

Introduction to C

Abu Sayeed
Lecturer
Department of Computer Science

C – Language History



- The C language is a structure oriented programming language, developed at Bell Lab (AT&T) in 1972 by Dennis Ritchie
- C language features were derived from an earlier language called “B” (Basic Combined Programming Language – BCPL)
- C language was invented for implementing UNIX operating system
- In 1978, Dennis Ritchie and Brian Kernighan published the first edition “The C Programming Language” and commonly known as K&R C
- In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or “ANSI C”, was completed late 1988.

C language standards

- C89/C90 standard – First standardized specification for C language was developed by the American National Standards Institute in 1989. C89 and C90 standards refer to the same programming language.
- C99 standard – Next revision was published in 1999 that introduced new features like advanced data types and other changes.

Features of C language

- Reliability
- Portability
- Flexibility
- Modularity
- Efficiency and Effectiveness

Uses of C language

- The C language is used for developing system applications that forms a major portion of operating systems such as Windows, UNIX and Linux. Below are some examples of C being used.
 - Database systems
 - Graphics packages
 - Word processors
 - Spreadsheets
 - Operating system development
 - Compilers and Assemblers
 - Network drivers
 - Interpreters

C is Middle Level Language

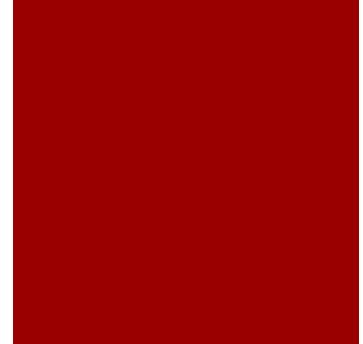
- There are following reason that C is called Middle Level Language as:
 - C programming language behaves as high level language through function, it gives a modular programming and breakup, increased the efficiency for resolvability.
 - C programming language support the low level language i.e. Assembly Language.
 - C language also gives the facility to access memory through pointer.
 - Its combines the elements of high-level languages with the functionalism of assembly language.
- So, C language neither a High Level nor a Low level language but a **Middle Level Language**.

The C language is a structured language



S.no	Structure oriented	Object oriented	Non structure
1	In this type of language, large programs are divided into small programs called functions	In this type of language, programs are divided into objects	There is no specific structure for programming this language
2	Prime focus is on functions and procedures that operate on the data	Prime focus is in the data that is being operated and not on the functions or procedures	N/A
3	Data moves freely around the systems from one function to another	Data is hidden and cannot be accessed by external functions	N/A
4	Program structure follows "Top Down Approach"	Program structure follows "Bottom UP Approach"	N/A
5	Examples: C, Pascal, ALGOL and Modula-2	C++, JAVA and C# (C sharp)	BASIC, COBOL, FORTRAN

Key points to remember in C language



- The C language is structured, middle level programming language developed by Dennis Ritchie
- Operating system programs such as Windows, Unix, Linux are written in C language
- C89/C90 and C99 are two standardized editions of C language
- C has been written in assembly language

C language tutorial reference E-books & research papers

- [ANSI 89] American National Standards Institute, American National Standard for Information Systems Programming Language C, X3.159-1989.
- [Kernighan 78] B. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice-Hall: Englewood Cliffs, NJ, 1978. Second edition, 1988.
- [Thinking 90] C* Programming Guide, Thinking Machines Corp.: Cambridge Mass., 1990.

Variables in C

Topics

- What is Variable
- Naming Variables
- Declaring Variables
- Using Variables
- The Assignment Statement

What Are Variables in C?

- Variables are the names that refer to sections of memory into which **data can be stored**.
- **Variables** in C have the same meaning as variables in algebra. That is, they represent some unknown, or variable, value.

$$x = a + b$$

$$z + 2 = 3(y - 5)$$

- Remember that variables in algebra are represented by a single alphabetic character.

Naming Variables

- Rules for variable naming:
 - Can be composed of letters (both uppercase and lowercase letters), digits and underscore only.
 - The first character should be either a letter or an underscore(not any digit).
 - Punctuation and special characters are not allowed except underscore.
 - Variable name should not be keywords.
 - names are case sensitive.
 - There is no rule for the length of a variable name. However, the first 31 characters are discriminated by the compiler. So, the first 31 letters of two name in a program should be different.

Reserved Words (Keywords) in C

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

Naming Conventions

- C programmers generally agree on the following **conventions** for naming variables.
 - Begin variable names with lowercase letters
 - Use meaningful identifiers
 - Separate “words” within identifiers with underscores or mixed upper and lower case.
 - Examples: surfaceArea surface_Area
 surface_area
 - Be consistent!

Naming Conventions (con't)

- Use all uppercase for **symbolic constants** (used in **#define** preprocessor directives).
- Examples:
 - `#define PI 3.14159`
 - `#define AGE 52`

Case Sensitivity

- C is **case sensitive**
 - It matters whether an **identifier**, such as a variable name, is uppercase or lowercase.
 - Example:

area

Area

AREA

ArEa

are all seen as different variables by the compiler.

Which Are Legal Identifiers?

- AREA area_under_the_curve
- 3D num45
- Last-Chance #values
- x_yt3 pi
- num\$ %done
- lucky***

Declaring Variables

- Before using a variable, you must give the compiler some information about the variable; i.e., you must **declare** it.
- The **declaration statement** includes the **data type** of the variable.
- Examples of variable declarations:
 - int meatballs ;
 - float area ;

Declaring Variables (con't)

- When we declare a variable
 - Space is set aside in memory to hold a value of the specified data type
 - That space is associated with the variable name
 - That space is associated with a unique **address**
- Visualization of the declaration

```
int meatballs ;
```

meatballs

garbage

FE07

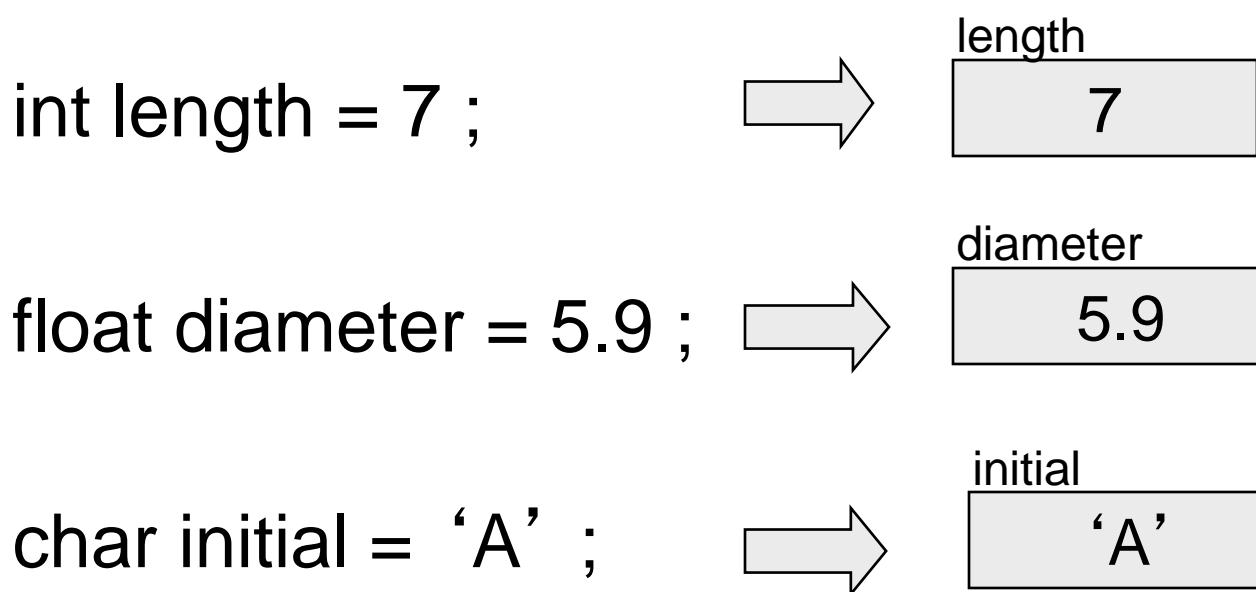
More About Variables

C has three basic predefined data types:

- Integers (whole numbers)
 - **Int**
- Floating point (real numbers)
 - **float**,
 - **double**
- Characters
 - **char**

Using Variables: Initialization

- Variables may be given initial values, or **initialized**, when declared. Examples:



Using Variables: Initialization (con't)

- Do not “hide” the initialization
 - put initialized variables on a separate line
 - a comment is always a good idea
 - Example:

```
int height ;      /* rectangle height */  
int width = 6 ;  /* rectangle width */  
int area ;       /* rectangle area */
```

NOT int height, width = 6, area ;

