**UNIT I BASICS OF C PROGRAMMING**

Introduction to programming paradigms - Structure of C program - C programming: Data Types – Storage classes - Constants – Enumeration Constants - Keywords – Operators: Precedence and Associativity - Expressions - Input/Output statements, Assignment statements – Decision making statements - Switch statement - Looping statements – Pre-processor directives - Compilation process

**Structure of C program**

// comments

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Program: add.c

Author: Ashok.R

Verion: 1.0.1

This program adds two numbers and print the result

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

// pre-processor directives

#include <stdio.h>

// Global declarations

int result **=** 0**;**

void add**(**int**,** int**);**

// main() function

int main**()**

**{**

// local declarations

int a **=** 10**;**

int b **=** 20**;**

// statements

add**(**a**,**b**);**

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

**return** 0**;**

**}**

// other functions

void add**(**int x**,** int y**)**

**{**

result **=** x **+** y**;**

**}**

**Comments**

Comments are included to improve the readability of the program

1. Single line comments

starts with //  
 Example:  
 // This is single line comment

2. Multi line comments  
 starts with /\* and ends with \*/  
 Example:

As shown in the example program

**Pre-processor directives**

* Starts with #
* It is executed before the compiler compiles the source code
* Example:  
  #include directive includes the specific header file to the program before compilation.

**Global declarations**

Variables can be declared globally and can be accessed anywhere within the program

**Functions**

A C program contains one or more functions. The statements inside a function are written in a sequence to perform a specific task. The main() function must be included in every program. The execution of a C program begins at main() function.

Every function has two parts:

1. Header:   
 Format:  
 return type function name (arguments)

Example:

void add(int x, int y)

2. Body:

A sequence of statements enclosed within curly braces {}

**C Tokens**

The smallest individual units of a C program.  
Types of tokens: keywords, identifiers, constants, string, special symbols, operators

**Keywords**

Reserved words to imply special meaning to the compiler. There are 32 keywords in C.  
Examples: auto, float, int

All the keywords must be written in lower case.

**Identifiers**

Names to identify the program elements such as variables, arrays and functions.

**Identifier naming rules**

* It must begin with a letter
* Only letters, digits and underscore( \_ ) are permitted to be used in an identifier name
* No other special characters are allowed
* Case sensitive
* Keywords can not be used as identifiers

**Escape sequences**

Non printable characters

\a alert

\b back space

\t horizontal tab

\v vertical tab

\n new line

\r carriage return

**Constants**

The value is known at the compile time

Example

-1, 12.8, ‘a’

**Enumeration Constants**

“Enumerated” data type is the user defined data type which can take on only finite set of values.

Example:

enum size(S=30, M=32, L=34, XL=36);

enum size mydress = M;

An enumeration is a list of constant integer values.

**Variables**

A variable is an entity whose value can vary during the execution of a program.

Example:

int a; int b=5;

a = 10;

a = b+a;

**Data types**

**Primitive data types**

**char:** The most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.

**int:**As the name suggests, an int variable is used to store an integer.

**float:** It is used to store decimal numbers (numbers with floating point value) with single precision.

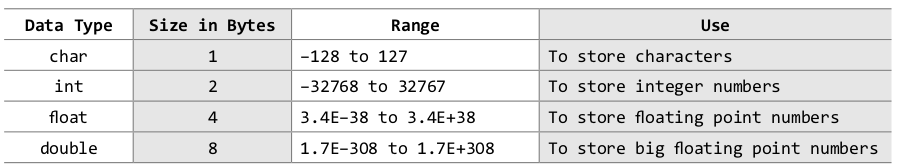
**double:** It is used to store decimal numbers (numbers with floating point value) with double precision.

Examples:

char grade **=** 'A'**;**

int a **=** 10**;**

float height **=** 154.8**;**



**Derived data types**

Arrays, functions and pointers are examples for derived data types.

**User defined data types**

Structures and unions are examples for user defined data types.

**Storage classes**

Storage Classes are used to describe about the features of a variable/function. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program.

C language uses 4 storage classes.

**auto**: This is the default storage class for all the variables declared inside a function or a block. They are assigned a garbage value by default whenever they are declared.

**extern**: Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program.

**static**: Static variables preserve the value of their last use in their scope. They are initialized only once and exist till the termination of the program. The value of static variables persists between function calls. By default, they are assigned the value 0 by the compiler.

