TATA ELXSI

engineering creativity

# C Programming

## Day -1

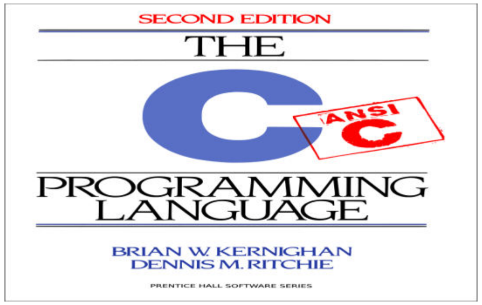
## Introduction to Programming

## Disclaimer

- This material is developed for in-house training at TATA ELXSI.
- The training material therefore was solely developed for educational purposes. Currently, commercialization of the prototype does not exist, nor is the prototype available for general usage. TATA ELXSI, its staff, its students or any other participants can use it as a part of training. This material should not be found on any website.
- For the preparation of this material we have referred from the below mentioned links or books. Excerpts of these material has been taken where there is no copy right infringements. Some examples and concepts have been sourced from the below links and are open source material

http://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

ANSI C K&R



## Day 1- Agenda

- Introduction to C Programming Language
- Basic Data Types
- Constants
- Variables
- Storage Classes
- Specifiers

## What is programming?

- Computers do what we tell them to do
  - Computer Programming involves writing instructions and giving them to the computer to complete a task.

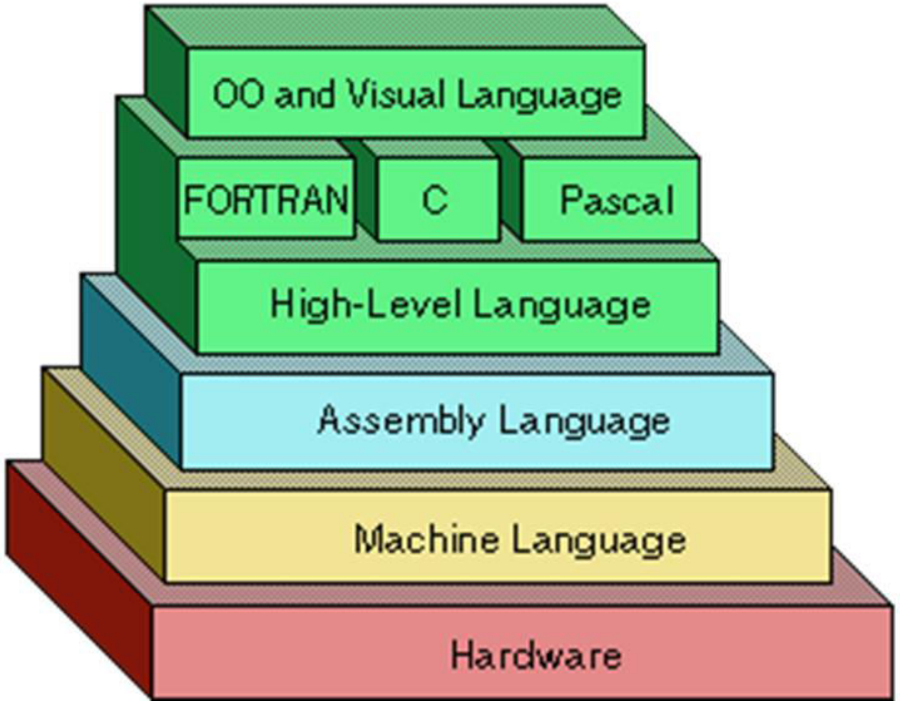
- A computer program or software is:
  - A set of instructions written in a computer language in order to be executed by a computer to perform a useful task

- Ex: Application software packages, such as word processors, spreadsheets and databases are all computer programs

## Types of programming errors

- *Syntax of a programming language* is the set of rules that to be followed while writing a program
- syntax error occurs when the rules are violated
- *run-time errors occur :*
- *logic error :*
- The process of finding and correcting errors in a program is called *debugging*

- Categories of ***programming languages are***
  - ***Machine***
  - Assembly
  - high-level

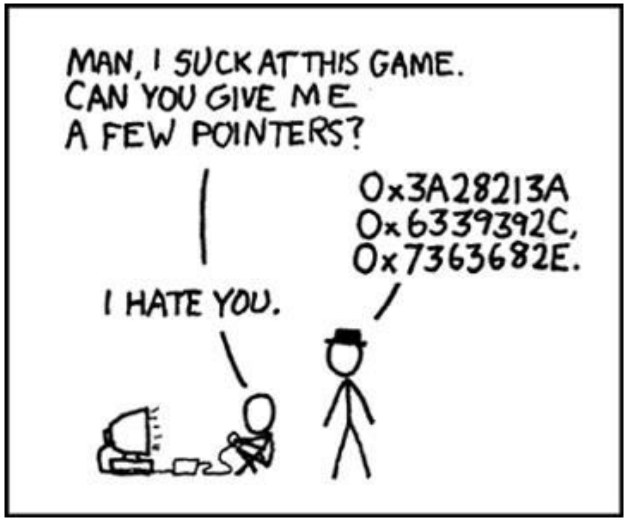


## Machine language is

- Made up of binary 1s and 0s this is the only programming language the computers can understand

- ***Advantages of machine languages are:***
  - fast execution speed and efficient use of main memory

- ***Disadvantages of machine languages are***
  - writing machine language is tedious, difficult and time consuming


MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A 0x6339392C, 0x7363682E.
I HATE YOU.

## Types of language-translator *programs*

- ***Assemblers***
  - translates the program in ***assembly-language into machine-language***
- ***compilers***
  - translates ***high-level language program into machine-language all at once***
- ***Interpreters***
  - translates ***high-level language into machine-language a line at a time***

## Stages of compilation

- mainly the C’s program building process involves four stages and utilizes different ‘tools’ such as a preprocessor, compiler, assembler, and linker.

- **Preprocessing** is the first phase of any C compilation.
  - It processes include-files, conditional compilation instructions and macros.