Using Variables: Assignment

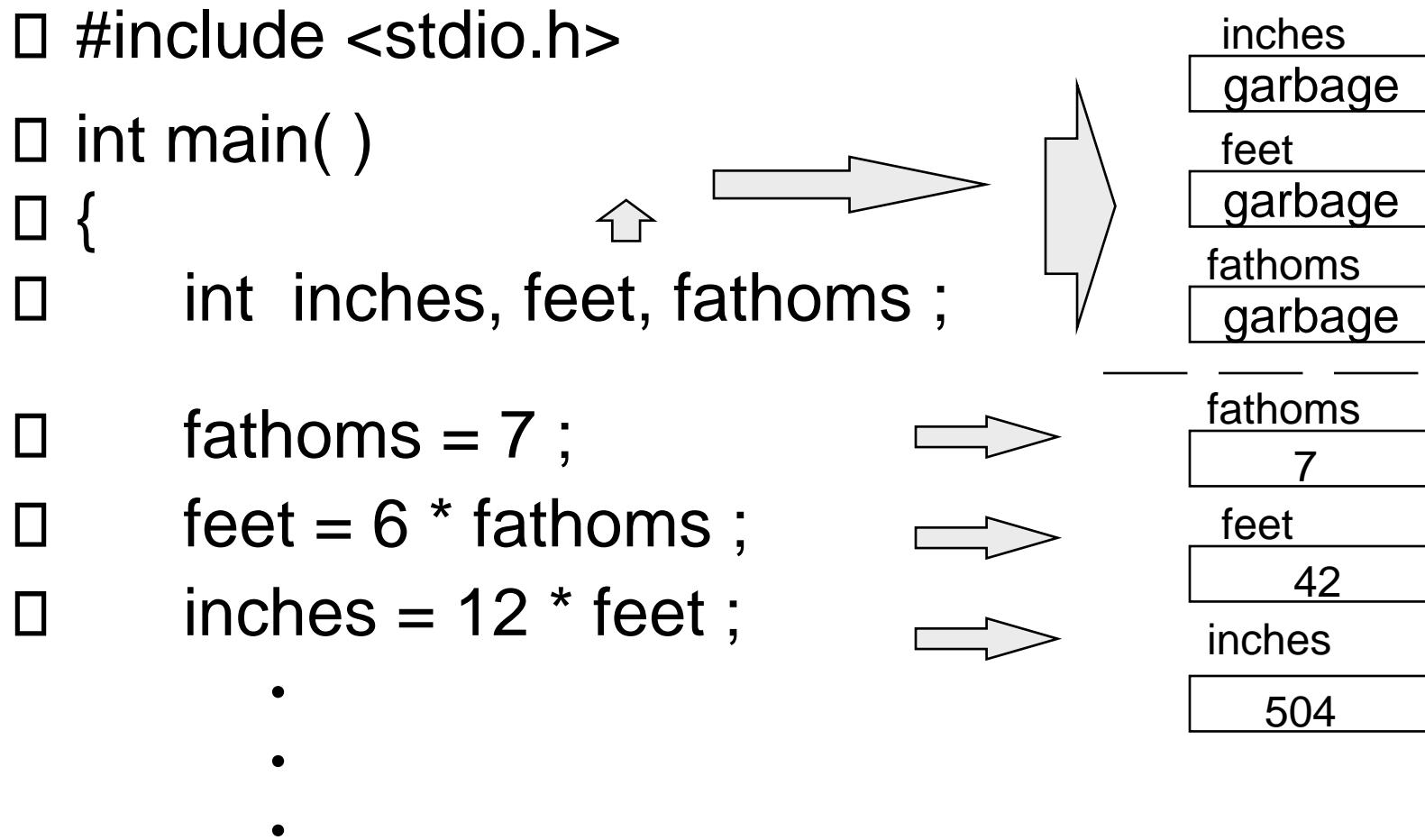
- Variables may have values assigned to them through the use of an **assignment statement**.
- Such a statement uses the **assignment operator** =
- This operator does not denote equality. It assigns the value of the righthand side of the statement (**the expression**) to the variable on the lefthand side.
- Examples:

diameter = 5.9 ;

area = length * width ;

Note that only single variables may appear on the lefthand side of the assignment operator.

Example: Declarations and Assignments



Example: Declarations and Assignments (cont'd)

```
    :  
    :  
    :  
□ printf ("Its depth at sea: \n") ;  
□ printf ("%d fathoms \n", fathoms) ;  
□ printf ("%d feet \n", feet) ;  
□ printf ("%d inches \n", inches) ;  
  
□ return 0 ;  
□ }
```

Enhancing Our Example

- What if the depth were really 5.75 fathoms?
Our program, as it is, couldn't handle it.
- Unlike integers, floating point numbers can contain decimal portions. So, let's use floating point, rather than integer.
- Let's also ask the user to enter the number of fathoms, rather than "**hard-coding**" it in.

Enhanced Program

```
#include <stdio.h>
int main ()
{
    float inches, feet, fathoms ;
    printf ("Enter the depth in fathoms : ");
    scanf ("%f", &fathoms) ;
    feet = 6 * fathoms ;
    inches = 12 * feet ;
    printf ("Its depth at sea: \n") ;
    printf ("%f fathoms \n", fathoms) ;
    printf ("%f feet \n", feet) ;
    printf ("%f inches \n", inches) ;
    return 0 ;
}
```

Final “Clean” Program

```
#include <stdio.h>
int main( )
{
    float inches ;      /* number of inches deep */
    float feet ;        /* number of feet deep */
    float fathoms ;     /* number of fathoms deep */

    /* Get the depth in fathoms from the user */
    printf ("Enter the depth in fathoms : ");
    scanf ("%f", &fathoms);

    /* Convert the depth to inches */
    feet = 6 * fathoms ;
    inches = 12 * feet ;
```

Final “Clean” Program (con’t)

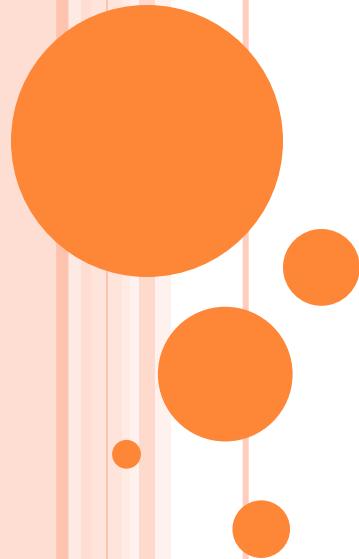
```
/* Display the results */  
printf (“Its depth at sea: \n”) ;  
printf (“    %f fathoms \n”, fathoms) ;  
printf (“    %f feet \n”, feet);  
printf (“    %f inches \n”, inches);  
  
return 0 ;  
}
```

Good Programming Practices

- Place each variable declaration on its own line with a descriptive comment.
- Place a comment before each logical “chunk” of code describing what it does.
- Do not place a comment on the same line as code (with the exception of variable declarations).
- Use spaces around all arithmetic and assignment operators.
- Use blank lines to enhance readability.

Good Programming Practices (con't)

- Place a blank line between the last variable declaration and the first executable statement of the program.
- Indent the body of the program 3 to 4 tab stops -- be consistent!



C – PRINTF AND SCANF FUNCTIONS

Abu Sayeed

Lecturer

Department of Computer Science

C - PRINTF() AND SCANF() FUNCTIONS

- printf() and scanf() functions are inbuilt library functions in C which are available in C library by default.
- These functions are declared and related macros are defined in “stdio.h” which is a header file.
- We have to include “stdio.h” file as shown in below C program to make use of these printf() and scanf() library functions.

```
#include <stdio.h>

int main()
{
    int a, b, c;
    a = 5;
    b = 7;
    c = a + b;
    printf("%d + %d = %d\n", a, b, c);
    return 0;
}
```

```
#include <stdio.h>

int main()
{
    int a, b, c;
    a = 5;
    b = 7;
    c = a + b;
    printf("%d + %d = %d\n", a, b, c);
    return 0;
}
```

- Here is an explanation of the different lines in this program:
- The line **int a, b, c;** declares three integer variables named **a**, **b** and **c**. Integer variables hold whole numbers.
- The next line initializes the variable named **a** to the value 5.
- The next line sets **b** to 7.
- The next line adds **a** and **b** and "assigns" the result to **c**. The computer adds the value in **a** (5) to the value in **b** (7) to form the result 12, and then places that new value (12) into the variable **c**. The variable **c** is assigned the value 12. For this reason, the **=** in this line is called "the assignment operator."
- The **printf** statement then prints the line "5 + 7 = 12." The **%d** placeholders in the **printf** statement act as placeholders for values. There are three **%d** placeholders, and at the end of the **printf** line there are the three variable names: **a**, **b** and **c**. C matches up the first **%d** with **a** and substitutes 5 there. It matches the second **%d** with **b** and substitutes 7. It matches the third **%d** with **c** and substitutes 12. Then it prints the completed line to the screen: 5 + 7 = 12. The **+**, the **=** and the spacing are a part of the format line and get embedded automatically between the **%d** operators as specified by the programmer.

