

# History Of C

- The C programming language was devised in the early 1970s by Dennis M. Ritchie in Bell Labs (AT&T).
- The programming language C was written down, by **Kernighan and Ritchie**, in a book called “**The C Programming Language**, 1st edition”.
- For years the book “The C Programming Language, 1st edition” was the standard on the language C.
- In 1983 a committee was formed by the **American National Standards Institute** (ANSI) to develop a modern definition for the programming language C.

# History Of C

- In 1988 they delivered the final standard definition **ANSI C**. (The standard was based on the book from **K&R** 1st ed.).
- The standard ANSI C made little changes on the original design of the C language.
- Later on, the ANSI C standard was adopted by the International Standards Organization (ISO).

# Importance Of C Language

- **Robust language:** Rich setup of built-in functions and operators.
- **Portable language:** Programs written for one computer system can be run on another system, with little or no modification
- **Flexible language:** Has its ability to extend itself. A c program is basically a **collection of functions** that are supported by the C library. We can continuously add our own functions to the library with the availability of the large number of functions

# Importance Of C Language

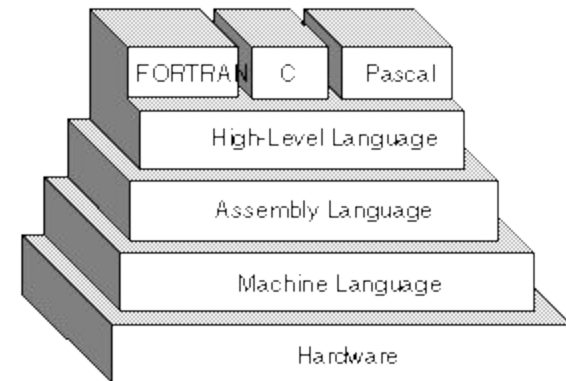
Has several variety of data types and powerful operators

Suitable to write both application software as well as system software

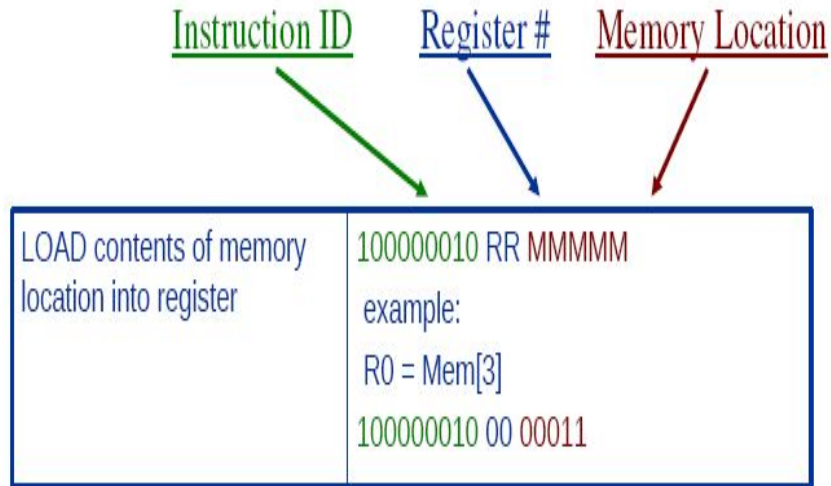
Well suited for structured programming. This makes the programmer to think in terms of modules or blocks. This in turn makes program debugging, testing and maintaining easier.

# Languages

- **Machine languages** consist entirely of numbers and are almost impossible for humans to read and write.
- **Assembly languages** have the same structure and set of commands as machine languages, but they enable a programmer to use names instead of numbers.
- Each type of CPU has its own machine language and assembly language, so an assembly language program written for one type of CPU won't run on another.



# Ex. Machine Language & Assembly Language



100000010	01	00101	Load Memory 5 -> R1
100000010	10	00101	Load Memory 5 -> R2
1010000100	00	01 10	R1 + R2 -> R0
100000100	00	00110	Store R0 -> Memory 6
1111111111111111			

<u>Machine Language</u>	<u>Equivalent Assembly</u>
1000000100100101	LOAD R1 5
1000000101000101	LOAD R2 5
1010000100000110	ADD R0 R1 R2
1000001000000110	SAVE R0 6
1111111111111111	HALT

- **High-level and Low-level term are used to differentiate any computer programming language whether it is easily understandable to human or not.**

High Level Language	Low Level Language	Middle level language
<ul style="list-style-type: none"><li>• <b>High Level Language means the language is easily understandable by human</b></li><li>• High level programming languages are more structured, are closer to spoken language</li><li>• Higher level languages are also easier to read.</li></ul>	<ul style="list-style-type: none"><li>• Low Level Language means the language is more to a machine language than human understandable language.</li><li>• Ex: Machine Level Language</li></ul>	<ul style="list-style-type: none"><li>▪ It supports both <b>high level language</b> and low level language that is machine level language.</li></ul>

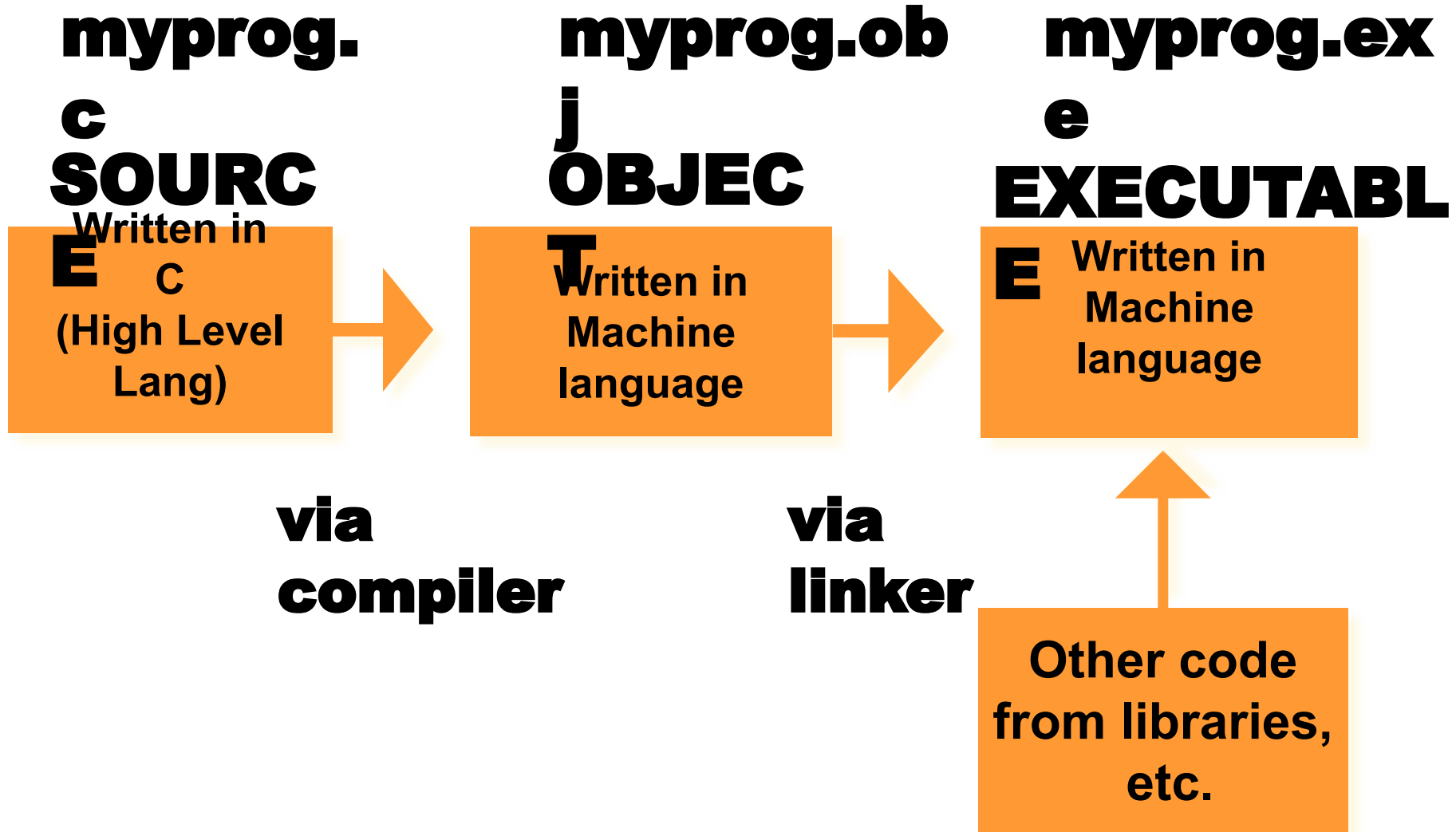
# C Language

- C Programming language is called as Middle Level Language
  1. It gives or behaves as High Level Language through Functions - gives a modular programming and breakup, increased efficiency for re-usability
  2. It gives access to the low level memory through Pointers. Moreover it does support the Low Level programming i.e, Assembly Language.

As its a combination of these two aspects, its neither a High Level nor a Low level language but a Middle Level Language.



# Three C Program Stages



# Compiler, Interpreter, Linker, Loader

## Compiler:

- It is a program which translates a high level language program into a machine language program.
- It checks all kinds of limits, ranges, errors etc.
- It has slow speed. Because a compiler goes through the entire program and then translates the entire program into machine codes.