**register**: The compiler tries to store these variables in the register of the microprocessor if a free register is available. The access of register variables are much faster than that of the variables stored in the memory during the runtime of the program. If a free register is not available, these are then stored in the memory only. We cannot obtain the address of a register variable using pointers.

Examples:

auto char grade **=** 'A'**;**

int a **=** 10**;** // All variables are auto by default

register int b **=** 200**;**

static int c **=** 10**;**

extern int d**;**

|  |  |  |  |
| --- | --- | --- | --- |
| **Storage class** | **Storage** | **initial value** | **life time** |
| auto | RAM | garbage | automatic |
| register | CPU registers | garbage | automatic |
| static | RAM | zero | static |
| extern | RAM | zero | static |

**Operators**

**Arithmetic Operators**: These are the operators used to perform arithmetic/mathematical operations on operands. Examples: (+, -, \*, /, %,++,–).  
Arithmetic operator are of two types:

1. **Unary Operators**: Operators that operates or works with a single operand are unary operators.  
   For example: (++ , –)
2. **Binary Operators**: Operators that operates or works with two operands are binary operators.For example: (+ , – , \* , /)

**Relational Operators:** Relational operators are used for comparison of the values of two operands. For example: checking if one operand is equal to the other operand or not, an operand is greater than the other operand or not etc. Some of the relational operators are (==, > , = , <= ).

**Logical Operators:** Logical Operators are used to combine two or more conditions. The result of the operation of a logical operator is a boolean value either true or false.

**Assignment Operators:** Assignment operators are used to assign value to a variable. It evaluates the expression in RHS(right hand side) and assigns the result to the variable in the LHS (left hand side).

For example:

a = 10;

b = 20;

ch = 'y';

Compound assignment operators are expanded as follows.

b += 2;

* b = b+2;

a <<= 2;

* a = a << 2;

**Bitwise Operators:** The Bitwise operators is used to perform bit-level operations on the operands.

& bitwise AND

| bitwise OR

^ bitwise EX-OR

<< left shift

>> right shift

Example:

a = 4 << 2;

|  |  |  |
| --- | --- | --- |
| 4 |  | 0000 0000 0000 0100 |
| 16 | after right shift 4 by 2 | 0000 0000 0001 0000 |

The ‘a’ will be assigned with ‘16’.

**Other operators**

sizeof operator

comma operator

Array subscript operator [ ]

direct member access operator (dot)

Indirect member access (arrow - > )

Address-of operator (&)

Indirection operator (\*)

**Ternary operator (?:)**

The ternary operator is an operator that takes three arguments. The first argument is a comparison argument, the second is the result upon a true comparison, and the third is the result upon a false comparison.

Example:

c = a < b ? a : b ;

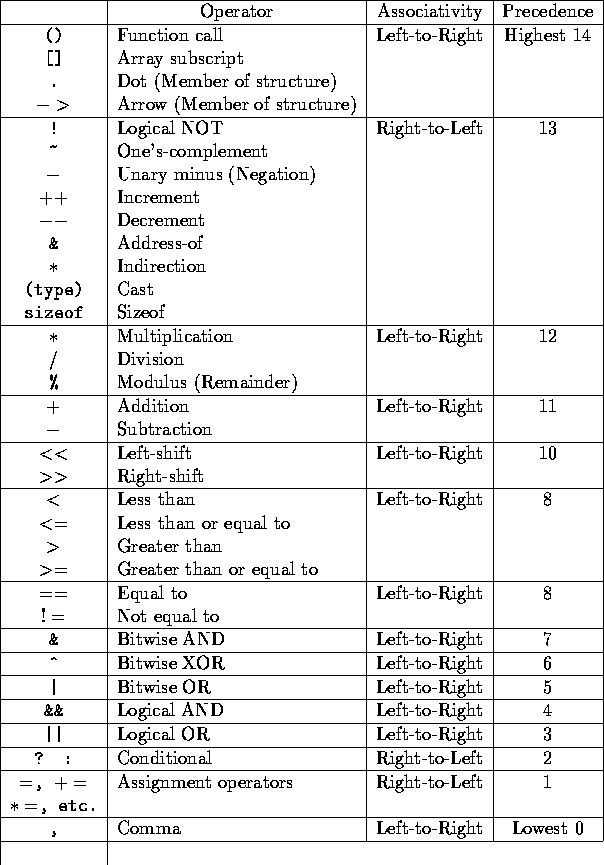
The above statement assigns the minimum of ‘a’ and ‘b’ to ‘c’.