PRINTF() FUNCTION

○ C printf() function:

- The **printf statement allows you to send output to standard out**. For us, standard out is generally the screen (although you can redirect standard out into a text file or another command).
- printf() function is used to print the “character, string, float, integer, octal and hexadecimal values” onto the output screen.
- We use printf() function with %d format specifier to display the value of an integer variable.
- Similarly %c is used to display character, %f for float variable, %s for string variable, %lf for double and %x for hexadecimal variable.
- To generate a newline, we use “\n” in C printf() statement.

○ Note:

- C language is case sensitive. For example, printf() and scanf() are different from **Printf()** and **Scanf()**. All characters in printf() and scanf() functions must be in lower case.

EXAMPLE PROGRAM `PRINTF()` FUNCTION

```
#include <stdio.h>

int main()
{
    char ch = 'A';
    char str[20] = "fresh2refresh.com";
    float flt = 10.234;
    int no = 150;

    double dbl = 20.123456;
    printf("Character is %c \n", ch);
    printf("String is %s \n" , str);
    printf("Float value is %f \n", flt);
    printf("Integer value is %d\n" , no);

    printf("Double value is %lf \n", dbl);
    printf("Octal value is %o \n", no);
    printf("Hexadecimal value is %x \n", no);
    return 0;
}
```

Output:

Character is A
String is fresh2refresh.com
Float value is 10.234000
Integer value is 150
Double value is 20.123456
Octal value is 226
Hexadecimal value is 96

FORMAT SPECIFIER

- You can see the output with the same data which are placed within the double quotes of printf statement in the program except
 - %d got replaced by value of an integer variable (no),
 - %c got replaced by value of a character variable (ch),
 - %f got replaced by value of a float variable (flt),
 - %lf got replaced by value of a double variable (dbl),
 - %s got replaced by value of a string variable (str),
 - %o got replaced by a octal value corresponding to integer variable (no),
 - %x got replaced by a hexadecimal value corresponding to integer variable
 - \n got replaced by a newline.



EXAMPLE PROGRAM : PRINTF() AND SCANF() FUNCTIONS

○ **scanf()** function:

- The **scanf function allows you to accept input from standard in**, which for us is generally the keyboard.
- **scanf()** function is used to read character, string, numeric data from keyboard
- Consider below example program where user enters a character. This value is assigned to the variable “ch” and then displayed.
- Then, user enters a string and this value is assigned to the variable ”str” and then displayed.

```
#include <stdio.h>

int main()
{
    char ch;
    char str[100];

    printf("Enter any character \n");
    scanf("%c", &ch);
    printf("Entered character is %c \n", ch);

    printf("Enter any string ( upto 100 character ) \n");
    scanf("%s", &str);
    printf("Entered string is %s \n", str);

    return 0;
}
```

Enter any character

a

Entered character is a

Enter any string (upto 100 character)
hai

Entered string is hai

EXAMPLE PROGRAM : PRINTF() AND SCANF() FUNCTIONS

- The format specifier %d is used in scanf() statement. So that, the value entered is received as an integer and %s for string.
- Ampersand is used before variable name “ch” in scanf() statement as &ch , this operator is called *address of operator*.



BASIC IO FUNCTIONS

I/O Library Functions	Meanings
getch()	Inputs a single character (most recently typed) from standard input (usually console).
getche()	Inputs a single character from console and echoes (displays) it.
getchar()	Inputs a single character from console and echoes it, but requires <i>Enter</i> key to be typed after the character.
putchar() or putch()	Outputs a single character on console (screen).
scanf()	Enables input of formatted data from console (keyboard). Formatted input data means we can specify the data type expected as input. Format specifiers for different data types are given in Figure 21.6.
printf()	Enables obtaining an output in a form specified by programmer (formatted output). Format specifiers are given in Figure 21.6. Newline character "\n" is used in <i>printf()</i> to get the output split over separate lines.
gets()	Enables input of a string from keyboard. Spaces are accepted as part of the input string, and the input string is terminated when <i>Enter</i> key is hit. Note that although <i>scanf()</i> enables input of a string of characters, it does not accept multi-word strings (spaces in-between).
puts()	Enables output of a multi-word string

understanding the execution of a C Program



Flowchart

Abu Sayeed
Lecturer
Department of Computer Science

Flowchart

- An organized combination of shapes, lines and text which graphically illustrate a process/program.
- A type of diagram that represents an algorithm, workflow or process, showing the steps as boxes of various kinds, and their order by connecting them with arrows. This diagrammatic representation illustrates a solution to a given problem.
- Flowcharts are used in analyzing, designing, documenting or managing a process or program. Like other types of diagrams, they help visualize what is going on and thereby help the people to understand a process, and perhaps also find flaws, bottlenecks, and other less-obvious features within it.
- Very helpful in explaining program to others.

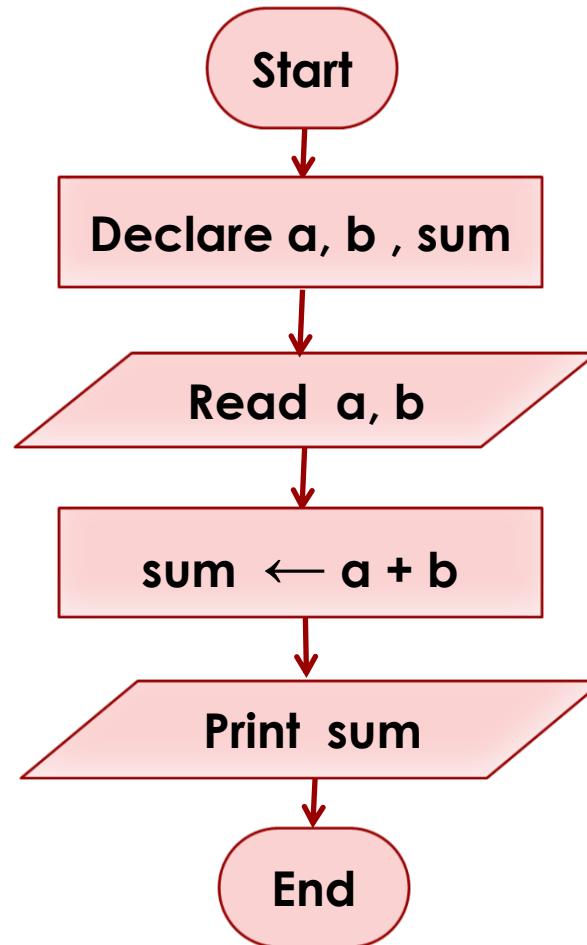
Symbols Used In Flowchart

Different symbols are used for different states in flowchart, For example: Input/Output and decision making has different symbols. The table below describes most of the symbols that are used in making flowchart

Symbol	Shape	Terminology	Purpose of the Symbol
	Oval	Terminator	To begin/end (start and stop)
	Arrow	Connector line	To connect any two symbols
	Rectangle	Process	Initialization and Computation
	Parallelogram	Data	Read and Print Data
	Rhombus	Decision	Conditionals – branching
	Circle	On page connector	To have a flowchart connectivity on the same page
	Pentagon	Off page connector	Connecting part of flowchart (which is in one page) to the remaining part (which is in the next page)

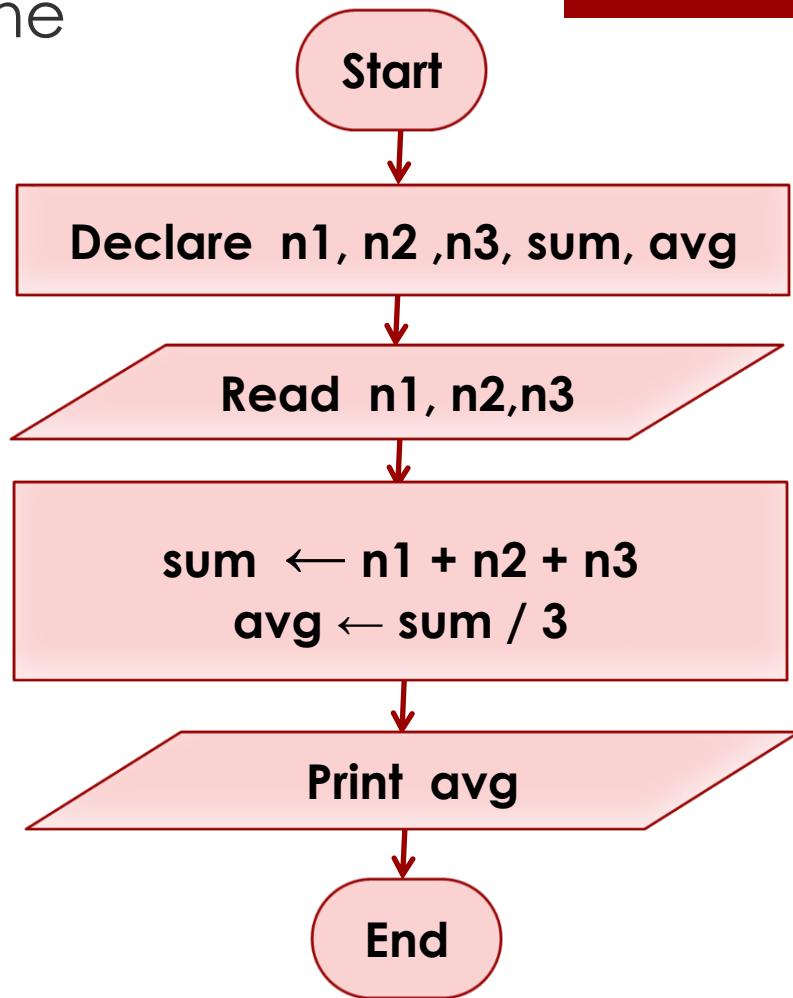
Example of flowchart in

- Draw a flowchart to add two numbers entered by user.