- **Interpreter:** An interpreter is a program which translates **statements of a program into machine code.**
- It translates only one statement of the program at a time.
- It reads only one statement of program, translates it and executes it.
- Then it reads the next statement of the program again translates it and executes it. In this way it proceeds further till all the statements are translated and executed.
- On the other hand, a **compiler goes** through the entire program and then translates the entire program into machine codes. A compiler is 5 to 25 times faster than an interpreter.
- By the compiler, the machine codes are saved permanently for future reference.
- On the other hand, the machine codes produced by interpreter are not saved.

<u><b>Linker</b></u>	<u><b>Loader</b></u>
<ul style="list-style-type: none"> <li>•In high level languages, some built in <b>header files or libraries</b> are stored.</li> <li>•These libraries are predefined and these contain <b>basic functions</b> which are essential for executing the program.</li> <li>•These functions are linked to the libraries by a program called Linker.</li> <li>•If linker does not find a library of a function then it informs to compiler and then compiler generates an error.</li> <li>• The compiler automatically invokes the linker as the last step in compiling a program.</li> </ul>	<ul style="list-style-type: none"> <li>•Loader is a program that loads <b>machine codes</b> of a program into the system memory.</li> <li>•In Computing, a <b>loader</b> is the part of an Operating System that is responsible for loading programs.</li> <li>•It is one of the essential stages in the process of starting a program.</li> <li>•Because it places programs into memory and prepares them for execution.</li> <li>•Loading a program involves reading the contents of executable file into memory.</li> </ul>

# Reading Assignment

- Difference between:
- Compiler & Assembler
- Compiler & Interpreter
- High Level & Low Level
- High Level & Middle Level
- Linker & Loader
- Assembler & Interpreter

**Your First**

**C PROGRAM !!!!!**

# C Program File

- All the C programs are written into text files with extension ".c" for example *hello.c*.
- **C Compilers**
- When you write any program in C language then to run that program you need to compile that program using a C Compiler which converts your program into a language understandable by a computer.
- This is called machine language (ie. binary format). So before proceeding, make sure you have C Compiler available at your computer.

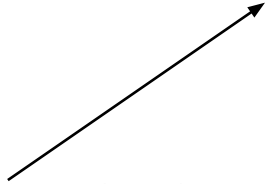
# C Program

- A C program basically has the following form:
- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comments



# Simplest C Program

Hash



```
#include<stdio.h>
int main()
{
printf(“ Programming in C is Interesting”);
return 0;
}
```

# Preprocessor Directive

- **# include:** Is a **preprocessor directive** which directs to include the designated file which contains the declaration of library functions (pre defined).
- **stdio.h (standard input output)** : A header file which contains declaration for standard input and output functions of the program. Like printf, scanf
- When one or more **library functions** are used, the corresponding **header file** where the information about these functions will be present are to be included.

# Functions

- Every C Program will have one or more functions and there is one mandatory function which is called *main()* function.
- This function is prefixed with keyword *int* which means this function returns an integer value when it exits. This integer value is returned using *return statement*.

- **Variables:** are used to hold numbers, strings and complex data for manipulation.
- **Statements & Expressions :** Expressions combine variables and constants to create new values. Statements are expressions, assignments, function calls, or control flow statements which make up C programs.
- **Comments:** are used to give additional useful information inside a C Program. All the comments will be put inside `/*...*/` as given in the example above. A comment can span through multiple lines.

# Note the following:

- C is a case **sensitive programming language**. It means in C *printf* and *Printf* will have different meanings.
- End of each C statement must be **marked with a semicolon**.
- Multiple statements can be one the same line.
- Statements can continue over multiple lines.

# Defining main( )

- When a **C program** is executed, system first calls the **main( )** function, thus a C program **must always** contain the function **main( )** somewhere.

- A function definition has:

**heading**

**{**

**declarations ;**

**statements;**

**}**

# Basic Structure of a C program

**preprocessor directive**

**int main( )**

**{**

**declarations;**

**\_\_\_\_\_;**

**\_\_\_\_\_body\_\_\_\_\_;**

**\_\_\_\_\_;**

**return 0;**

**}**

# Concept of Comment

- Comments are inserted in program to **maintain the clarity** and for future references. They help in easy debugging.
- Comments are **NOT compiled**, they are just for the programmer to maintain the readability of the source code.

Comments are included as

```
/* .....  
..... */
```



# Check this out!

```
#include<stdio.h>
```

```
int main()
```

```
{
```

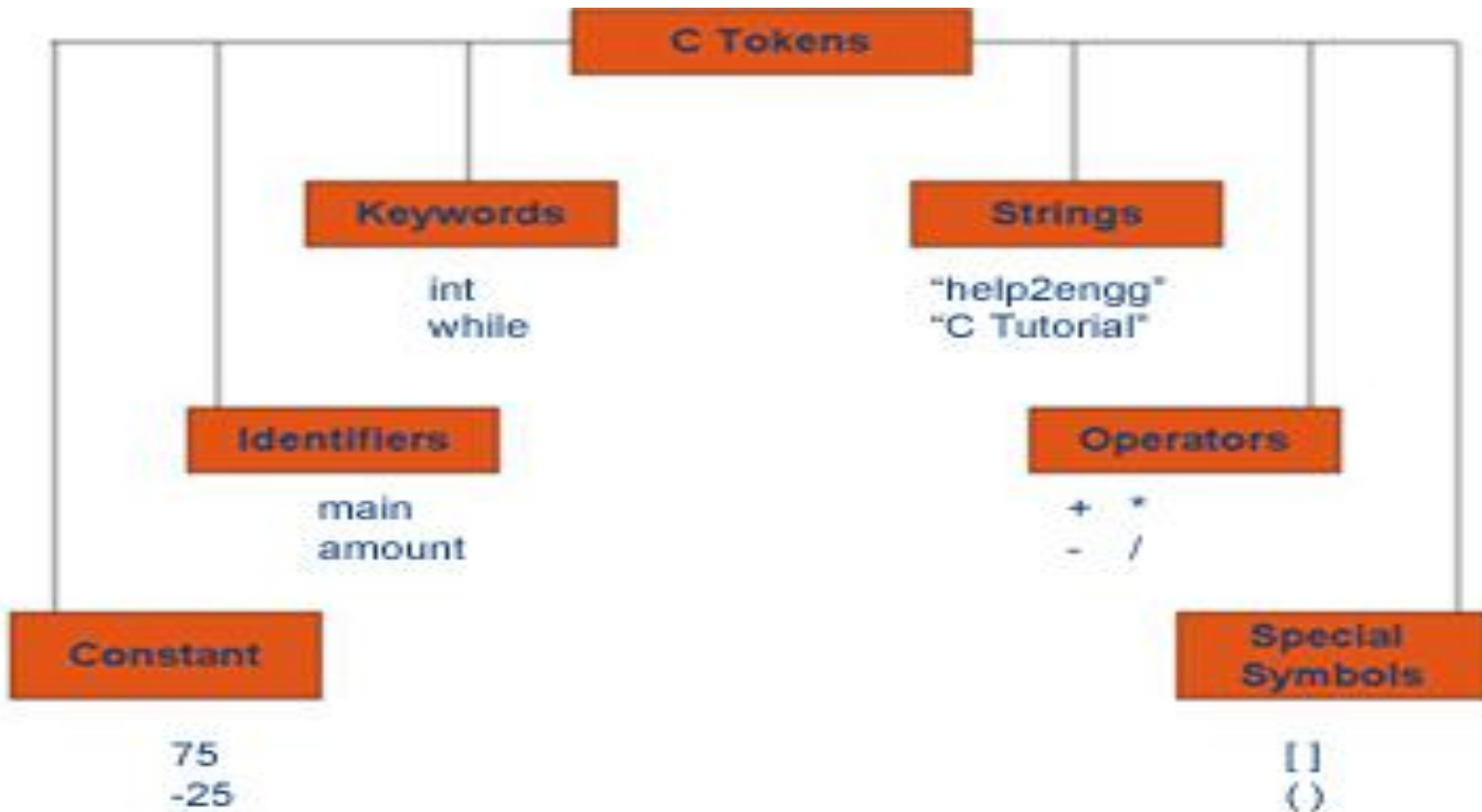
```
printf(" India won the final cricket match in  
Sri Lanka ");
```

```
return 0;
```

```
}
```

# C Tokens

- Tokens are individual words and punctuation marks in passage of text. In C, program the smallest individual units are known as C Tokens.
- C has **Six types of Tokens**. The Tokens are shown in figure.



# Notion of keywords

- Keywords are certain **reserved words**, that have standard predefined meanings in C.
- All keywords are in **lowercase**.
- Some keywords in C:

<b>auto</b>	<b>extern</b>	<b>sizeof</b>	<b>break</b>	<b>static</b>	
<b>case</b>	<b>for</b>	<b>struct</b>	<b>goto</b>	<b>switch</b>	
<b>const</b>	<b>if</b>	<b>typedef</b>	<b>enum</b>	<b>signed</b>	
<b>default</b>	<b>int</b>	<b>union</b>	<b>long</b>	<b>continue</b>	
<b>unsigned</b>	<b>do</b>	<b>register</b>	<b>void</b>	<b>double</b>	<b>return</b>
<b>volatile</b>	<b>else</b>	<b>short</b>	<b>while</b>	<b>float</b>	<b>char</b>

# Identifiers

- In C programming, identifiers are **names given to C entities**, such as **variables, functions, structures** etc.
- Identifier are created to give unique name to C entities to identify it during the execution of program.

- For example:

*int money;*

*int mango\_tree;*

# Identifiers and Variables

- **Identifier :**

- A name has to be devised for program elements such as variables or functions.