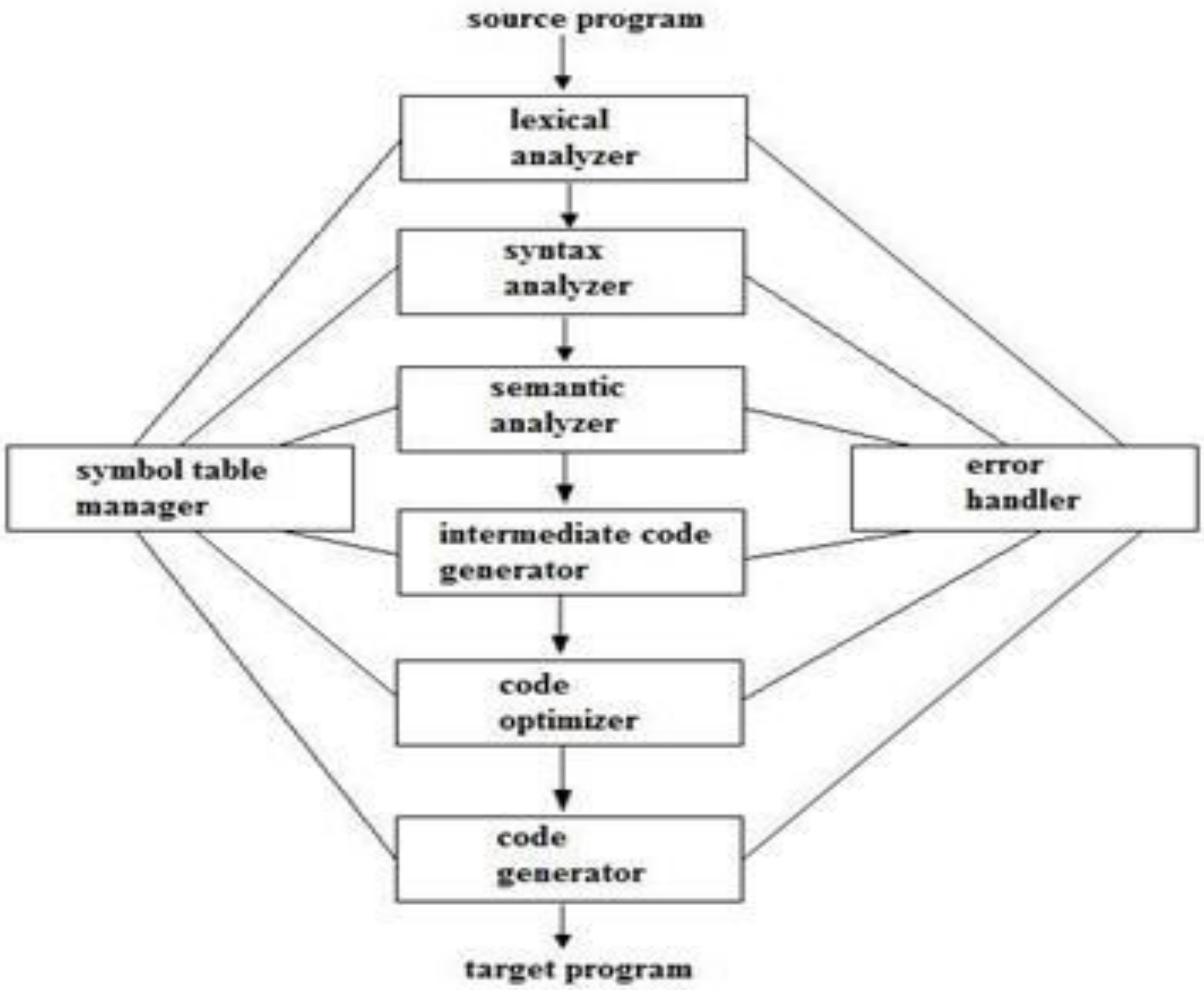
- **Compilation** is the second phase.
  - It takes the output of the preprocessor, and the source code, and generates assembler source code.

## Stages of compilation

- **Assembly** is the third stage of compilation.
  - It takes the assembly source code and produces an assembly listing with offsets.
  - The assembler output is stored in an object file.

- **Linking** is the final stage of compilation.
  - It takes one or more object files or libraries as input and combines them to produce a single (usually executable) file.

  - In doing so, it resolves references to external symbols, assigns final addresses to procedures/functions and variables, and revises code and data to reflect new addresses (a process called relocation).

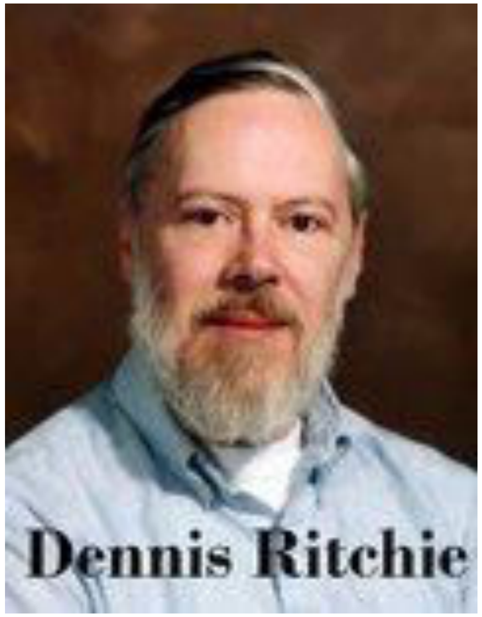

source program
lexical analyzer
syntax analyzer
semantic analyzer
symbol table manager
error handler
intermediate code generator
code optimizer
code generator
target program

## Introduction to  C Language

## History of C Language

- C programming language was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T and BELL LABS.

- Dennis Ritchie is known as the founder of c language.



## History of C Language

- It was developed to overcome the problems of previous languages such as B, BCPL etc.

- Initially, C language was developed to be used in UNIX operating system.
    - It inherits many features of previous languages such as B and BCPL.