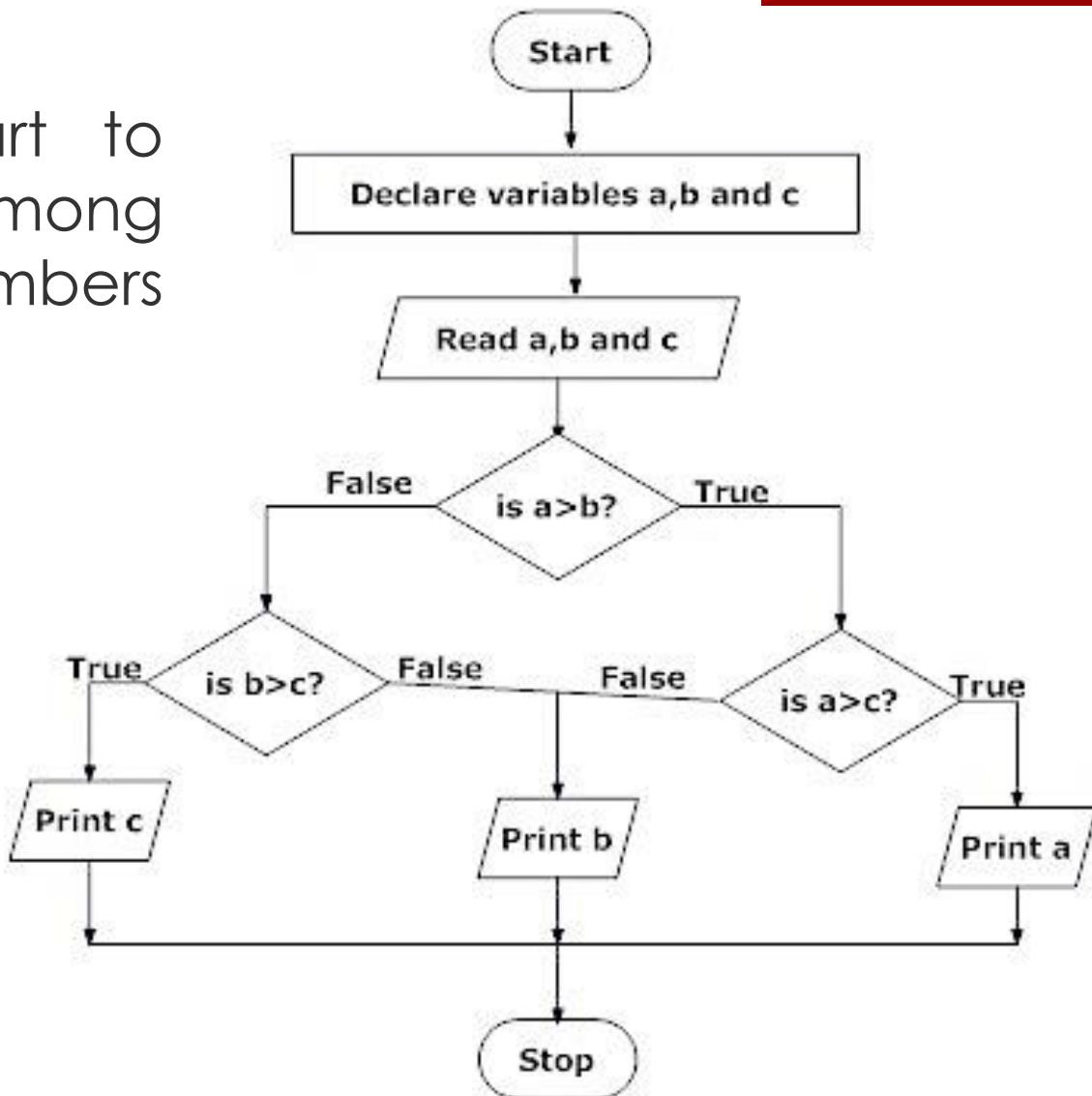
Example 2

- Draw a flowchart to print the average of three numbers entered by user.



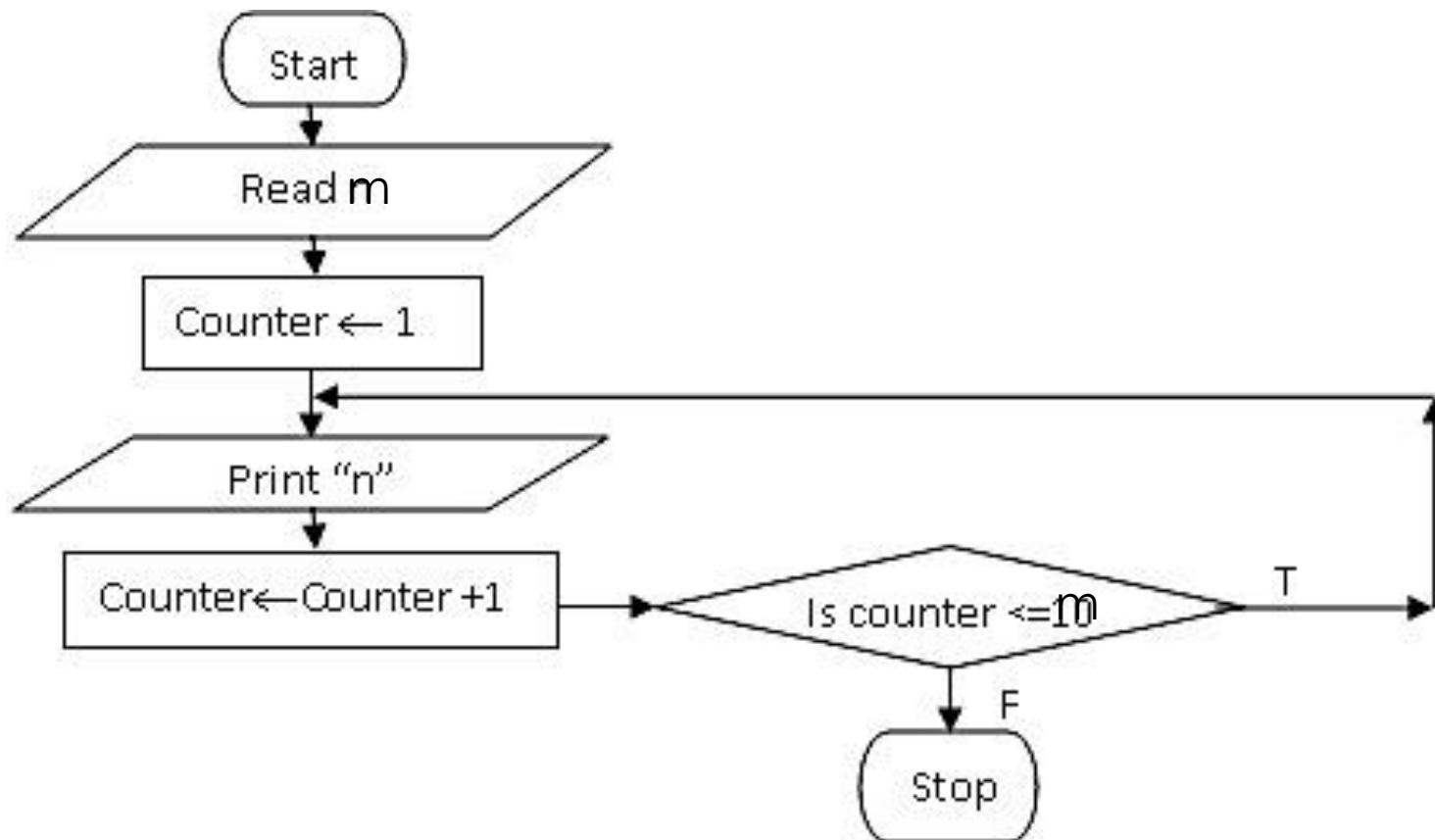
Example 3

- Draw a flowchart to find the largest among three different numbers entered by user.



Example 4

- Draw a flowchart to print a letter 'n' m times.