- **Variable:**

- Variables are memory regions you can use to hold data while your program is running.
- Thus variable has an **unique address** in memory region.
- For your convenience, you give them names

- Contains only letters, digits and under score characters,  
**example1, amount, hit\_count**

# Rules for Identifiers

1. Must begin with either a letter of the alphabet or an underscore character.
1. Uppercase letters are different than lowercase, **example amount, Amount and AMOUNT** all three are different identifiers.
1. Maximum length can be **31** characters.
1. Should not be the same as one already defined in the library, **example it can not be printf / scanf.**
1. No special characters are permitted. **e.g. blank space, period, semicolon, comma etc.**

## Naming Variables

There are several rules that you must follow when naming variables:

Variable names....	Example
CANNOT start with a number	2times
CAN contain a number elsewhere	times2
CANNOT contain any arithmetic operators...	a*b+c
... or any other punctuation marks...	#@%£!!
... but may contain or begin with an underscore	_height
CANNOT be a C keyword	while
CANNOT contain a space	stupid me
CAN be of mixed cases	HelloWorld

# Character set

- The character set in C Language can be grouped into the following categories.

**1. Letters 2. Digits 3. Special Characters 4. White Spaces**

## Alphabets:

Uppercase: A B C ..... X Y Z

Lowercase: a b c ..... x y z

## Digits:

0 1 2 3 4 5 6 8 9

## Special Characters:

Special Characters in C language , < > . \_ ( ) ; \$ : % [ ] # ? ‘ & { } “ ^ ! \* / | - \ ~ +

**White space Characters:** blank space, new line, horizontal tab, carriage return



# Constants

- Constants are the terms that can't be changed during the execution of a program.
- In C, constants can be classified as:

**Integer constants:** Integer constants are the numeric constants without any fractional part or exponential part.

There are *3 types of integer constants* in C language: decimal constant(base 10), octal constant(base 8) & hexadecimal constant(base 16).

- Decimal digits: 0 1 2 3 4 5 6 7 8 9
- Octal digits: 0 1 2 3 4 5 6 7
- Hexadecimal digits: 0 1 2 3 4 5 6 7 8 9 A B C D E F.
- For example:

Decimal constants: 0, -9, 22 etc

Octal constants: 021, 077, 033 etc

Hexadecimal constants: 0x7f, 0x2a,

- **Floating-point constants**

Floating point constants are the numeric constants that has either fractional form or exponent form.

For example: 1.0, 1.23, 100.002.

*E.g.*, 1.0e10, 1.23e-2, 1.002e+2  
-0.22E-5

**Note:** Here, E-5 represents  $10^{-5}$ . Thus,  $-0.22E-5 = -0.0000022$ .

- **Character constants**

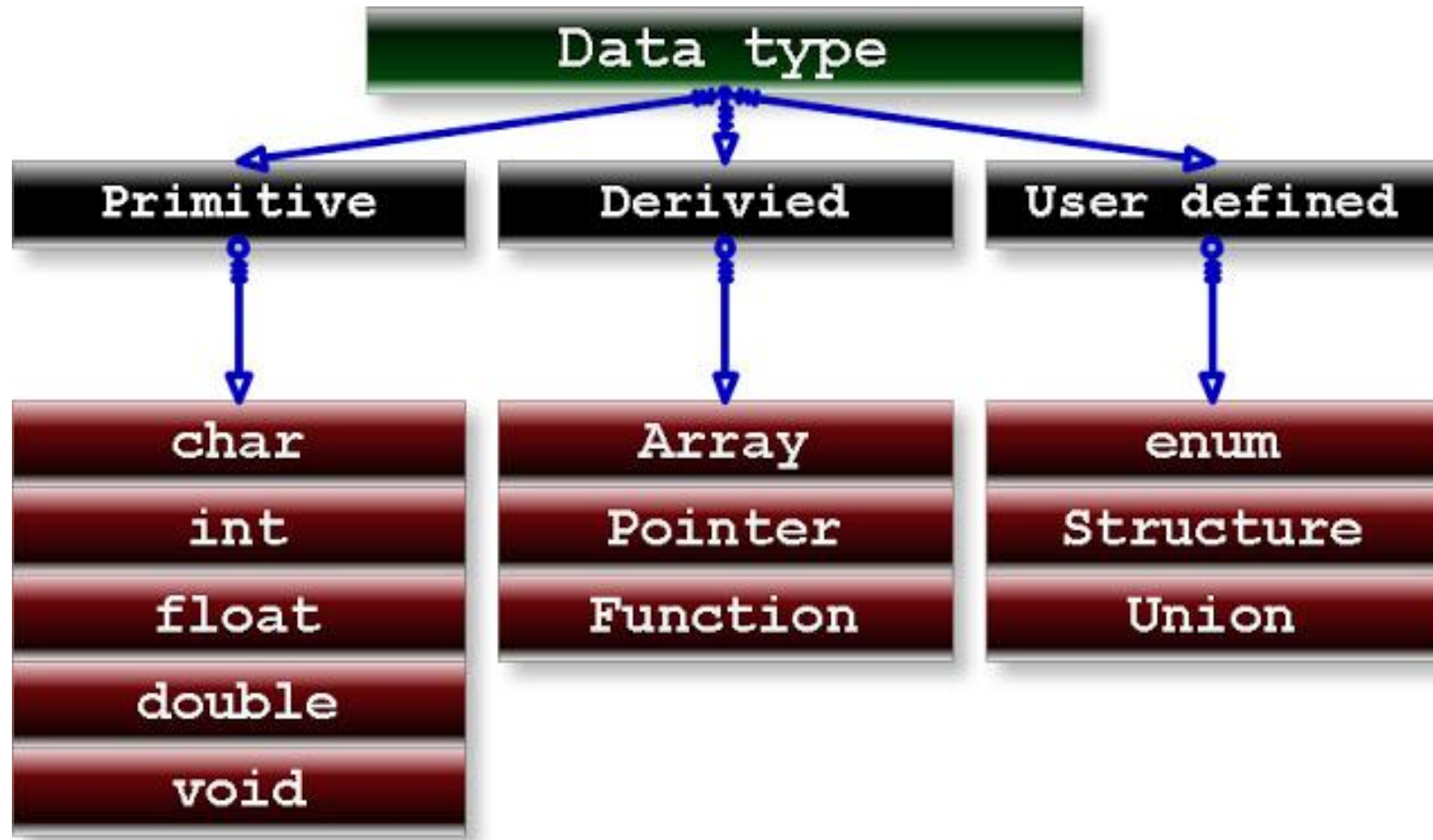
Character constants are the constant which use single quotation around characters. For example: 'a', 'l', 'm', 'F' etc.

- **Escape Sequences are used for**
  - newline(enter),
  - tab,
  - quotation mark etc. in the program which either cannot be typed or has special meaning in C programming.
- For example: `\n` is used for newline.
- The backslash( `\` ) causes "escape" from the normal way the characters are interpreted by the compiler.

Escape Sequences

Escape Sequences	Character
<code>\b</code>	Backspace
<code>\f</code>	Form feed
<code>\n</code>	Newline
<code>\r</code>	Return
<code>\t</code>	Horizontal tab
<code>\v</code>	Vertical tab
<code>\\</code>	Backslash
<code>\'</code>	Single quotation mark
<code>\"</code>	Double quotation mark
<code>\?</code>	Question mark
<code>\0</code>	Null character

# Data Types



# Primitive data type

- A data type is a classification of data, which can store a specific type of information.
- Data types are primarily used in computer programming, in which variables are created to store data.
- Each variable is assigned a data type that determines what type of data the variable may contain.
- **Primitive data types** are predefined types of data, which are supported by the programming language.
- For example, integer For example, integer, character For example, integer, character, and string are all primitive data types.

# Derived Data type

- Data types that are **derived from fundamental data types** are called derived data types.
- Derived data types **don't create a new data type** but, instead they **add some functionality to the basic data types**.
- **Ex: Array** : An array is a collection of variables of same type. They are stored in contiguous memory allocation.
  - `int a[10];`
  - **User Defined Data type**: allows us to define our own types based on other existing data types. In order to do that we shall use keyword **typedef**

# Basic Data Types

- In C, there are 5 basic data types:

1. **char,**

2. **int,**

3. **float**

4. **double**

5. **void**

Each one has its own properties. For instance, they all have different sizes.

The size of a variable can be pictured as the number of memory slots that are required to store it.

# Format specifiers

- There are several format specifiers-The one you use should depend on the type of the variable you wish to print out. The common ones are as follows:

Format Specifier	Type
%d	int
%c	char
%f	float
%lf	double
%ld	long
%s	string

To display a number in scientific notation, use %e.

To display a percentage sign, use %%



# printf( )

1. It is used to print message or result on the output screen. It is define in **stdio.h** header file.
2. Returns the number of characters it outputs on the screen.

Example:

```
printf( “Enter the number “);
```

```
printf( “ Balance in your account is %d”, bal);
```

```
printf(“Balance =%d,Total tax  is %f ” ,bal, tax);
```

# scanf( )

- **scanf( )** : It is used to take input from the user such as numerical values, characters or strings. It is defined in `stdio.h`
- The function returns the number of data items that have been entered successfully.

Example:

```
int num1,num2,num3;  
char var;  
printf("enter the numbers ");  
scanf("%d%d%d", &num1,&num2,&num3);  
printf("enter a character");  
scanf("%c", &var);
```

# Char Data type

- A variable of type **char** can store a single character.
- All character have a **numeric code associated** with them, so in reality while storing a character variable its **numeric value gets stored**.
- The set of numeric code is called as “ASCII”, American Standard Code for Information Interchange.