- Let's see the programming languages that were developed before C language.

| Language | Year | Developed By |
|---|---|---|
| Algol | 1960 | International Group |
| BCPL | 1967 | Martin Richard |
| B | 1970 | Ken Thompson |
| Traditional C | 1972 | Dennis Ritchie |
| K & R C | 1978 | Kernighan & Dennis Ritchie |
| ANSI C | 1989 | ANSI Committee |
| ANSI/ISO C | 1990 | ISO Committee |
| C99 | 1999 | Standardization Committee |

## Writing and Running Programs

1. Write text of program (source code) using an editor such as emacs, save as file e.g. my_program.c

2. Run the compiler to convert program from source to an “executable” or “binary”:

```
$ gcc –Wall –g my_program.c –o my_program
```

-Wall –g ?

3-N. Compiler gives errors and warnings; edit source file, fix it, and re-compile

N. Run it and see if it works ☺

```
$ ./my_program
Hello World
$ █
```

. / ?

What if it doesn’t work?

TATA ELXSI engineering creativity

## C Syntax and Hello World

#include inserts another file. ".h" files are called "header" files. They contain stuff needed to interface to libraries and code in other ".c" files.

Can your program have more than one .c file?

What do the < > mean?

This is a comment. The compiler ignores this.

```
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
  printf("Hello World\n");
  return 0;
}
```

The main() function is always where your program starts running.

Blocks of code ("lexical scopes") are marked by { … }

Return '0' from this function

Print out a message. '\n' means "new line".

# A Quick Digression About the Compiler

```
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
  printf("Hello World\n");
  return 0;
}
```

**Preprocess**

```
__extension__ typedef  unsigned long long int   __dev_t;
__extension__ typedef  unsigned int   __uid_t;
__extension__ typedef  unsigned int   __gid_t;
__extension__ typedef  unsigned long int   __ino_t;
__extension__ typedef  unsigned long long int   __ino64_t;
__extension__ typedef  unsigned int   __nlink_t;
__extension__ typedef  long int   __off_t;
__extension__ typedef  long long int   __off64_t;
extern void flockfile (FILE *__stream)  ;
extern int ftrylockfile (FILE *__stream)  ;
extern void funlockfile (FILE *__stream)  ;
int main(int argc, char **argv)
{
  printf("Hello World\n");
  return 0;
}
```

**Compile**

my_program

Compilation occurs in two steps: "Preprocessing" and "Compiling"

Why ?

In Preprocessing, source code is "expanded" into a larger form that is simpler for the compiler to understand. Any line that starts with '#' is a line that is interpreted by the Preprocessor.

• Include files are "pasted in" (#include)
• Macros are "expanded" (#define)
• Comments are stripped out ( /*  */ , // )
• Continued lines are joined ( \ )

\ ?

The compiler then converts the resulting text into binary code the CPU can run directly.

TATA ELXSI engineering creativity

# OK, We're Back.. What is a Function?

A Function is a series of instructions to run. You pass Arguments to a function and it returns a Value.

"main()" is a Function. It's only special because it always gets called first when you run your program.

Return type, or void

Function Arguments

```
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
  printf("Hello World\n");
  return 0;
}
```

Calling a Function: "printf()" is just another function, like main(). It's defined for you in a "library", a collection of functions you can call from your program.

Returning a value

## Lexical Elements

- lexical elements that make up C source code after preprocessing.
- These elements are called tokens.
    - Identifiers
    - Keywords
    - Constants
    - Operators
    - Separators
    - White Space

## Identifiers

- Identifiers are used for naming variables,  functions , new data types and preprocessor macros.

- Rules to name the  identifiers:
  - It can  include letters, decimal digits, and the underscore character '_' in identifiers.
  - The first character of an identifier cannot be a digit.
  - Lowercase letters and uppercase letters are distinct.

## Keywords

- Keywords are special identifiers reserved for use as part of the programming language itself.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Auto** | **break** | **case** | **char** | **const** | **continue** | **default** | **do** | **double** | **else** |
| **enum** | **extern** | **float** | **for** | **goto** | **if** | **int** | **long** | **register** | **return** |
| **short** | **signed** | **sizeof** | **static** | **struct** | **switch** | **typedef** | **union** | **unsigned** | **void** |
| **volatile** | **while** | | | | | | | | |

## Constants

- Integer Constants
- Character Constants
- (float) Real Number Constants
- String Constants

## Constants

- A *constant* is a data storage location used by the program.
- Unlike a variable, the value stored in a constant can't be changed during program execution

**C has two types of constants**

1. Literal Constants
2. Symbolic Constants

## Literal Constants

- A *literal constant* is a value that is typed directly into the source code wherever it is needed.
- count = 20; tax_rate = 0.28;

- Decimal constants can contain the digits 0 through 9 and a leading minus or plus sign.

- Octal constants can contain the digits 0 through 7 and a leading minus or plus sign.

- Hexadecimal constants can contain the digits 0 through 9, the letters A through F, and a leading minus or plus sign

## Integer constants

- An integer constant is a sequence of digits, with an optional prefix to denote a number base.

- If the sequence of digits is preceded by 0x or 0X
  - 0xAB43
  - 0xAbCd
- If the first digit is 0 (zero) then it is Octal
  - 057
  - 012
- In all other cases
  - 459
  - 23901

## Character Constants

- A character constant is usually a single character enclosed within single quotation marks, such as 'Q'

- \b: Backspace character.
- \f :Form feed.
- \n :Newline character.
- \r :Carriage return.
- \t :Horizontal tab.
- \o, \oo, \ooo : Octal number.
- \xh, \xhh, \xhhh : Hexadecimal number.

## String Constants

- A string constant is a sequence of zero or more characters, digits, and escape sequences enclosed within double quotation marks.

- A string constant is of type “array of characters”.

- All string constants contain a null termination character (\0) as their last character.

  - "Hello World"
  - "007 Bond"

## float numbers

- Real number constants can also be followed by e or E, and an integer exponent. The exponent can be either positive or negative.

- Float constants:
  - 3.145f
  - 0.55f
  - 100000.45f

- double constants:
  - x = 5e2;   /* x is 5 * 100, or 500.0. */
  - y = 5e-2;  /* y is 5 * (1/100), or 0.05. */
  - 3.14567
  - 10000.000
  - 78263.61253

## Constants

**2. Symbolic Constants**

A *symbolic constant* is a constant that is represented by a name (symbol) in the program. Like a literal constant, a symbolic constant can't change.

