12*

*~a = 4294967290*

*b<<1 = 18*

*b>>1 = 4*

**SPECIAL OPERATORS**

In the C programming language, special operators are used to perform specific operations that cannot be done with normal arithmetic or logical operators. There are few other important operators including sizeof and ? : supported by C Language.

|  |  |
| --- | --- |
| **Operators** | **Description** |
| & | This is used to get the address of the variable.  Example : &a will give address of a. |
| \* | This is used as pointer to a variable.  Example : \* a  where, \* is pointer to the variable a. |
| Sizeof () | This gives the size of the variable.  Example : size of (char) will give us 1. |
| ?: | Conditional Operator to check whetherthe given expression is true or not. |

*//Program to illustrate sizeof operator*

*#include <stdio.h>*

*int main()*

*{*

*int a;*

*float b;*

*double c;*

*char d;*

*printf("Size of int=%lu bytes\n",sizeof(a));*

*printf("Size of float=%lu bytes\n",sizeof(b));*

*printf("Size of double=%lu bytes\n",sizeof(c));*

*printf("Size of char=%lu byte\n",sizeof(d));*

*return 0;*

*}*

*Output:*

*Size of int = 4 bytes*

*Size of float = 4 bytes*

*Size of double = 8 bytes*

*Size of char = 1 byte*

*//program to illustrate \* operator*

*#include <stdio.h>*

*int main()*

*{*

*int \*ptr, q;*

*q = 50;*

*/\* address of q is assigned to ptr \*/*

*ptr = &q;*

*/\* display q's value using ptr variable \*/*

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

*return 0;*

*}*

*Output:*

*50*

**EXPRESSION**

An expression is a combination of variables, constants, operators and function call. It can be arithmetic, logical and relational for example:-

int z= x+y; // arithmetic expression

(a>b) //relational

(a==b) // logical

func(a, b) // function call

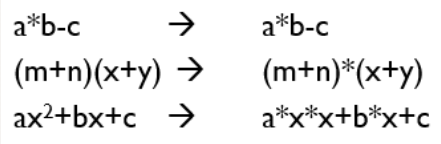
Expressions consisting entirely of constant values are called constant expressions. So, the expression 121 + 17 - 110 is a constant expression because each of the terms of the expression is a constant value. But if i were declared to be an integer variable, the expression 180 + 2 – j would not represent a constant expression. There are four types of expression in C.

* Arithmetic expressions
* Relational expressions
* Logical expressions
* Conditional expressions

**ARITHMETIC EXPRESSION**

An arithmetic expression is an expression that consists of operands and arithmetic operators. An arithmetic expression computes a value of type int, float or double.

When an expression contains only integral operands, then it is known as pure integer expression when it contains only real operands, it is known as pure real expression, and when it contains both integral and real operands, it is known as mixed mode expression.



**EVALUATION OF EXPRESSION**

The expressions are evaluated by performing one operation at a time. The precedence and associativity of operators decide the order of the evaluation of individual operations.

**When individual operations are performed, the following cases can be happened:**

* When both the operands are of type integer, then arithmetic will be performed, and the result of the operation would be an integer value. For example, 3/2 will yield 1 not 1.5 as the fractional part is ignored.
* When both the operands are of type float, then arithmetic will be performed, and the result of the operation would be a real value. For example, 2.0/2.0 will yield 1.0, not 1.
* If one operand is of type integer and another operand is of type real, then the mixed arithmetic will be performed. In this case, the first operand is converted into a real operand, and then arithmetic is performed to produce the real value. For example, 6/2.0 will yield 3.0 as the first value of 6 is converted into 6.0 and then arithmetic is performed to produce 3.0.

Let's understand through an example.

6\*2/ (2+1 \* 2/3 + 6) + 8 \* (8/4)

|  |  |
| --- | --- |
| **Evaluation of expression** | **Description of each operation** |
| 6\*2/( 2+1 \* 2/3 +6) +8 \* (8/4) | An expression is given. |
| 6\*2/(2+2/3 + 6) + 8 \* (8/4) | 2 is multiplied by 1, giving value 2. |
| 6\*2/(2+0+6) + 8 \* (8/4) | 2 is divided by 3, giving value 0. |
| 6\*2/ 8+ 8 \* (8/4) | 2 is added to 6, giving value 8. |
| 6\*2/8 + 8 \* 2 | 8 is divided by 4, giving value 2. |
| 12/8 +8 \* 2 | 6 is multiplied by 2, giving value 12. |
| 1 + 8 \* 2 | 12 is divided by 8, giving value 1. |
| 1 + 16 | 8 is multiplied by 2, giving value 16. |
| 17 | 1 is added to 16, giving value 17. |

**PRECEDENCE OF ARITHMETIC OPERATORS**

At first, the expressions within parenthesis are evaluated. If no parenthesis is present, then the arithmetic expression is evaluated from left to right. There are two priority levels of operators in C.

High priority: \* / %

Low priority: + -

The evaluation procedure of an arithmetic expression includes two left to right passes through the entire expression. In the first pass, the high priority operators are applied as they are encountered and in the second pass, low priority operations are applied as they are encountered. Suppose, we have an arithmetic expression as:

x = 9 – 12 / 3 + 3 \*2 - 1

This expression is evaluated in two left to right passes as:

First Pass

Step 1: x = 9-4 + 3 \* 2 – 1

Step 2: x = 9 – 4 + 6 – 1

Second Pass

Step 1: x = 5 + 6 – 1

Step 2: x = 11 – 1

Step 3: x = 10

But when parenthesis is used in the same expression, the order of evaluation gets changed.

**RELATIONAL EXPRESSIONS**

* A relational expression is an expression used to compare two operands.
* It is a condition which is used to decide whether the action should be taken or not.
* In relational expressions, a numeric value cannot be compared with the string value.
* The result of the relational expression can be either zero or non-zero value. Here, the zero value is equivalent to a false and non-zero value is equivalent to true.

|  |  |
| --- | --- |
| **Relational Expression** | **Description** |
| x%2 = = 0 | This condition is used to check whether the x is an even number or not. The relational expression results in value 1 if x is an even number otherwise results in value 0. |
| a!=b | It is used to check whether a is not equal to b. This relational expression results in 1 if a is not equal to b otherwise 0. |
| a+b = = x+y | It is used to check whether the expression "a+b" is equal to the expression "x+y". |
| a>=9 | It is used to check whether the value of a is greater than or equal to 9. |

**Let's see a simple example:**

*#include <stdio.h>*

*int main()*

*{*

*int x=4;*

*if(x%2==0)*

*{*

*printf("The number x is even");*

*}*

*else*

*printf("The number x is not even");*

*return 0;*

*}*

**Output**

The number x is even

**LOGICAL EXPRESSIONS**

* A logical expression is an expression that computes either a zero or non-zero value.
* It is a complex test condition to take a decision.

**Let's see some example of the logical expressions.**

|  |  |
| --- | --- |
| **Logical Expressions** | **Description** |
| ( x > 4 ) && ( x < 6 ) | It is a test condition to check whether the x is greater than 4 and x is less than 6. The result of the condition is true only when both the conditions are true. |
| x > 10 || y <11 | It is a test condition used to check whether x is greater than 10 or y is less than 11. The result of the test condition is true if either of the conditions holds true value. |
| ! ( x > 10 ) && ( y = = 2 ) | It is a test condition used to check whether x is not greater than 10 and y is equal to 2. The result of the condition is true if both the conditions are true. |

**Let's see a simple program of "&&" operator.**

*#include <stdio.h>*

*int main()*

*{*

*int x = 4;*

*int y = 10;*

*if ( (x <10) && (y>5))*

*{*

*printf("Condition is true");*

*}*

*else*

*printf("Condition is false");*

*return 0;*

*}*

**Output**

Condition is true

**Let's see a simple example of "| |" operator**

*#include <stdio.h>*

*int main()*

*{*

*int x = 4;*

*int y = 9;*

*if ( (x <6) || (y>10))*

*{*

*printf("Condition is true");*

*}*

*else*

*printf("Condition is false");*

*return 0;*

*}*

**Output**

Condition is True

**CONDITIONAL EXPRESSIONS**

* A conditional expression is an expression that returns 1 if the condition is true otherwise 0.
* A conditional operator is also known as a ternary operator.

**The Syntax of Conditional operator**

**Suppose exp1, exp2 and exp3 are three expressions.**

exp1 ? exp2 : exp3

The above expression is a conditional expression which is evaluated on the basis of the value of the exp1 expression. If the condition of the expression exp1 holds true, then the final conditional expression is represented by exp2 otherwise represented by exp3.

**Let's understand through a simple example.**

*#include<stdio.h>*

*#include<string.h>*

*int main()*

*{*

*int age = 25;*

*char status;*

*status = (age>22) ? 'M': 'U';*

*if(status == 'M')*

*printf("Married");*

*else*

*printf("Unmarried");*

*return 0;*

*}*

**Output**

Married

**TYPE CONVERSION**

When variables and constants of different types are combined in an expression then they are converted to same data type. The process of converting one predefined type into another is called type conversion. Type conversion in C can be classified into the following two types:

**Implicit Type Conversion**

When the type conversion is performed automatically by the compiler without programmer‘s intervention, such type of conversion is known as implicit type conversion or type promotion. The compiler converts all operands into the data type of the largest operand. The sequence of rules that are applied while evaluating expressions are given below:

1. All short and char are automatically converted to int, then,
2. If either of the operand is of type long double, then others will be converted to long double and result will be long double.
3. Else, if either of the operand is double, then others are converted to double.
4. Else, if either of the operand is float, then others are converted to float.
5. Else, if either of the operand is unsigned long int, then others will be converted to unsigned long int.
6. Else, if one of the operand is long int, and the other is unsigned int, then if a long int can represent all values of an unsigned int, the unsigned int is converted to long int. otherwise, both operands are converted to unsigned long int.
7. Else, if either operand is long int then other will be converted to long int.
8. Else, if either operand is unsigned int then others will be converted to unsigned int.

It should be noted that the final result of expression is converted to type of variable on left side of assignment operator before assigning value to it. Also,

* conversion of float to int causes truncation of fractional part,
* conversion of double to float causes rounding of digits and the
* conversion of long int to int causes dropping of excess higher order bits.

**Example:**

*#include <stdio.h>*

*int main() {*

*// Automatic conversion: int to float*

*float myFloat = 9;*

*printf("%f", myFloat);*

*return 0;*

*}*

*Output:*

*9.000000*

**Explicit Type Conversion**

The type conversion performed by the programmer by posing the data type of the expression of specific type is known as explicit type conversion. The explicit type conversion is also known as type casting. Type casting in c is done in the following form:

*(data\_type)expression;*

where, data\_type is any valid c data type, and expression may be constant, variable or expression. For example,

*int a=7,b=3;*

*x=(float) a / b;*

The following rules have to be followed while converting the expression from one type to another to avoid the loss of information: All integer types to be converted to float. All float types to be converted to double. All character types to be converted to integer

**Example:**

*#include <stdio.h>*

*int main() {*

*// Manual conversion: int to float*

*float sum = (float) 5 / 2;*

*printf("%f", sum);*

*return 0;*

*}*

*Output:*

*2.500000*

**OPERATOR PRECEDENCE AND ASSOCIATIVITY**

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator. For example, x = 7 + 3 \* 2; here, x is assigned 13, not 20 because operator \* has 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.

+2, 3+4, ++a

|  |  |  |  |
| --- | --- | --- | --- |
| **Precedence** | **Operator** | **Description** | **Associativity** |
| 1 | **()** | Parentheses (function call) | Left-to-Right |
| **[]** | Array Subscript (Square Brackets) |
| **.** | Dot Operator |
| **->** | Structure Pointer Operator |
| **++ , —** | Postfix increment, decrement |
| 2 | **++ / —** | Prefix increment, decrement | Right-to-Left |
| **+ / –** | Unary plus, minus |
| **! , ~** | Logical NOT,  Bitwise complement |
| **(type)** | Cast Operator |
| **\*** | Dereference Operator |
| **&** | Addressof Operator |
| **sizeof** | Determine size in bytes |
| 3 | **\*,/,%** | Multiplication, division, modulus | Left-to-Right |
| 4 | **+/-** | Addition, subtraction | Left-to-Right |
| 5 | **<< , >>** | Bitwise shift left, Bitwise shift right | Left-to-Right |
| 6 | **< , <=** | Relational less than, less than or equal to | Left-to-Right |
| **> , >=** | Relational greater than, greater than or equal to |
| 7 | **== , !=** | Relational is equal to, is not equal to | Left-to-Right |
| 8 | **&** | Bitwise AND | Left-to-Right |
| 9 | **^** | Bitwise exclusive OR | Left-to-Right |
| 10 | **|** | Bitwise inclusive OR | Left-to-Right |
| 11 | **&&** | Logical AND | Left-to-Right |
| 12 | **||** | Logical OR | Left-to-Right |
| 13 | **?:** | Ternary conditional | Right-to-Left |
| 14 | **=** | Assignment | Right-to-Left |
| **+= , -=** | Addition, subtraction assignment |
| **\*= , /=** | Multiplication, division assignment |
| **%= , &=** | Modulus, bitwise AND assignment |
| **^= , |=** | Bitwise exclusive, inclusive OR assignment |
| **<<=, >>=** | Bitwise shift left, right assignment |
| 15 | **,** | comma (expression separator) | Left-to-Right |

**MANAGING INPUT AND OUTPUT OPERATORS**

**FORMATTED INPUT/OUTPUT**

The C language provides both low and high level features it provides both the power of low-level languages and the flexibility and simplicity of high-level languages. C provides standard functions scanf() and printf(), for performing formatted input and output. These functions accept, as parameters, a format specification string and a list of variables. The format specification string is a character string that specifies the data type of each variable to be input or output and the size or width of the input and output.

**Formatted Output**

The function printf() is used for formatted output to standard output based on a format specification. The format specification string, along with the data to be output, are the parameters to the printf() function.

*Syntax:*

*printf (format, data1, data2,……..);*

In this syntax format is the format specification string. This string contains, for each variable to be output, a specification beginning with the symbol % followed by a character called the conversion character.

Example: printf (“%c”, data1); The character specified after % will display the value of data1 on the output screen.

**Example:**

*// C program to print a variable*

*#include <stdio.h>*

*int main()*

*{*

*int num1 = 99;*

*int num2 = 1;*

*printf("The sum of %d and %d is %d\n", num1, num2,*

*num1 + num2);*

*return 0;*

*}*

**Formatted Input**

The function scanf() is used for formatted input from standard input and provides many of the conversion facilities of the function printf().

*Syntax*

scanf (format, num1, num2,……);

The function scanf() reads and converts characters from the standards input depending on the format specification string and stores the input in memory locations represented by the other arguments (num1, num2,….).

For Example:

scanf(“%c %d”,&Name, &Roll No);

Note: the data names are listed as &Name and &Roll No instead of Name and Roll No 54 respectively. This is how data names are specified in a scanf() function. In case of string type data names, the data name is not preceded by the character &.

**Example:**

*// C program to implement*

*// scanf*

*#include <stdio.h>*

*// Driver code*

*int main()*

*{*

*int a, b;*

*printf("Enter first number: ");*

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

*printf("Enter second number: ");*

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

*printf("A : %d \t B : %d" ,*

*a , b);*

*return 0;*

*}*

**UNFORMATTED INPUT**

The unformatted input functions in C programming language are explained below −

## GETCHAR()

It reads a character from the keyboard. C getchar is a standard library function that takes a single input character from standard input. The major difference between getchar and getc is that getc can take input from any no of input streams but getchar can take input from a single standard input stream.

The syntax of getchar() function is as follows −

Variablename=getchar();

For example,

Char a;

a = getchar();

## Example Program

## *// C program to implement getchar()*

## *// function to read single character*

## *#include <stdio.h>*

## *int main()*

## *{*

## *int character;*

## *character = getchar();*

## *printf("The entered character is : %c", character);*

## *return 0;*

## *}*

## Input

## f

## Output

## The entered character is : f

## GETS()

It reads a string from the keyboard. It reads characters from the standard input (stdin) and stores them as a C string into str until a newline character or the end-of-file is reached.

The syntax for gets() function is as follows −

gets(variablename);

## Example

## *#include<stdio.h>*

## *#include<string.h>*

## *main(){*

## *char str[10];*

## *printf("Enter your name: \n");*

## *gets(str);*

## *printf("Hello %s, welcome to C programming tutorial", str);*

## *}*

## Output

Enter your name:

Madhu

Hello Sangi, welcome to welcome to C programming tutorial

**GETCHE()**

Reads a *single* character entered by the user at the console, and echoing it. getche() function is a function in C programming language which waits for any character input from keyboard and it will also echo the input character on to the output screen.

The syntax for getche() function is as follows −

Variablename=getche();

## Example:

## *#include <conio.h>*

## *#include <stdio.h>*

## *int main()*

## *{*

## *printf("%c", getche());*

## *return 0;*

## *}*

## Output:

## Input: g(without enter key as it is not buffered)

## Output: Program terminates immediately.

## But when you use DOS shell in Turbo C,

## double g, i.e., 'gg'

## UNFORMATTED OUTPUT FUNCTIONS

The unformatted output functions in C programming language are as follows −

## PUTCHAR()

It displays a character on the monitor. The putchar(int ch) method in C is used to write a character, of unsigned char type, to stdout. This character is passed as the parameter to this method.

The syntax for putchar() function is as follows −

Putchar(variablename);

For example,

Putchar(‘a’);

## Example:

## *// C program to demonstrate putchar() method*

## *#include <stdio.h>;*

## *int main()*

## *{*

## *// Get the character to be written*

## *char ch = '1';*

## *// Write the Character to stdout*

## *for (ch = '1'; ch <= '9'; ch++)*

## *putchar(ch);*

## *return (0);*

## *}*

## *Output*

## *123456789*

## PUTS()

It displays a string on the monitor. In C programming language, puts() is a function defined in header <stdio.h> that prints strings character by character until the NULL character is encountered. The puts() function prints the newline character at the end of the output string.

The syntax for puts() function is as follows −

puts(variablename);

For example,

puts("tutorial");

## Sample Program

## *// C program to illutrate the use of puts() function*

## *#include <stdio.h>*

## *int main()*

## *{*

## *// using puts to print hello world*

## *char\* str1 = "Hello Mizoram";*

## *puts(str1);*

## 

## *puts("Welcome Mizoram");*

## 

## *return 0;*

## *}*

## *Output*

## *Hello Mizoram*

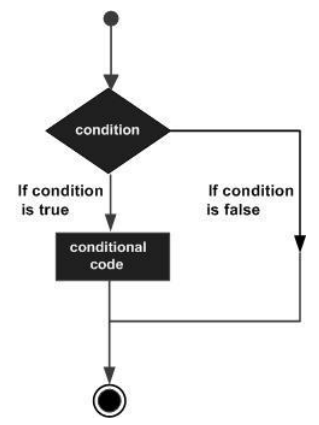
## *Welcome Mizoram*

**UNIT 2**

**DECISION MAKING AND LOOPING**

**DECISION MAKING STATEMENT**

Decision making structures require that the programmer specify 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. Following is the general form of a typical decision making structure found in most of the programming languages:



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. C programming language provides following types of decision making statements:

1. If statement
2. If…elseif statement
3. Switch statement

**IF STATEMENT**

The "if" statement in C is a control flow statement that is used to execute a block of code based on a condition. It evaluates an expression that results in a boolean value (true or false). If the expression evaluates to true (any non-zero value), the block of code inside the "if" statement is executed. An if statement consists of a boolean expression followed by one or more statements.