**Precedence and associativity**

When an expression has more than one operator, they will be evaluated in an order defined by their precedence and associativity.

Precedence: denotes the priority (which operator gets evaluated first)

Associativity: defines the order for the operators with the same precedence or same operators (Left to right or right to left).



Example:

5 \* 8 / 2 + 3

The expression is evaluated based on the precedence of the operators and associativity.

* 40 / 2 + 3
* 20 + 3
* 23

The result after evaluating the given expression is 23.

**Formatted Input & Output using printf() and scanf()**

## printf()

This function is used to print text as well as value of the variables on the standard output device (monitor), printf is very basic library function in c language that is declared in stdio.h header file.

**Syntax:**

printf(“message”);

printf(“message + format-specifier”,variable-list);

First printf() style printf the simple text on the monitor, while second printf() prints the message with values of the variable list.

#include <stdio.h>

**int** main()

{

printf("Message-1");

printf("Message-2");

printf("Message-3");

**return** 0;

}

Output

Message-1Message-2Message-3

## How to print value of the variables?

To print values of the variables, you need to understand about **format specifiers** are the special characters followed by % sign, which are used to print values of the variable s from variable list.

### Format specifiers

Here are the list some of the format specifiers, use them in printf() & scanf() to format & print values of the variables:

Character (char) %c

Integer (int) %d

Insigned integer (unsigned int) %ld

Long (long) %ld

Unsigned long (unsigned long) %lu

Float (float) %f

Double (double) %lf

Octal Value (octal value) %o

Hexadecimal Value (hex value) %x

String (char[]) %s

\*\*NOTE\*\* Use ‘u’ for unsigned type modifier, ‘l’ for long.

### Escape Sequences

To print special extra line/spaces etc, we use escape sequences, these characters are followed by ‘\’ (slash).

\\ \

\” “

\’ ‘

\? ?

\a Alert

\b Back space

\n New Line

\t Horizontal tab

\v Vertical tab

\r Carriage return

**Consider the following examples:**

#include <stdio.h>

**int** main()

{

**int** num=100;

**float** val=1.23f;

**char** sex='M';

//print values using different printf

printf("Output1:");

printf("%d",num);

printf("%f",val);

printf("%c",sex);

//print values using single printf

printf("\nOutput2:"); // \n: for new line in c

printf("%d,%f,%c",num,val,sex);

**return** 0;

}

Output

Output1:1001.230000M

Output2:100,1.230000,M

### scanf()

This function is used to get (input) value from the keyboard. We pass format specifiers, in which format we want to take input.

**Syntax:**

scanf(“format-specifier”, &var-name);

scanf(“fromat-specifier-list”, &var-name-list);

First type of scanf() takes the single value for the variable and second type of scanf() will take the multiple values for the variable list.

**Consider the following examples:**

#include <stdio.h>

**int** main()

{

**int** a;

**float** b;

**char** c;

printf("Enter an integer number (value of a)?:");

scanf("%d",&a);

printf("Enter a float number (value of b)?:");

scanf("%f",&b);

printf("Enter a character (value of c)?:");

fflush(stdin); // to flush (clear) input buffer

scanf("%c",&c);

printf("\na=%d,b=%f,c=%c",a,b,c);

**return** 0;

}

Output

Enter an integer number (value of a)?:1234

Enter a float number (value of b)?:1.2345

Enter a character (value of c)?:G

a=1234,b=1.234500,c=G

**Consider the following examples to read multiple value in single scanf statement:**

#include <stdio.h>

**int** main()

{

**int** a;

**float** b;

**char** c;

printf("\nEnter value of a,b,c (an integer, a float, a character):");

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

printf("\na=%d,b=%f,c=%c",a,b,c);

**return** 0;

}

Output

Enter value of a,b,c (an integer, a float, a character):1234 1.2345 G

a=1234,b=1.234500,c=

Here, G will not store into c variable, because we are not flushing input buffer here. So either you will have to take input of c first or you will have to read value of c separately.

# Unformatted I/O

C provides many low level functions to read and write unformatted data. These functions are explained next.

## getchar()

getchar() function will read a single character from the standard input. The return value of getchar() is the first character in the standard input.The input is read until the Enter key is pressed, but only the first character in the input will be returned.