# ASCII codes

0	<b>nul</b>
1	<b>soh</b>
2	<b>stx</b>
3	<b>etx</b>
4	<b>eot</b>
5	<b>enq</b>
6	<b>ack</b>
7	<b>bel</b>
8	<b>bs</b>
9	<b>ht</b>
10	<b>nl</b>
11	<b>vt</b>
12	<b>mp</b>
13	<b>cr</b>
14	<b>so</b>
15	<b>si</b>
16	<b>dle</b>
17	<b>dc1</b>
18	<b>dc2</b>
19	<b>dc3</b>

20	<b>dc4</b>
21	<b>nak</b>
22	<b>syn</b>
23	<b>etb</b>
24	<b>can</b>
25	<b>em</b>
26	<b>sub</b>
27	<b>esc</b>
28	<b>fs</b>
29	<b>gs</b>
30	<b>rs</b>
31	<b>us</b>
32	<b>sp</b>
33	<b>!</b>
34	<b>"</b>
35	<b>#</b>
36	<b>\$</b>
37	<b>%</b>
38	<b>&amp;</b>
39	<b>'</b>

40	<b>(</b>
41	<b>)</b>
42	<b>*</b>
43	<b>+</b>
44	<b>,</b>
45	<b>-</b>
46	<b>.</b>
47	<b>/</b>
48	<b>0</b>
49	<b>1</b>
50	<b>2</b>
51	<b>3</b>
52	<b>4</b>
53	<b>5</b>
54	<b>6</b>
55	<b>7</b>
56	<b>8</b>
57	<b>9</b>
58	<b>:</b>
59	<b>;</b>

60	<b>&lt;</b>
61	<b>=</b>
62	<b>&gt;</b>
63	<b>?</b>
64	<b>@</b>
65	<b>A</b>
66	<b>B</b>
67	<b>C</b>
68	<b>D</b>
69	<b>E</b>
70	<b>F</b>
71	<b>G</b>
72	<b>H</b>
73	<b>I</b>
74	<b>J</b>
75	<b>K</b>
76	<b>L</b>
77	<b>M</b>
78	<b>N</b>
79	<b>O</b>

80	<b>P</b>
81	<b>Q</b>
82	<b>R</b>
83	<b>S</b>
84	<b>T</b>
85	<b>U</b>
86	<b>V</b>
87	<b>W</b>
88	<b>X</b>
89	<b>Y</b>
90	<b>Z</b>
91	<b>[</b>
92	<b>\</b>
93	<b>]</b>
94	<b>^</b>
95	<b>_</b>
96	<b>`</b>
97	<b>a</b>
98	<b>b</b>
99	<b>c</b>

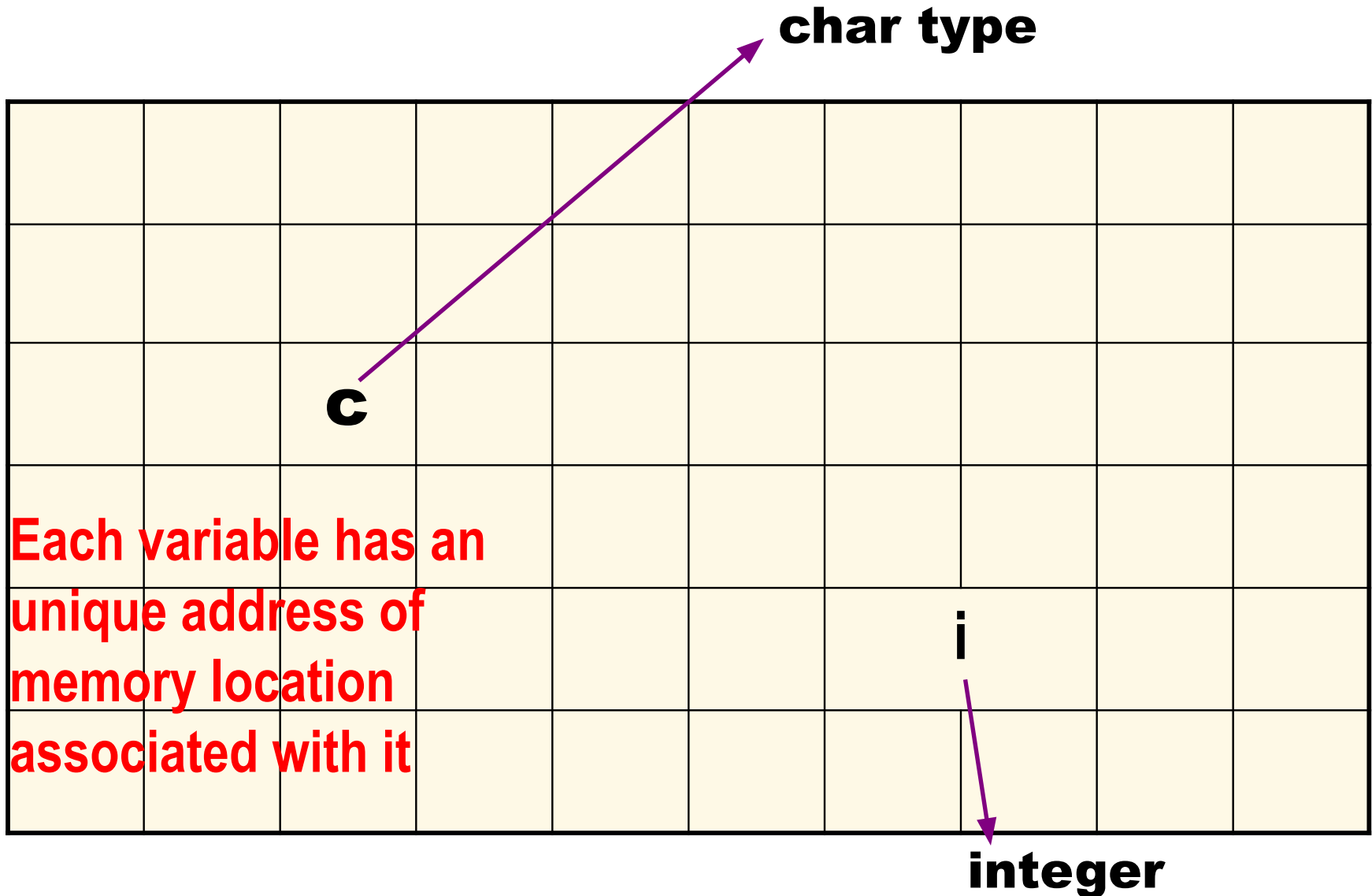
100	<b>d</b>
101	<b>e</b>
102	<b>f</b>
103	<b>g</b>
104	<b>h</b>
105	<b>i</b>
106	<b>j</b>
107	<b>k</b>
108	<b>l</b>
109	<b>m</b>
110	<b>n</b>
111	<b>o</b>
112	<b>p</b>
113	<b>q</b>
114	<b>r</b>
115	<b>s</b>
116	<b>t</b>
117	<b>u</b>
118	<b>v</b>
119	<b>w</b>

120	<b>x</b>
121	<b>y</b>
122	<b>z</b>
123	<b>{</b>
124	<b> </b>
125	<b>}</b>
126	<b>~</b>
127	<b>del</b>

# Declaration of Variable

- To declare a variable means to **reserve memory** space for it.
- In the declaration variable is provided a name which is used through out the program.
- Example:  
**char c;**  
OR  
**char c, k, l;**
- Declaring the character **does not initialize** the variable with the desired value.

# Memory view



# Data types and Ranges

Data Type	Range	Bytes	Format
signed char	-128 to +127	1	%c
unsigned char	0 to 255	1	%c
short signed int	-32768 to +32767	2	%d
short unsigned int	0 to 65535	2	%u
signed int	-32768 to +32767	2	%d
unsigned int	0 to 65535	2	%u
long signed int	-2147483648 to +2147483647	4	%ld
long unsigned int	0 to 4294967295	4	%lu
float	-3.4e38 to +3.4e38	4	%f
double	-1.7e308 to +1.7e308	8	%lf
long double	-1.7e4932 to +1.7e4932	10	%Lf

# Important about Variable

- Always remember in C , a variable **must always** be declared before it used.
- Declaration **must specify the data type** of the value of the variable.
- Name of the variable should match with its functionality /purpose.

Example    **int count ;**    for counting the iterations.

**float per\_centage ;**    for percentage of student



# Char data type

- Characters (**char**) – To store a character into a char variable, you must enclose it with **SINGLE QUOTE** marks i.e. by **' '**.
- The double quotes are reserved for string.
- The character that you assign are called as character constants.
- You can assign a char variable an integer.i.e. its integer value.

**'a', 'z', 'A', 'Z', '?', '@', '0', '9'**

# Initialization

- Initializing a variable involves assigning(putting in) a value for the first time. This is done by using the **ASSIGNMENT OPERATOR**, denoted by the equals sign, **=**.
- **Equality operator is denoted by (==)**
- Declaration and initializing can be done on separate lines, or on one line.
- **char c='x'; or char c;  
c='x'**

# Printing value of char variable

```
printf( “%c”, a);
```

This causes the “ %c” to be replaced by value of character variable a.

```
printf(“\n %c”, a);
```

“\n” is a new line character  
which will print the value of variable on newline.