**Syntax**

The syntax of an if statement in C programming language is:

*if(boolean\_expression)*

*{*

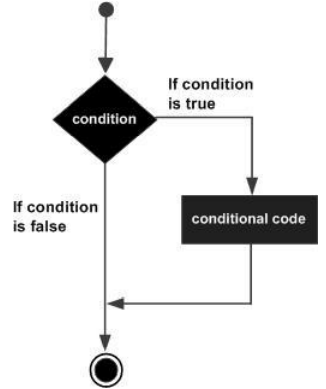
*/\* statement(s) will execute if the boolean expression is true \*/*

*}*

If the boolean expression evaluates to true, then the block of code inside the if statement will be executed. If 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.

**Flow Diagram**



**Example**

*#include<stdio.h>*

*int main ()*

*{*

*/\* local variable definition \*/*

*int a = 10;*

*/\* check the boolean condition using if statement \*/*

*if( a < 20 )*

*{*

*/\* if condition is true then print the following \*/*

*printf("a is less than 20\n" );*

*}*

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

*return 0;*

*}*

When the above code is compiled and executed, it produces the following result:

a is less than 20;

value of a is : 10

**IF ELSE STATEMENT**

An if statement can be followed by an optional else statement, which executes when the boolean expression is false.

**Syntax**

The syntax of an if...else statement in C programming language is:

*if(boolean\_expression)*

*{*

*/\* statement(s) will execute if the boolean expression is true \*/*

*}*

*else*

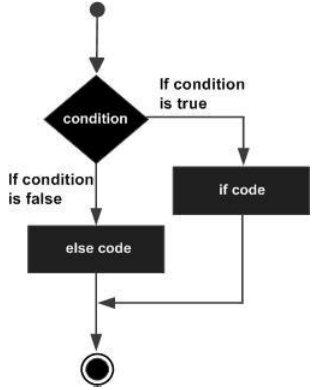
*{*

*/\* statement(s) will execute if the boolean expression is false \*/*

*}*

If the boolean expression evaluates to true, then the if block of code will be executed, otherwise else block of code 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.

Flow Diagram



**Example**

*#include <stdion.h>*

*int main ()*

*{*

*/\* local variable definition \*/*

*int a = 100;*

*/\* check the boolean condition \*/*

*if( a < 20 )*

*{*

*/\* if condition is true then print the following \*/*

*printf("a is less than 20\n" );*

*}*

*else*

*{*

*/\* if condition is false then print the following \*/*

*printf("a is not less than 20\n" );*

*}*

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

*return 0;*

*}*

When the above code is compiled and executed, it produces the following result:

a is not less than 20;

value of a is : 100

**NESTED IF ELSE 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).

**Syntax**

The syntax for a nested if statement is as follows:

*if( boolean\_expression 1)*

*{*

*/\* Executes when the boolean expression 1 is true \*/*

*if(boolean\_expression 2)*

*{*

*/\* Executes when the boolean expression 2 is true \*/*

*}*

*}*

You can nest else if...else in the similar way as you have nested if statement.

**Example**

*#include<stdio.h>*

*int main ()*

*{*

*/\* local variable definition \*/*

*int a = 100;*

*int b = 200;*

*/\* check the boolean condition \*/*

*If(a == 100)*

*{*

*/\* if condition is true then check the following \*/*

*if( b == 200 )*

*{*

*/\* if condition is true then print the following \*/*

*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 );*

*return 0;*

*}*

When the above code is compiled and executed, it produces the following result:

Value of a is 100 and b is 200

Exact value of a is : 100

Exact value of b is : 200

**ELSE IF LADDER**

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.

**Syntax**

The syntax of an if...else if...else statement in C programming language is:

*if(boolean\_expression 1)*

*{*

*/\* Executes when the boolean expression 1 is true \*/*

*}*

*else if( boolean\_expression 2)*

*{*

*/\* Executes when the boolean expression 2 is true \*/*

*}*

*else if( boolean\_expression 3)*

*{*

*/\* Executes when the boolean expression 3 is true \*/*

*}*

*else*

*{*

*/\* executes when the none of the above condition is true \*/*

*}*

**Example**

*#include<stdio.h>*

*int main ()*

*{*

*/\* local variable definition \*/*

*int a = 100;*

*/\* check the boolean condition \*/*

*if( a == 10 )*

*{*

*/\* if condition is true then print the following \*/*

*printf("Value of a is 10\n" );*

*}*

*else if( a == 20 )*

*{*

*/\* if else if condition is true \*/*

*printf("Value of a is 20\n" );*

*}*

*else if( a == 30 )*

*{*

*/\* if else if condition is true \*/*

*printf("Value of a is 30\n" );*

*}*

*else*

*{*

*/\* if none of the conditions is true \*/*

*printf("None of the values is matching\n" );*

*}*

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

*return 0;*

*}*

When the above code is compiled and executed, it produces the following result:

None of the values is matching

Exact value of a is: 100

**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**

The syntax for a switch statement in C programming language is as follows:

*switch(expression)*

*{*

*case constant-expression :*

*statement(s);*

*break; /\* optional \*/*

*case constant-expression :*

*statement(s);*

*break; /\* optional \*/*

*/\* you can have any number of case statements \*/*

*default : /\* Optional \*/*

*statement(s);*

*}*

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.

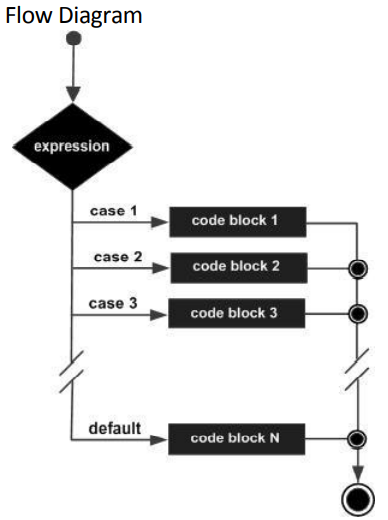
• When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.

• 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.

**Flow Diagram**



**Example 1**

*#include<stdio.h>*

*int main ()*

*{*

*int day = 4;*

*switch (day) {*

*case 1:*

*printf("Monday");*

*break;*

*case 2:*

*printf("Tuesday");*

*break;*

*case 3:*

*printf("Wednesday");*

*break;*

*case 4:*

*printf("Thursday");*

*break;*

*case 5:*

*printf("Friday");*

*break;*

*case 6:*

*printf("Saturday");*

*break;*

*case 7:*

*printf("Sunday");*

*break;*

*}*

**Example 2**

*#include<stdio.h>*

*int main ()*

*{*

*/\* local variable definition \*/*

*char grade = 'B';*

*switch(grade)*

*{*

*case 'A' :*

*printf("Excellent!\n" );*

*break;*

*case 'B' :*

*case 'C' :*

*printf("Well done\n" );*

*break;*

*case 'D' :*

*printf("You passed\n" );*

*break;*

*case 'F' :*

*printf("Better try again\n" );*

*break;*

*default :*

*printf("Invalid grade\n" );*

*}*

*printf("Your grade is %c\n", grade );*

*return 0;*

*}*

When the above code is compiled and executed, it produces the following result:

Well done Your grade is B

**THE ? : OPERATOR (Ternary Operator)**

The conditional operator in C is kind of similar to the if-else statement as it follows the same algorithm as of if-else statement but the conditional operator takes less space and helps to write the if-else statements in the shortest way possible. It is also known as the ternary operator in C as it operates on three operands. This operator can be used to replace if...else statements. It has the following general form:

**Exp1 ? Exp2 : Exp3;**

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon. 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.

**Example 1:**

*int main()*

*{*

*int m = 5, n = 4;*

*(m > n) ? printf("m is greater than n that is %d > %d", m, n) : printf("n is greater than m that is %d > %d", n, m);*

*return 0;*

*}*

**Example 2:**

*int main()*

*{*

*int a=5,b; // variable declaration*

*b=((a==5)?(3):(2)); // conditional operator*

*printf("The value of 'b' variable is : %d",b);*

*return 0;*

*}*

**BREAK STATEMENT**

The break statement is one of the four jump statements in the C language. The purpose of the break statement in C is for unconditional exit from the loop. The break statement in C programming language has the following two usages:

1. When the break statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.

2. It can be used to terminate a case in the switch statement.

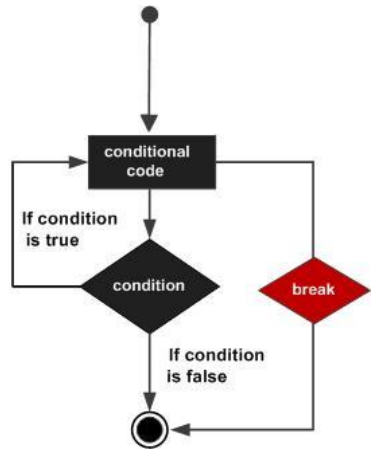
If you are using nested loops (i.e., one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax

The syntax for a break statement in C is as follows:

break;

Flow Diagram



**Example**

*#include<stdio.h>*

*int main ()*

*{*

*/\* local variable definition \*/*

*int a = 10;*

*/\* while loop execution \*/*

*while( a < 20 )*

*{*

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

*a++;*

*if( a > 15)*

*{*

*/\* terminate the loop using break statement \*/*

*break;*

*}*

*}*

*return 0;*

*}*

When the above code is compiled and executed, it produces the following result:

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

**CONTINUE STATEMENT**

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

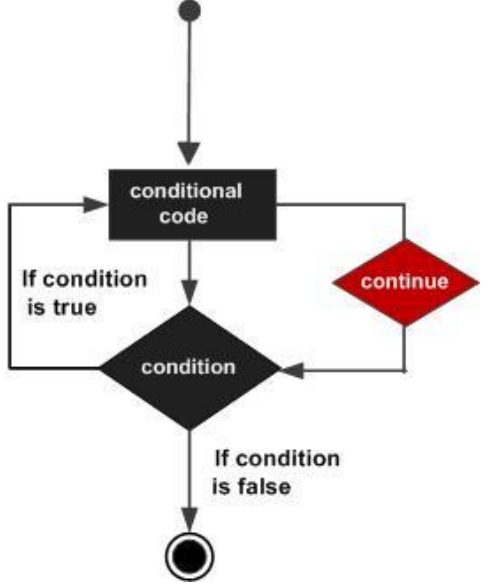
For the for loop, continue statement causes the conditional test and increment portions of the loop to execute. For the while and do...while loops, continue statement causes the program control passes to the conditional tests.

**Syntax**

The syntax for a continue statement in C is using the keyword:

continue;

**Flow Diagram**



**Example**

*#include<stdio.h?*

*int main ()*

*{*

*/\* local variable definition \*/*

*int a = 10;*

*/\* do loop execution \*/*

*do*

*{*

*if( a == 15)*

*{*

*/\* skip the iteration \*/*

*a = a + 1;*

*continue;*

*}*

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

*a++;*

*}while( a < 20 );*

*return 0;*

*}*

When the above code is compiled and executed, it produces the following result:

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 16

value of a: 17

value of a: 18

value of a: 19

**GOTO STATEMENT**

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

NOTE: 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. Any program that uses a goto can be rewritten so that it doesn't need the goto.

**Syntax**

The syntax for a goto statement in C is as follows:

goto label;

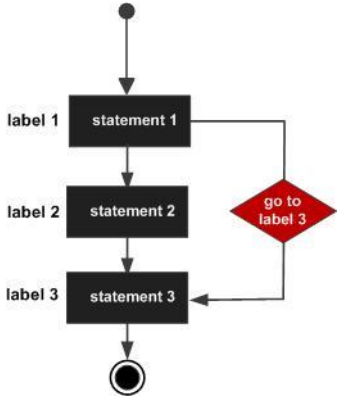
..

.

label: statement;

Here label can be any plain text except C keyword and it can be set anywhere in the C program above or below to goto statement.

**Flow Diagram**



**Example**

*#include<stdio.h>*

*int main ()*

*{*

*/\* local variable definition \*/*

*int a = 10;*

*/\* do loop execution \*/*

*LOOP:do*

*{*

*if( a == 15)*

*{*

*/\* skip the iteration \*/*

*a = a + 1;*

*goto LOOP;*

*}*

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

*a++;*

*}while( a < 20 );*

*return 0;*

*}*

When the above code is compiled and executed, it produces the following result:

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 16

value of a: 17

value of a: 18

value of a: 19

**ENUM**

An enumeration is a user-defined data type consists of integral constants and each integral constant is give a name. Keyword enum is used to defined enumerated data type.

enum type\_name{ value1, value2,...,valueN };

Here, type\_name is the name of enumerated data type or tag. And value1,value2,....,valueN are values of type type\_name. By default, value1 will be equal to 0, value2 will be 1 and so on but, the programmer can change the default value.

*enum boolean*

*{*

*false;*

*true;*

*}*

Here value of false is 0 and true is 1.

// Changing the default value of enum elements

*enum suit*

*{*

*club=0; diamonds=10; hearts=20; spades=3;*

*};*

**Example 1**

*#include<stdio.h>*

*enum week{ sunday, monday, tuesday, wednesday, thursday, friday, saturday};*

*int main()*

*{*

*enum week today;*

*today=wednesday;*

*printf("%d day",today+1);*

*return 0;*

*}*

Output

4 day

**Example 2**

*#include<stdio.h>*

*enum year{Jan=1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};*

*int main()*

*{*

*int i;*

*for (i=Jan; i<=Dec; i++)*

*printf("%d ", i);*

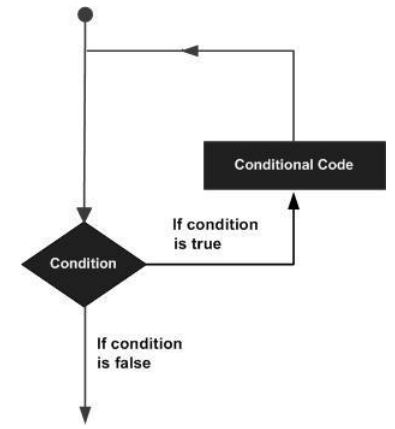
*return 0;*

*}*

**LOOPING STATEMENT**

There may be a situation, when you need to execute a block of code 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.

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 and following is the general form of a loop statement in most of the programming languages



**WHILE**

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

**Syntax**

The syntax of a while loop in C programming language is:

while(condition)

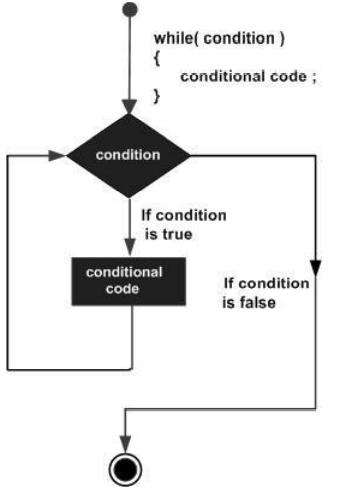
{

statement(s);

}

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, program control passes to the line immediately following the loop.

**Flow Diagram**



Here, key point of the while loop is that the loop might not ever run. 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.

**Example**

*#include<stdio.h>*

*int main ()*

*{*

*/\* local variable definition \*/*

*int a = 10;*

*/\* while loop execution \*/*

*while( a < 20 )*

*{*

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

*a++;*

*}*

*return 0;*

*}*

When the above code is compiled and executed, it produces the following result:

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19

**DO…WHILE**

Unlike for and while loops, which test the loop condition at the top of the loop, the do...while loop in C programming language checks its condition at the bottom of the loop. A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

**Syntax**

The syntax of a do...while loop in C programming language is:

do

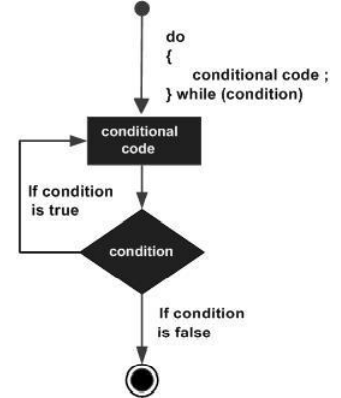
{

statement(s);

}while( condition );

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute 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 execute again. This process repeats until the given condition becomes false.

**Flow Diagram**



**Example**

*#include<stdio.h>*

*int main ()*

*{*

*/\* local variable definition \*/*

*int a = 10;*

*/\* do loop execution \*/*

*do*

*{*

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

*a = a + 1;*

*}while( a < 20 );*

*return 0;*

*}*

When the above code is compiled and executed, it produces the following result:

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19

**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**

The syntax of a for loop in C programming language is:

for ( init; condition; increment )

{

statement(s);

}

Here is the flow of control in a for loop:

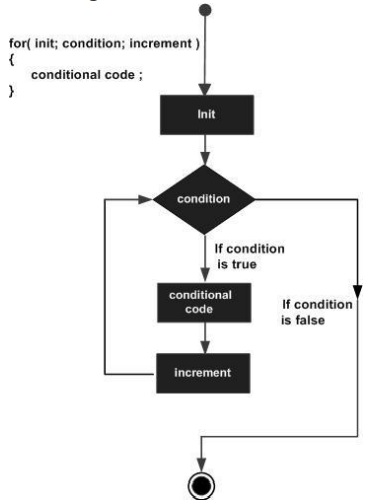
1. The init step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

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 flow of control jumps to the next statement just after the for loop.

3. After the body of the for loop executes, the flow of control jumps back up to the increment statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.

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.

**Flow Diagram**



**Example**

*#include<stdio.h>*

*int main ()*

*{*

*/\* for loop execution \*/*

*for( int a = 10; a < 20; a = a + 1 )*

*{*

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

*}*

*return 0;*

*}*

When the above code is compiled and executed, it produces the following result:

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19

**NESTING AND JUMPS IN LOOP**

C programming language allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax

The syntax for a nested for loop statement in C is as follows:

*for ( init; condition; increment )*

*{*

*for ( init; condition; increment )*

*{*

*statement(s);*

*}*

*statement(s);*

*}*

The syntax for a nested while loop statement in C programming language is as follows: *while(condition)*

*{*

*while(condition)*

*{*

*statement(s);*

*}*

*statement(s);*

*}*

The syntax for a nested do...while loop statement in C programming language is as follows:

*do*

*{*

*statement(s);*

*do*

*{*

*statement(s);*

*}while( condition );*

*}while( condition );*

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example, a for loop can be inside a while loop or vice versa.

Example

The following program uses a nested for loop to find the prime numbers from 2 to 100:

*#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;*

*}*

When the above code is compiled and executed, it produces the following result:

2 is prime

3 is prime

5 is prime

7 is prime

11 is prime

13 is prime

17 is prime

19 is prime

23 is prime

29 is prime

31 is prime

37 is prime

41 is prime

43 is prime

47 is prime

53 is prime

59 is prime

61 is prime

67 is prime

71 is prime

73 is prime

79 is prime

83 is prime

89 is prime

97 is prime

**INFINITE LOOP**

A loop becomes infinite loop if a condition never becomes false. The for loop is traditionally used for this purpose. Since none of the three expressions that form the for loop are required, you can make an endless loop by leaving the conditional expression empty.