C has two methods for defining a symbolic constant: the #define directive and the const keyword.

**#define *CONSTNAME literal***

This creates a constant named *CONSTNAME* with the value of *literal*. *literal* represents a literal constant

e.g.. **#define PI 3.14159** /* a constant for PI is defined. */

**Defining Constants with the const Keyword**

```
const int count = 100;
const float pi = 3.14159;
const long debt = 12000000;
```

## Variable Declarations

A variable declaration has the following form:

***typename varname*;** //typename specifies the type of variable
// *varname* is the variable name

**e.g..**

**int** count, number, start; /* three integer variables */
**float** percent, total; /* two float variables */

## Initializing Numeric Variables

If the value of the variable—isn’t defined. It might be zero, or it might be some random “garbage” value. Before using a variable, always initialize it to a known value.

```
int count;   /* Set aside storage space for count */
count = 0;   /* Store 0 in count */
```

Be careful not to initialize a variable with a value outside the allowed range. Here are two examples of out-of-range initializations:

```
int weight = 10000000000;
unsigned char value = 2500;
```

The C compiler doesn’t catch such errors. The program might compile and link, but unexpected results might be obtained when the program is made to run.

## Expressions

- An *expression* consists of at least one operand and zero or more operators.

- Operands are objects such as constants, variables, and function calls that return values.

  - 47
  - 2 + 2
  - cosine(3.14159) /* We presume this returns a floating point value. */

- Parentheses group subexpressions:
  - ( 2 * ( ( 3 + 10 ) - ( 2 * 6 ) ) )

## Operators

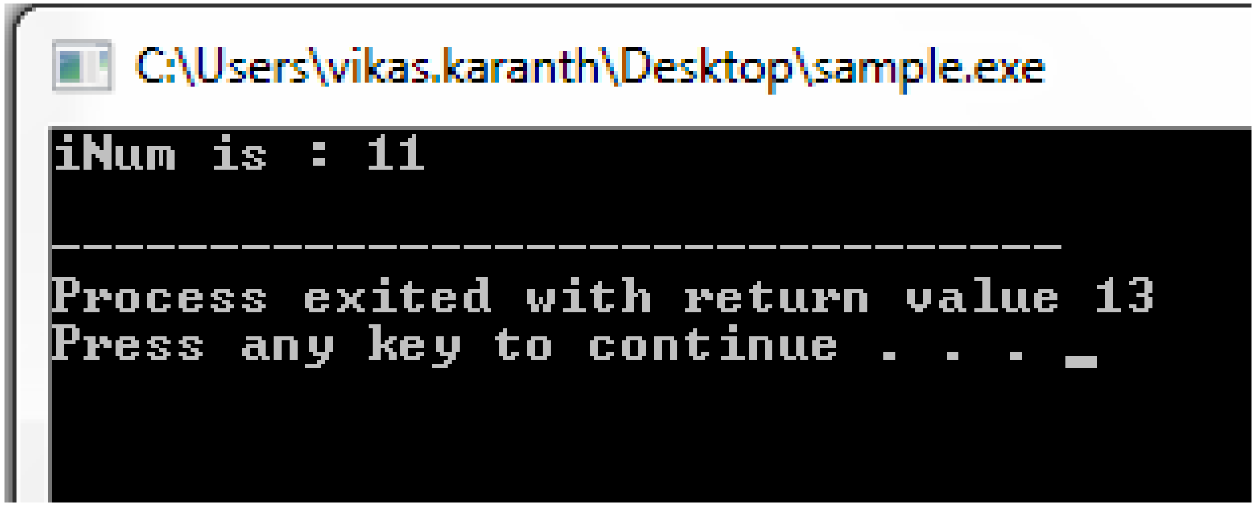
- There are three categories of operators in C.
  - Unary operators
  - Binary Operators
  - Ternary Operators

## Types of operators

- Unary : [ + - ! ~ ++ - - (type)* & sizeof ]
- Arithmetic Operators : [ * , / , + , - ]
- Relational Operators: [ <,>,==,!=,<=,>= ]
- Logical Operators [ &&,||,! ]
- Bitwise Operators [ << ,>>,^,&,|, ]
- Ternary or Conditional Operators: [ :? ]
- Assignment Operator : [ = += -= *= /= %=>>= <<= &= ^= |= ]

## Example pre - Incrementing

```
#include<stdio.h>
main()
{
        int iNum = 10;
        printf("iNum is : %d\n",++iNum);
}
```



C:\Users\vikas.karanth\Desktop\sample.exe

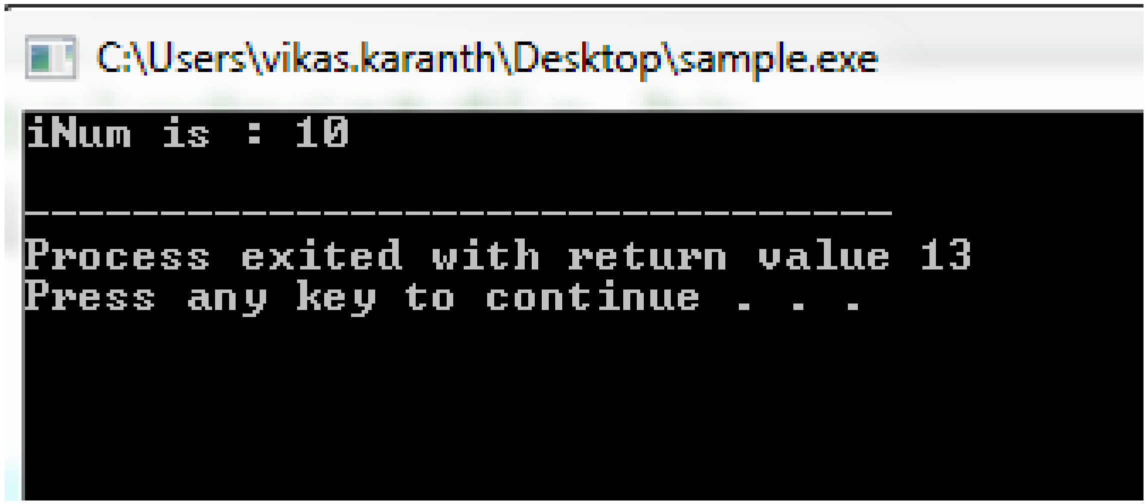
```
iNum is : 11

--------------------------------
Process exited with return value 13
Press any key to continue . . .
```