# Expressions

- Expressions consist of a mixture of constants, variables and operators .They return values.
- Examples are:
  - 17
  - $-X*B/C+A$
  - $X+17$

# Program

```
#include<stdio.h>
```

```
main()
```

```
{   int a,b,c,d,e;
```

```
    a=10;
```

```
    b=4.3;
```

```
    c=4.8;
```

```
    d='A';
```

```
    e= 4.3 +4.8;
```

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

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

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

```
    printf("\n d=%d",d);
```

```
    printf("\n e=%d",e);
```

```
    return 0;
```

```
}
```

- a=10
- b=4 // b is integer
- c=4 // c is integer
- d=65 //ASCII value of A is 65
- e=9

# ASCII codes

0	<b>nul</b>
1	<b>soh</b>
2	<b>stx</b>
3	<b>etx</b>
4	<b>eot</b>
5	<b>enq</b>
6	<b>ack</b>
7	<b>bel</b>
8	<b>bs</b>
9	<b>ht</b>
10	<b>nl</b>
11	<b>vt</b>
12	<b>mp</b>
13	<b>cr</b>
14	<b>so</b>
15	<b>si</b>
16	<b>dle</b>
17	<b>dc1</b>
18	<b>dc2</b>
19	<b>dc3</b>

20	<b>dc4</b>
21	<b>nak</b>
22	<b>syn</b>
23	<b>etb</b>
24	<b>can</b>
25	<b>em</b>
26	<b>sub</b>
27	<b>esc</b>
28	<b>fs</b>
29	<b>gs</b>
30	<b>rs</b>
31	<b>us</b>
32	<b>sp</b>
33	<b>!</b>
34	<b>"</b>
35	<b>#</b>
36	<b>\$</b>
37	<b>%</b>
38	<b>&amp;</b>
39	<b>'</b>

40	<b>(</b>
41	<b>)</b>
42	<b>*</b>
43	<b>+</b>
44	<b>,</b>
45	<b>-</b>
46	<b>.</b>
47	<b>/</b>
48	<b>0</b>
49	<b>1</b>
50	<b>2</b>
51	<b>3</b>
52	<b>4</b>
53	<b>5</b>
54	<b>6</b>
55	<b>7</b>
56	<b>8</b>
57	<b>9</b>
58	<b>:</b>
59	<b>;</b>

60	<b>&lt;</b>
61	<b>=</b>
62	<b>&gt;</b>
63	<b>?</b>
64	<b>@</b>
65	<b>A</b>
66	<b>B</b>
67	<b>C</b>
68	<b>D</b>
69	<b>E</b>
70	<b>F</b>
71	<b>G</b>
72	<b>H</b>
73	<b>I</b>
74	<b>J</b>
75	<b>K</b>
76	<b>L</b>
77	<b>M</b>
78	<b>N</b>
79	<b>O</b>

80	<b>P</b>
81	<b>Q</b>
82	<b>R</b>
83	<b>S</b>
84	<b>T</b>
85	<b>U</b>
86	<b>V</b>
87	<b>W</b>
88	<b>X</b>
89	<b>Y</b>
90	<b>Z</b>
91	<b>[</b>
92	<b>\</b>
93	<b>]</b>
94	<b>^</b>
95	<b>_</b>
96	<b>`</b>
97	<b>a</b>
98	<b>b</b>
99	<b>c</b>

100	<b>d</b>
101	<b>e</b>
102	<b>f</b>
103	<b>g</b>
104	<b>h</b>
105	<b>i</b>
106	<b>j</b>
107	<b>k</b>
108	<b>l</b>
109	<b>m</b>
110	<b>n</b>
111	<b>o</b>
112	<b>p</b>
113	<b>q</b>
114	<b>r</b>
115	<b>s</b>
116	<b>t</b>
117	<b>u</b>
118	<b>v</b>
119	<b>w</b>

120	<b>x</b>
121	<b>y</b>
122	<b>z</b>
123	<b>{</b>
124	<b> </b>
125	<b>}</b>
126	<b>~</b>
127	<b>del</b>

# Program using character constants

```
#include<stdio.h>
```

```
int main()
```

```
{
```

```
    char a,b,c,d;
```

```
    char t='o';
```

```
    a='H';
```

```
    b=e;
```

```
    c="l";
```

```
    d=108;
```

```
    printf("%c%c%c%c%c",a,b,c,d,t);
```

```
    return 0;
```

```
}
```

# Program using character constants

```
#include<stdio.h>
```

```
int main()
```

```
{
```

```
    char a,b,c,d;        /* declare char variables*/
```

```
    char e='o';          /* declare char variable*/
```

```
    a='H';                /* initialize the rest */
```

```
    b='e';                /* b=e is incorrect */
```

```
    c='l';                /* c="l" is incorrect*/
```

```
    d=108;                /* the ASCII value of l is 108 */
```

```
    printf("%c%c%c%c%c",a,b,c,d,e);
```

```
    return 0;
```

```
}
```

Output : Hello would be printed



# Integer Data Type

- Variables of the **int data type** represent whole numbers.
- If you try to **assign a fraction to an int variable, the decimal part is ignored** and the value assigned is rounded down from the actual value.
- Also assigning a character constant to an **int variable** assigns the ASCII value.

# Float data type

- For **storing decimal** numbers **float data type** is used.
- Floats are relatively easy to use but problems tend to occur when performing division

In general:

An **int** divided by an **int** returns an **int**.

An **int** divided by a **float** returns a **float**.

A **float** divided by an **int** returns a **float**.

A **float** divided by a **float** returns a **float**.

# Program

```
#include<stdio.h>
```

```
main( )
```

```
{
```

```
    float a=3.0;
```

```
    float b= 4.00 + 7.00;
```

```
    printf("a=%f" ,a);
```

```
    printf("b=%f" ,b);
```

```
    return 0;
```

```
}
```

**a=3.000000    b=11.000000**

# int and float

- Float is “communicable type”
  - Example:

$$4.0/5 + 2 - 1$$

$$= 0.8 + 2 - 1$$

$$= 1.8$$

- Integer division in C rounds down:

$$4/5 = 0$$

# Example

$$1 + 2 * 3 - 4.0 / 5$$

$$= 1 + (2 * 3) - (4.0 / 5)$$

$$= 1 + 6 - 0.8$$

$$= 6.2$$

# Multiple Format specifiers

- You could use as many format specifiers as you want with **printf**-just as long as you pass the correct number of arguments.
- The ordering of the arguments matters. The first one should correspond to the first format specifier in the string and so on. Take this example:

```
printf( “a=%d,b=%d, c=%d\n”, a , b, c);
```

If a,b and c were integers,this statement will print the values in the correct order.

- Rewriting the statement as

```
printf( “a=%d,b=%d, c=%d\n”, c,a,b);
```

Would still cause the program to compile OK, but the values of a,b and c would be displayed in the wrong order.

# Statements

**STATEMENTS** are instructions and are terminated with a semicolon, `;`. Statements consist of a mixture of expressions, operators, function calls and various keywords. Here are some examples of statements:

```
x = 1 + 8;
```

```
printf("We will learn printf soon!\n");
```

```
int x, y, z; /* more on "int" later */
```

# Statements

Which of these are valid:

`int = 314.562 * 50;`

Not valid

`3.14 * r * r = area;`

Not valid

`k = a * b + c(2.5a + b);`

Not valid

`m_inst = rate of int * amt in rs;`

Not valid

`km = 4;`

valid

`area = 3.14 * r * r;`

valid

`S.I. = 400;`

Not valid



# **OVERVIEW OF**

## **C**

# The C language

- General purpose and Structure programming language.
- Rich in **library functions** and allow user to create additional library functions which can be added to existing ones.
- Highly portable: Most C programs can run on **many different machines** with almost no alterations.
- It gives user the flexibility to create the functions, which give C immense power and scope.

# Size of data types

- Size of data types is **compiler** and **processor types** dependent.
- The size **of data type** can be determined by using **sizeof** keyword **specified in between parenthesis**
- For Turbo 3.0 compiler, usually in bytes
  - char -> 1 bytes
  - int -> 2 bytes
  - float -> 4 bytes
  - double -> 8 bytes

# Example

```
1 #include <stdio.h>
2 int main()
3 {
4     printf("sizeof(char) == %d\n", sizeof(char));
5     printf("sizeof(short) == %d\n", sizeof(short));
6     printf("sizeof(int) == %d\n", sizeof(int));
7     printf("sizeof(long) == %d\n", sizeof(long));
8     printf("sizeof(long long) == %d\n", sizeof(long long));
9
10    return 0;
11 }
```

# Output

```
sizeof(char) == 1
sizeof(short) == 2
sizeof(int) == 4
sizeof(long) == 8
sizeof(long long) == 8
```

```
int main()
{
    int a, b, c;
    printf("Enter the first value:");
    scanf("%d", &a);
    printf("Enter the second value:");
    scanf("%d", &b);
    c = a + b;
    printf("%d + %d = %d\n", a, b, c);
    return 0;
}
```

>

**This animation shows the execution of a simple C program.**

```
int main()
```

```
{
```

```
    int a, b, c;
```

```
    printf("Enter the first value:");
```

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

```
    printf("Enter the second value:");
```

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

```
    c = a + b;
```

```
    printf("%d + %d = %d\n", a, b, c);
```

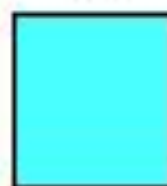
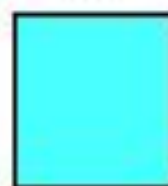
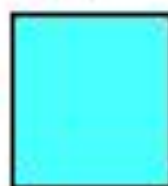
```
    return 0;
```

```
}
```

**a**

**b**

**c**

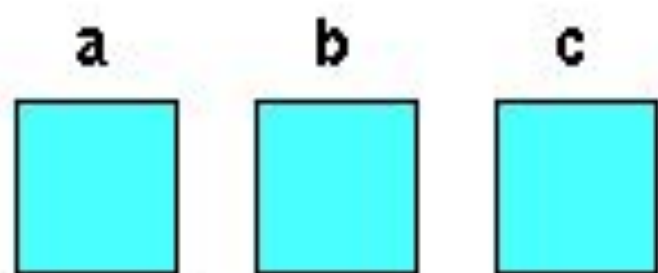


>add

Execution begins. The 3 variables are created.