**putchar()**

putchar() function will print a single character on standard output. The character to be printed is passed to putchar() function as an argument. The return value of putchar() is the character which was written to the output.

getchar() and putchar() functions are part of the standard C library header stdio.h

### Example

#include <stdio.h>

int main()

{

char ch;

printf("Input some character and finish by pressing the Enter key.\n");

ch = getchar();

printf("The input character is ");

putchar(ch);

return 0;

}

If we run this program and enter the characters "apple" at the prompt then the output will be as shown below

Input some character and finish by pressing the Enter key

apple

The input character is a

Please note that when you entered the word apple those characters were echoed on the screen. You also terminated the input by pressing the Enter key. However getchar() function took only the first character, which is "a" from the input and putchar() function printed it out.

## getch(), getche() and putch()

These functions are similar to getchar() and putchar(), but they come from the library header conio.h. The header file conio.h is not a standard C library header and it is not supported by compilers that target Unix. However it is supported in DOS like environments.

getch() reads a single character from the standard input. The input character is not displayed (echoed) on the screen. Unlike the getchar() function, getch()returns when the first character is entered and does not wait for the Enter key to be pressed.

getche() is same as getch() except that it echoes the input character.

putch() writes a character that is passed as an argument to the console.

### Example

#include <conio.h>

#include <stdio.h>

int main()

{

char ch;

printf("Press any key\n");

ch = getch();

printf("The key pressed is: ");

putch(ch);

return 0;

}

If we run this program and press any key, let's say "x", then the output will be as shown below:

Press any key

The key pressed is x

Please note here that the character "x" was not echoed when you entered it.

**Sequence Selection and looping**

**Sequence**

The statements are executed one after another in a sequence of steps

**Example :**

A=10

B=20

C=a+b

After the above sequence of statements are executed the value of C is 30.

**Selection or Decision making**

The program flow is controlled by the selection or decision made depending on a condition.

**Example :**

if(a>b)

c=a;

else

c=b;

Based on the condition (a>b) the bigger number is assigned to c.

Selection statements help to make alternative actions based on a condition.

**Repetition or Iteration or Looping**

The loop statement repeats a particular block of statements for specific number of times or until a condition is met.

The following statements make iteration

1.for loop

2.while loop

3.do..while loop

Example

for(i=1;i<=10;i++)

{

Printf(“%d x 5 =%d\n”,i,i\*5);

}

The above code snippet prints the multiplication table of 5

1 x 5 = 5

2 x 5 = 10

.

.

.

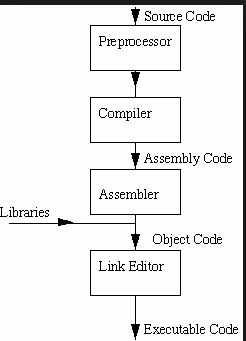
10 x 5 = 50

**Compilation process**

A compiler is used to translate a source code (.c) into a machine executable object code (.obj).

The linker links the runtime libraries and creates the executable file(.exe).

The executable file can be directly run on the computer system.



The four stages of compilation process is shown above.

**Pre-processor**

-includes content of header files

#include<stdio.h>

Includes standard library for I/O.

-Expands macros

#define SIZE 100

Defines symbolic name or constant and constitutes its occurrence.

**Compiler**

Translates source to assembly code(.s extension).

**Assembler**

The assembler creates the machine object code (.o extension)

[.OBJ in MSDOS]

**Linker**

If a source file references the library functions, the linker combines these functions to create an executable file.

**Preproceesor directives**

**#include directive**

Includes a copy of a specified file in place of the directive.

#include<stdio.h>

Includes standard library for I/O.

**#define directive**

Creates symbolic constants

#define PI 3.14159

Replaces all subsequent occurrences of symbolic constant PI with the numeric constant 3.14159

**Macros**

A macro is created with #define directive. The macro identifier is replaced in the program with the replacement text before the program is compiled.

#define AREA(X) (PI\*X\*X)

Wherever the macro AREA(X) appears it is replaced with the text PI\*X\*X

For example

a=AREA(10)

is expanded as

a=PI\*10\*10;

before compilation. Further the symbolic constants PI is also replaced with 3.14159 before compilation.

**#undef directive**

Undefines symbolic constant or a nacro(if defined elsewhere)

Once undefined a name can be redefined with# define.

Conditional pre processor

Conditional compilation is commonly used for debugging

#ifdef DEBUG

printf(“X=%d\n”,X)

#endif

If the symbolic constant DEBUG has been defined before the #ifdef directive, the printf statement will be executed.