TATA ELXSI engineering creativity

## Example Post - increment

```
#include<stdio.h>
main()
{
        int iNum = 10;
        printf("iNum is : %d\n",iNum++);
}
```



C:\Users\vikas.karanth\Desktop\sample.exe

```
iNum is : 10

--------------------------------
Process exited with return value 13
Press any key to continue . . .
```

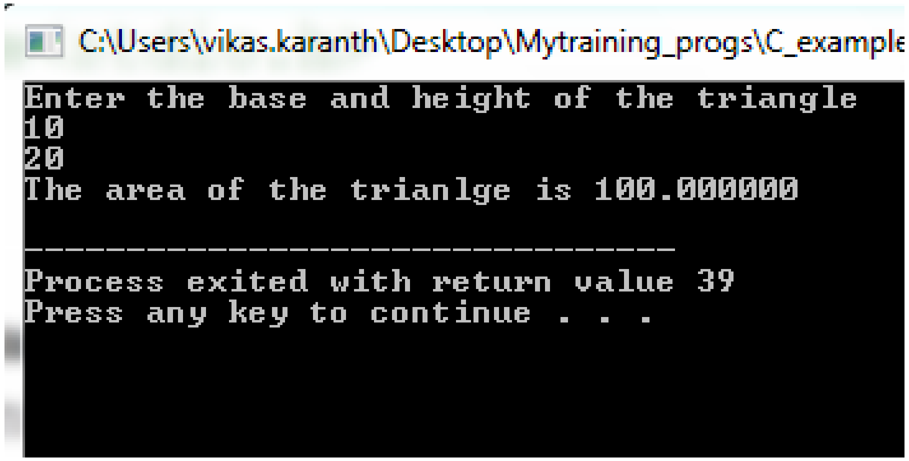
TATA ELXSI engineering creativity

## What is the output ?

```
#include<stdio.h>
main()
{
        int iNum = 10;
        printf("iNum is : %d\n",iNum++);
        printf("iNum is : %d\n",++iNum);
        printf("iNum is : %d\n",iNum--);
        printf("iNum is : %d\n",--iNum);
}
```

## Program to find Area of triangle

```
#include<stdio.h>
main()
{
        float base,height,area;
        printf("Enter the base and height of the triangle\n");
        scanf("%f%f",&base,&height);
        area = 1 / 2.0 * base * height;
        printf("The area of the trianlge is %f\n",area);
}
```

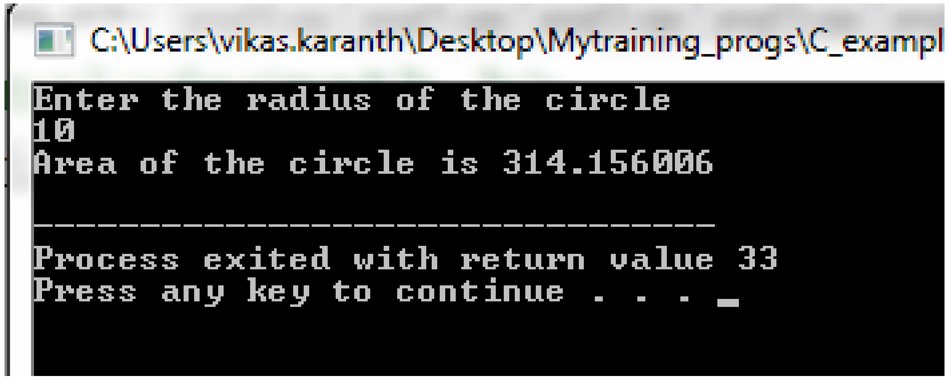


```
C:\Users\vikas.karanth\Desktop\Mytraining_progs\C_example
Enter the base and height of the triangle
10
20
The area of the trianlge is 100.000000

--------------------------------
Process exited with return value 39
Press any key to continue . . .
```

## Program to find Area of circle

```
#include<math.h>
#include<stdio.h>
main()
{
    const float pi=3.14156;
    float area,radius;
    printf("Enter the radius of the circle\n");
    scanf("%f",&radius);
    area=pi*(pow(radius,2));
    printf("Area of the circle is %f\n",area);
}
```



C:\Users\vikas.karanth\Desktop\Mytraining_progs\C_exampl

```
Enter the radius of the circle
10
Area of the circle is 314.156006

--------------------------------
Process exited with return value 33
Press any key to continue . . . _
```

```
$gcc –lm circle.c –o circle.out
```

TATA ELXSI engineering creativity

## What is the output ?

```
main()
{
    int num1,num2;

    printf("Enter any 2 numbers :");
    scanf("%d %d",&num1,&num2);
    printf("num1 = %d \t  num2 = %d\n",num1,num2);

    num1 = num1 + num2;
    num2 = num1 - num2;
    num1 = num1 - num2;

    printf("num1 = %d \t  num2 = %d\n",num1,num2);
}
```

## Relational Operators

| **Operator** | **Symbol** | **Question Asked** | **Example** |
|---|---|---|---|
| Equal | **==** | Is operand 1 equal to operand 2? | x==y |
| Greater than | **>** | Is operand 1 greater than operand 2? | x>y |
| Less than | **<** | Is operand 1 less than operand 2? | x<y |
| Greater than or equal to | **>=** | Is operand 1 greater than or equal to operand 2? | x>=y |
| Less than or equal to | **<=** | Is operand 1 less than or equal to operand 2? | x<=y |
| Not equal to | **!=** | Is operand 1 not equal to operand 2? | x!=y |