**This animation shows the execution of a simple C program.**

```
int main()
{
    int a, b, c;
    ▶ printf("Enter the first value");
    scanf("%d", &a);
    printf("Enter the second value:");
    scanf("%d", &b);
    c = a + b;
    printf("%d + %d = %d\n", a, b, c);
    return 0;
}
```



>add  
Enter the first value:

**This animation shows the execution of a simple C program.**

```
int main()
```

```
{
```

```
    int a, b, c;
```

```
    printf("Enter the first value");
```

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

```
    printf("Enter the second value.");
```

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

```
    c = a + b;
```

```
    printf("%d + %d = %d\n", a, b, c);
```

```
    return 0;
```

```
}
```

**a**

**5**

**b**

**c**

>add

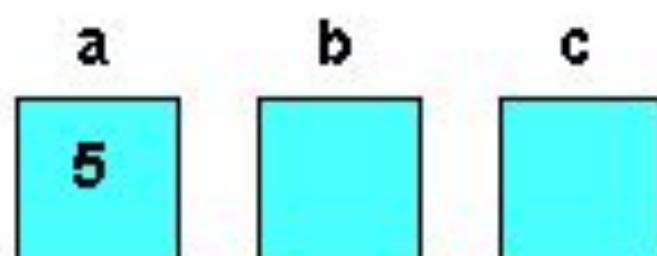
Enter the first value. 5

The user enters a value. It is stored in "a".

**This animation shows the execution of a simple C program.**



```
int main()
{
    int a, b, c;
    printf("Enter the first value:");
    scanf("%d", &a);
    ▶ printf("Enter the second value:");
    scanf("%d", &b);
    c = a + b;
    printf("%d + %d = %d\n", a, b, c);
    return 0;
}
```

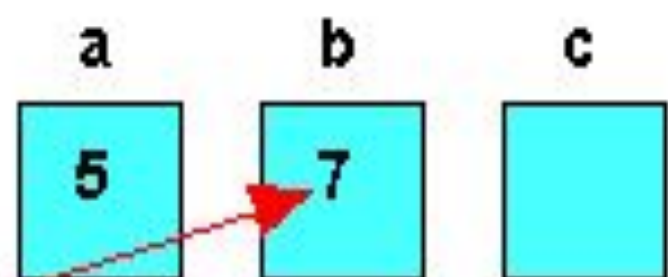


>add  
Enter the first value: 5  
Enter the second value:

The second prompt is displayed to the user.

**This animation shows the execution of a simple C program.**

```
int main()
{
    int a, b, c;
    printf("Enter the first value:");
    scanf("%d", &a);
    printf("Enter the second value:");
    scanf("%d", &b);
    c = a + b;
    printf("%d + %d = %d\n", a, b, c);
    return 0;
}
```

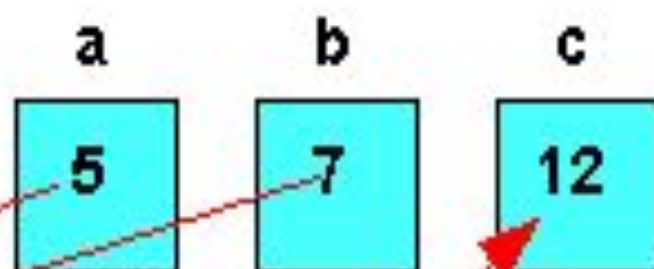


>add  
Enter the first value: 5  
Enter the second value: 7

The user enters a 7. It is stored in "b".

**This animation shows the execution of a simple C program.**

```
int main()
{
    int a, b, c;
    printf("Enter the first value:");
    scanf("%d", &a);
    printf("Enter the second value:");
    scanf("%d", &b);
    ▶ c = a + b;
    printf("%d + %d = %d\n", a, b, c);
    return 0;
}
```



>add  
Enter the first value: 5  
Enter the second value: 7

"a" is added to "b" and the result is stored in "c".

**This animation shows the execution of a simple C program.**

```
int main()
{
    int a, b, c;
    printf("Enter the first value:");
    scanf("%d", &a);
    printf("Enter the second value:");
    scanf("%d", &b);
    c = a + b;
    ▶ printf("%d + %d = %d\n", a, b, c);
    return 0;
}
```

a	b	c
5	7	12

>add  
Enter the first value: 5  
Enter the second value: 7  
5 + 7 = 12

The line "5+7=12" is formed and displayed to the user.

**This animation shows the execution of a simple C program.**



```
int main()
{
    int a, b, c;
    printf("Enter the first value:");
    scanf("%d", &a);
    printf("Enter the second value:");
    scanf("%d", &b);
    c = a + b;
    printf("%d + %d = %d\n", a, b, c);
    return 0;
}
```

**a**

**5**

**b**

**7**

**c**

**12**

>add

Enter the first value: 5

Enter the second value: 7

5 + 7 = 12

>

The program completes.

**This animation shows the execution of a simple C program.**

# Output??

```
#include <stdio.h>
```

```
int main() {
```

```
    int a = 72;
```

```
    char b = 'A';
```

```
    printf("a equals %d \n", a);
```

```
    printf("a equals %c \n", a);
```

```
    printf("b equals %d \n", b);
```

```
    printf("b equals %c \n", b);
```

```
}
```

- a equals 72
- a equals H
- b equals 65
- b equals A

# Example Formatting : Printf statement

```
#include<stdio.h>

main()
{
    int a,b;
    float c,d;

    a = 15;
    b = a / 2;
    printf("%d\n",b);
    printf("%3d\n",b);
    printf("%03d\n",b);

    c = 15.3;
    d = c / 3;
    printf("%3.2f\n",d);
}
```

Output of the source above:

```
7
 7
007
5.10
```

- The first printf statement we print a decimal.
- In the second printf statement we print the same decimal, **but we use a width (%3d)** to say that we want three digits (positions) reserved for the output.
- The result **is that two “space characters”** are placed before printing the character.
- In the third printf statement we say almost the same as the previous one.
- Print the output with a width of **three digits, but fill the space with 0.**
- In the fourth printf statement we want to print a float.
- In this printf statement we want to print **three position before the decimal point (called width) and two positions behind the decimal point (called precision).**

# Formatting Floating Points

- `%d` (print as a decimal integer)
- `%6d` (print as a decimal integer with a width of at least 6 wide)
- `%f` (print as a floating point)
- `%4f` (print as a floating point with a width of at least 4 wide)
- `%.4f` (print as a floating point with a precision of four characters after the decimal point)
- `%3.2f` (print as a floating point at least 3 wide and a precision of 2)



# Assignment: Practice on your machine

- `printf(“%d”,9876)`

9	8	7	6
---	---	---	---

- `printf(“%6d”,9876)`

		9	8	7	6
--	--	---	---	---	---

- `printf(“%2d”,9876)`

9	8	7	6
---	---	---	---

- `printf(“%-6d”,9876)`

9	8	7	6		
---	---	---	---	--	--

- `printf(“%06d”,9876)`

0	0	9	8	7	6
---	---	---	---	---	---

If the number is greater than the specified field width, it will be printed in full overriding the minimum specification

# Example

`printf("%7.4f",98.7654)`

9	8	.	7	6	5	4
---	---	---	---	---	---	---

`printf("%7.2f",98.7654)`

		9	8	.	7	7
--	--	---	---	---	---	---

`printf("%-7.2f",98.7654)`

9	8	.	7	7		
---	---	---	---	---	--	--

`printf("%f",98.7654)`

9	8	.	7	6	5	4	0	0
---	---	---	---	---	---	---	---	---

**Note:-Using precision in a conversion specification in the format control string of a scanf statement is wrong.**

```
#include<stdio.h>

main()
{
    printf(":%s:\n", "Hello, world!");
    printf(":%15s:\n", "Hello, world!");
    printf(":%.10s:\n", "Hello, world!");
    printf(":%-10s:\n", "Hello, world!");
    printf(":%-15s:\n", "Hello, world!");
    printf(":%.15s:\n", "Hello, world!");
    printf(":%15.10s:\n", "Hello, world!");
    printf(":%-15.10s:\n", "Hello, world!");
}
```

```

{
    printf(":%s:\n", "Hello, world!");
    printf(":%15s:\n", "Hello, world!");
    printf(":%.10s:\n", "Hello, world!");
    printf(":%-10s:\n", "Hello, world!");
    printf(":%-15s:\n", "Hello, world!");
    printf(":%.15s:\n", "Hello, world!");
    printf(":%15.10s:\n", "Hello, world!");
    printf(":%-15.10s:\n", "Hello, world!");
}

```

The output of the example above:

```

:Hello, world!:
:  Hello, world!:
:Hello, wor:
:Hello, world!:
:Hello, world! :
:Hello, world!:
:      Hello, wor:
:Hello, wor      :