*#include<stdio.h>*

*int main ()*

*{*

*for( ; ; )*

*{*

*printf("This loop will run forever.\n");*

*}*

*return 0;*

*}*

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the for(;;) construct to signify an infinite loop. You can terminate an infinite loop by pressing Ctrl + C keys

**UNIT 3**

**FUNCTIONS**

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. Some of the property of function are:

* A C programming code can be divided into one or more separate functions.
* Each function performs a specific task.
* 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.
* The C standard library provides numerous built-in functions that your program can call. For example, function strcat() to concatenate two strings, function memcpy() to copy one memory location to another location and many more functions.
* A function is known with various names like a method or a sub-routine or a procedure, etc.

**PROTOTYPE OF 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 language consists of a function header and a function body. Here are all the 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.

**Example**

Following is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum between the two:

*/\* function returning the max between two numbers \*/*

*int max(int num1, int num2)*

*{*

*/\* local variable declaration \*/*

*int result;*

*if (num1 > num2)*

*result = num1;*

*else*

*result = num2;*

*return result;*

*}*

**Function Declarations**

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately. A function declaration has the following parts:

*return\_type function\_name( parameter list );*

For the above defined function max(), following is the function declaration:

*int max(int num1, int num2);*

Parameter names are not important in function declaration only their type is required, so following is also valid declaration:

*int max(int, int);*

Function declaration is required when you define a function in one source file and you call that function in another file. In such case you should declare the function at the top of the file calling the function.

**STANDARD LIBRARY FUNCTION**

Library functions are built-in functions that are grouped together and placed in a common location called library. Each function here performs a specific operation. We can use this library functions to get the pre-defined output.

All C standard library functions are declared by using many header files. These library functions are created at the time of designing the compilers.

We include the header files in our C program by using **#include<filename.h>**. Whenever the program is run and executed, the related files are included in the C program.

Some of the header file functions are as follows −

1. **stdio.h** − It is a standard i/o header file in which Input/output functions are declared
2. **conio.h** − This is a console input/output header file.
3. **string.h** − All string related functions are in this header file.
4. **stdlib.h** − This file contains common functions which are used in the C programs.
5. **math.h** − All functions related to mathematics are in this header file.
6. **time.h** − This file contains time and clock related functions.
7. **ctype**.h - The ctype.h header file of the C Standard Library declares several functions that are useful for testing and mapping characters.

#### **LIST OF INBUILT C FUNCTIONS IN STDIO.H FILE:**

|  |  |  |
| --- | --- | --- |
| **SN** | **Function** | **Description** |
| 1 | printf() | This function is used to print the character, string, float, integer, octal and hexadecimal values onto the output screen |
| 2 | scanf() | This function is used to read a character, string, numeric data from keyboard. |
| 3 | getc() | It reads character from file |
| 4 | gets() | It reads line from keyboard |
| 5 | getchar() | It reads character from keyboard |
| 6 | puts() | It writes line to o/p screen |
| 7 | putchar() | It writes a character to screen |
| 8 | putc() | writes a character to file |

#### **LIST OF INBUILT C FUNCTIONS IN CONIO.H FILE:**

|  |  |  |
| --- | --- | --- |
| **SN** | **Functions** | **Description** |
| 1 | clrscr() | This function is used to clear the output screen. |
| 2 | getch() | It reads character from keyboard |
| 3 | getche() | It reads character from keyboard and echoes to o/p screen |
| 4 | textcolor() | This function is used to change the text color |
| 5 | textbackground() | This function is used to change text background |

#### **LIST OF INBUILT C FUNCTIONS IN STRING.H FILE:**

|  |  |  |
| --- | --- | --- |
| **SN** | **String functions** | **Description** |
| 1 | [strcat ( )](http://fresh2refresh.com/c/c-strings/c-strcat-function/) | Concatenates str2 at the end of str1 |
| 2 | [strcpy ( )](http://fresh2refresh.com/c/c-strings/c-strcpy-function/) | Copies str2 into str1 |
| 3 | [strlen ( )](http://fresh2refresh.com/c/c-strings/c-strlen-function/) | Gives the length of str1 |
| 4 | [strcmp ( )](http://fresh2refresh.com/c/c-strings/c-strcmp-function/) | Returns 0 if str1 is same as str2. Returns <0 if strl < str2. Returns >0 if str1 > str2 |
| 5 | [strlwr ( )](http://fresh2refresh.com/c/c-strings/c-strlwr-function/) | Converts string to lowercase |
| 6 | [strupr ( )](http://fresh2refresh.com/c/c-strings/c-strupr-function/) | Converts string to uppercase |
| 7 | [strrev ( )](http://fresh2refresh.com/c/c-strings/c-strrev-function/) | Reverses the given string |

#### **LIST OF INBUILT C FUNCTIONS IN STDLIB.H FILE:**

|  |  |  |
| --- | --- | --- |
| **SN** | **Function** | **Description** |
| **1** | **malloc()** | This function is used to allocate space in memory during the execution of the program. |
| **2** | **calloc()** | This function is also like malloc () function. But calloc () initializes the allocated memory to zero. But, malloc() doesn’t |
| **3** | **realloc()** | This function modifies the allocated memory size by malloc () and calloc() functions to new size |
| **4** | **free()** | This function frees the allocated memory by malloc (), calloc (), realloc () functions and returns the memory to the system. |
| **5** | **abort()** | It terminates the C program |
| **6** | **exit()** | This function terminates the program and does not return any value |
| **7** | **rand()** | This function returns the random integer numbers |
| **8** | **delay()** | This function Suspends the execution of the program for particular time |

#### **LIST OF INBUILT C FUNCTIONS IN MATH.H FILE:**

|  |  |  |
| --- | --- | --- |
| **SN** | **Function** | **Description** |
| 1 | [floor ( )](http://fresh2refresh.com/c/c-arithmetic-functions/c-floor-function/) | This function returns the nearest integer which is less than or equal to the argument passed to this function. |
| 2 | [round ( )](http://fresh2refresh.com/c/c-arithmetic-functions/c-round-function/) | This function returns the nearest integer value of the float/double/long double argument passed to this function. |
| 3 | [ceil ( )](http://fresh2refresh.com/c/c-arithmetic-functions/c-ceil-function/) | This function returns nearest integer value which is greater than or equal to the argument passed to this function. |
| 4 | [sin ( )](http://fresh2refresh.com/c/c-arithmetic-functions/c-sin-cos-tan-exp-log-function/) | This function is used to calculate sine value. |
| 5 | [cos ( )](http://fresh2refresh.com/c/c-arithmetic-functions/c-sin-cos-tan-exp-log-function/) | This function is used to calculate cosine. |
| 6 | [exp ( )](http://fresh2refresh.com/c/c-arithmetic-functions/c-sin-cos-tan-exp-log-function/) | This function is used to calculate the exponential “e” to the xthpower. |
| 7 | [log ( )](http://fresh2refresh.com/c/c-arithmetic-functions/c-sin-cos-tan-exp-log-function/) | This function is used to calculates natural logarithm. |
| 8 | [sqrt ( )](http://fresh2refresh.com/c/c-arithmetic-functions/c-sqrt-function/) | This function is used to find square root of the argument passed to this function. |
| 9 | [pow ( )](http://fresh2refresh.com/c/c-arithmetic-functions/c-pow-function/) | This is used to find the power of the given number. |

#### **LIST OF INBUILT C FUNCTIONS IN TIME.H FILE:**

|  |  |  |
| --- | --- | --- |
| **SN** | **Functions** | **Description** |
| **1** | **setdate()** | This function used to modify the system date |
| **2** | **getdate()** | This function is used to get the CPU time |
| **3** | **clock()** | This function is used to get current system time |
| **4** | **time()** | This function is used to get current system time as structure |
| **5** | **difftime()** | This function is used to get the difference between two given times |

#### **LIST OF INBUILT C FUNCTIONS IN CTYPE.H FILE:**

|  |  |  |
| --- | --- | --- |
| **SN** | **Functions** | **Description** |
| 1 | [isalpha()](http://fresh2refresh.com/c/c-int-char-validation/c-isalpha-function/) | checks whether character is alphabetic |
| 2 | [isdigit()](http://fresh2refresh.com/c/c-int-char-validation/c-isdigit-function/) | checks whether character is digit |
| 3 | [isalnum()](http://fresh2refresh.com/c/c-int-char-validation/c-isalnum-function/) | Checks whether character is alphanumeric |
| 4 | [isspace()](http://fresh2refresh.com/c/c-int-char-validation/c-isspace-function/) | Checks whether character is space |
| 5 | [islower()](http://fresh2refresh.com/c/c-int-char-validation/c-islower-function/) | Checks whether character is lower case |
| 6 | [isupper()](http://fresh2refresh.com/c/c-int-char-validation/c-isupper-function/) | Checks whether character is upper case |
| 7 | [tolower()](http://fresh2refresh.com/c/c-int-char-validation/c-tolower-function/) | Checks whether character is alphabetic & converts to lower case |
| 8 | [toupper()](http://fresh2refresh.com/c/c-int-char-validation/c-toupper-function/) | Checks whether character is alphabetic & converts to upper case |

**USER DEFINED FUNCTIONS**

The functions that we create in a program are known as user defined functions or in other words you can say that a function created by user is known as user defined function.

**Syntax of a function**

return\_type function\_name (argument list)

{

Set of statements – Block of code

}

**return\_type:** Return type can be of any data type such as int, double, char, void, short etc.

**function\_name:** It can be anything, however it is advised to have a meaningful name for the functions so that it would be easy to understand the purpose of function just by seeing it’s name.

**argument list:** Argument list contains variables names along with their data types. These arguments are kind of inputs for the function. For example – A function which is used to add two integer variables, will be having two integer argument.

The following example shows how to declare a user-defined function called addition. This function perform addition of two numbers which is passed as a parameter.

*int addition(int num1, int num2)*

*{*

*int sum;*

*/\* Arguments are used here\*/*

*sum = num1+num2;*

*return sum; //returns a result*

*}*

You call user-defined functions in the same way that you call other types of functions such as the main function as follows:

*int res = addition(var1, var2);*

**NEED FOR USER DEFINED FUNCTIONS**

1. It provides modularity to your program's structure.
2. It makes your code reusable. You just have to call the function by its name to use it, wherever required.
3. In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.
4. It makes the program more readable and easy to understand.

**RETURN VALUES AND THEIR TYPES**

A function may or may not send back any value to the calling function. If it does, it is done through the return statement. A return value is a value that a function returns to the calling function when it completes its task. A return value can be any one of the four variable types: handle, integer, object, or string. The type of value your function returns depends largely on the task it performs.

The return statement can take one of the following forms:

*return;*

Or

*return (expression);*

The first return statement does not return any value; it acts much as the closing brace of the function. When the return is encountered, the control is immediately passed back to the calling function. **Example**

*If(error)*

*return;*

The second form of return with an expression returns the value of the expression. Example

*int mul(int x, int y)*

*{*

*int p;*

*P=x\*y;*

*return (p);*

*}*

Return the value of p which is the product of the values of x and y. the last two statement can be combined into one statement as follows:

*return (x\*y);*

**CATEGORY OF FUNCTIONS**

All the **C** functions can be called either with **arguments** or without arguments in a C program. These functions may or may not **return values** to the calling function. Depending on the arguments and return values functions are classified into 4 categories:

1. Function without arguments and without a return value.
2. Function with arguments and without a return value.
3. Function without arguments and with a return value.
4. Function with arguments and with a return value.

When a function has **no arguments**, it does not receive any data from the calling function. similarly, when a function **does not return a value**, the calling function does not receive any data from the called function.

## 1. Functions without arguments and without return value:

When there is no argument passed in the calling function, there is no return value. In effect, there is no data transfer between the calling function and the called function in the category **function without arguments and without a return value**.

Let’s consider an example of a function without arguments and without a return value:

*#include <stdio.h>*

*void country\_capital(void);*

*void main() {*

*country\_capital();*

*}*

*void country\_capital() {*

*printf("New Delhi is the capital of India\n");*

*}*

**Output:**

*New Delhi is the capital of India*

In the above sample code the function

***void country\_capital(void);***

specifies that the function does not receive any arguments and does not return any value to the **main()** function.

## 2. Functions with arguments and without a return value:

When a function definition has **arguments**, it receives data from the calling function.

The **actual arguments** in the function call must correspond to the **formal parameters** in the function definition, i.e. the number of actual arguments must be the same as the number of formal parameters, and each actual argument must be of the same data type as its corresponding formal parameter. The formal parameters must be valid variable names in the function definition and the actual arguments may be variable names, expressions, or constants in the function call. The variables used in actual arguments must be assigned values before the function call is made. When a function call is made, copies of the values of actual arguments are passed to the called function.

Let’s consider an example of a function with arguments and without return value:

*#include <stdio.h>*

*void largest(int, int);*

*void main() {*

*int p, q;*

*printf("Enter two numbers : ");*

*scanf("%d%d" , &p, &q);*

*largest(p, q);*

*}*

*void largest(int x, int y) {*

*if (x > y)*

*{*

*printf("Largest element = %d\n", x);*

*}*

*else*

*{*

*printf("Largest element = %d\n", y);*

*}*

*}*

Output:

*Enter two numbers : 1*

*10*

*Largest element = 10*

In the above sample code the function

*void largest(int, int);*

specifies that the function receives two integer arguments from the calling function and does not return any value to the called function. When the function call largest(p, q) is made in the main() function, the values of actual arguments p and q are copied into the formal parameters x and y. After completion of execution of largest(int x, int y) function, it does not return any value to the main() function. Simply the control is transferred to the main() function.

**3. Functions without arguments and with a return value:**

When a function has no arguments, it does not receive any data from the calling function. When a function returns a value, the calling function receives data from the called function.

Let’s consider an example of a function without arguments and with a return value:

*#include <stdio.h>*

*int sum(void);*

*void main()*

*{*

*printf("\nSum of two given values = %d\n", sum());*

*}*

*int sum()*

*{*

*int a, b, total;*

*printf("Enter two numbers : ");*

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

*total = a + b;*

*return total;*

*}*

**Output:**

*Enter two numbers : 1 2*

*Sum of two given values = 3*

In the above sample code the function

*int sum(void);*

specifies that the function does not receive any arguments but returns a value to the calling function.

**4. Functions with arguments and with a return value:**

When a function definition has arguments, it receives data from the calling function.

After taking some desired action, only one value will be returned from called function to the calling a function through the return statement.

If a function returns a value, the function call may appear in any expression, and the returned value used as an operand in the evaluation of the expression.

Let us consider an example of a function with arguments and with return value:

*#include <stdio.h>*

*int largest(int, int, int);*

*void main() {*

*int a, b, c;*

*printf("Enter three numbers : ");*

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

*printf(" Largest of the given three numbers = %d\n", largest(a, b, c));*

*}*

*int largest(int x, int y, int z) {*

*if ((x > y) && (x > z)) {*

*return x;*

*} else if (y > z) {*

*return y;*

*} else {*

*return z;*

*}*

*}*

**Output:**

*Enter three numbers : 1 4 7*

*Largest of the given three numbers = 7*

In the above sample code the function

*int largest(int, int, int);*

specifies that the function receives three values and returns a value to the calling function.

**HANDLING OF NON INTEGER FUNCTIONS**

Functions can return different types of data other than integer. Many numerical functions like sqrt, sin, and cos return double; other specialized functions return other types than int. For functions returning non-integers values

* function must declare the type of value it returns (ie. double function ( ) )
* Calling routine must know that the function returns a non-int value
* Declarations must match definitions
* If functions take argument, declare them; if functions take no argument, use void

A function with larger primitive return type can return any smaller primitive type values, as long as it doesn't lead to a loss in the value of smaller primitive types.

*#include<stdio.h>*

*int main()*

*{*

*long funct(); /\*function prototype declaration\*/*

*long a = funct();*

*printf("Value in a : %d ", a);*

*}*

*long funct()*

*{*

*int i = 90;*

*return i;*

*}*

## *Output-*

*Value in a : 90*

Here we are returning an **int** value out of a function named **funct()**, which has an **long** return type, this works because -

* An **long** data type is larger than **int**, hence the **long** return type is large enough to hold a **int** value and return it from the function.
* There is no loss in the value of **int** even if its value is widened to an **long**.

***A char value is returned from a function with double return type***

*#include<stdio.h>*

*int main()*

*{*

*double funct(); /\*function prototype declaration\*/*

*double d = funct();*

*printf("Value in d : %lf ", a);*

*}*

*double funct()*

*{*

*char c ='A';*

*return c;*

*}*

Output-

*Value in d : 65.000000*

Here we are returning a char value out of a function that has a double return type, this works because -

* Data type double is larger than char, hence, double return type is large enough to hold a char value and return it from the function.
* There is no loss in the value of char even if its value is widened to a double. Hence, ASCII value of A(65) is printed by printf() function after converting it to a double value.

**CALLING A FUNCTION**

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, program control is transferred to the called function. A called function performs defined task, and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

*#include <stdio.h>*

*/\* function declaration \*/*

*int max(int num1, int num2);*

*int main ()*

*{*

*/\* local variable definition \*/*

*int a = 100;*

*int b = 200;*

*int ret;*

*/\* calling a function to get max value \*/*

*ret = max(a, b);*

*printf( "Max value is : %d\n", ret );*

*return 0;*

*}*

*/\* function returning the max between two numbers \*/*

*int max(int num1, int num2)*

*{*

*/\* local variable declaration \*/*

*int result;*

*if (num1 > num2)*

*result = num1;*

*else result = num2;*

*return result;*

*}*

The max() function is called from the main() function and would produce the following result:

*Max value is : 200*

**FUNCTION ARGUMENTS**

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function:

* Function call by value
* Function call by reference

**FUNCTION 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.

By default, C programming language uses call by value method to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function swap() definition as follows.

*#include<stdio.h>*

*/\* function declaration \*/*

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

*int main ()*

*{*

*/\* local variable definition \*/*

*int a = 100;*

*int b = 200;*

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

*printf("Before swap, value of b : %d\n", b );*

*/\* calling a function to swap the values \*/*

*swap(a, b);*

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

*printf("After swap, value of b : %d\n", b );*

*return 0;*

*}*

*/\* function definition to swap the values \*/*

*void swap(int x, int y)*

*{*

*int temp;*

*temp = x; /\* save the value of x \*/*

*x = y; /\* put y into x \*/*

*y = temp; /\* put x into y \*/*

*return;*

*}*

Let us put above code in a single C file, compile and execute it, it will produce the following result:

*Before swap, value of a :100*

*Before swap, value of b :200*

*After swap, value of a :100*

*After swap, value of b :200*

Which shows that there is no change in the values though they had been changed inside the function.