## Relational operators

- Relational operators  evaluates either 1 or 0,
- meaning true or false, respectively.

```
if (x == y)
      puts (``x is equal to y");
else
      puts (``x is not equal to y");
if (x != y)
      puts (``x is not equal to y");
else
      puts (``x is equal to y");
```

```
if (x < y)
      puts (``x is less than y");
 if (x <= y)
      puts (``x is less than or equal to y");
if (x > y)
      puts (``x is greater than y");
if (x >= y)
      puts (``x is greater than or equal to y");
```

## Logical operators – and (&&)

- Logical conjunction  operators check the Boolean value of the two expression.
- Any nonzero expression is considered true in C.
- If the first expression is false, then the second expression is not evaluated.

```
if ((x == 5) && (y == 10))
     printf ("x is 5 and y is 10);
```

## Logical operators – OR (||)

- The logical conjunction operator || tests if at least one of two expressions it true.
- If the first expression is true, then the second expression is not evaluated.

```
if ((x == 5) || (y == 10))
    printf (``x is 5 or y is 10");
```

## Logical operators –NOT (!)

- You can prepend a logical expression with a negation operator ! to flip the truth value:

- if (!(x == 5))
- printf ("x is not 5");

# Compound Assignment Operators

C's compound assignment operators provide a shorthand method for combining a binary mathematical operation with an assignment operation

e.g.. using shorthand operator we can write x +=5;

| When Written as... | It Is Equivalent To This |
|---|---|
| x *= y | x = x * y |
| y -= z + 1 | y = y - z + 1 |
| a /= b | a = a / b |
| x += y / 8 | x = x + y / 8 |
| y %= 3 | y = y % 3 |

## The Conditional Operator

The conditional operator is C's only *ternary* operator, meaning that it takes three operands. Its syntax is

*exp1* ? *exp2* : *exp3*;

If *exp1* evaluates to true (that is, nonzero), the entire expression evaluates to the value of *exp2*. If *exp1* evaluates to false (that is, zero), the entire expression evaluates as the value of *exp3*.

1. assigns the value 1 to x if y is true and assigns 100 to x if y is false

e.g. `x = y ? 1 : 100;`

2. to make z equal to the larger of x and y, one could write

```
z = (x > y) ? x : y;
```

## Max 3 using ternary operator

```
main()
{
int num1,num2,num3,max;
printf("Enter any 3 numbers :");
scanf("%d %d %d",&num1,&num2,&num3);
        max=((num1>num2)&(num1>num3)?num1:(num2>num3)?num2:num3);
printf("The greatest of three numbers is %d\n",max);
}
```

# The Comma Operator

The comma is frequently used in C as a simple punctuation mark, serving to separate variable declarations, function arguments, and so on

Separating two sub expressions with a comma can form an expression. The result is as follows:

Both expressions are evaluated, with the left expression being evaluated first.

e.g. i=(j=3 , j+2);

here first value 3 is assigned to j and then the expression j+2 is evaluated giving 5

## Bitwise operators

- In C, following 6 operators are bitwise operators (work at bit-level)

- **& (bitwise AND) :** Takes two numbers as operand and does AND on every bit of two numbers.
- **The result of** AND is 1 only if both bits are 1.

- **| (bitwise OR) :** Takes two numbers as operand and does OR on every bit of two numbers.
- **The result of OR is 1** any of the two bits is 1.

## Bitwise operators

- **^ (bitwise XOR) :** Takes two numbers as operand and does XOR on every bit of two numbers.
- **The result of** XOR is 1 if the two bits are different.

- << (left shift) : Takes two numbers, left shifts the bits of first operand, the second operand decides the number of
- places to shift.

- >> (right shift) Takes two numbers, right shifts the bits of first operand, the second operand decides the number of places to shift.

- ~ (bitwise NOT) Takes one number and inverts all bits of it

```
/* C Program to demonstrate use of bitwise operators */
#include<stdio.h>
int main()
{
     unsigned char a = 5, b = 9; // a = 5(00000101), b = 9(00001001)
     printf("a = %d, b = %d\n", a, b);
     printf("a&b = %d\n", a&b); // The result is 00000001
     printf("a|b = %d\n", a|b); // The result is 00001101
     printf("a^b = %d\n", a^b); // The result is 00001100
    printf("~a = %d\n", a = ~a); // The result is 11111010
    printf("b<<1 = %d\n", b<<1); // The result is 00010010
    printf("b>>1 = %d\n", b>>1); // The result is 00000100
    return 0;
}
```

## Following are interesting facts about bitwise operators.

- 1) The left shift and right shift operators should not be used for negative numbers (the resulting value is implementation-defined.)
- Also, if the number is shifted more than the size of integer, the behaviour is undefined.
- For example, 1 << 33 is undefined if integers are stored using 32 bits.

## Bitwise operators – left shift right shift

```
#include<stdio.h>
int main()
{
    int x = 31;
    printf ("x << 1 = %d\n", x << 1); //62
    printf ("x >> 3 = %d\n", x >> 3); //3 = 31 / 8
    return 0;
}
```

## Precedence and Order of Evaluation

| Operators | Associativity |
|---|---|
| () [] -> . | left to right |
| ! ~ ++ -- +(u) -(u) *(u) sizeof() | right to left |
| * / % | left to right |
| + - | left to right |
| << >> | left to right |
| < <= > >= | left to right |
| == != | left to right |
| & , ^ , \| | left to right |
| && , \|\| | left to right |
| ?: | right to left |
| = += -= *= /= %= &= ^= \|= <<= >>= | right to left |

## Type cast

- Type cast is to explicitly cause an expression to be of a specified data type.
- A type cast consists of a type specifier enclosed in parentheses, followed by an expression.
- To ensure proper casting, you should also enclose the expression that follows the type specifier in parentheses.

```
float x;
int y = 7;
int z = 3;
x = ((float) (y) / z));
```

## statement

- We write statements to cause actions and to control flow within programs.
- You can also write statements that do not do anything at all, or do things that are uselessly trivial.
- iTotal = 100 + 200; // simple statement
- while(1) {   }  // compound stement

## Expression and Statements

- Primary expressions are identifiers, constants, strings, or expressions in parentheses. *primary-expression*
  *identifier*
  *constant*
  *string*
  *(expression)*
- An identifier is a primary expression, provided it has been suitably declared as discussed below.