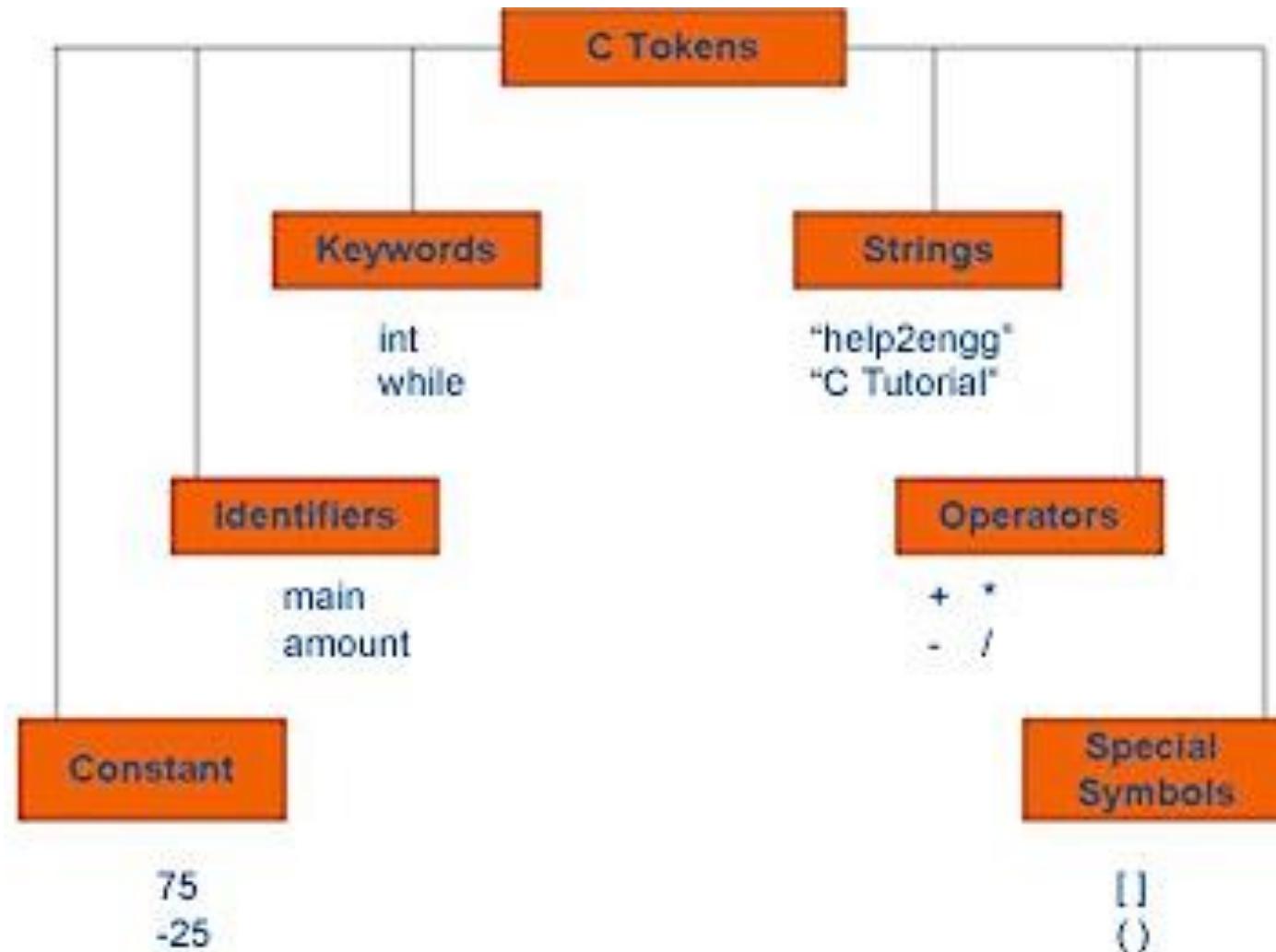
C Token

Dr. Sheak Rashed Haider Noori
Assistant Professor
Department of Computer Science

C tokens

- C tokens are the basic buildings blocks in C language which are constructed together to write a C program.
- Each and every smallest individual units in a C program is known as C token or a lexical unit.
- C tokens can be classified in six types as follows:
 - ① Keywords (e.g., int, while, do ...),
 - ② Identifiers (e.g., main, total, my_var ...),
 - ③ Constants (e.g., 10, 20 , - 25.5 ...),
 - ④ Strings (e.g., “total”, “hello” , “DIU” ...),
 - ⑤ Special symbols (e.g., (), {}, [] ...),
 - ⑥ Operators (e.g., +, /, -, * ...)

C tokens



C tokens example program

```
int main()
{
    int x, y, total;
    x = 10, y = 20;
    total = x + y;
    printf("Total %d \n", total);
    return 0;
}
```

- where,
 - Main, x, y, total – identifier
 - {}, .() – special symbols
 - Int , return – keyword
 - 10, 20 – constant
 - =,+ – operator
 - “Total 30” – String
 - main, {, }, (,), int, x, y, total etc– all these are various tokens

C Keywords

- (1) **Keywords**
- (2) Identifiers
- (3) Constants
- (4) Strings
- (5) Special symbols
- (6) Operators

- There are some reserved words in C language whose meaning are predefined in C compiler, those are called C keywords.
- Each keyword is meant to perform a specific function in a C program.
- Each keyword has fixed meaning and that cannot be changed by user.
- We cannot use a keyword as a variable name.
- Since upper case and lowercase characters are not considered same in C, we can use an uppercase keyword as an identifier. But it is not considered as good programming practice.

- ① **Keywords**
- ② Identifiers
- ③ Constants
- ④ Strings
- ⑤ Special symbols
- ⑥ Operators

Keywords

- There are 32 keywords in C which are given below. keywords are all lowercase.

Keywords in C Language

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

- ① Keywords
- ② **Identifiers**
- ③ Constants
- ④ Strings
- ⑤ Special symbols
- ⑥ Operators

Identifiers

- Each program elements in a C program are given a name called identifiers.
- Names given to identify variables, functions etc. are examples for identifiers.
 - e.g., `int x ;` here `x` is a name given to an integer variable.
- **Rules for constructing identifier name in C:**
 - An identifier can be composed of letters (both uppercase and lowercase letters), digits and underscore only.
 - The first character of identifier should be either a letter or an underscore(not any digit). But, it is discouraged to start an identifier name with an underscore though it is legal. It is because, identifier that starts with underscore can conflict with system names. In such cases, compiler will complain about it.
 - Punctuation and special characters are not allowed except underscore.
 - Identifiers should not be keywords.
 - Identifiers are case sensitive.
 - There is no rule for the length of an identifier. However, the first 31 characters of an identifier are discriminated by the compiler. So, the first 31 letters of two identifiers in a program should be different.

Constants

- ① Keywords
- ② Identifiers
- ③ **Constants**
- ④ Strings
- ⑤ Special symbols
- ⑥ Operators

- Constants in C refer to fixed values that do not change during the execution of a program.
- Example: 1, 2.5 , "Programming is fun." etc. are the example of constants.
- In C, constants can be classified as follows:
 - Numeric constants
 - Integer constant (Ex: 102, - 5)
 - Real constant (Ex: 3.14, 5.5)
 - Character constants
 - Single character constants (Ex: 'A` , ';' , '5`)
 - String constants (Ex: "Hello" , "5+4")
 - Backslash character constants (Ex: \n, \r)

- ① Keywords
- ② Identifiers
- ③ **Constants**
- ④ Strings
- ⑤ Special symbols
- ⑥ Operators

Integer constants

- Integer constants are the numeric constants (constant associated with number) without any fractional part or exponential part.
- There are three types of integer constants in C language:
 - decimal constant(base 10),
 - octal constant(base 8) and
 - hexadecimal constant(base 16).
- For example:
 - Decimal constants: 0, -9 , 22 etc
 - Octal constants: 021, 077, 033 etc
 - Hexadecimal constants: 0x7f, 0x2a, 0x521 etc
- Note: Every octal constant starts with 0 and hexadecimal constant starts with 0x in C programming.

Real Constants

- ① Keywords
- ② Identifiers
- ③ **Constants**
- ④ Strings
- ⑤ Special symbols
- ⑥ Operators

- Real constant, also called Floating point constants are the numeric constants that has either fractional form or exponent form.
- For example:
 - -2.0
 - 0.0000234
 - -0.22E-5
- **Note:** Here, E-5 represents 10^{-5} . Thus, -0.22E-5 = -0.0000022.

Single Character Constants

- ① Keywords
- ② Identifiers
- ③ **Constants**
- ④ Strings
- ⑤ Special symbols
- ⑥ Operators

- Single character constants are the constant which use single quotation around characters.
- For example: 'a', 'l', 'm', 'F' etc.
- All character constants have an equivalent integer value which are called ASCII Values.

String constants

- ① Keywords
- ② Identifiers
- ③ **Constants**
- ④ Strings
- ⑤ Special symbols
- ⑥ Operators

- A string is a sequence of characters enclosed in double quotes.
- The sequence of characters may contain letters, numbers, special characters and blank spaces.
- String constants are the constants which are enclosed in a pair of double-quote marks.
- For example:
 - "good" // string constant
 - "" // null string constant
 - " " " // string constant of six white space
 - "x" // string constant having single character.