```

# Formatting Strings

```
#include<stdio.h>

main()
{
    printf(":%s:\n", "Hello, world!");
    printf(":%15s:\n", "Hello, world!");
    printf(":%.10s:\n", "Hello, world!");
    printf(":%-10s:\n", "Hello, world!");
    printf(":%-15s:\n", "Hello, world!");
    printf(":%.15s:\n", "Hello, world!");
    printf(":%15.10s:\n", "Hello, world!");
    printf(":%-15.10s:\n", "Hello, world!");
}
```

The output of the example above:

```
:Hello, world!:
:  Hello, world!:
:Hello, wor:
:Hello, world!:
:Hello, world! :
:Hello, world!:
:   Hello, wor:
:Hello, wor   :
```

- The printf(":%s:\n", "Hello, world!"); statement prints the string (nothing special happens.)
  - The printf(":%15s:\n", "Hello, world!"); statement prints the string, but print 15 characters. If the string is smaller the “empty” positions will be filled with “whitespace.”
  - The printf(":%.10s:\n", "Hello, world!"); statement prints the string, but print only 10 characters of the string.
  - The printf(":%-10s:\n", "Hello, world!"); statement prints the string, but prints at least 10 characters. If the string is smaller “whitespace” is added at the end.
  - The printf(":%-15s:\n", "Hello, world!"); statement prints the string, but prints at least 15 characters. The string in this case is shorter than the defined 15 character, thus “whitespace” is added at the end (defined by the minus sign.)
  - The printf(":%.15s:\n", "Hello, world!"); statement prints the string, but print only 15 characters of the string. In this case the string is shorter than 15, thus the whole string is printed.
  - The printf(":%15.10s:\n", "Hello, world!"); statement prints the string, but print 15 characters.
- If the string is smaller the “empty” positions will be filled with “whitespace.” But it will only print a maximum of 10 characters, thus only part of new string (old string plus the whitespace positions) is printed.
- The printf(":%-15.10s:\n", "Hello, world!"); statement prints the string, but it does the exact same thing as the previous statement, accept the “whitespace” is added at the end.

# Example

```
#include<stdio.h>
```

```
main()
```

```
{
```

```
    printf(":%s:\n", "Hello, world!");
```

```
    printf(":%15s:\n", "Hello, world!");
```

```
    printf(":%.10s:\n", "Computer");
```

```
    printf(":%-10s:\n", "Computer");
```

```
    printf(":%-15s:\n", "Computer");
```

```
    printf(":%.15s:\n", "Computer");
```

```
    printf(":%15.10s:\n", "Computer");
```

```
    printf(":%15.10s:\n", "Computer Science");
```

```
    printf(":%-15.10s:\n", "Computer");
```

```
    printf(":%-15.10s:\n", "Computer Science");
```

```
}
```

# Exercise

- **Write a C program to compute the percentage of a student in five subjects.**

**Input the marks in five subjects from the user.**

**Maximum marks in all the subjects is 100.**

```
#include<stdio.h>
#include<conio.h>
void main()
{ int m1,m2,m3,m4,m5,total;
float average, percentage;
printf("Enter marks for subject one - ");
scanf("%d",&m1);
printf("Enter marks for subject two - ");
scanf("%d",&m2);
printf("Enter marks for subject three - ");
scanf("%d",&m3);
printf("Enter marks for subject four - ");
scanf("%d",&m4);
printf("Enter marks for subject five - ");
scanf("%d",&m5);
total=m1+m2+m3+m4+m5;
average=total/5;
percentage=(total*100)/500;
printf("\nThe average of five subjects is %f",average);
printf("\nPercentage=%f%%",percentage);
getch(); }
```



# Operators & Expressions

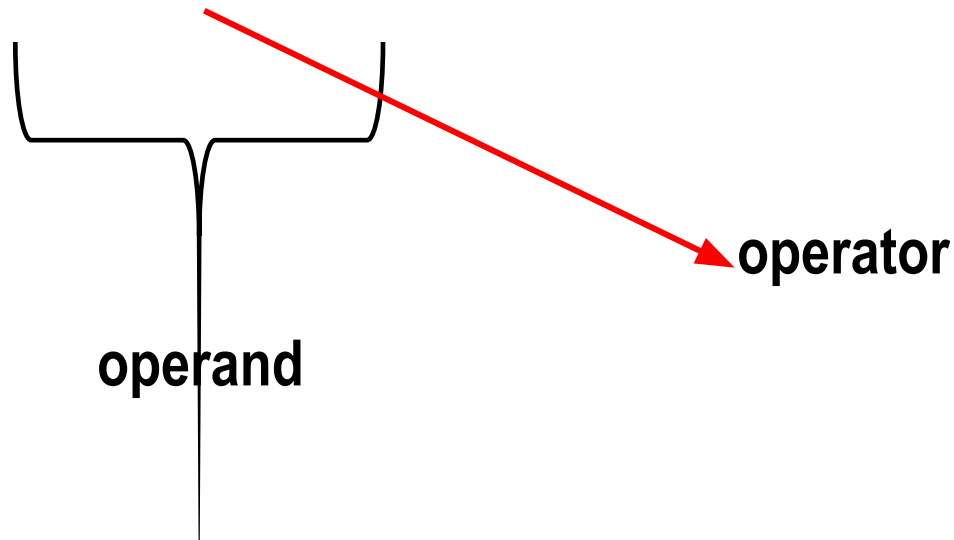
# Operator

- An operator is a symbol used to indicate a specific operation on variables in a program.
- Example : symbol “ + ” is an **add operator** that adds two data items called operands.

# Expression

- An expression is a combination of operands ( constants, variables, numbers) connected by operators and parenthesis.
- Example :

**A    +    B**



# Types of expression

- **Arithmetic expression** : An expression that involves arithmetic operators. The computed result of this expression is a numeric value.
- **Logical or Boolean expression** : An expression that involves relational and/ or logical operators. The result of this expression is a logical value i.e either **1** (TRUE) or **0** (FALSE)

# Operators

- C language is very rich in operators.
- Four main classes of operators :
  - » Arithmetic
  - » Relational
  - » Logical
  - » Bit wise

# Assignment operator

- It is used to **assign** variable a value:  
variable\_name = expression;
- **lvalue** : In compiler lvalue error messages means that an object on left hand side of assignment operator is missing.
- **rvalue** : In compiler rvalue error messages means that expression on right hand side of assignment operator is erroneous.

# Two cases of assignment

- **Multiple assignment:**

`int j=k=m=0;`

- **Compound assignment:**

`j= j+10;` this expression can be written as

**`j += 10;`**

similarly

`m= m-100;` is equivalent to **`m -= 100;`**

# Arithmetic operators

- Following operators are used for arithmetic operations on all built in data types :
  - + (addition)
  - (subtraction)
  - \* (multiplication)
  - / (division or quotient)
  - % (modulus or remainder)
  - (decrement)
  - ++ (increment)



# Precedence

Defines the order in which an expression is evaluated

<b>operators</b>	<b>precedence</b>
<b>* , / , %</b>	High and are on same level
<b>+ , -</b>	Lower and are on same level

# Precedence in Expressions -- Example

$$1 + 2 * 3 - 4 / 5 =$$

$$1 + (2 * 3) - (4 / 5)$$

B.O.D.M.A.S.

B stands for brackets,  
O for Order (exponents),  
D for division,  
M for multiplication,  
A for addition, and  
S for subtraction.



# Associativity

- Associativity is defined as the order in which consecutive operations within the same precedence group will be carried out.
- In the groups discussed, the associativity is left to right.

E.g. in the expression

$$= 2 + 3 - 4$$

$$= 5 - 4$$

$$= 1$$

# Example

```
int x;  
x= 7 + 3 * 5;
```

output:  
x = 22

```
int x;  
x= ( 7 + 3 ) * 5;
```

output:  
x = 50

```
int x;  
x= 7 / 3 * 5;
```

output:  
x = 10

# Modulus operator ( % )

- This operator has same priority as that of multiplication and division
- $a \% b$  = output remainder after performing  $a / b$
- Example:

$$7 \% 10 = 7$$

$$7 \% 1 = 0$$

$$7 \% 2 = 1$$

$$7 \% 7 = 0$$

# Exercise

```
int y;  
y = 10 % 4 * 3;
```

Output:  
6

```
int y;  
y = 3 * 10 % 4;
```

Output::  
2

# Important

- Modulus operator ( % ) : It produces remainder of an integer division. This operator **can not** be used with floating point numbers.

```
int main( ){  
    float  f1=3.2, f2=1.1, f3 ;  
    f3 = f1 % f2;  
    printf(“ %f”, f3);  
    return 0; }
```

Output: **error at line 3**  
illegal use of floating point

# Invalid arithmetic expressions

- $a * + b$  : Invalid as two operators can not be used in continuation.
- $a( b * c )$  : Invalid as there is no operator between a and b.



# Unary arithmetic operators

- A unary operator requires only one operand or data item.
- Unary arithmetic operators are:
  - » Unary minus ( - )
  - » Increment ( + + )
  - » decrement ( - - )
- Unary operators are **RIGHT** associative that is they are **evaluated from right to left** when operators of same precedence are encountered in an expression.

# Unary minus ( - )

- It is written before a numeric value, variable or expression
- Its effect is NEGATION of the operand to which it is applied.
- Example:

**- 57          - 2.933      -x**

**-( a \* b)          8 \* ( - ( a+b))**

# Increment operator ( ++ )

- The increment ( ++ ) operator adds **1** to its operand.

**n = n + 1 ;            =>    ++ n ;**

- Postfix Increment ( **n ++** ) : It increments the value of n after its value is used.
- Prefix Increment ( **++ n** ) : It increments the value of n before it is used.