**FUNCTION 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. This means that changes made to the parameter affect the passed argument. To pass the value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

Let us call the function swap() by passing values by reference as in the following example:

*#include<stdio.h>*

*/\* function declaration \*/*

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

*int main ()*

*{*

*/\* local variable definition \*/*

*int a = 100;*

*int b = 200;*

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

*printf("Before swap, value of b : %d\n", b );*

*/\* calling a function to swap the values.*

*\* &a indicates pointer to a ie. address of variable a and*

*\* &b indicates pointer to b ie. address of variable b. \*/*

*swap(&a, &b);*

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

*printf("After swap, value of b : %d\n", b );*

*return 0;*

*}*

*/\* function definition to swap the values \*/*

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

*{*

*int temp;*

*temp = \*x; /\* save the value at address x \*/*

*\*x = \*y; /\* put y into x \*/*

*\*y = temp; /\* put x into y \*/*

*return;*

*}*

**Output:**

*Before swap, value of a :100*

*Before swap, value of b :200*

*After swap, value of a :200*

*After swap, value of b :100*

Which shows that there is no change in the values though they had been changed inside the function.

**RECURSION**

Recursion is the process of repeating items in a self-similar way. Same applies in programming languages as well where if a programming allows you to call a function inside the same function that is called recursive call of the function as follows.

*void recursion()*

*{*

*recursion(); /\* function calls itself \*/*

*}*

*int main()*

*{*

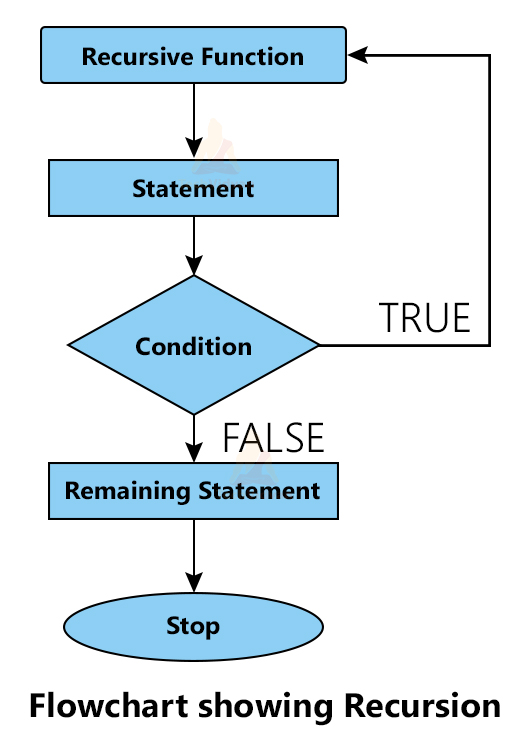
*recursion();*

*}*

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go in infinite loop.

Recursive function are very useful to solve many mathematical problems like to calculate factorial of a number, generating Fibonacci series, etc.

**Flowchart of recursion**



**Example 1: Write a C program to find the Sum of n Natural Numbers**

*#include<stdio.h>*

*int sum(int k);*

*int main() {*

*int n;*

*printf("Enter any number:");*

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

*int result = sum(n);*

*printf("Sum of natural no from 1 to %d is %d", n,result);*

*return 0;*

*}*

*int sum(int k) {*

*if (k > 0) {*

*return k + sum(k - 1);*

*}*

*else {*

*return 0;*

*}*

*}*

**Example 1: Write a C program to find the Factorial of a given number**

*#include<stdio.h>*

*int factorial(unsigned int i)*

*{*

*if(i <= 1)*

*{*

*return 1;*

*}*

*return i \* factorial(i - 1);*

*}*

*int main()*

*{*

*int i = 15;*

*printf("Factorial of %d is %d\n", i, factorial(i));*

*return 0;*

*}*

When the above code is compiled and executed, it produces the following result:

*Factorial of 15 is 2004310016*

**STORAGE CLASSES OF VARIABLES**

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. These specifiers precede the type that they modify. These are the following storage classes, which can be used in a C Program

• auto

• register

• static

• extern

**The auto Storage Class**

It is the local variable known only to the function in which it is declared. The auto storage class is the default storage class for all local variables.

{

int mount;

auto int month;

}

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

**Example:**

*#include<stdio.h>*

*main ( ){*

*auto int i=1;*

*{*

*auto int i=2;*

*{*

*auto int i=3;*

*printf ("%d",i);*

*}*

*printf("%d", i);*

*}*

*printf("%d", i);*

*}*

*Output: 321*

**The register Storage Class**

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).

{

register int miles;

}

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.

**Example:**

* *#include<stdio.h>*
* *main ( ){*
* *register int i;*
* *for (i=1; i<=5; i++)*
* *printf ("%d ",i);*
* *}*

***Output:*** *12345*

**The static Storage Class**

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 objects of its class.

*#include <stdio.h>*

*/\* function declaration \*/*

*void func(void);*

*static int count = 5; /\* global variable \*/*

*main()*

*{*

*while(count--)*

*{*

*func();*

*}*

*return 0;*

*}*

*/\* function definition \*/*

*void func( void )*

*{*

*static int i = 5; /\* local static variable \*/*

*i++;*

*printf("i is %d and count is %d\n", i, count);*

*}*

When the above code is compiled and executed, it produces the following result:

*i is 6 and count is 4*

*i is 7 and count is 3*

*i is 8 and count is 2*

*i is 9 and count is 1*

*i is 10 and count is 0*

**The extern Storage Class**

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 as all it does is point 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 be used in other files also, then extern will be used in another file to give reference of defined variable or function.

**Example:**

* *#include <stdio.h>*
* *int main()*
* *{*
* *extern int a;*
* *printf("%d",a);*
* *}*
* *int a = 20;*

***Output:20***

**STORAGE CLASS: SCOPE AND LIFETIME**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Storage Classes** | **Storage Place** | **Default Value** | **Scope** | **Lifetime** |
| auto | RAM | Garbage Value | Local | Within function |
| extern | RAM | Zero | Global | Till the end of the main program Maybe declared anywhere in the program |
| static | RAM | Zero | Local | Till the end of the main program, Retains value between multiple functions call |
| register | Register | Garbage Value | Local | Within the function |

**UNIT 4**

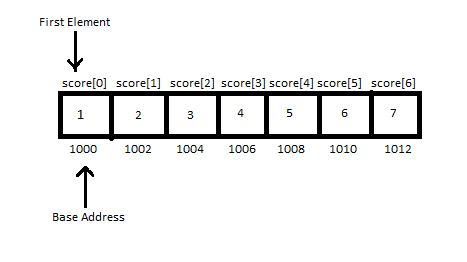
**ARRAYS AND POINTERS**

**ARRAYS**

Array is the collection of similar data types or collection of similar entity stored in contiguous memory location. C programming language provides a data structure called the array, which 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.



**DECLARATION ANAD INITIALIZATION**

**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];**

Now balance is a variable array which is sufficient to hold up-to 10 double numbers.

**Initializing Arrays**

You can initialize array in C either one by one or using a single statement as follows:

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

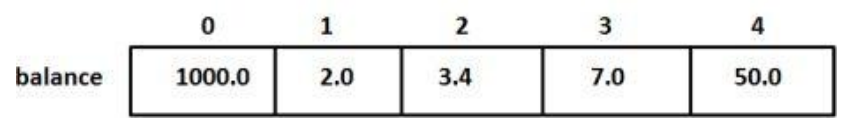
The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

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

You will create exactly the same array as you did in the previous example.

**balance[4] = 50.0;**

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th i.e. last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representation of the same array we discussed above:



**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 10th element from the array and assign the value to salary variable. **Following is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays:**

*#include <stdio.h>*

*int main ()*

*{*

*int n[ 10 ]; /\* n is an array of 10 integers \*/*

*int i,j;*

*/\* initialize elements of array n to 0 \*/*

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

*{*

*n[ i ] = i + 100; /\* set element at location i to i + 100 \*/*

*}*

*/\* output each array element's value \*/*

*for (j = 0; j < 10; j++ )*

*{*

*printf("Element[%d] = %d\n", j, n[j] );*

*}*

*return 0;*

*}*

*When the above code is compiled and executed, it produces the following result:*

*Element[0] = 100*

*Element[1] = 101*

*Element[2] = 102*

*Element[3] = 103*

*Element[4] = 104*

*Element[5] = 105*

*Element[6] = 106*

*Element[7] = 107*

*Element[8] = 108*

*Element[9] = 109*

**TYPES OF ARRAYS**

An array with fixed size is said to be a static array. Types of static array:

1. One Dimensional Array

2. Two Dimensional Array.

3. Multi Dimensional Array.

**1. One Dimensional Array**

One-dimensional arrays, also known as single arrays, are arrays with only one dimension or a single row. A one dimensional array in C contains a series of elements of the same datatype. The elements are zero-indexed, meaning the first element is in position or index 0, and the last element is in position or index (array size) - 1.

### **Syntax for declaration of One Dimensional Array**

datatype array\_name[size];

Here, array\_name is an array of type datatype and size is the number of elements in that array.

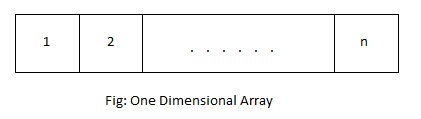
**For example**,

*int a[5]; // declares an integer array with 5 elements*

*float f[5]; // declares an float array with 5 elements*

*char c[10]; // declares an character array with 10 elements*

### **Arrangement of One Dimensional Array**



### **Example of One Dimensional Array**

**Program:** **C Program to enter 5 numbers and print them in reverse order.**

*#include<stdio.h>*

*int main()*

*{*

*int i,a[10];*

*printf("Enter 5 numbersn");*

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

*{*

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

*}*

*printf("Reverse ordern");*

*for(i=4;i>=0;i--)*

*{*

*printf("%dn",a[i]);*

*}*

*return 0;*

*}*

In this program, a one dimensional integer array with 5 elements is declared where these values are entered by the user. These values are then stored in respective index of array a using a [for loop](http://www.programtopia.net/c-programming/docs/for-loop). Then, these values are printed in reverse order, i.e. the value with higher index is printed at first, by traversing through the array.

**Output**

Enter 5 numbers

11

34

-21

6

90

Reverse order

90

6

-21

34

11

**2. Two Dimensional Array.**

An array where data is arranged in two dimensions is called two dimensional array. It stores data in column and row form .

**Syntax and Declaration of Two Dimensional Array**

datatype array\_name[d1][d2];

Here, *array\_name*is an array of type *datatype* and it has 2 dimensions. The number of elements in *array\_name* is equal to *d1\*d2*.

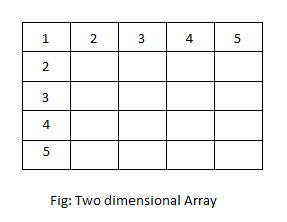
**For example**,

int i[5][5]; // declares an integer array with 25 elements

float f[5][10]; // declares an float array with 50 elements

char n[5][10]; // declares an character array with 50 elements

**Arrangement of Two Dimensional Array**



**Example: C Program to find the sum of all the elements of a 2 x 3 matrix.**

*#include<stdio.h>*

*int main()*

*{*

*int i,j,a[2][3],sum=0;*

*printf("Enter elements of matrixn");*

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

*{*

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

*{*

*printf("a[%d][%d] = ",i+1,j+1);*

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

*sum = sum + a[i][j];*

*}*

*}*

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

*return 0;*

*}*

In this program, a 2 x 3 matrix is represented by a two dimensional array *a[2][3]*. The elements of matrix are entered by user. A variable *sum* is declared and initialized to zero that holds the sum. A nested for loop is used to enter the elements of matrix and to add them with *sum.*Finally, the sum is printed after the loop is terminated.

**Output**

Enter elements of matrix

a[1][1] = 10

a[1][2] = 4

a[1][3] = 21

a[2][1] = 7

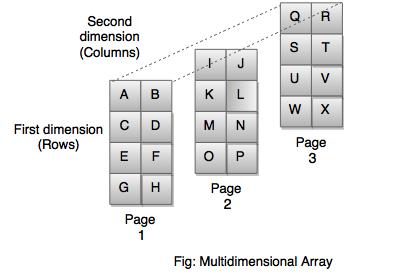
a[2][2] = 19

a[2][3] = 0

Sum = 61

**3. Multi Dimensional Array.**

An array where data are arranged in multiple dimensions is called multi-dimensional array. In other words, a multi-dimensional array is an array of another array with one dimension less i.e. a two dimensional array is an array of one dimensional array, a three dimensional array is an array of two dimensional array and so on. An array can have as many dimensions as required. However, two dimensional and three dimensional array are commonly used.



### **Syntax for declaration of Multi-Dimensional Array**

datatype array\_name[d1][d2]...[dn];

Here, array\_name is an array of type datatype and it has n dimensions.

The number of elements in a multi dimensional array is equal to the product of size of all dimensions i.e. total number of elements in array array\_name is d1\*d2\* … dn.

**For example**,

*int i[5][5][10]; // declares an integer array with 250 elements*

*float f[2][5][10]; // declares an float array with 100 elements*

*char n[5][5][10]; // declares an character array with 250 elements*

**Example: C program to show the concept of three dimensional array.**

*#include <stdio.h>*

*int main()*

*{*

*int a[10][10][10],d1,d2,d3,i,j,k;*

*printf("Enter size of three dimensions: ");*

*scanf("%d%d%d",&d1,&d2,&d3);*

*printf("Enter elements of arrayn");*

*for(i=0;i<d1;i++)*

*{*

*for(j=0;j<d2;j++)*

*{*

*for(k=0;k<d3;k++)*

*{*

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

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

*}*

*}*

*}*

*printf("Displaying elements of arrayn");*

*for(i=0;i<d1;i++)*

*for(j=0;j<d2;j++)*

*for(k=0;k<d3;k++)*

*printf("a[%d][%d][%d] = %dn",i,j,k,a[i][j][k]);*

*getch();*

*return 0;*

*}*

This example show how data is stored and accessed from a three dimensional array. The values of size of three dimensions: *d1*, *d2* and *d3* are entered by user. According to these values, a nested loop is created to enter the value of elements of array and display them. The outermost loop runs *d1* times, middle loop runs *d2* times and the innermost loop runs *d3* times printing the values of the array *a* as they go accessed by their index.

**Output**

*Enter size of three dimensions: 2 2 2*

*Enter elements of array*

*a[0][0][0] = 5*

*a[0][0][1] = 9*

*a[0][1][0] = -11*

*a[0][1][1] = 23*

*a[1][0][0] = 45*

*a[1][0][1] = -111*

*a[1][1][0] = 2*

*a[1][1][1] = 0*

*Displaying elements of array*

*a[0][0][0] = 5*

*a[0][0][1] = 9*

*a[0][1][0] = -11*

*a[0][1][1] = 23*

*a[1][0][0] = 45*

*a[1][0][1] = -111*

*a[1][1][0] = 2*

*a[1][1][1] = 0*

**POINTERS**

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte. 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 you can use it to store any variable address.

**ACCESSING THE ADDRESS OF VARIABLE**

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory.

Consider the following example, which will print the address of the variables defined:

*#include <stdio.h>*

*int main ()*

*{*

*int var1;*

*char var2[10];*

*printf("Address of var1 variable: %x\n", &var1 );*

*printf("Address of var2 variable: %x\n", &var2 );*

*return 0;*

*}*

When the above code is compiled and executed, it produces result something as follows:

*Address of var1 variable: bff5a400*

*Address of var2 variable: bff5a3f6*

**DECLARING AND INITIALIZING POINTERS**

**POINTER DECLARATION**

The general form of a pointer variable declaration is:

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 \* you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

*int \*ip; /\* pointer to an integer \*/*

*double \*dp; /\* pointer to a double \*/*

*float \*fp; /\* pointer to a float \*/*

*char \*ch /\* pointer to a character \*/*

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

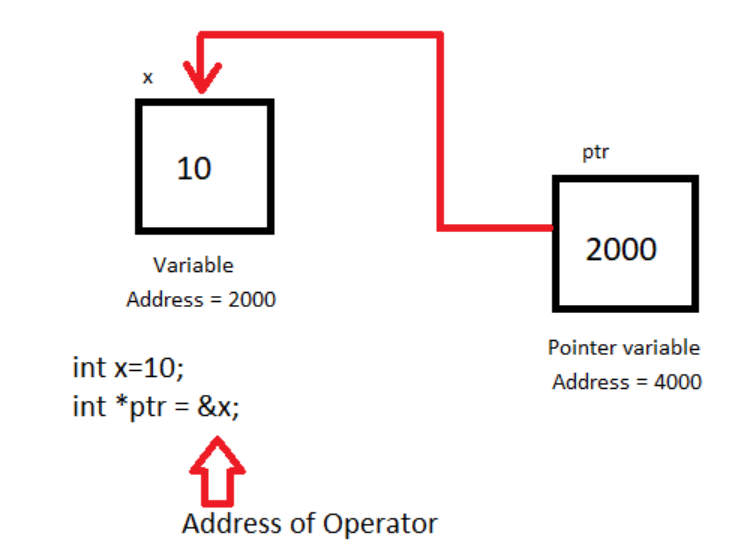
**INITIALIZATION OF POINTER**

**Pointer Initialization** is the process of assigning address of a variable to a **pointer** variable. It contains the address of a variable of the same data type. In C language **address operator** & is used to determine the address of a variable. The & (immediately preceding a variable name) returns the address of the variable associated with it.

*int a = 10;*

*int \*ptr; //pointer declaration*

*ptr = &a; //pointer initialization*

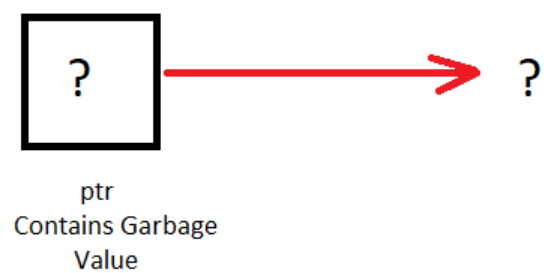
****

Pointer variable always points to variables of the same datatype. For example:

*float a;*

*int \*ptr = &a; // ERROR, type mismatch*

While declaring a pointer variable, if it is not assigned to anything then it contains garbage value. Therefore, it is recommended to assign a NULL value to it,

****

A pointer that is assigned a NULL value is called a NULL pointer in C.

int \*ptr = NULL;

**ACCESSING A VARIABLE THROUGH ITS POINTER**

There are a few important operations, which we will do with the help of pointers very frequently. (a) We define a pointer variable, (b) assign the address of a variable to a pointer and (c) finally access the value at the address available in the pointer variable. This is done by using unary operator \* that returns the value of the variable located at the address specified by its operand. The following example makes shows an example of pointers −

*#include <stdio.h>*

*int main () {*

*int var = 20; /\* actual variable declaration \*/*

*int \*ip; /\* pointer variable declaration \*/*

*ip = &var; /\* store address of var in pointer variable\*/*

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

*/\* address stored in pointer variable \*/*

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

*/\* access the value using the pointer \*/*

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

*return 0;*

*}*

*When the above code is compiled and executed, it produces the following result −*

*Address of var variable: bffd8b3c*

*Address stored in ip variable: bffd8b3c*

*Value of \*ip variable: 20*

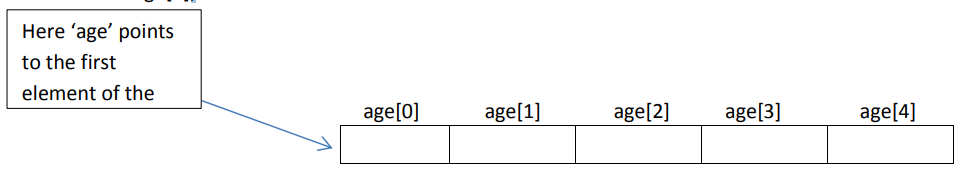
**POINTERS AND ARRAYS**

Arrays and pointers are closely related in C. In fact an array declared as

int A[10];

can be accessed using its pointer representation. The name of the array A is a constant pointer to the first element of the array. So A can be considered a const int\*. Since A is a constant pointer, A = NULL would be an illegal statement. Arrays and pointers are synonymous in terms of how they use to access memory. But, the important difference between them is that, a pointer variable can take different addresses as value whereas, in case of array it is fixed.