- Example Primary expression
- int x=5 ;
- If(x) { //…..} x evolvates as 5 hence TRUE

## Expression and Statements

- Expressions can be turned into a statement by adding a semicolon to the end of the expression.

- This will not be evaluated due to optimization
  - 3 / 0 ;
  - 2 + 2;
  - 10 >= 9;

- Expression statements are only useful when they have some kind of side effect, such as storing a value, calling a function, or (this is esoteric) causing a fault in the program.
  - x = 3 / 0;
  - y = 2 + 2;
  - puts ("Hello, world");
  - *iPtr;

## Type qualifiers and Modifiers

## Type qualifiers

- Specifiers modify the meanings of the predefined built-in data types and expand them to a much larger set.

- There are four data type modifiers in C:

  - long
  - short
  - signed
  - unsigned

## Short and long type modifiers

- The intent is that short and long should provide different lengths of integers where practical;

- int will normally be the natural size for a particular machine ( int either 16 or 32 bits. )
- short is often 16 bits long.

- Each compiler is free to choose appropriate sizes for its own hardware,

- subject only to the the restriction that shorts and ints are at least 16bits

- longs are at least 32 bits.

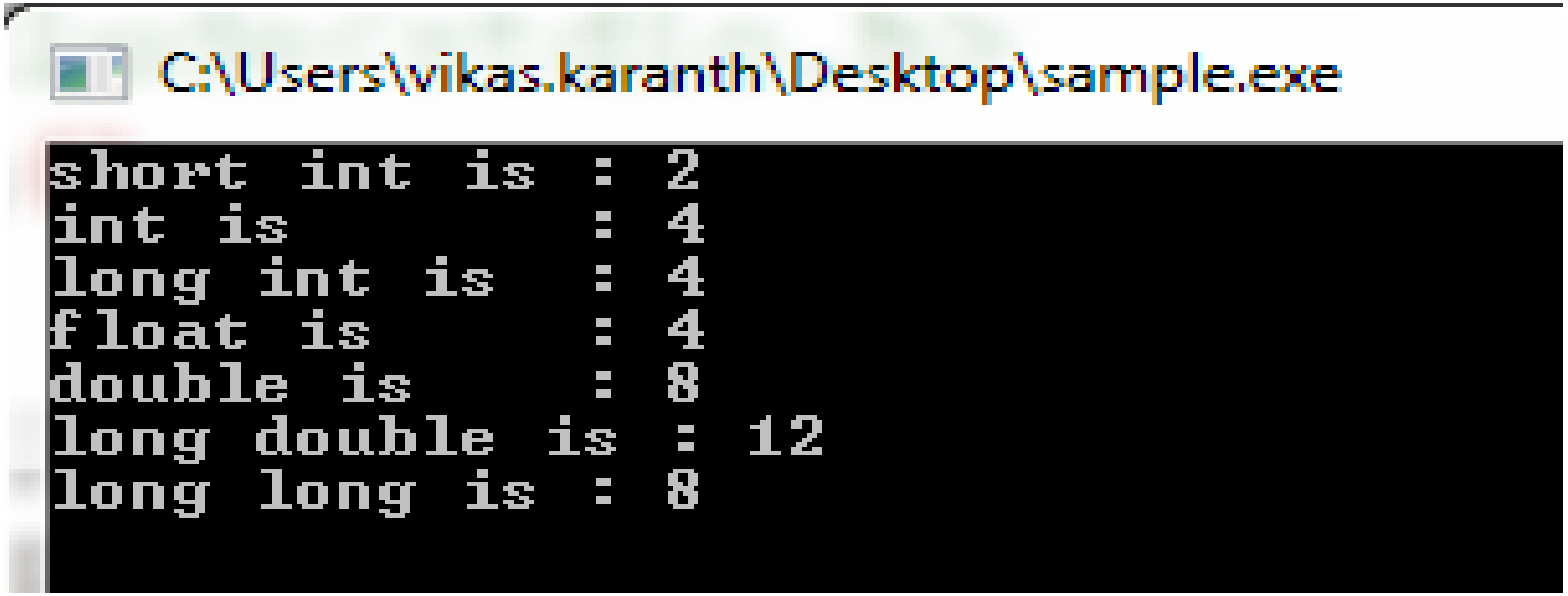
- short is no longer than int, which is no longer than long.

courtesy Dennis Ritchie – ANSI - C

## Example program to check size of data types

```
#include<stdio.h>
main()
{
        printf("short int is : %d\n",sizeof(short int));
        printf("int is       : %d\n",sizeof(int));
        printf("long int is  : %d\n",sizeof(long int));
        printf("float is      : %d\n",sizeof(float));
        printf("double is     : %d\n",sizeof(double));
        printf("long double is : %d\n",sizeof(long double));
        printf("long long is : %d\n",sizeof(long long));
}
```

Output Screen shot: next page


C:\Users\vikas.karanth\Desktop\sample.exe

```
short int is  : 2
int is        : 4
long int is   : 4
float is      : 4
double is     : 8
long double is : 12
long long is  : 8
```

- The type long double specifies extended-precision floating point.
- As with integers, the sizes of floating-point objects are implementation-defined ;
- float, double and long double could represent one, two or three distinct sizes.

## Answer the following

- Is it possible to use:
  - Short char
  - Short float
  - Long char
  - Long float
  - Short double

## Signed and unsigned qualifiers

- The qualifier signed or unsigned may be applied to char or any integer.
- unsigned numbers are always positive or zero, and obey the laws of arithmetic modulo 2^n, where n is the number of bits in the type.
- So, for instance, if chars are 8 bits, unsigned char variables have values between 0 and 255,
- while signed chars have values between -128 and 127 (in a two's complement machine.)
- Whether plain chars are signed or unsigned is machine-dependent,
- but printable characters are always positive.

courtesy Dennis Ritchie – ANSI - C

# Type qualifiers

- Type Qualifiers : which can prepend to variable declarations which change how the variables may be accessed:

- There are two type qualifiers.
  - const
  - volatile.

- const causes the variable to be read-only; after initialization, its value may not be changed.
  - const float pi = 3.14159f;

- In addition to helping to prevent accidental value changes, declaring variables with const can aid the compiler in code optimization.