①	Keywords
②	Identifiers
③	Constants
④	Strings
⑤	Special symbols
⑥	Operators

Backslash Character Constant

- Sometimes, it is necessary to use newline(enter), tab, quotation mark etc. in the program which either cannot be typed or has special meaning in C programming.
- In such cases, backslash character constant (escape sequence) are used.
- For example: \n is used for newline.
- The backslash(\) causes "escape" from the normal way the characters are interpreted by the compiler.

List of Escape Sequences

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

- ① Keywords
- ② Identifiers
- ③ **Constants**
- ④ Strings
- ⑤ Special symbols
- ⑥ Operators

Special Symbols

- ① Keywords
- ② Identifiers
- ③ Constants
- ④ Strings
- ⑤ Special symbols

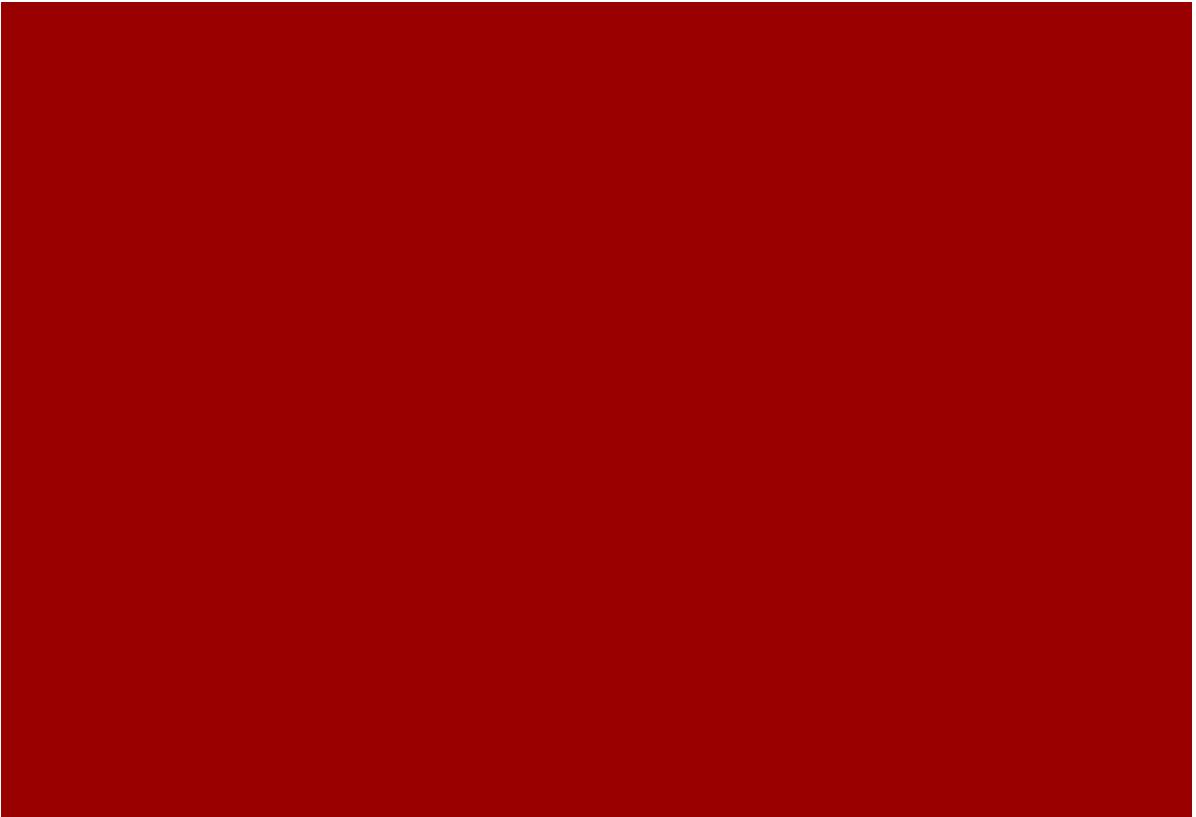
- ⑥ Operators

- The following special symbols are used in C having some special meaning and thus, cannot be used for some other purpose.
- [] () {} , ; : * ... = #
- **Braces{ }:** These opening and ending curly braces marks the start and end of a block of code containing more than one executable statement.
- **Parentheses():** These special symbols are used to indicate function calls and function parameters.
- **Brackets[]:** Opening and closing brackets are used as array element reference. These indicate single and multidimensional subscripts.

①	Keywords
②	Identifiers
③	Constants
④	Strings
⑤	Special symbols
⑥	Operators

Operators

- C operators are symbols that triggers an action when applied to C variables and other objects. The data items on which operators act upon are called operands.
- Depending on the number of operands that an operator can act upon, operators can be classified as follows:
 - **Unary Operators:** Those operators that require only single operand to act upon are known as unary operators.
 - **Binary Operators:** Those operators that require two operands to act upon are called binary operators.
 - **Ternary Operators:** These operators requires three operands to act upon.
- There are many operators, some of which are single characters ~ ! @ % ^ & * - + = | / : ? < >
- While others require two characters ++ -- << >> <= += -= *= /= == |= %= &= ^= || && !=
- Some even require three characters <<= >>=
- The multiple-character operators can not have white spaces or comments between the characters.



Operators And Expression

Dr. Sheak Rashed Haider Noori
Assistant Professor
Department of Computer Science

Operators and Expressions

- Consider the expression $A + B * 5$, where,

- $+$, $*$ are **operators**,
- A, B are **variables**,
- 5 is **constant**,
- A, B and 5 are called **operand**, and
- $A + B * 5$ is an **expression**.

$(a+b) *c$

Operator is $*$, operands are $(a+b)$ and c

$(a+b)$

Operator is $()$, operand is $a+b$

$a+b$

Operator is $+$, operands are a and b

Types of C operators

- C language offers many types of operators, such as:
 - Arithmetic operators
 - Assignment operators
 - Increment/decrement operators
 - Relational operators
 - Logical operators
 - Bit wise operators
 - Conditional operators (ternary operators)
 - Special operators

Arithmetic Operators

Arithmetic operators
Assignment operators
Inc/dec operators
Relational operators
Logical operators
Bit wise operators
Conditional operators
Special operators

- C Arithmetic operators are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus in C programs.

Arithmetic Operators	Operation	Example
+	Addition	A+B
-	Subtraction	A-B
*	multiplication	A*B
/	Division	A/B
%	Modulus	A%B

Arithmetic Operators

Arithmetic operators
Assignment operators
Inc/dec operators
Relational operators
Logical operators
Bit wise operators
Conditional operators
Special operators

- There are three types of arithmetic operations using arithmetic operators:
 - ① **Integer arithmetic** : when all operands are integer. If $a=15$, $b=10$,
 - $a + b = 25$
 - $a / b = 1$ (decimal part)
 - $a \% b = 5$ (remainder of division)
 - ② **Real arithmetic** : All operands are only real number. If $a=15.0$, $b=10.0$
 - $a / b = 1.5$
 - ③ **Mixed model arithmetic** : when one operand is real and another is integer. If $a=15$ and $b= 10.0$
 - $a / b = 1.5$ whereas, $15/10=1$
- Note: The modulus operator % gives you the remainder when two integers are divided: $1 \% 2$ is 1 and $7 \% 4$ is 3.
- The modulus operator can only be applied to integers.

Arithmetic Operators

① Integer arithmetic :

- When an arithmetic operation is performed on two whole numbers or integers than such an operation is called as integer arithmetic.
- It always gives an integer as the result.
- Let $x = 27$ and $y = 5$ be 2 integer numbers. Then the integer operation leads to the following results.

- $x + y = 32$
- $x - y = 22$
- $x * y = 115$
- $x \% y = 2$
- $x / y = 5$

Example program for C arithmetic operators

Arithmetic operators
Assignment operators
Inc/dec operators
Relational operators
Logical operators
Bit wise operators
Conditional operators
Special operators

```
#include <stdio.h>
int main()
{
    int a=40,b=20, add,sub,mul,div,mod;

    add = a+b;
    sub = a-b;
    mul = a*b;
    div = a/b;
    mod = a%b;

    printf("Addition of a, b is : %d\n", add);
    printf("Subtraction of a, b is : %d\n", sub);
    printf("Multiplication of a, b is : %d\n", mul);
    printf("Division of a, b is : %d\n", div);
    printf("Modulus of a, b is : %d\n", mod);
}
```

Output:

Addition of a, b is : 60
Subtraction of a, b is : 20
Multiplication of a, b is : 800
Division of a, b is : 2
Modulus of a, b is : 0

Arithmetic Operators

Arithmetic operators
Assignment operators
Inc/dec operators
Relational operators
Logical operators
Bit wise operators
Conditional operators
Special operators

② Real arithmetic :

- When an arithmetic operation is preformed on two real numbers or fraction numbers such an operation is called real or floating point arithmetic.
- The modulus (remainder) operator is not applicable for floating point arithmetic operands.
- Let $x = 14.0$ and $y = 4.0$ then
 - $x + y = 18.0$
 - $x - y = 10.0$
 - $x * y = 56.0$
 - $x / y = 3.50$