Consider the following array: Int age[5];



In C , name of the array always points to the first element of an array. Here, address of first element of an array is &age[0]. Also, age represents the address of the pointer where it is pointing. Hence, &age[0] is equivalent to age. Note, value inside the address &age[0] and address age are equal. Value in address &age[0] is age[0] and value in address age is \*age. Hence, age[0] is equivalent to \*age.

C arrays can be of any type. We define array of ints, chars, doubles etc. We can also define an array of pointers as follows. Here is the code to define an array of n char pointers or an array of strings.

char\* A[n];

each cell in the array A[i] is a char\* and so it can point to a character. Now if you would like to assign a string to each A[i] you can do something like this.

A[i] = malloc(length\_of\_string + 1);

Again this only allocates memory for a string and you still need to copy the characters into this string. So if you are building a dynamic dictionary (n words) you need to allocate memory for n char\*’s and then allocate just the right amount of memory for each string.

In C, you can declare an array and can use pointer to alter the data of an array. This program declares the array of six element and the elements of that array are accessed using pointer, and returns the sum.

*//Program to find the sum of six numbers with arrays and pointers.*

*#include <stdion.h>*

*int main()*

*{*

*int i,class[6],sum=0;*

*printf("Enter 6 numbers:\n");*

*for(i=0;i<6;++i)*

*{*

*scanf("%d",(class+i)); // (class+i) is equivalent to &class[i]*

*sum += \*(class+i); // \*(class+i) is equivalent to class[i]*

*}*

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

*return 0;*

*}*

***Output***

*Enter 6 numbers:*

*2*

*3*

*4*

*5*

*3*

*4*

*Sum=21*

Traditionally, we access the array elements using its index, but this method can be eliminated by using pointers. Pointers make it easy to access each array element.

*#include <stdio.h>*

*int main()*

*{*

*int a[5]={1,2,3,4,5}; //array initialization*

*int \*p; //pointer declaration*

*/\*the ptr points to the first element of the array\*/*

*p=a; /\*We can also type simply ptr==&a[0] \*/*

*printf("Printing the array elements using pointer\n");*

*for(int i=0;i<5;i++) //loop for traversing array elements*

*{*

*printf("\n%x",\*p); //printing array elements*

*p++; //incrementing to the next element, you can also write p=p+1*

*}*

*return 0;*

*}*

***Output:***

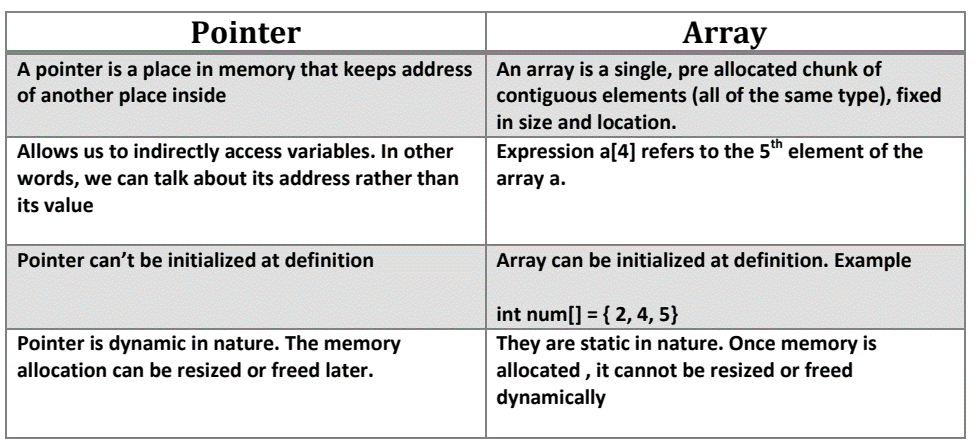
*1*

*2*

*3*

*4*

*5*

****

**ARRAYS OF POINTERS**

An array of pointers is an array that consists of variables of pointer type, which means that the variable is a pointer addressing to some other element. Before we understand the concept of arrays of pointers, let us consider the following example, which makes use of an array of 3 integers:

*#include <stdio.h>*

*const int MAX = 3;*

*int main ()*

*{*

*int var[] = {10, 100, 200};*

*int i;*

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

*{*

*printf("Value of var[%d] = %d\n", i, var[i] );*

*}*

*return 0;*

*}*

*When the above code is compiled and executed, it produces the following result:*

*Value of var[0] = 10*

*Value of var[1] = 100*

*Value of var[2] = 200*

There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer:

int \*ptr[MAX];

This declares ptr as an array of MAX integer pointers. Thus, each element in ptr, now holds a pointer to an int value. Following example makes use of three integers, which will be stored in an array of pointers as follows:

*#include <stdio.h>*

*const int MAX = 3;*

*int main ()*

*{*

*int var[] = {10, 100, 200};*

*int i, \*ptr[MAX];*

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

*{*

*ptr[i] = &var[i]; /\* assign the address of integer. \*/*

*}*

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

*{*

*printf("Value of var[%d] = %d\n", i, \*ptr[i] );*

*}*

*return 0;*

*}*

*When the above code is compiled and executed, it produces the following result:*

*Value of var[0] = 10*

*Value of var[1] = 100*

*Value of var[2] = 200*

You can also use an array of pointers to character to store a list of strings as follows:

*#include <stdio.h>*

*const int MAX = 4;*

*int main ()*

*{*

*char \*names[] = {*

*"Zara Ali", "Hina Ali", "Nuha Ali", "Sara Ali",*

*};*

*int i = 0;*

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

*{*

*printf("Value of names[%d] = %s\n", i, names[i] );*

*}*

*return 0;*

*}*

*When the above code is compiled and executed, it produces the following result:*

*Value of names[0] = Zara Ali*

*Value of names[1] = Hina Ali*

*Value of names[2] = Nuha Ali*

*Value of names[3] = Sara Ali*

**POINTERS AND FUNCTIONS**

As we know that we can create a pointer of any data type such as int, char, float, we can also create a pointer pointing to a function. The code of a function always resides in memory, which means that the function has some address. We can get the address of memory by using the function pointer.

Let's see a simple example.

*#include <stdio.h>*

*int main()*

*{*

*printf("Address of main() function is %p",main);*

*return 0;*

*}*

The above code prints the address of main() function.

**Output:**

*Address of main() function is 0x400027*

In the above output, we observe that the main() function has some address. Therefore, we conclude that every function has some address.

**Declaration of a function pointer**

Till now, we have seen that the functions have addresses, so we can create pointers that can contain these addresses, and hence can point them.

Syntax of function pointer

return type (\*ptr\_name)(type1, type2…);

For example:

int (\*ip) (int);

In the above declaration, \*ip is a pointer that points to a function which returns an int value and accepts an integer value as an argument.

float (\*fp) (float);

In the above declaration, \*fp is a pointer that points to a function that returns a float value and accepts a float value as an argument.

We can observe that the declaration of a function is similar to the declaration of a function pointer except that the pointer is preceded by a '\*'. So, in the above declaration, fp is declared as a function rather than a pointer.

Till now, we have learnt how to declare the function pointer. Our next step is to assign the address of a function to the function pointer.

*float (\*fp) (int , int); // Declaration of a function pointer.*

*float func( int , int ); // Declaration of function.*

*fp = func; // Assigning address of func to the fp pointer.*

In the above declaration, 'fp' pointer contains the address of the 'func' function.

Note: Declaration of a function is necessary before assigning the address of a function to the function pointer.

**Calling a function through a function pointer**

Now, we will see how to call a function using a function pointer.

Suppose we declare a function as given below:

float func(int , int); // Declaration of a function.

Calling an above function using a usual way is given below:

result = func(a , b); // Calling a function using usual ways.

Calling a function using a function pointer is given below:

result = (\*fp)( a , b); // Calling a function using function pointer.

Or

result = fp(a , b); // Calling a function using function pointer, and indirection operator can be removed.

The effect of calling a function by its name or function pointer is the same. If we are using the function pointer, we can omit the indirection operator as we did in the second case. Still, we use the indirection operator as it makes it clear to the user that we are using a function pointer.

Let's understand the function pointer through an example.

*#include <stdio.h>*

*int add(int,int);*

*int main()*

*{*

*int a,b;*

*int (\*ip)(int,int);*

*int result;*

*printf("Enter the values of a and b : ");*

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

*ip=add;*

*result=(\*ip)(a,b);*

*printf("Value after addition is : %d",result);*

*return 0;*

*}*

*int add(int a,int b)*

*{*

*int c=a+b;*

*return c;*

*}*

***Output:***

*Enter the values of a and b: 10*

*20*

*Value after addition is : 30*

**Passing a function's address as an argument to other function**

We can pass the function's address as an argument to other functions in the same way we send other arguments to the function.

Let's understand through an example.

*# include <stdio.h>*

*void func1(void (\*ptr)());*

*void func2();*

*int main()*

*{*

*func1(func2);*

*return 0;*

*}*

*void func1(void (\*ptr)())*

*{*

*printf("Function1 is called");*

*(\*ptr)();*

*}*

*void func2()*

*{*

*printf("\nFunction2 is called");*

*}*

In the above code, we have created two functions, i.e., func1() and func2(). The func1() function contains the function pointer as an argument. In the main() method, the func1() method is called in which we pass the address of func2. When func1() function is called, 'ptr' contains the address of 'func2'. Inside the func1() function, we call the func2() function by dereferencing the pointer 'ptr' as it contains the address of func2.

***Output***

*Function1 is called*

*Function2 is called*

**ADVANTAGES OF POINTERS IN C**

* Pointers provide direct access to memory
* Pointers provide a way to return more than one value to the functions
* Reduces the storage space and complexity of the program
* Reduces the execution time of the program
* Provides an alternate way to access array elements
* Pointers allows us to resize the dynamically allocated memory block.
* Addresses of objects can be extracted using pointers
* Pointers provide an efficient way for accessing the elements of an array structure.
* Pointers are used for dynamic memory allocation as well as deallocation.
* Pointers are used to form complex data structures such as linked list, graph, tree, etc.

**DISADVANTAGES OF POINTERS IN C**

* Pointers are a little complex to understand.
* Pointers can lead to various errors such as segmentation faults or can access a memory location which is not required at all.
* If an incorrect value is provided to a pointer, it may cause memory corruption.
* Pointers are also responsible for memory leakage.
* Pointers are comparatively slower than that of the variables.
* Programmers find it very difficult to work with the pointers; therefore it is programmer’s responsibility to manipulate a pointer carefully.

**UNIT 5**

**STRUCTURES, UNIONS AND FILE MANAGEMENT IN C**

**USER DEFINED DATA TYPES**

The C language allows a feature known as the type definition. This basically allows a programmer to provide a definition to an identifier that will represent a data type which already exists in a program. The C program consists of the following types of user defined datatypes:

1. Structures
2. Union
3. Typedef
4. Enum

**1.STRUCTURES**

We use the structure for organising a group of various data items into one single entity – for grouping those data items that are related but might belong to different data types.

The structure data types are related (usually), such as the different sets of information regarding a person, an account, or a part, etc. Every data item present in a structure is known as a member. These members are sometimes also known as fields.

We use the struct keyword for creating a structure. The primary advantage of using a structure is that it becomes very easy for a person to access the members. It is because the allocation of all the members that belong to a specific structure is in the continuous memory. Thus, structure helps in minimising the memory access time for any program.

Generally, we declare a structure as follows:

struct name\_of\_tag {

data\_typea svar\_a;

data\_typeb svar\_b;

data\_typec svar\_c;

….

data\_typez svar\_z;

};

The beginning of the declaration of a structure typically begins with the struct keyword. We must enclose the declaration of all the members of the structure in braces. Here, the tag\_name refers to the identifier that would specify the name of the new structure.

The structure declaration reserves no storage space – but the structure definition would lead to the creation of the structure variables.

We can define the structure variables as follows:

struct tag\_name svar\_a, svar\_b …svar\_z;

An example would help us understand this better:

*struct sample {*

*int p, q;*

*float r, s;*

*char t, u;*

*};*

*struct sample vA, vB, vC; //definition of structure*

**2.UNION**

A union is basically a collection of various different data types that are present in the C language, but not very similar to each other. The union mainly allows the storage of those data types that are different from each other in the very same memory location. Any user will be able to define a union using various members, but at any given time, just a single member would be able to contain the given value. The unions provide us with an efficient way in which we can use the very same memory location for multiple purposes.

Note that the structures and unions are very similar to each other. The only difference between them is that we can access just a single member of the Union at any given time. It is because the creation of memory is only for one member that has the biggest size (or the highest number of bytes).

We declare the union using the union keyword, and we can access all the members of a Union using the (.) dot operator.

The definition and declaration of a union would go like this:

*union tag\_name {*

*data\_typea uvar\_a;*

*data\_typeb uvar\_b;*

*……*

*data\_typez uvar\_z;*

*};*

union tag\_name uvar\_a, uvar\_b,….uvar\_z;

We will understand this better with the help of an example,

*union sample {*

*int duration;*

*float cost;*

*char keyword;*

*};*

*union sample x;*

In the example mentioned above, the allocation of about 4 bytes of memory occurs to the union variable x. Thus, we can access the members as x.duration, x.cost, x.keyword, etc., but we can only access one of the members at any given time since the very same memory exists for all three members here.

**3.TYPEDEF**

We use the keyword typedef for creating an alias (a new name) for a data type that already exists. The typedef won’t create any new form of data type. When using the typedef data type, the syntax would be:

typedef existing\_data\_type new\_type;

Let us consider the example as follows:

*#include <stdio.h>*

*void main()*

*{*

*typedef int Lessons; //statement-1*

*Lessons x = 17;*

*printf(“Given value =%d\n”, x);*

*}*

Let us look at the statement here. We have used the typedef statement for creating Lessons as an alias for int. After we use this statement, Lessons will become the new name for the int data type in the program given above. Also, the variables that are declared in the form of Lessons data type will behave like the int variables in all the practical implications of the program.

**4.ENUM**

The enum refers to a keyword that we use for creating an enumerated data type. The enum is basically a special data type (enumerated data type) that consists of a set of various named values – known as members or elements. We mainly use it for assigning different names to the integral constants. This way, the program becomes much more readable.

Here is the format that we use for creating the enum type:

enum identifier (value\_a, value\_b, …. , value\_z);

The enumerated data types basically allow a user to create symbolic names of their own for a list of all the constants that are related to each other.

For instance, you can create the enumerated data type for the true and false conditions this way:

*enum Boolean {*

*true,*

*false*

*};*

In case we don’t assign values explicitly to the enum names; the compiler, by default, would assign the values that start from 0.

In this case, true and false would be automatically assigned to 1 and 0 respectively. In simpler words, the first field of the enum value here will be replaced by 1, and then the next field will be replaced with 0, and so on.

Let us take a look at an example to display how to use the enum data value.

*#include <stdio.h>*

*void main()*

*{*

*enum week {MON = 1, TUE, WED, THURS, FRI, SAT, SUN};*

*enum week off = FRI;*

*printf(“Week Off = %d”, birthday);*

*}*

Here, the field name MON will be assigned with the value 1. Thus, the next field name TUE will be assigned with the value 2 automatically, and so on.

The program mentioned above will print the following output:

Week Off = 5

**STRUCTURES**

C arrays allow you to define type of variables that can hold several data items of the same kind but structure is another user defined data type available in C programming, which allows you to combine data items of different kinds.

Structures are used to represent a record, suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

• Title

• Author

• Subject

• Book ID

**DEFINITION**

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member for your program. The format of the struct statement is this:

*struct [structure tag]*

*{*

*member definition;*

*member definition;*

*...*

*member definition;*

*} [one or more structure variables];*

The structure tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure:

*struct Books*

*{*

*char title[50];*

*char author[50];*

*char subject[100];*

*int book\_id;*

*} book;*

**STURCTURE INITIALIZATION**

C language supports multiple ways to initialize a structure variable. You can use any of the initialization method to initialize your structure.

1. Initialize using dot operator
2. Value initialized structure variable
3. Variant of value initialized structure variable

1**.Initialize structure using dot operator**

In C, we initialize or access a structure variable either through dot . or arrow -> operator. This is the most easiest way to initialize or access a structure.

Example:

*// Declare structure variable*

*struct student stu1;*

*// Initialize structure members*

*stu1.name = "Pankaj";*

*stu1.roll = 12;*

*stu1.marks = 79.5f;*

**2.Value initialized structure variable**

The above method is easy and straightforward to initialize a structure variable. However, C language also supports value initialization for structure variable. Means, you can initialize a structure to some default value during its variable declaration.

Example:

*// Declare and initialize structure variable*

*struct student stu1 = { "Pankaj", 12, 79.5f };*

Note: The values for the value initialized structure should match the order in which structure members are declared.

Invalid initialization:

*// Declare and initialize structure variable*

*struct student stu1 = { 12, "Pankaj", 79.5f };*

The above code will throw compilation error. Since the order of member type in structure is character array, integer finally float. But, we aren't initializing the structure variable in the same order.

**3.Variant of value initialized structure variable**

In addition, C language also supports flexibility to initialize structure members in any order.

This approach is an extension of the other initialization method. Here, you can specify member name along with the value.

Example:

*// Declare and initialize structure variable*

*struct student stu1 = {*

*.roll = 12,*

*.name = "Pankaj",*

*.marks = 79.5f*

*};*

**STRUCTURE DEFAULT INITIALIZATION**

Default initialization of a variable considered as good programming practice. However, C doesn't support any programming construct for default structure initialization. You manually need to initialize all fields to 0 or NULL.

*Example:*

*// Define macro for default structure initialization*

*#define NEW\_STUDENT { "", 0, 0.0f }*

*// Default initialization of structure variable*

*struct student stu1 = NEW\_STUDENT;*

**ARRAYS WITHIN STRUCTURES**

In C programming, we can have structure members of type arrays. Any structure having an array as a structure member is known as an array within the structure. We can have as many members of the type array as we want in C structure.

To access each element from an array within a structure, we write structure variables followed by dot (.) and then array name along with index.