## volatile

The modifier volatile tells the compiler that a variable's value may be changed in ways not explicitly specified by the program.

For example, a global variable's address may be passed to the operating system's clock routine and used to hold the real time of the system.

In this situation, the contents of the variable are altered without any explicit assignment statements in the program.

Volatile modifier is discussed much in detail in later slides

## Control Structures

## The if Statement

- Statements in a C program normally execute from top to bottom, in the same order as they appear in source code file. A program control statement modifies the order of statement execution.

- In its basic form, the if statement evaluates an expression and directs program execution depending on the result of that evaluation. The form of an if statement is as follows:

- if (*expression*)
- *statement*;

- If *expression* evaluates to true, *statement* is executed. If *expression* evaluates to false, *statement* is not executed.

## The if Statement

a block is a group of two or more statements enclosed in braces.

```
    if (expression)
{
    statement1;
    statement2;
    /* additional code goes here */
    statementn;
}
```

- An if statement can optionally include an else clause. The else

clause is included as follows:

```
    if (expression)
            statement1;
    else
            statement2;
```

- If *expression* evaluates to true, *statement1* is executed. If *expression* evaluates to false, *statement2* is executed

# If statements in different format

The if Statement

**Form 1**

```
if( expression )
    statement1;
next_statement;
```

**Form 2**

```
if(expression)
        statement1;
else{
         statement2;
         next statement;
}
```

**Form 3**

```
if( expression1 )
                 statement1;
else if( expression2 )
                statement2;
else
                statement3;
next_statement;
```

## A program to find max of 2 numbers

```
main()
{
int num1,num2;
printf("\n Enter any two numbers");
scanf("%d %d",&num1,&num2);
if(num1 > num2)
        printf("%d is greater than %d\n",num1,num2);
else
        printf("%d is greater than %d\n",num2,num1);
}
```

## Find Max of 3 numbers

```
main()
{
int num1,num2,num3,max;
printf("Enter any 3 numbers :");
scanf("%d%d%d",&num1,&num2,&num3);
if(num1>num2)
        if(num1>num3)
                max = num1;
        else
                max = num3;
        else if(num2> num3)
                max = num2;
        else
                max = num3;
        printf("The greatest of three numbers is %d\n",max);
}
```

## The switch Statement

C's most flexible program control statement is the switch statement, which lets the program to execute different statements based on an expression that can have more than two values.

```
 switch  (expression)
{
   case  template_1: statement(s);
    case  template_2: statement(s);
    -----
   case  template_n: statement(s);
   default: statement(s);
}
```

The switch statement allows for multiple branches from a single expression. It's more efficient and easier to follow than a multileveled if statement.

## The do...while Loop

the do...while loop, which executes a block of statements as long as a specified condition is true. The do...while loop tests the condition at the end of the loop rather than at the beginning, as is done by the for loop and the while loop.

```
do
        statements
while(condition);
```

**1.** The statements in *statement* are executed.

**2.** *condition* is evaluated. If it's true, execution returns to step 1. If it's false, the loop terminates.

The statements associated with a do...while loop are always executed at least once. This is because the test condition is evaluated at the end, instead of the beginning, of the loop.

## A program to find reverse of a number

```
#include<stdio.h>
main()
{
int num,rev,r;
char rep='y';
do
{
        system("clear");
        printf("\n ENTER ANY number:");
        scanf("%d",&num);
        printf("THE reverse OF THE NUMBER i.e '%d' IS: ",num);
        rev=0;
```

# A program to find reverse of a number

```
        while(num!=0)
        {
                r=(num % 10);
                num=(num / 10);
                rev=rev*10+r;
        }
        printf("%d\n",rev);
        printf("Do you wish to continue [y/n] or [Y/N]:")
        _fpurge(stdin);
        scanf("%c",&rep);
    }
    while(rep=='y' || rep=='Y');
    printf("Press any key return back to the program..........");
    }
```

## Exiting the Program

A C program normally terminates when execution reaches the closing brace of the main() function.

However, a program can be terminated at any time by calling the library function exit().

- **The exit() Function**

  exit(0) : If *status* has a value of 0, it indicates that the program terminated normally.

  exit(1) : A value of 1 indicates that the program terminated with some sort of error.

  To use the exit() function, a program must include the header file stdlib.h.

## Ending loops early:  by Break

- In loops, termination or exit of the loop occurs only when a certain condition occurs.
- To exert more control over loop execution.
- The break and continue statements provide this control.

- break is used inside a loop or switch statement. It causes the control of a program to skip past the end of the current loop (for, while, or do...while) or switch statement. No further iterations of the loop execute;
- The following is an example:

```
for (count = 0; count < 10; count++)
          if ( count == 5 )
              break;
```

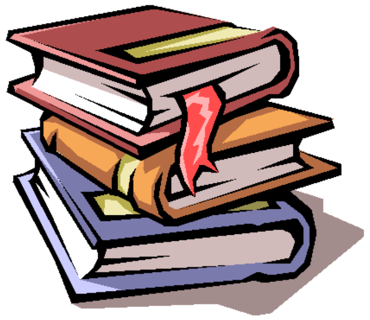
## The continue Statement

The continue statement can be placed only in the body of a for loop.

When a continue statement executes, the next iteration of the enclosing loop begins immediately.
The statements between the continue statement and the end of the loop aren't executed.

```
E.g.int x;
      printf("Printing only the even numbers from 1 to 10\n");
      for( x = 1; x <= 10; x++ )
      {
            if( x % 2 != 0 )    /* See if the number is NOT even */
                  continue;      /* Get next instance x */
            printf( "\n%d", x );
      }
```

## references



- The GNU C Reference Manual : https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

- The GNU C Reference Manual : http://www.cprogramming.com/

- C Programming Language

by Brian W. Kernighan , Dennis Ritchie

- C: The Complete Reference

by Herbert Schildt

## Thank you

TATA ELXSI

ITPB Road  Whitefield

Bangalore 560 048  India
Tel +91 80 2297 9123
Fax +91 80 2841 1474
e-mail info@tataelxsi.com

www.tataelxsi.com