**Example Programs**

**1.Find the sum of individual digits of a number.**

#include<stdio.h>

#include<conio.h>

void main()

{

    int n,sum=0;

    printf("enter a +ve integer"); // enter a integer value

    scanf("%d",&n);

    while(n>0)  // checks the condition

    {

        sum=sum+n%10;   // sum + remainder value

        n=n/10;

    }

    printf("sum of individual digits of a positive integer is %d",sum);  // prints the sum of individual digits

    getch();

}

#### Output:

Enter any number

1234

Sum of individual digits of a given number is 10

**2.Check whether a number is prime or not**

#include <stdio.h>

int main()

{

int n, i, flag = 0;

printf("Enter a positive integer: ");

scanf("%d",&n);

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

{

// condition for nonprime number

if(n%i==0)

{

flag=1;

break;

}

}

if (flag==0)

printf("%d is a prime number.",n);

else

printf("%d is not a prime number.",n);

return 0;

}

**Output**

Enter a positive integer: 29

29 is a prime number.

**3.Find the factorial of a number**

#include <stdio.h>

int main()

{

int n, i;

unsigned long long factorial = 1;

printf("Enter an integer: ");

scanf("%d",&n);

// show error if the user enters a negative integer

if (n < 0)

printf("Error! Factorial of a negative number doesn't exist.");

else

{

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

{

factorial \*= i; // factorial = factorial\*i;

}

printf("Factorial of %d = %llu", n, factorial);

}

return 0;

}

**Output**

Enter an integer: 10

Factorial of 10 = 3628800

**4.Generate Fibonacci series**

#include <stdio.h>

int main()

{

int i, n, t1 = 0, t2 = 1, nextTerm;

printf("Enter the number of terms: ");

scanf("%d", &n);

printf("Fibonacci Series: ");

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

{

printf("%d, ", t1);

nextTerm = t1 + t2;

t1 = t2;

t2 = nextTerm;

}

return 0;

}

**Output**

Enter the number of terms: 10

Fibonacci Series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34,

**5.Checking leap year**

#include <stdio.h>

int main()

{

int year;

printf("Enter a year: ");

scanf("%d",&year);

if(year%4 == 0)

{

if( year%100 == 0)

{

// year is divisible by 400, hence the year is a leap year

if ( year%400 == 0)

printf("%d is a leap year.", year);

else

printf("%d is not a leap year.", year);

}

else

printf("%d is a leap year.", year );

}

else

printf("%d is not a leap year.", year);

return 0;

}

**Output 1**

Enter a year: 1900

1900 is not a leap year.

**Output 2**

Enter a year: 2012

2012 is a leap year.

**6. To solve quadratic equation**

#include <stdio.h>

#include <math.h>

int main()

{

double a, b, c, determinant, root1,root2, realPart, imaginaryPart;

printf("Enter coefficients a, b and c: ");

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

determinant = b\*b-4\*a\*c;

// condition for real and different roots

if (determinant > 0)

{

// sqrt() function returns square root

root1 = (-b+sqrt(determinant))/(2\*a);

root2 = (-b-sqrt(determinant))/(2\*a);

printf("root1 = %.2lf and root2 = %.2lf",root1 , root2);

}

//condition for real and equal roots

else if (determinant == 0)

{

root1 = root2 = -b/(2\*a);

printf("root1 = root2 = %.2lf;", root1);

}

// if roots are not real

else

{

realPart = -b/(2\*a);

imaginaryPart = sqrt(-determinant)/(2\*a);

printf("root1 = %.2lf+%.2lfi and root2 = %.2f-%.2fi", realPart, imaginaryPart, realPart, imaginaryPart);

}

return 0;

}

**Output**

Enter coefficients a, b and c: 2.3

4

5.6

Roots are: -0.87+1.30i and -0.87-1.30i

**7. To check Armstrong number**

#include <stdio.h>

int main()

{

int number, originalNumber, remainder, result = 0;

printf("Enter a three digit integer: ");

scanf("%d", &number);

originalNumber = number;

while (originalNumber != 0)

{

remainder = originalNumber%10;

result += remainder\*remainder\*remainder;

originalNumber /= 10;

}

if(result == number)

printf("%d is an Armstrong number.",number);

else

printf("%d is not an Armstrong number.",number);

return 0;

}

**Output**

Enter a three digit integer: 371

371 is an Armstrong number.