The following example shows the usage of array within a structure in C programming −

*struct student*

*{*

*char name[20]; /\* This is array within structure \*/*

*int roll;*

*float marks[5]; /\* This is array within structure \*/*

*};*

**Example**

Following is the C program to demonstrate the use of an array within a structure −

*#include <stdio.h>*

*// Declaration of the structure candidate*

*struct candidate {*

*int roll\_no;*

*char grade;*

*// Array within the structure*

*float marks[4];*

*};*

*// Function to displays the content of*

*// the structure variables*

*void display(struct candidate a1){*

*printf("Roll number : %d\n", a1.roll\_no);*

*printf("Grade : %c\n", a1.grade);*

*printf("Marks secured:\n");*

*int i;*

*int len = sizeof(a1.marks) / sizeof(float);*

*// Accessing the contents of the*

*// array within the structure*

*for (i = 0; i < len; i++) {*

*printf("Subject %d : %.2f\n",*

*i + 1, a1.marks[i]);*

*}*

*}*

*// Driver Code*

*int main(){*

*// Initialize a structure*

*struct candidate A= { 1, 'A', { 98.5, 77, 89, 78.5 } };*

*// Function to display structure*

*display(A);*

*return 0;*

*}*

*Output*

*When the above program is executed, it produces the following result −*

*Roll number : 1*

*Grade : A*

*Marks secured:*

*Subject 1 : 98.50*

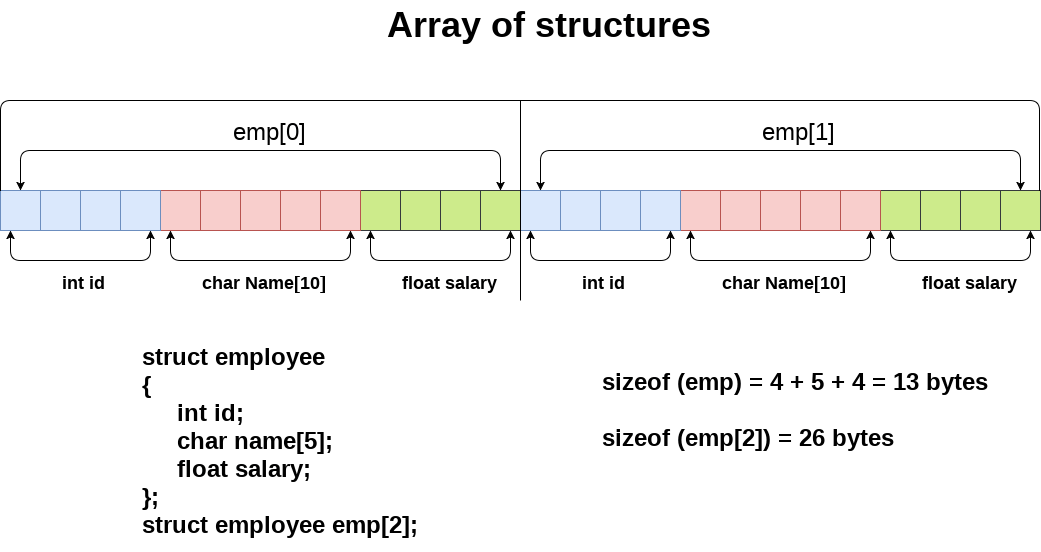
*Subject 2 : 77.00*

*Subject 3 : 89.00*

*Subject 4 : 78.50*

**ARRAYS OF STRUCTURES**

An array of structure in C programming is a collection of different datatype variables, grouped together under a single name.



**Example**

The most common use of structure in C programming is an array of structures. To declare an array of structure, first the structure must be defined and then an array variable of that type should be defined.

For Example − struct book b[10]; //10 elements in an array of structures of type ‘book’

Example

The following program shows the usage of array of structures.

*#include <stdio.h>*

*#include <string.h>*

*struct student{*

*int id;*

*char name[30];*

*float percentage;*

*};*

*int main(){*

*int i;*

*struct student record[2];*

*// 1st student's record*

*record[0].id=1;*

*strcpy(record[0].name, "Bhanu");*

*record[0].percentage = 86.5;*

*// 2nd student's record*

*record[1].id=2;*

*strcpy(record[1].name, "Priya");*

*record[1].percentage = 90.5;*

*// 3rd student's record*

*record[2].id=3;*

*strcpy(record[2].name, "Hari");*

*record[2].percentage = 81.5;*

*for(i=0; i<3; i++){*

*printf(" Records of STUDENT : %d \n", i+1);*

*printf(" Id is: %d \n", record[i].id);*

*printf(" Name is: %s \n", record[i].name);*

*printf(" Percentage is: %f\n\n",record[i].percentage);*

*}*

*return 0;*

*}*

***Output***

*When the above program is executed, it produces the following result −*

*Records of STUDENT : 1*

*Id is: 1*

*Name is: Bhanu*

*Percentage is: 86.500000*

*Records of STUDENT : 2*

*Id is: 2*

*Name is: Priya*

*Percentage is: 90.500000*

*Records of STUDENT : 3*

*Id is: 3*

*Name is: Hari*

*Percentage is: 81.500000*

**STRUCTURES WITHIN STRUCTURES**

Structure inside another structure is called nested structure. One structure can be declared inside another structure in the same way structure members are declared inside a structure.

Consider the following example, all the items comes under allowances can be grouped together and declared under a sub – structure as shown below.

*stuct emp{*

*int eno;*

*char ename[30];*

*float sal;*

*struct allowance{*

*float da;*

*float hra;*

*float ea;*

*}a;*

*}e;*

The inner most member in a nested structure can be accessed by changing all the concerned structure variables (from outer most to inner most) with the member using dot operator.

**Program**

Following program is to demonstrate nested structure (structure within the structure) –

*#include<stdio.h>*

*struct address {*

*char city[20];*

*int pin;*

*char phone[14];*

*};*

*struct employee {*

*char name[20];*

*struct address add;*

*};*

*void main ()*

*{*

*struct employee emp;*

*printf("Enter employee information?\nName, city, pincode,phone no:\n");*

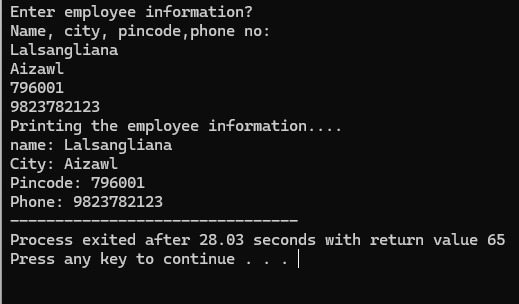
*scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.phone);*

*printf("Printing the employee information....\n");*

*printf("name: %s\nCity: %s\nPincode: %d\nPhone: %s",emp.name,emp.add.city, emp.add.pin,emp.add.phone);*

*}*

**Output:**



**STRUCTURES AND FUNCTIONS**

C allows programmers to pass a single or entire structure information to or from a function. A structure information can be passed as a function arguments. The structure variable may be passed as a value or reference. The function will return the value by using the return statement.

*#include <stdio.h>*

*int add(int, int) ; //function declaration*

*int main()*

*{*

*//structures declartion*

*struct addition{*

*int a, b;*

*int c;*

*}sum;*

*printf("Enter the value of a : ");*

*scanf("%d",&sum.a);*

*printf("\nEnter the value of b : ");*

*scanf("%d",&sum.b);*

*sum.c = add(sum. a, sum.b); //passing structure members as arguments to function*

*printf("\nThe sum of two value are : ");*

*printf("%d ", sum.c);*

*return 0;*

*}*

*//Function definition*

*int add(int x, int y)*

*{*

*int sum1;*

*sum1 = x + y;*

*return(sum1);*

*}*

***Output***

*Enter the value of a 10*

*Enter the value of b 20*

*The sum of two values 30*

The structure addition have three variables( a, b, c) which are also known as structure members. The variables are reads through scanf function. The next statement illustrates that the structure members can be passed to the function add. The add function defined to perform addition operation between two values.

**C program - Passing The Entire Structure To Functions**

*#include <stdio.h>*

*//structures declaration*

*typedef struct {*

*int a, b;*

*int c;*

*}sum;*

*void add(sum) ; //function declaration with struct type sum*

*int main()*

*{*

*sum s1;*

*printf("Enter the value of a : ");*

*scanf("%d",&s1.a);*

*printf("\nEnter the value of b : ");*

*scanf("%d",&s1.b);*

*add(s1); //passing entire structure as an argument to function*

*return 0;*

*}*

*//Function Definition*

*void add(sum s)*

*{*

*int sum1;*

*sum1 = s.a + s.b;*

*printf("\nThe sum of two values are :%d ", sum1);*

*}*

***Output***

*Enter the value of a 10*

*Enter the value of b 20*

*The sum of two values 30*

The program uses the keyword typedef to create the structure type sum. The variable "s1" is declared as structure type sum.

**PASSING STRUCTURES USING POINTERS**

The Call by value method is inefficient to pass the large structures to functions. C provides the efficient method, known as call by reference to pass large structures to functions. Call by reference method is achieved by passing pointers to functions. Pointer is a variable, used to hold or store the address of another variable.

*C program - passing Structures By Using Pointers*

*#include <stdio.h>*

*//Structure declartion*

*struct employee {*

*char name[40];*

*int empid;*

*int experience;*

*}emp;*

*void displaydetails(struct employee\*); //function declaration*

*int main()*

*{*

*struct employee \*empptr; //pointer declaration*

*empptr = &emp; //initial*

*printf("\nEnter the name of the Employee : ");*

*scanf("%s", empptr->name);*

*printf("\nEnter the Employee Id : ");*

*scanf("%d",&empptr->empid);*

*printf("\nEnter Experience of the Employee : ");*

*scanf("%d",&empptr->experience);*

*displaydetails(empptr);*

*return 0;*

*}*

*//Function Definition*

*void displaydetails(struct employee \*empptr)*

*{*

*printf("\n---------Details List--------- \n ");*

*printf("Employee Name : %s",empptr->name);*

*printf("\nEmployee ID : %d ",empptr->empid);*

*printf("\nEmployee Experience : %d ",empptr->experience);*

*}*

***Output***

*Enter the name of the Employee : Jiju*

*Enter the Employee Id : 16*

*Enter Experience of the Employee : 3*

*---------Details List---------*

*Employee Name : Jiju*

*Employee Id : 16*

*Employee Experience : 3*

The structure employee has the variable "emp". The pointer variable is declared inside the main function and initialized as &emp. Now the pointer empptr hold the address of structure. The employee structure members can be accessed by using the "empptr->membername". The calling function displaydetails has a pointer "empptr" as an argument. Finally the function will display the details of employee.

**POINTERS AND STRUCTURES**

Pointer to structure holds the add of the entire structure.

It is used to create complex data structures such as linked lists, trees, graphs and so on.

The members of the structure can be accessed using a special operator called as an arrow operator ( -> ).

**Declaration**

Following is the declaration for pointers to structures in C programming −

struct tagname \*ptr;

For example : struct student \*s;

**Accessing**

It is explained below how to access the pointers to structures.

Ptr-> membername;

For example − s->sno, s->sname, s->marks;

**Example Program**

The following program shows the usage of pointers to structures −

*#include<stdio.h>*

*struct student{*

*int sno;*

*char sname[30];*

*float marks;*

*};*

*main ( ){*

*struct student s;*

*struct student \*st;*

*printf("enter sno, sname, marks:");*

*scanf ("%d%s%f", & s.sno, s.sname, &s. marks);*

*st = &s;*

*printf ("details of the student are");*

*printf ("Serial No = %d\n", st ->sno);*

*printf ("name = %s\n", st->sname);*

*printf ("marks =%f\n", st ->marks);*

*getch ( );*

*}*

**Output**

Let us run the above program that will produce the following result −

*enter sno, sname, marks:1 Lucky 98*

*details of the student are:*

*Number = 1*

*name = Lucky*

*marks =98.000000*

**Example 2**

Consider another example which explains the functioning of pointers to structures.

*#include<stdio.h>*

*struct person{*

*int age;*

*float weight;*

*};*

*int main(){*

*struct person \*personPtr, person1;*

*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);*

*return 0;*

*}*

**Output**

Let us run the above program that will produce the following result −

*Enter age: 45*

*Enter weight: 60*

*Displaying:*

*Age: 45*

*weight: 60.000000*

**SELF REFERENTIAL STRUCTURES**

A structure can have members which point to a structure variable of the same type. These types of structures are called self referential structures and are widely used in dynamic data structures like trees, linked list, etc. The following is a definition of a self referential structure.

*struct node*

*{*

*int data;*

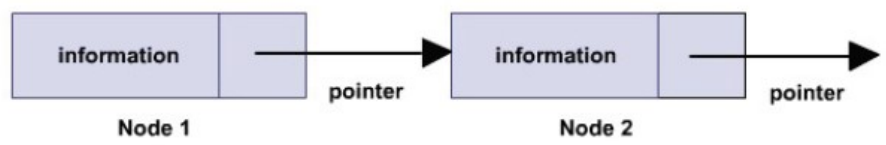
*struct node \*next;*

*};*

Here, next is a pointer to a struct node variable.

It should be remembered that a pointer to a structure is similar to a pointer to any other

variable. A self referential data structure is essentially a structure definition which includes at least one member that is a pointer to the structure of its own kind.



Such self referential structures are very useful in applications that involve linked data structures, such as lists and trees. Unlike a static data structure such as array where the number of elements that can be inserted in the array is limited by the size of the array, a self referential structure can dynamically be expanded or contracted.

Operations like insertion or deletion of nodes in a self referential structure involve simple

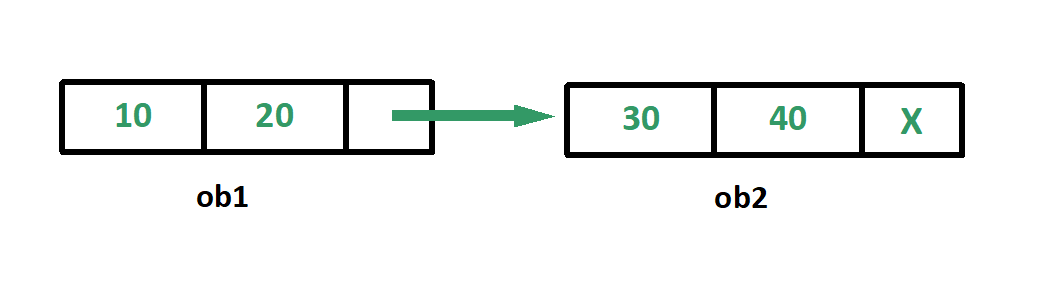
and straight forward alteration of pointers.

**Types of Self Referential Structures**

1. Self Referential Structure with Single Link
2. Self Referential Structure with Multiple Links

**1.Self Referential Structure with Single Link:**

These structures can have only one self-pointer as their member. The following example will show us how to connect the objects of a self-referential structure with the single link and access the corresponding data members. The connection formed is shown in the following figure.



**Sample Code**

*#include <stdio.h>*

*struct node {*

*int data1;*

*int data2;*

*struct node\* link;*

*};*

*int main() {*

*struct node ob1; // Node1*

*ob1.link = NULL;*

*ob1.data1 = 10;*

*ob1.data2 = 20;*

*struct node ob2; // Node2*

*ob2.link = NULL;*

*ob2.data1 = 30;*

*ob2.data2 = 40;*

*printf("\ndata of ob1");*

*printf("\n%d", ob1.data1);*

*printf("\n%d", ob1.data2);*

*// Linking ob1 and ob2*

*ob1.link = &ob2;*

*//Accessing data members of ob2 using ob1*

*printf("\ndata of ob2");*

*printf("\n%d", ob1.link->data1);*

*printf("\n%d", ob1.link->data2);*

*return 0;*

*}*

***Output:***

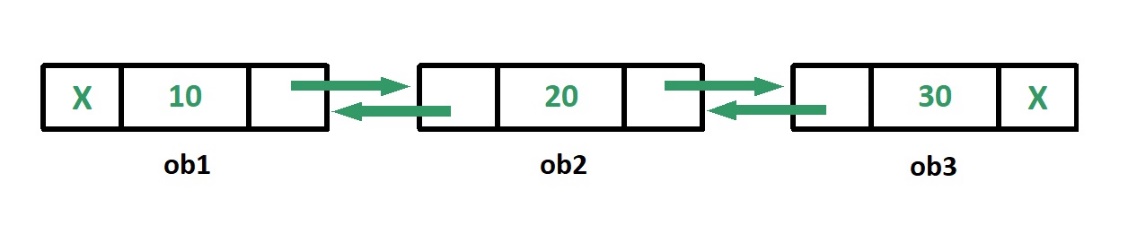
*30*

*40*

**2.Self Referential Structure with Multiple Links:**

Self referential structures with multiple links can have more than one self-pointers. Many complicated data structures can be easily constructed using these structures. Such structures can easily connect to more than one nodes at a time. The following example shows one such structure with more than one links.

The connections made in the above example can be understood using the following figure.



**Sample Code**

*#include <stdio.h>*

*struct node {*

*int data;*

*struct node\* prev\_link;*

*struct node\* next\_link;*

*};*

*int main()*

*{*

*struct node ob1; // Node1*

*ob1.prev\_link = NULL;*

*ob1.next\_link = NULL;*

*ob1.data = 10;*

*struct node ob2; // Node2*

*ob2.prev\_link = NULL;*

*ob2.next\_link = NULL;*

*ob2.data = 20;*

*struct node ob3; // Node3*

*ob3.prev\_link = NULL;*

*ob3.next\_link = NULL;*

*ob3.data = 30;*

*// Forward links*

*ob1.next\_link = &ob2;*

*ob2.next\_link = &ob3;*

*// Backward links*

*ob2.prev\_link = &ob1;*

*ob3.prev\_link = &ob2;*

*// Accessing data of ob1, ob2 and ob3 by ob1*

*printf("%d\t", ob1.data);*

*printf("%d\t", ob1.next\_link->data);*

*printf("%d\n", ob1.next\_link->next\_link->data);*

*// Accessing data of ob1, ob2 and ob3 by ob2*

*printf("%d\t", ob2.prev\_link->data);*

*printf("%d\t", ob2.data);*

*printf("%d\n", ob2.next\_link->data);*

*// Accessing data of ob1, ob2 and ob3 by ob3*

*printf("%d\t", ob3.prev\_link->prev\_link->data);*

*printf("%d\t", ob3.prev\_link->data);*

*printf("%d", ob3.data);*

*return 0;*

*}*

***Output:***

*10 20 30*

*10 20 30*

*10 20 30*

In the above example we can see that ‘ob1’, ‘ob2’ and ‘ob3’ are three objects of the self referential structure ‘node’. And they are connected using their links in such a way that any of them can easily access each other’s data. This is the beauty of the self referential structures. The connections can be manipulated according to the requirements of the programmer.

**Applications:**

Self referential structures are very useful in creation of other complex data structures like:

1. Linked Lists
2. Stacks
3. Queues
4. Trees
5. Graphs etc

**UNION**

A union is a special data type available in C that enables you to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multi-purpose.