# Example 1

1.  $x = n++$  ;

1.  $x = ++n$  ;

Where  $n = 5$ ;

- case 1: It sets the value of  $x$  to 5 and **then increments**  $n$  to 6.

$x = 5$  and  $n = 6$

- case 2: It **increments the** value of  $n$  and then sets the value of  $x$  to 6.

$x = 6$  and  $n = 6$

# Example

**sum=x++;**

**sum = ++x;**



**Sum = x;**

**x=x+1;**



**x=x+1;**

**Sum=x;**

# Decrement operator

- The decrement ( - - ) operator subtracts **1** from its operand.  
**j = j - 1 ;        =>        -- j;**
- Postfix decrement ( **y - -** ) : In this case value of operand is fetched before subtracting 1 from it.
- Prefix decrement ( **-- y** ) : In this case value of operand is fetched after subtracting 1 from it.

# Example

**sum=x--;**



**sum = --x;**



**sum = x;**  
**x=x-1;**

**x=x-1;**  
**sum=x;**

# Precedence of Arithmetic operators

Highest :    ++    --

             - (unary minus)

             \*       /       %

Lowest +    -

Operators on same level of precedence are evaluated by the compiler from left to right.



# Exercise

```
int x= 34.9;  
printf ( “\n\t ++x=%d and x++ = %d”,++ x, x++);
```

**++x = 36 and x++ = 34**

# Relational & Logical operators

- A relational operator is used to compare two values and the result of such operation is always **logical** either **TRUE ( 1 )** or **FALSE ( 0 )**.

- |   |    |                          |          |
|---|----|--------------------------|----------|
|   | <  | less than                | $x < y$  |
| ● | >  | greater than             | $x > y$  |
| ● | <= | less than or equal to    | $x <= y$ |
| ● | >= | greater than or equal to | $x >= y$ |
| ● | == | is equal to              | $x == y$ |
| ● | != | is not equal to          | $x != y$ |

# Exercise

- Suppose that  $i$ ,  $j$ , and  $k$  are integer variables whose values are 1, 2 and 3, respectively.

Expression	Value	Interpretation
$i < j$	1	true
$(i + j) \geq k$	1	true
$(j + k) > (i + 5)$	0	false
$k \neq 3$	0	false
$j == 2$	1	true

# Logical operator

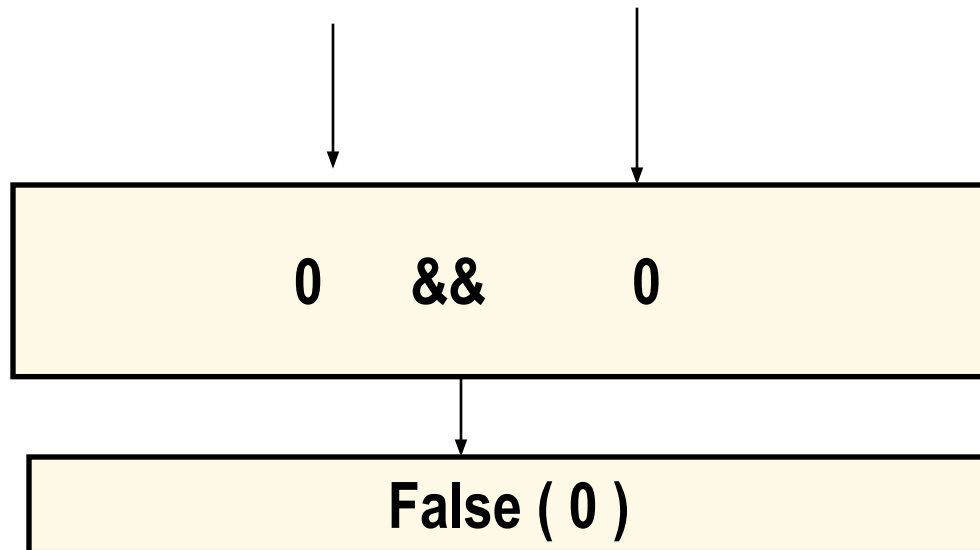
- A logical operator is used to connect two relational expressions or logical expressions.
- The result of logical expressions is always an integer value either TRUE ( 1 ) or FALSE( 0 ).

<b>&amp;&amp;</b>	<b>Logical AND</b>	<b>x &amp;&amp; y</b>
<b>  </b>	<b>Logical OR</b>	<b>x    y</b>
<b>!</b>	<b>Logical NOT</b>	<b>!x</b>

# Logic AND

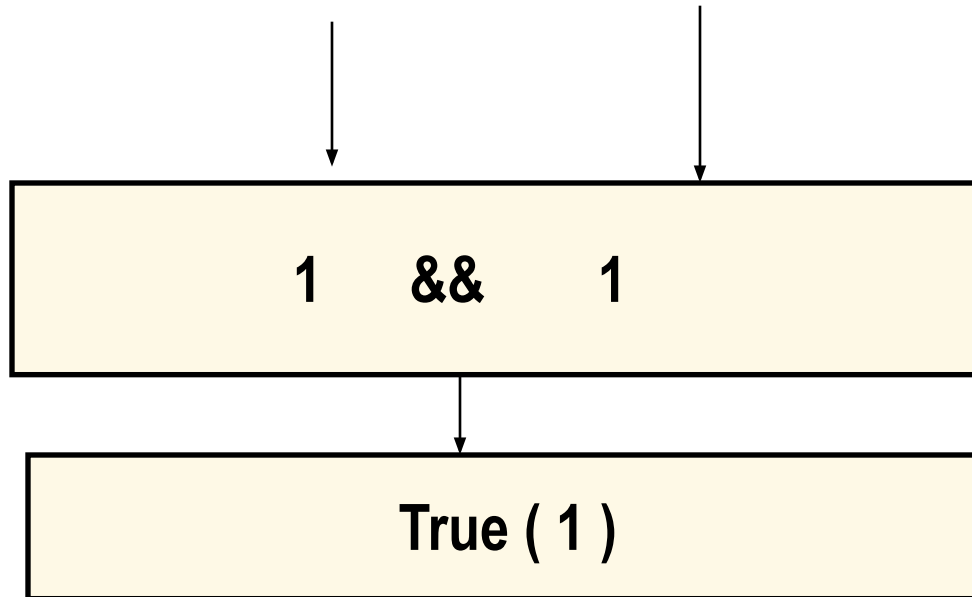
- The output of AND operation is **TRUE** if **BOTH** the operands are true.

Example:      **( 8 < 7 ) && ( 6 > 7 )**



# Example 2

**$(8 > 7) \ \&\& \ (6 < 7)$**



# Logical OR

- The result of a *logical or* operation will be true if either operand is true or if both operands are true.

**$(8 < 7) \parallel (6 > 7)$  is false**

**$(8 > 7) \parallel (6 > 7)$  is true**

**$(8 > 7) \parallel (6 < 7)$  is true**

# Logical NOT

- The Logical NOT ( ! ) is a unary operator. It negates the value of the logical expression or operand.
- If value of  $X = 0$        $! X = ?$   
     $! X = 1$
- $! ( 5 < 6 ) \parallel ( 7 > 7 ) = ???$   
     $! ( 1 ) \parallel ( 0 ) = ! 1 = 0 \rightarrow \text{false}$
- $! ( 5 > 3 ) = ??$   
     $\rightarrow 0 \rightarrow \text{false}$
- $!(34 \geq 765) = ??$   
     $\rightarrow 1 \rightarrow \text{True}.$



# Exercise

**x = 10 and y = 25**

**( x >= 10 ) && ( x < 20 )**

**( x >= 10) && ( y <= 15)**

**( x == 10 ) && ( y > 20 )**

**( x==10) || ( y < 20)**

**True**

**False**

**True**

**True**

# Precedence & Associativity

!(logical NOT)   ++   --     sizeof( )	Right to left	<div>highest</div> <div>P R E C E D E N C E</div> <div>lowest</div>
* (multiplication) / ( division) %( modulus)	Left to right	
+   - (binary )	Left to right	
<     <=     >     >=	Left to right	
==(equal to)   !=( Not equal to )	Left to right	
&&(AND)	Left to right	
(OR)	Left to right	
? : (conditional )	Right to left	
=   +=   -=   *=   /=   %=	Right to left	
,	Left to right	

# Exercise

- Suppose that

**j = 7**, an integer variable

**f = 5.5**, a float variable

**c = 'w'**

**Interpret the value of the following expressions:**

<b>( j &gt;= 6) &amp;&amp; (c == 'w')</b>	<b>1</b>
<b>( j &gt;= 6)    ( c == 'w')</b>	<b>1</b>
<b>(f &lt; 11) &amp;&amp; ( j &gt; 100)</b>	<b>0</b>
<b>(c != 'p')    ((j + f) &lt;= 10)</b>	<b>1</b>
<b>f &gt; 5</b>	<b>1</b>
<b>!(f &gt; 5)</b>	<b>0</b>
<b>j &lt;= 3</b>	<b>0</b>
<b>!( j &lt;= 3)</b>	<b>1</b>
<b>j &gt; (f +1)</b>	<b>1</b>
<b>!( j &gt; (f +1))</b>	<b>0</b>

- Suppose that

$j = 7$ , an integer variable

$f = 5.5$ , a float variable

$c = 'w'$

Interpret the value of the following expressions:

$j + f \leq 10$

0

$j \geq 6 \ \&\& \ c == 'w'$

1

$f < 11 \ \&\& \ j > 100$

0

$!0 \ \&\& \ 0 \ || \ 0$

0

$!(0 \ \&\& \ 0) \ || \ 0$

1