Arithmetic Operators

Arithmetic operators
Assignment operators
Inc/dec operators
Relational operators
Logical operators
Bit wise operators
Conditional operators
Special operators

③ Mixed mode arithmetic :

- When one of the operand is real and other is an integer and if the arithmetic operation is carried out on these 2 operands then it is called as mixed mode arithmetic.
- If any one operand is of real type then the result will always be real
- Let $x = 15$ and $y = 10.0$ then
 - $x / y = 1.5$
- Note that: $15 / 10 = 1$ (since both of the operands are integer)

Assignment Operators

- In C programs, values for the variables are assigned using assignment operators.
- For example, if the value “10” is to be assigned for the variable “sum”, it can be assigned as **sum = 10;**

Arithmetic operators
Assignment operators
Inc/dec operators
Relational operators
Logical operators
Bit wise operators
Conditional operators
Special operators

Operators	Example	Explanation
Simple assignment operator	= sum=10	10 is assigned to variable sum
Shorthand or Compound assignment operators	+ = sum+=10	This is same as sum=sum+10
	- = sum-=10	This is same as sum = sum-10
	* = sum*=10	This is same as sum = sum*10
	/ = sum/=10	This is same as sum = sum/10
	% = sum%=10	This is same as sum = sum%10
	& = sum&=10	This is same as sum = sum&10
	^ = sum^=10	This is same as sum = sum^10

Increment and Decrement Operators

Arithmetic operators
Assignment operators
Inc/dec operators
Relational operators
Logical operators
Bit wise operators
Conditional operators
Special operators

- There are two more shorthand operators:
 - Increment `--`
 - Decrement `++`
- These two operators are for incrementing and decrementing a variable by 1.
- For example, the following code increments `i` by 1 and decrements `j` by 1.

```
int i = 3, j = 3;  
i++; // i becomes 4  
j--; // j becomes 2
```

Increment and Decrement Operators

Arithmetic operators
Assignment operators
Inc/dec operators
Relational operators
Logical operators
Bit wise operators
Conditional operators
Special operators

- The ++ and -- operators can be used in **prefix** or **suffix** mode, as shown in Table

<i>Operator</i>	<i>Name</i>	<i>Description</i>	<i>Example (assume i = 1)</i>
<code>++var</code>	preincrement	Increment <code>var</code> by <code>1</code> and use the new <code>var</code> value	<code>int j = ++i; // j is 2, // i is 2</code>
<code>var++</code>	postincrement	Increment <code>var</code> by <code>1</code> , but use the original <code>var</code> value	<code>int j = i++; // j is 1, // i is 2</code>
<code>--var</code>	predecrement	Decrement <code>var</code> by <code>1</code> and use the new <code>var</code> value	<code>int j = --i; // j is 0, // i is 0</code>
<code>var--</code>	postdecrement	Decrement <code>var</code> by <code>1</code> and use the original <code>var</code> value	<code>int j = ++i; // j is 1, // i is 0</code>

Increment and Decrement Operators

Arithmetic operators
Assignment operators
Inc/dec operators
Relational operators
Logical operators
Bit wise operators
Conditional operators
Special operators

- If the operator is *before* (prefixed to) the variable, the variable is incremented or decremented by 1, then the new value of the variable is returned.
- If the operator is *after* (suffixed to) the variable, then the variable is incremented or decremented by 1, but the original *old* value of the variable is returned.
- Therefore, the prefixes $++x$ and $--x$ are referred to, respectively, as the *preincrement* operator and the *predecrement* operator; and the suffixes $x++$ and $x--$ are referred to, respectively, as the *postincrement* operator and the *postdecrement* operator.

Increment and Decrement Operators

Arithmetic operators
Assignment operators
Inc/dec operators
Relational operators
Logical operators
Bit wise operators
Conditional operators
Special operators

The prefix form of ++ (or --) and the suffix form of ++ (or --) are the same if they are used in isolation, but they cause different effects when used in an expression. The following code illustrates this:

```
int i = 10;  
int newNum = 10 * i++;
```

Same effect as

```
int newNum = 10 * i;  
i = i + 1;
```

In this case, i is incremented by 1, then the *old* value of i is returned and used in the multiplication. So newNum becomes **100**. If i++ is replaced by ++i as follows,

```
int i = 10;  
int newNum = 10 * (++i);
```

Same effect as

```
i = i + 1;  
int newNum = 10 * i;
```

i is incremented by 1, and the new value of i is returned and used in the multiplication. Thus newNum becomes **110**.

Exercise on ++ and - -

Arithmetic operators
Assignment operators
Inc/dec operators
Relational operators
Logical operators
Bit wise operators
Conditional operators
Special operators

■ int x=2 , y = 5 , z = 0;

- x++ ; y++ ;
- x=y++ + x++;
- y=++y + ++x;
- y=++y + x++;
- y += ++y;
- y += 1 + (++x);
- y += 2 + x++;

Relational Operators

Arithmetic operators
Assignment operators
Inc/dec operators
Relational operators
Logical operators
Bit wise operators
Conditional operators
Special operators

- Relational operators are used to find the relation between two variables. i.e. to compare the values of two variables.

S.no	Operators	Example	Description
1	>	$x > y$	x is greater than y
2	<	$x < y$	x is less than y
3	\geq	$x \geq y$	x is greater than or equal to y
4	\leq	$x \leq y$	x is less than or equal to y
5	\equiv	$x \equiv y$	x is equal to y
6	\neq	$x \neq y$	x is not equal to y

Logical Operators

- These operators are used to perform logical operations on the given expressions.
- There are 3 logical operators in C language. They are, logical AND (`&&`), logical OR (`||`) and logical NOT (`!`).

Arithmetic operators
Assignment operators
Inc/dec operators
Relational operators
Logical operators
Bit wise operators
Conditional operators
Special operators

S.no	Operators	Name	Example	Description
1	<code>&&</code>	logical AND	<code>(x>5)&&(y<5)</code>	It returns true when both conditions are true
2	<code> </code>	logical OR	<code>(x>=10) (y>=10)</code>	It returns true when at-least one of the condition is true
3	<code>!</code>	logical NOT	<code>!((x>5)&&(y<5))</code>	It reverses the state of the operand "((x>5) && (y<5))" If "((x>5) && (y<5))" is true, logical NOT operator makes it false

Example program for logical operators in C

```
#include <stdio.h>
int main()
{
    int m=40,n=20;
    int o=20,p=30;

    if (m>n && m !=0)
    {
        printf("&& Operator : Both conditions are true\n");
    }
    if (o>p || p!=20)
    {
        printf("|| Operator : Only one condition is true\n");
    }
    if (!(m>n && m !=0))
    {
        printf("! Operator : Both conditions are true\n");
    }
    else
    {
        printf("! Operator : Both conditions are true. " \
               "But, status is inverted as false\n");
    }
}
```

Output:

&& Operator : Both conditions are true
|| Operator : Only one condition is true
! Operator : Both conditions are true. But, status is inverted as false

Arithmetic operators
Assignment operators
Inc/dec operators
Relational operators
Logical operators
Bit wise operators
Conditional operators
Special operators

Bit wise Operators

Arithmetic operators
Assignment operators
Inc/dec operators
Relational operators
Logical operators
Bit wise operators
Conditional operators
Special operators

- One of C's powerful features is a set of bit manipulation operators. These permit the programmer to access and manipulate individual bits within a piece of data to perform bit operations. The various Bitwise Operators available in C are shown in Figure
- Decimal values are converted into binary values which are the sequence of bits and bit wise operators work on these bits.
- These operators can operate upon **ints** and **chars** but not on **floats** and **doubles**.

Operator	Meaning
<code>~</code>	One's complement
<code>>></code>	Right shift
<code><<</code>	Left shift
<code>&</code>	Bitwise AND
<code> </code>	Bitwise OR
<code>^</code>	Bitwise XOR(Exclusive OR)

Special Operators

Arithmetic operators
Assignment operators
Inc/dec operators
Relational operators
Logical operators
Bit wise operators
Conditional operators
Special operators

S.no	Operators	Description
1	&	This is used to get the address of the variable. Example : &a will give address of a.
2	*	This is used as pointer to a variable. Example : * a where, * is pointer to the variable a.
3	Sizeof ()	This gives the size of the variable. Example : size of (char) will give us 1.