**Defining a Union**

To define a union, you must use the union statement in very similar was as you did while defining structure. The union statement defines a new data type, with more than one member for your program. The format of the union statement is as follows:

union [union tag]

{

member definition;

member definition;

...

member definition;

} [one or more union variables];

The union tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data which has the three members i, f, and str:

union Data

{

int i;

float f;

char str[20];

} data;

Now, a variable of Data type can store an integer, a floating-point number, or a string of characters. This means that a single variable i.e. same memory location can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in above example Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by character string. Following is the example which will display total memory size occupied by the above union:

*#include <stdio.h>*

*#include <conio.h>*

*union Data*

*{*

*int i;*

*float f;*

*char str[20];*

*};*

*int main( )*

*{*

*union Data data;*

*printf( "Memory size occupied by data : %d\n", sizeof(data));*

*return 0;*

*}*

*When the above code is compiled and executed, it produces the following result:*

*Memory size occupied by data : 20*

**Accessing Union Members**

To access any member of a union, we use the member access operator (.). The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use union keyword to define variables of union type. Following is the example to explain usage of union:

*#include <stdio.h>*

*#include <string.h>*

*union Data*

*{*

*int i;*

*float f;*

*char str[20];*

*};*

*int main( )*

*{*

*union Data data;*

*data.i = 10;*

*data.f = 220.5;*

*strcpy( data.str, "C Programming");*

*printf( "data.i : %d\n", data.i);*

*printf( "data.f : %f\n", data.f);*

*printf( "data.str : %s\n", data.str);*

*return 0;*

*}*

*When the above code is compiled and executed, it produces the following result:*

*data.i : 1917853763*

*data.f : 4122360580327794860452759994368.000000*

*data.str : C Programming*

Here, we can see that values of i and f members of union got corrupted because final value assigned to the variable has occupied the memory location and this is the reason that the value if str member is getting printed very well. Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having union:

*#include <stdio.h>*

*#include <string.h>*

*union Data*

*{*

*int i;*

*float f;*

*char str[20];*

*};*

*int main( )*

*{*

*union Data data;*

*data.i = 10;*

*printf( "data.i : %d\n", data.i);*

*data.f = 220.5;*

*printf( "data.f : %f\n", data.f);*

*strcpy( data.str, "C Programming");*

*printf( "data.str : %s\n", data.str);*

*return 0;*

*}*

When the above code is compiled and executed, it produces the following result:

*data.i : 10*

*data.f : 220.500000*

*data.str : C Programming*

Here, all the members are getting printed very well because one member is being used at a time.

**TYPEDEF**

The C programming language provides a keyword called typedef, which you can use to give a type a new name. Following is an example to define a term BYTE for one-byte numbers:

typedef unsigned char BYTE;

After this type definitions, the identifier BYTE can be used as an abbreviation for the

type unsigned char, for example:

BYTE b1, b2;

By convention, uppercase letters are used for these definitions to remind the user that the

type name is really a symbolic abbreviation, but you can use lowercase, as follows:

typedef unsigned char byte;

You can use typedef to give a name to user defined data type as well. For example you can

use typedef with structure to define a new data type and then use that data type to define

structure variables directly as follows:

*#include <stdio.h>*

*#include <string.h>*

*typedef struct Books*

*{*

*char title[50];*

*char author[50];*

*char subject[100];*

*int book\_id;*

*} Book;*

*int main( )*

*{*

*Book book;*

*strcpy( book.title, "C Programming");*

*strcpy( book.author, "Nuha Ali");*

*strcpy( book.subject, "C Programming Tutorial"*

*book.book\_id = 6495407;*

*printf( "Book title : %s\n", book.title);*

*printf( "Book author : %s\n", book.author);*

*printf( "Book subject : %s\n", book.subject);*

*printf( "Book book\_id : %d\n", book.book\_id);*

*return 0;*

*}*

When the above code is compiled and executed, it produces the following result:

*Book title : C Programming*

*Book author : Nuha Ali*

*Book subject : C Programming Tutorial*

*Book book\_id : 6495407*

**TYPEDEF VS #DEFINE**

The #define is a C-directive which is also used to define the aliases for various data types

similar to typedef but with three differences:

1. The typedef is limited to giving symbolic names to types only where as #define can be

used to define alias for values as well, like you can define 1 as ONE etc.

1. The typedef interpretation is performed by the compiler where as #define statements

are processed by the pre-processor.

1. The typedef is used for renaming a new name for the existinf keyword while #define is used to declare symbolic constant

Following is a simplest usage of #define:

*#include <stdio.h>*

*#define TRUE 1*

*#define FALSE 0*

*int main( )*

*{*

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

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

*return 0;*

*}*

When the above code is compiled and executed, it produces the following result:

*Value of TRUE : 1*

*Value of FALSE : 0*

**FILE MANAGEMENT IN C**

**FILE CONCEPT**

A file represents a sequence of bytes, does not matter if it is a text file or 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.

**INPUT/OUTPUT OPERATIONS ON FILES**

File in C programming language, can be anything from a disk file to a device. C language provide support for opening a file, closing a file, reading data from a file and writing data to a file through a set of standard library functions.

C file handling is designed to work with a number of devices and file formats like printer, screen, keyboard, disk file etc. C file system transforms different devices into a abstract logical device called stream. All streams behaves same irrespective of devices and files. It provides an abstract consistent interface for C program to interact by hiding the details of the device specific implementations. Disk file read function can be used to read data from keyboard as well.

**FILE Pointer in C**

A file pointer is a pointer to a structure of type FILE. A file pointer is associated with stream and manages all Input and Output operations of that stream. The FILE structure contains information which is required to perform any file I/O operations like name of the file, current location of position indicator in file, data transfer bugger, status flag etc.  
A file pointer is a bridge between c program and file/stream. We can declare a FILE pointer as follows:

**FILE \*file;**

**Facts about FILE Structure**

*To perform any I/O operation on file, we need FILE pointer.*

*FILE Structure is defined in stdio.h header file.*

*FILE Structure contains necessary information about a stream, which is required to perform I/O operation in that stream.*

*FILE pointer is an interface for our programs to interact with files.*

**OPENING A FILE IN C**

The stdio.h library function fopen() is used to create a new file or open an exiting file.

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

Function fopen() opens the file whose name is given in the filename argument and associate it with a stream and return a FILE pointer to be used in any future I/O operations on this stream. The fopen function opens a stream in a particular mode, which defines the operations that are allowed on the stream.

**Various Modes of Opening a File**

| Mode | Description |
| --- | --- |
| "r" | To read a file. Opens a file for reading. The file must exist. |
| "w" | To write on a file. Creates an empty file for writing. If a file already exists with same name, its content is removed and the file is considered as a new empty file. |
| "a" | To append data at the end of file. The file is created if it does not exist. |
| "r+" | To read and write on an existing file. Opens a file to update both reading and writing. The file must exist. |
| "w+" | To create a new file for reading and writing. |
| "a+" | To read and append data on a file. |

**C Program to Open a File using fopen Function**

The following program shows the use of fopen function to open a text file in read mode.

*#include <stdio.h>*

*int main(){*

*FILE \*file;*

*int ch;*

*/\* Open a file for reading \*/*

*file = fopen("textFile.txt","r");*

*if(file == NULL){*

*perror("Error: Unable to open a file");*

*} else {*

*while(!feof(file)){*

*ch = fgetc(file);*

*printf("%c", ch);*

*}*

*fclose(file);*

*}*

*return(0);*

*}*

**Output**

fopen C Standard library function

**CLOSING A FILE IN C**

The stdio.h library function fclose() is used to close a stream that was opened by fopen() function. Before closing a stream it flushes it's buffer.

int fclose(FILE \*stream);

Function fclose closes a stream whose file pointer is passed as argument. If a call to fclose is successful it returns zero, otherwise EOF is returned.

**C Program to Close a File using fclose Function**

The following program shows the use of **fclose function** to close a file(stream) after writing a sentence in it.

|  |
| --- |
| *#include <stdio.h>*  *int main (){*  *FILE \* pFile;*  */\* Creates a new file \*/*  *pFile = fopen ("TextFile.txt","w");*  */\* Write a sentence in a file \*/*  *fprintf (pFile, "fclose C standard library function");*  */\* Close a file \*/*  *fclose (pFile);*  *return 0;*  *}* |

**Output**

fclose C standard library function

**READING FROM A FILE IN C**

The stdio.h standard library provides a number of functions to read data from a stream.

Functions to read a single character from a file

| **Function** | **Description** |
| --- | --- |
| [fgetc()](http://www.techcrashcourse.com/2015/08/fgetc-stdio-c-library-function.html) | Reads a character from the given stream. |
| [getc()](http://www.techcrashcourse.com/2015/08/getc-stdio-c-library-function.html) | Reads a single character from given stream. |
| [getchar()](http://www.techcrashcourse.com/2015/08/getchar-stdio-c-library-function.html) | Reads a single character from stdin stream. |

**Functions to Read a String from a File**

| Function | Description |
| --- | --- |
| [fgets()](http://www.techcrashcourse.com/2015/08/fgets-stdio-c-library-function.html) | Reads a line from given stream and stores it into a character array. |
| [gets()](https://www.techcrashcourse.com/2015/08/gets-stdio-c-library-function.html) | Reads a line from stdin and stores it into given character array. |

**Functions to Read Formatted Data from a File**

| Function | Description |
| --- | --- |
| [fscanf()](https://www.techcrashcourse.com/2015/08/fscanf-stdio-c-library-function.html) | Read formatted data from given stream. |
| [scanf()](http://www.techcrashcourse.com/2015/08/scanf-stdio-c-library-function.html) | Reads formatted data from stdin. |

**C Program to read formatted data from a file using fscanf function**

The following program shows the use of fscanf function to read formatted data from a stream.

|  |
| --- |
| *#include <stdio.h>*  *int main()*  *{*  *char string[50];*  *int val;*  *float fval;*  *FILE \*file;*  *file = fopen ("textFile.txt", "w+");*  *fprintf(file, "%d %f %s", 5, 5.5, "TechCrashCourse.com");*  *rewind(file);*  *fscanf(file, "%d %f %s", &val, &fval, string);*  *printf("Integer : %d\n", val);*  *printf("Floating point number : %f\n", fval);*  *printf("String : %s\n", string);*  *fclose(file);*  *return(0);*  *}* |

*Output*

*Integer : 5*

*Floating point number : 5.500000*

*String : TechCrashCourse.com*

**WRITING TO A FILE IN C**

The stdio.h standard library provides a number of functions to write data on a stream.

Functions to write a single character on a file

| Function | Description |
| --- | --- |
| [fputc()](https://www.techcrashcourse.com/2015/08/fputc-stdio-c-library-function.html) | Writes a character to the given stream. |
| [putc()](https://www.techcrashcourse.com/2015/08/putc-stdio-c-library-function.html) | Writes a character to the given stream. |
| [putchar()](http://www.techcrashcourse.com/2015/08/putchar-stdio-c-library-function.html) | Writes a character to stdout stream. |

**Functions to write a string on a file**

| Function | Description |
| --- | --- |
| [fputs()](https://www.techcrashcourse.com/2015/08/fputs-stdio-c-library-function.html) | Writes a string to the given stream excluding the null terminating character. |
| [puts()](https://www.techcrashcourse.com/2015/08/puts-stdio-c-library-function.html) | Writes a string to stdout stream excluding null terminating character. |

**Functions to write formatted data on a file**

| Function | Description |
| --- | --- |
| [fprintf()](https://www.techcrashcourse.com/2015/08/fprintf-stdio-c-library-function.html) | Writes formatted output to a stream. |
| [printf()](https://www.techcrashcourse.com/2015/08/printf-stdio-c-library-function.html) | Print formatted data to stdout. |
| [vfprintf()](https://www.techcrashcourse.com/2015/08/vfprintf-stdio-c-library-function.html) | Writes formatted data to a stream using an argument list. |
| [vprintf()](https://www.techcrashcourse.com/2015/08/vprintf-stdio-c-library-function.html) | Print formatted data to stdout using an argument list. |
| [vsprintf()](https://www.techcrashcourse.com/2015/08/vsprintf-stdio-c-library-function.html) | Writes formatted data to a string using an argument list. |

**C Program to write formatted data on a file**

|  |
| --- |
| *#include <stdio.h>*  *int main (){*  *int c, counter = 0;*  *FILE \*file;*  *char \*string = "fprintf C Standard Library function";*    */\* Create a temporary file \*/*  *file = fopen("textFile.txt", "w");*  *fprintf(file, "%s", string);*  *fclose(file);*    */\* Opening textFile.txt and printing it's content \*/*  *file = fopen("textFile.txt", "r");*  *while(!feof(file)){*  *c = fgetc(file);*  *printf("%c", c);*  *}*  *return(0);*  *}* |

**Output**

fprintf C Standard Library function

**STANDARD LIBRARY FUNCTION FOR FILE HANDLING IN C**

There are many file handling functions defined in stdio.h header file. Here is the list of frequently used file handling functions:

| Function | Description |
| --- | --- |
| [clearerr()](https://www.techcrashcourse.com/2015/08/clearerr-stdio-c-library-function.html) | Clears error indicators associated with a given stream. |
| [fclose()](https://www.techcrashcourse.com/2015/08/fclose-stdio-c-library-function.html) | Closes the stream and flushes buffers associated with the given stream. |
| [feof()](https://www.techcrashcourse.com/2015/08/feof-stdio-c-library-function.html) | Checks the end-of-file indicator of the given stream. |
| [ferror()](https://www.techcrashcourse.com/2015/08/ferror-stdio-c-library-function.html) | Checks the error indicator of the given stream. |
| [fflush()](https://www.techcrashcourse.com/2015/08/fflush-stdio-c-library-function.html) | Flushes the content of the given stream. |
| [fgetc()](https://www.techcrashcourse.com/2015/08/fgetc-stdio-c-library-function.html) | Gets the next character from the given stream. |
| [fgetpos()](https://www.techcrashcourse.com/2015/08/fgetpos-stdio-c-library-function.html) | Gets current position of the given stream. |
| [fgets()](https://www.techcrashcourse.com/2015/08/fgets-stdio-c-library-function.html) | Reads a line from given stream and stores it into a character array. |
| [fopen()](https://www.techcrashcourse.com/2015/08/fopen-stdio-c-library-function.html) | Opens a file in the given mode. |
| [fprintf()](https://www.techcrashcourse.com/2015/08/fprintf-stdio-c-library-function.html) | Writes formatted output to a stream. |
| [fputc()](https://www.techcrashcourse.com/2015/08/fputc-stdio-c-library-function.html) | Writes a character to the given stream. |
| [fputs()](https://www.techcrashcourse.com/2015/08/fputs-stdio-c-library-function.html) | Writes a string to the given stream excluding the null terminating character. |
| [fread()](https://www.techcrashcourse.com/2015/08/fread-stdio-c-library-function.html) | Reads data from the given stream and stores it into an array. |
| [freopen()](https://www.techcrashcourse.com/2015/08/freopen-stdio-c-library-function.html) | Reopens a stream with different file or mode. |
| [fscanf()](https://www.techcrashcourse.com/2015/08/fscanf-stdio-c-library-function.html) | Read formatted data from given stream. |
| [fseek()](https://www.techcrashcourse.com/2015/08/fseek-stdio-c-library-function.html) | Changes the position indicator of the given stream. |
| [fsetpos()](https://www.techcrashcourse.com/2015/08/fsetpos-stdio-c-library-function.html) | Sets the position indicator of the given stream. |
| [ftell()](https://www.techcrashcourse.com/2015/08/ftell-stdio-c-library-function.html) | Returns the current position of the given stream. |
| [fwrite()](https://www.techcrashcourse.com/2015/08/fwrite-stdio-c-library-function.html) | Writes data from an array to the given stream. |

**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 specially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where argc refers to the number of arguments passed, and argv[] is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly:

*#include <stdio.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");*

*}*

*}*

When the above code is compiled and executed with a single argument, it produces the

following result.

*$./a.out testing*

*The argument supplied is testing*

When the above code is compiled and executed with a two arguments, it produces the

following result.

*$./a.out testing1 testing2*

*Too many arguments supplied.*

When the above code is compiled and executed without passing any argument, it produces

the following result.

*$./a.out*

*One argument expected*

It should be noted that argv[0] holds the name of the program itself and argv[1] is a

pointer to the first command line argument supplied, and \*argv[n] is the last argument. If

no arguments are supplied, argc will be one, otherwise and if you pass one argument

then argc is set at 2.

You pass all the command line arguments separated by a space, but if argument itself has

a space then you can pass such arguments by putting them inside double quotes "" or

single quotes ''. Let us re-write above example once again where we will print program

name and we also pass a command line argument by putting inside double quotes:

*#include <stdio.h>*

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

*{*

*printf("Program name %s\n", argv[0]);*

*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");*

*}*

*}*

When the above code is compiled and executed with a single argument separated by space

but inside double quotes, it produces the following result.

*$./a.out "testing1 testing2"*

*Progranm name ./a.out*

*The argument supplied is testing1 testing2*

**SAMPLE PROGRAM**

**Program 1**

Marks obtained by a student in five different subjects are input through the keyboard. Calculate the percentage and display on the screen. Assume the maximum marks in each subject is 100.

**Source Code:**

#include<stdio.h>

int main()

{

char student\_name[20];

int rollno,maths,cprog,english,accounting,cprac;

int total=0;

float average;

printf("Enter Student name:");

scanf("%s",&student\_name);

printf("Enter roll no:");

scanf("%s",&rollno);

printf("Enter mark for Mathematics:");

scanf("%s",&maths);

printf("Enter mark for C Programming:");

scanf("%s",&cprog);

printf("Enter mark for English:");

scanf("%s",&english);

printf("Enter mark for Accounting:");

scanf("%s",&accounting);

printf("Enter mark for C Practical:");

scanf("%s",&cprac);

total=maths+cprog+english+accounting+cprac;

average=(float)total/5;

printf("The percentage of a student is %0.2f",average);

return 0;

}

**Program 2**

Temperature of your city in Fahrenheit degrees is input through the keyboard. Write a program to convert this temperature into Centigrade degrees.

**Source Code:**

#include<stdio.h>

int main()

{

int Fahr;

float Celcius;

printf("Enter temperature of your city in fahrenheit:");

scanf("%d",&Fahr);

Celcius = (Fahr-32)\*(5.0/9.0);

printf("Temperature in Centigrate: %0.2f",Celcius);

return 0;

}

**Program 3**

Write a C program for demonstration of nested if…else statement to find out the greatest of three numbers input by the user.

**Source Code:**

#include<stdio.h>

main()

{

int a1,a2,a3;

printf("Enter three numbers:");

scanf("%d %d %d",&a1, &a2, &a3);

if (a1>a2)

{

if (a1>a3)

{

printf("%d is the greatest",a1);

}

else

{

printf("%d is the greatest",a3);

}

}

else if(a2>a3)

{

printf("%d is the greatest",a2);

}

else

printf("%d is the greatest",a3);

}

}

**Program 4**

Percentage of your attendance is input through the keyboard. Write C program to calculate and display internal mark based on percentage of attendance.

**Source Code:**

#include<stdio.h>

int main()

{

float attendance;

printf("Enter Your Attendance %%:");

scanf("%f",&attendance);

if (attendance>=95 && attendance<=100)

{printf(" Your Internal Mark is 5");

}

else if (attendance<=95 && attendance>=90)

{

printf("Your Internal Mark is 4");

}

else if (attendance<=90 && attendance>=85)

{

printf("Your Internal Mark is 3");

}

else if (attendance<=85 && attendance>=80)

{

printf("Your Internal Mark is 2");

}

else if (attendance<=80 && attendance>=75)

{

printf("Your Internal Mark is 1");

}

else if (attendance<75)

{

printf("Your Internal Mark is 0");

}

}

**Program 5**

Write a C program which input three numbers and display the largest number using conditional operator.

**Source Code:**

#include<stdio.h>

int main()

{

int a,b,c,larger;

printf("\tEnter the value of A:");

scanf("%d",&a);

printf("\tEnter the value of B:");

scanf("%d",&b);

printf("\tEnter the value of C:");

scanf("%d",&c);

larger=(a>b && a>c)?(a):((c>b && c>a)?(c):(b));

printf("\n\t%d is largest",larger);

return 0;

}

**Program 6**

Write a C program using switch to display day of the week by entering the corresponding number. For example, if input is 1, it will display Sunday and so on.

**Source Code:**

#include<stdio.h>

int main()

{

int week, c;

repeat:

printf("Enter week number(0-6):");

scanf("%d",&week);

if(week<=6)

switch(week)

{

case 0:

printf("Sunday");

break;

case 1:

printf("Monday");

break;

case 2:

printf("Tuesday");

break;

case 3:

printf("Wednesday");

break;

case 4:

printf("Thursday");

break;

case 5:

printf("Friday");

break;

case 6:

printf("Saturday");

break;

}

else

{

printf("Invalid input! Please enter week number between 0-6");

}

printf("\n Do you want to repeat: (type 0-yes)");

scanf("%d",&c);

if (c==0)

{

goto repeat;

}

return 0;

}

**Program 7**

Write a C program to print the Fibonacci series up to n terms using while loop.

**Source Code:**

#include<stdio.h>

int main()

{

int a,b,c,no,counter=1;

printf("Enter The Number of Term:");

scanf("%d",&no);

a=0;

b=1;

while(counter<=no)

{

c=a+b;

printf("%10ld\t",a);

a=b;

b=c;

counter++;

}

return 0;

}

**Program 8**

Write a C program to check whether the given number is prime or not.

**Source Code:**

#

include<stdio.h>

int main()

{

int a,b,c,d;

repeat:

b=1;

c=0;

printf("Enter The Number:");

scanf("%d",&a);

while(b<=a)

{

if(a%b==0)

{

c++;

printf("\t%d can be divided by %d\n",a,b);

}

b++;

}

if(c==2)

printf("\n\t%d is Prime Number",a);

else

printf("\n\t%d is not a Prime Number",a);

printf("\n Do you want to repeat: (type 0- yes)");

scanf("%d",&d);

if (d==0)

{

goto repeat;

}

}

**Program 9**

Write a program to perform Linear Search of any value in an array and display the position of the value if found.

**Source Code:**

#include<stdio.h>

void main ()

{

int n,a[50],x,i,found;

printf("Enter how many Items are there: ");

scanf("%d",&n);

printf("\n");

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

{

printf("Enter the value of a[%d] : ",i+1);

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

}

printf("Enter Number to be search: ");

scanf("%d",&x);

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

{

if(x==a[i])

{

found=1;

break;

}

}

if(found==1)

{

printf("%d Is found at position a[%d]",x,i+1);

}

else

{

printf("%d Is not found",x);

}

getch();

}

**Program 10**

Write a program using C to perform binary search for any item in an ordered array and display the position of the item if found.

**Source Code:**

#include<stdio.h>

void main()

{

int x,i,a[50],lower,upper,middle,found,numb;

printf("Enter The Number Of Items:");

scanf("%d",&x);

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

{

printf("Enter the value of a[%d]:",i);

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

}

printf("Enter the value to be search:");

scanf("%d",&numb);

lower=0;

upper=x-1;

middle=(lower+upper)/2;

while(lower<=upper)

{

if(a[middle]==numb){

printf("%d Is found at a[%d]",numb,middle);

break;}

else if(a[middle]<numb){

lower=middle+1;

middle = (lower + upper)/2;}

else{

upper=middle - 1;

middle = (lower + upper)/2;}

}

if (lower>upper)

printf("Not found! %d isn't present in the list.\n",numb); return 0;}

**Program 11**

Elements of two matrices are input by the user. Write a C program to find and display the addition and subtraction of the two matrices.

**Source Code:**

#include<stdio.h>

int main()

{

int a[2][3],b[2][3],add[2][3],sub[2][3];

int r,c;

printf("Enter Data For Matrix a:\n");

for(r=0;2>r;r++)

{

for(c=0;c<3;c++)

{

printf("\n\ta[%d][%d] = ",r,c);

scanf("%d",&a[r][c]);

}

}

printf("\nEnter Data For Matrix b:\n");

for(r=0;2>r;r++)

{

for(c=0;c<3;c++)

{

printf("\n\tb[%d][%d] = ",r,c);

scanf("%d",&b[r][c]);

}

}

for(r=0;2>r;r++)

{

for(c=0;c<3;c++)

{

add[r][c]=a[r][c]+b[r][c];

}

}

for(r=0;2>r;r++)

{

for(c=0;c<3;c++)

{

sub[r][c]=a[r][c]-b[r][c];

}

}

printf("\n The First Matrix Is:\n");

for(r=0;2>r;r++)

{

for(c=0;c<3;c++)

{

printf("\t%d",a[r][c]);

}

printf("\n");

}

printf("\n The Second Matrix Is:\n");

for(r=0;2>r;r++)

{

for(c=0;c<3;c++)

{

printf("\t%d",b[r][c]);

}

printf("\n");

}

printf("\n The Sum Of Two Matrix Is:\n");

for(r=0;2>r;r++)

{

for(c=0;c<3;c++)

{

printf("\t%d",add[r][c]);

}

printf("\n");

}

printf("\n The Subtract Of The Two Matrix Is:\n");

for(r=0;2>r;r++)

{

for(c=0;c<3;c++)

{

printf("\t%d",sub[r][c]);

}

printf("\n");

}

getch();

}

**Program 12**

Using C programming language, write a program to find the transpose of a matrix.

**Source Code:**

#include<stdio.h>

void main()

{

int a[10][10],i,j,colm,rows;

printf("Enter the Number of Rows:");

scanf("%d",&rows);

printf("Enter the Number of Columns:");

scanf("%d",&colm);

printf("Enter the Matrices:");

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

{

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

{

printf("\na[%d][%d] = ",i,j);

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

}

}

printf("\nThe Given Matrix Is:\n");

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

{

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

{

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

}

printf("\n");

}

printf("\nTranspose of the matrices is:\n");

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

{

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

{

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

}

printf("\n");

} }

**Program 13**

Write C program to find the multiplication of two matrices using two dimensional array.

**Source Code:**

#include<stdio.h>

void main()

{

int a[2][2],b[2][3],ab[2][3];

int r,c,k,sum=0;

printf("Enter Data For Matrix a:");

printf("\n");

for(r=0;r<2;r++)

{

for(c=0;c<2;c++)

{

printf("a[%d][%d]\t= ",r,c);

scanf("%d",&a[r][c]);

}

}

printf("Enter Data For Matrix b:");

printf("\n");

for(r=0;r<2;r++)

{

for(c=0;c<3;c++)

{

printf("a[%d][%d]\t= ",r,c);

scanf("%d",&b[r][c]);

}

}

for(r=0;r<2;r++)

{

for(c=0;c<3;c++)

{

for(k=0;k<2;k++)

{

sum=sum+a[r][k]\*b[k][c];

}

ab[r][c]=sum;

sum=0;

}

printf("\n");

}

printf("\n The First Matrix Is:\n");

for(r=0;r<2;r++)

{

for(c=0;c<2;c++)

{

printf("\t%d",a[r][c]);

}

printf("\n");

}

printf("\n The Second Matrix Is:\n");

for(r=0;r<2;r++)

{

for(c=0;c<3;c++)

{

printf("\t%d",b[r][c]);

}

printf("\n");

}

printf("\n The multiplication of the two Matrix Is Matrix Is:\n");

for(r=0;r<2;r++)

{

for(c=0;c<3;c++)

{

printf("\t%d",ab[r][c]);

}

printf("\n");

}

getch();

}

**Program 14**

Write a program in C to accept lines of text and then display the number of alphabets, digits and special characters.

**Source Code:**

#include <stdio.h>

int main()

{

char s[100];

int i, alphabets=0, digits=0, specialcharacters=0;

printf("Enter the string : ");

gets(s);

for(i=0; s[i]; i++) //0 is considered as false

{//&& and || or

if((s[i]>=65 && s[i]<=90)|| (s[i]>=97 && s[i]<=122))

alphabets++; //+1

else if(s[i]>=48 && s[i]<=57)

digits++;

else

specialcharacters++;

}

printf("Alphabets = %d\n", alphabets);

printf("Digits = %d\n", digits);

printf("Special characters = %d", specialcharacters);

return 0;

}

**Program 15**

Write a recursive function program using C to find the factorial value of any number entered through the keyboard.

**Source Code:**

#include<stdio.h>

void fact(int x);

int ans=1;

void main()

{

int x;

printf("\nEnter any No to find the factorial value:");

scanf("%d",&x);

printf("\nFactorial value of %d is:\n", x);

fact(x);

printf("\b\b = %d",ans);

getch();

}

void fact(int x)

{

ans=ans\*x;

printf("%d x ",x);

x--;

if(x>0)

fact(x); }

**Program 16**

Write a C program to demonstrations functions call by value and call by references.

**Source Code:**

#include <stdio.h>

void swapv(int, int); //call by value function

void swapr(int \*, int \*); //call by reference function

int main()

{

int a = 10;

int b = 20;

printf("Before swapping the values in main a = %d, b = %d\n",a,b);

swapv(a,b);

printf("\nAfter swapping values in main a = %d, b = %d\n",a,b);

swapr(&a,&b);

printf("\nAfter swapping values in main a=%d, b = %d\n",a,b);

getch();

}

void swapv (int x, int y)

{

int temp;

temp = x;

x=y;

y=temp;

printf("\nAfter swapping(using call by value) a = %d, b = %d\n",x,y);

}

void swapr (int \*x, int \*y)

{

int temp;

temp = \*x;

\*x=\*y;

\*y=temp;

printf("\nAfter swapping (using call by reference) a = %d, b = %d\n", \*x, \*y);

}

**Program 17**

Write a C program to reverse the accepted number using function.

**Source Code:**

#include<stdio.h>

int Reverse (int n)

{

int sum=0;

while (n!=0)

{

sum = sum\*10 + n%10;

n = n/10;

}

return sum;

}

void main()

{

int rev, num;

printf("Enter a Positive Number: ");

scanf("%d", &num);

rev = Reverse (num);

printf("The Reverse of given number %d is: %d", num, rev);

getch();

}

**Program 18**

Write a C program to convert decimal number to its equivalent binary number using function.

**Source Code:**

#include<stdio.h>

long tobinary(int);

int main()

{

long bno;

int dno;

printf("Enter any decimal number : ");

scanf("%d",&dno);

bno = tobinary(dno);

printf("\n The Binary Value is : %ld\n\n",bno);

return 0;

}

long tobinary(int dno)

{

long bno=0,rem,f=1;

while(dno != 0)

{

rem = dno%2;

bno = bno+rem\*f;

f=f\*10;

dno=dno/2;

}

return bno;

}

**Program 19**

Define a structure called student with data members RollNo, Name and Percentage. Also write two functions getdata() to input data and printdata() to display the data.

**Source Code:**

#include<stdio.h>

int i,n;

struct student

{

int RollNo;

char Name[30];

float Percentage;

}stud[];

main()

{

getdata();

printdata();

getch();

}

getdata()

{

printf("Enter the number of students:\t");

scanf("%d",&n);

printf("\nFill up the information\n\n");

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

{

printf("Enter the roll no:");

scanf("%d",&stud[i].RollNo);

printf("Enter the name of students:");

scanf("%s",&stud[i].Name);

printf("Enter the percentage:");

scanf("%f",&stud[i].Percentage);

printf("\n\n");

}

}

printdata()

{

printf("\nThe information of the students are:\n\n");

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

{

printf("\nRoll no:\t%d\nName:\t\t%s\nPercentage:\t%0.2f%%\n",stud[i].RollNo,stud[i].Name,stud[i].Percentage);

}

}

**Program 20**

Write a C program to read and print student details such as roll number as integer, name as character and percentage as float using structure pointer.

**Source Code:**

#include<stdio.h>

struct student

{

char name[30];

int roll;

float perc;

};

int main()

{

struct student std;

struct student \*ptr;

ptr= &std;

printf("Enter details of student: ");

printf("\nName : ");

gets(ptr->name);

printf("Roll No : ");

scanf("%d",&ptr->roll);

printf("Percentage : ");

scanf("%f",&ptr->perc);

printf("\nEntered details :");

printf("\nName : %s\nRollNo : %d \nPercentage : %0.2f%%\n",ptr->name,ptr->roll,ptr->perc); return 0; }

**Program 21**

Write a C program to create file "Student.txt" which will save student information such as student roll number, name and total marks for n students and display lists of students on screen by reading from the same file.

**Source Code:**

#include<stdio.h>

struct s{

int rollno;

char name[50];

float mark;

};

int main()

{

struct s a[5],b[5];

FILE \*fptr;

int i=0;

fptr=fopen("student.txt","w");

for (i-0;i<2;i++)

{

fflush(stdin);

printf("Enter rollno : ");

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

printf("Enter Student Name : ");

scanf("%s",&a[i].name);

printf("Enter mark obtained : ");

scanf("%f",&a[i].mark);

printf("\n\n");

}

fwrite(a,sizeof(a),1,fptr);

fclose(fptr);

fptr=fopen("student.txt","r");

fread(b,sizeof(b),1,fptr);

printf("\nReading student.txt file");

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

{

printf("\nRollNo : %d\n

Name : %s\nMark Obtained: %0.2f%%\n\n",b[i].rollno,b[i].name,b[i].mark);

}

fclose(fptr);

**Program 22**

Write a program in C to accept lines of text to save into a file and then read from the same file to display the number of alphabets, digits and special characters on the screen.

**Source Code:**

#include<stdio.h>

void main()

{

char ch;

int alphabets=0, digits=0, specialcharacters=0;

FILE \*fptr;

fptr=fopen("text.txt","w");

if (fptr==NULL)

{

printf("File can't be created\a");

getch();

exit(0);

}

printf("Enter some text and press enter the key:\t");

while((ch=getche())!='\r')

{

fputc(ch,fptr);

}

fclose(fptr);

fptr=fopen("text.txt","r");

printf("\n\nContents of the File is:");

while((ch=fgetc(fptr))!=EOF)

{

if((ch>=65 && ch<=90)|| (ch>=97 && ch<=122) )

alphabets++;

else if(ch>=48 && ch<=57)

digits++;

else

specialcharacters++;

printf("%c",ch);

}

fclose(fptr);

printf("\nAlphabets = %d\n",alphabets);

printf("Digits = %d\n",digits);

printf("special Characters = %d", specialcharacters);

getch();

}

**Program 23**

**Programs to Print Patterns in C**

**Right Half Pyramid Pattern in C**

The right-half pyramid is nothing but a right-angle triangle whose hypotenuse is in the right direction. We can print the right half pyramid pattern using numbers, alphabets, or any other character like a star (\*).

// C program to print right half pyramid pattern of star

#include <stdio.h>

**int** main()

{

**int** rows = 5;

   // first loop for printing rows

**for** (**int** i = 0; i < rows; i++)

{

        // second loop for printing character in each rows

**for** (**int** j = 0; j <= i; j++) {

**printf**("\* ");

        }

**printf**("\n");

    }

**return** 0;

}

**Output**

\*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

**// c program to print left half pyramid pattern of star**

#include <stdio.h>

**int** main()

{

**int** rows = 5;

    // first loop is for printing the rows

**for** (**int** i = 0; i < rows; i++)

{

          // loop for printing leading whitespaces

**for** (**int** j = 0; j < 2 \* (rows - i) - 1; j++) {

**printf**(" ");

        }

        // loop for printing \* character

**for** (**int** k = 0; k <= i; k++) {

**printf**("\* ");

        }

**printf**("\n");

    }

**return** 0;

}

**Output**

\*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

**// C program to print the full pyramid pattern of stars**

#include <stdio.h>

**int** main()

{

**int** rows = 5;

    // first loop to print all rows

**for** (**int** i = 0; i < rows; i++) {

        // inner loop 1 to print white spaces

**for** (**int** j = 0; j < 2 \* (rows - i) - 1; j++) {

**printf**(" ");

        }

       // inner loop 2 to print star \* character

**for** (**int** k = 0; k < 2 \* i + 1; k++) {

**printf**("\* ");

        }

**printf**("\n");

    }

**return** 0;

}

**Output**

\*

\* \* \*

\* \* \* \* \*

\* \* \* \* \* \* \*

\* \* \* \* \* \* \* \* \*

**// C program to print the inverted right half pyramid of stars**

#include <stdio.h>

**int** main()

{

**int** rows = 5;

    // first loop to print all rows

**for** (**int** i = 0; i < rows; i++) {

        // first inner loop to print the \* in each row

**for** (**int** j = 0; j < rows - i; j++) {

**printf**("\* ");

        }

**printf**("\n");

    }

}

**Output**

\* \* \* \* \*

\* \* \* \*

\* \* \*

\* \*

\*

**// C program to print the inverted left half pyramid pattern of stars**

#include <stdio.h>

**int** main()

{

**int** rows = 5;

    // first loop for printing all rows

**for** (**int** i = 0; i < rows; i++) {

        // first inner loop for printing white spaces

**for** (**int** j = 0; j < 2 \* i; j++) {

**printf**(" ");

        }

        // second inner loop for printing star \*

**for** (**int** k = 0; k < rows - i; k++) {

**printf**("\* ");

        }

**printf**("\n");

    }

**return** 0;

}

**Output**

\* \* \* \* \*

\* \* \* \*

\* \* \*

\* \*

\*

**// C program to print the inverted full pyramid pattern of stars**

#include <stdio.h>

**int** main()

{

**int** rows = 5;

    // first loop for printing all rows

**for** (**int** i = 0; i < rows; i++) {

        // first inner loop for printing leading white

        // spaces

**for** (**int** j = 0; j < 2 \* i; j++) {

**printf**(" ");

        }

      // second inner loop for printing stars \*

**for** (**int** k = 0; k < 2 \* (rows - i) - 1; k++) {

**printf**("\* ");

        }

**printf**("\n");

    }

}

**Output**

\* \* \* \* \* \* \* \* \*

\* \* \* \* \* \* \*

\* \* \* \* \*

\* \* \*

\*

**// C program to print hollow full pyramid using star \***

#include <stdio.h>

**int** main()

{

**int** rows = 5;

    // first outer loop to iterate through each loop

**for** (**int** i = 0; i < rows; i++) {

       // first inner loop to print leading whitespaces

**for** (**int** j = 0; j < 2 \* (rows - i) - 1; j++) {

**printf**(" ");

        }

        // second inner loop to print stars \* and inner

        // whitespaces

**for** (**int** k = 0; k < 2 \* i + 1; k++) {

**if** (k == 0 || k == 2 \* i || i == rows - 1) {

**printf**("\* ");

            }

**else** {

**printf**("  ");

            }

        }

**printf**("\n");

    }

**return** 0;

}

**Output**

\*

\* \*

\* \*

\* \*

\* \* \* \* \* \* \* \* \*