Example program for Special operators in C

```
#include <stdio.h>

int main()
{
    int *ptr, q;
    q = 50;
    /* address of q is assigned to ptr */
    ptr = &q;
    /* display q's value using ptr variable */
    printf("%d", *ptr);
    return 0;
}
```

Example program for sizeof() operator in C

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

int main()
{
    int a;
    char b;
    float c;
    double d;
    printf("Storage size for int data type:%d \n", sizeof(a));
    printf("Storage size for char data type:%d \n", sizeof(b));
    printf("Storage size for float data type:%d \n", sizeof(c));
    printf("Storage size for double data type:%d\n", sizeof(d));
    return 0;
}
```

Output:

Storage size for int data type:4
Storage size for char data type:1
Storage size for float data type:4
Storage size for double data type:8

Operators And Expression

Bit wise Operators

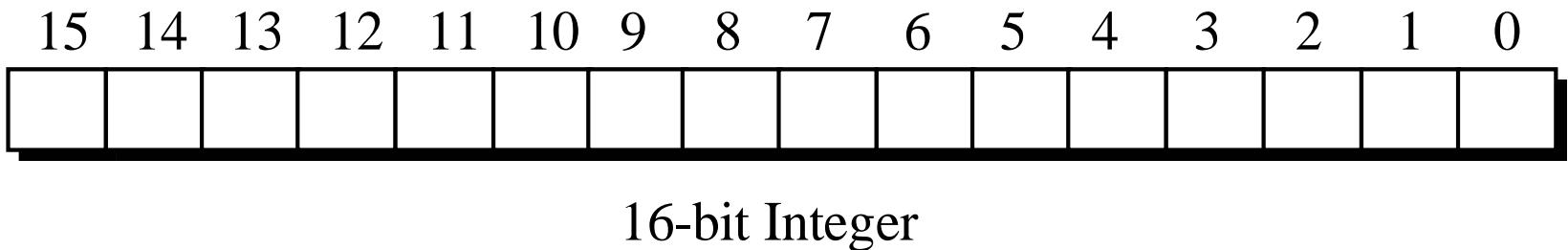
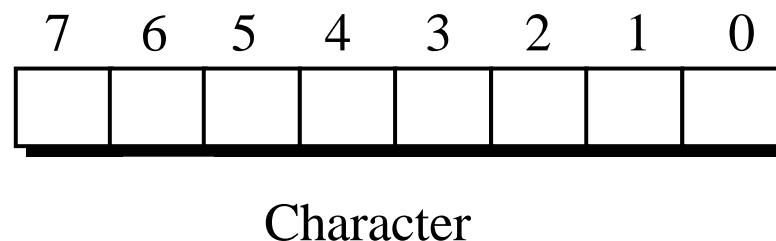
Bit wise Operators

- One of C's powerful features is a set of bit manipulation operators.
- These permit the programmer to access and manipulate individual bits within a piece of data to perform bit operations.
- Decimal values are converted into binary values which are the sequence of bits and bit wise operators work on these bits.
- These operators can operate upon **ints** and **chars** but not on **floats** and **doubles**.

Operator	Meaning
<code>~</code>	One's complement
<code>>></code>	Right shift
<code><<</code>	Left shift
<code>&</code>	Bitwise AND
<code> </code>	Bitwise OR
<code>^</code>	Bitwise XOR(Exclusive OR)

Bit wise Operators

- Let us first take a look at the bit numbering scheme in integers and characters. Bits are numbered from zero onwards, increasing from right to left as shown below:



One's Complement Operator (\sim)

- One's complement operator is represented by the symbol \sim
- On taking one's complement of a number, all 1's present in the number are changed to 0's and all 0's are changed to 1's.
- For example one's complement of 1010 is 0101. Similarly, one's complement of 1111 is 0000. Note that here when we talk of a number we are talking of binary equivalent of the number.
- Thus, one's complement of 65 means one's complement of 0000 0000 0100 0001, which is binary equivalent of 65. One's complement of 65 therefore would be, 1111 1111 1011 1110.

Use of one's complement operator

- In real-world situations where could the one's complement operator be useful?
- Since it changes the original number beyond recognition, one potential place where it can be effectively used is in development of a file encryption utility

Right Shift Operator (>>)

- The right shift operator is represented by >>.
- It needs two operands. It shifts each bit in its left operand to the right.
- The number of places the bits are shifted depends on the number following the operator (i.e. its right operand).
- Thus, **ch >> 3** would shift all bits in **ch** three places to the right. Similarly, **ch >> 5** would shift all bits 5 places to the right.
- For example, if the variable **ch** contains the bit pattern 11010111, then, **ch >> 1** would give 01101011 and **ch >> 2** would give 00110101.
- Note that as the bits are shifted to the right, blanks are created on the left. These blanks must be filled somehow. They are always filled with zeros.

Left Shift Operator (<<)

- The left shift operator is represented by <<.
- This is similar to the right shift operator, the only difference being that the bits are shifted to the left, and for each bit shifted, a 0 is added to the right of the number.

Bitwise AND Operator (&)

- This operator is represented as **&**.
- Remember it is different than **&&**, the logical AND operator.
- The **&** operator operates on two operands. While operating upon these two operands they are compared on a bit-by-bit basis. Hence both the operands must be of the same type (either **char** or **int**).
- The second operand is often called an AND mask. The **&** operator operates on a pair of bits to yield a resultant bit.

Bitwise AND Operator (&)

- The rules that decide the value of the resultant bit are shown below:

First bit	Second bit	First bit & Second bit
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise AND Operator (&)

- The example given below shows more clearly what happens while ANDing one operand with another.

7	6	5	4	3	2	1	0
1	0	1	0	1	0	1	0

This operand when
ANDed bitwise

7	6	5	4	3	2	1	0
1	1	0	0	0	0	1	1

With this operand
yields

7	6	5	4	3	2	1	0
1	0	0	0	0	0	1	0

this result

Bitwise AND Operator (&)

- Probably, the best use of the AND operator is to check whether a particular bit of an operand is ON or OFF. This is explained in the following example.
- Suppose, from the bit pattern 10101101 of an operand, we want to check whether bit number 3 is ON (1) or OFF (0).

Then the ANDing operation would be,

10101101

00001000

00001000

Original bit pattern

AND mask

Resulting bit pattern

Bitwise AND Operator (&)

- In every file entry present in the directory, there is an attribute byte. The status of a file is governed by the value of individual bits in this attribute byte. The AND operator can be used to check the status of the bits of this attribute byte. The meaning of each bit in the attribute byte is shown in Figure

Bitwise AND Operator (&)

- Now, suppose we want to check whether a file is a hidden file or not. A hidden file is one, which is never shown in the directory, even though it exists on the disk.
- From the above bit classification of attribute byte, we only need to check whether bit number 1 is ON or OFF.
- So, our first operand in this case becomes the attribute byte of the file in question, whereas the second operand is the $1 * 2^1 = 2$.
- Similarly, it can be checked whether the file is a system file or not, whether the file is read-only file or not, and so on.
- The second important use of the AND operator is in changing the status of the bit, or more precisely to switch OFF a particular bit.

Bit wise AND operator

- Let's summarize the uses of bitwise AND operator:
 - ① It is used to check whether a particular bit in a number is ON or OFF.
 - ② It is used to turn OFF a particular bit in a number.

Bitwise XOR Operator (\wedge)

- The XOR operator is represented as \wedge and is also called an Exclusive OR Operator.
- The OR operator returns 1, when any one of the two bits or both the bits are 1, whereas XOR returns 1 only if one of the two bits is 1.
- XOR operator is used to toggle a bit ON or OFF.
- The truth table for the XOR operator is given below.

x	y	$x \text{ XOR } y$
0	0	0
0	1	1
1	0	1
1	1	0

Truth table for bit wise operations

x	y	x y	x & y	x ^ y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

Bit wise Operators

```
#include <stdio.h>

int main()
{
    unsigned int a = 60; /* 60 = 0011 1100 */
    unsigned int b = 13; /* 13 = 0000 1101 */
    int c = 0;

    c = a & b;          /* 12 = 0000 1100 */
    printf("Line 1 - Value of c is %d\n", c);

    c = a | b;          /* 61 = 0011 1101 */
    printf("Line 2 - Value of c is %d\n", c);

    c = a ^ b;          /* 49 = 0011 0001 */
    printf("Line 3 - Value of c is %d\n", c);

    c = ~a;             /*-61 = 1100 0011 */
    printf("Line 4 - Value of c is %d\n", c);

    c = a << 2;         /* 240 = 1111 0000 */
    printf("Line 5 - Value of c is %d\n", c);

    c = a >> 2;         /* 15 = 0000 1111 */
    printf("Line 6 - Value of c is %d\n", c);
}
```

Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15

Example program for bit wise operators in C

```
#include <stdio.h>
int main()
{
    int m=40,n=80,AND_opr,OR_opr,XOR_opr,NOT_opr ;

    AND_opr = (m&n);
    OR_opr = (m|n);
    NOT_opr = (~m);
    XOR_opr = (m^n);

    printf("AND_opr value = %d\n",AND_opr );
    printf("OR_opr value = %d\n",OR_opr );
    printf("NOT_opr value = %d\n",NOT_opr );
    printf("XOR_opr value = %d\n",XOR_opr );
    printf("left_shift value = %d\n", m << 1);
    printf("right_shift value = %d\n", m >> 1);
}
```

Output:
AND_opr value = 0
OR_opr value = 120
NOT_opr value = -41
XOR_opr value = 120
left_shift value = 80
right_shift value = 20

Summary Bit wise Operators

- ① To help manipulate hardware oriented data—individual bits rather than bytes a set of bitwise operators are used.
- ② The bitwise operators include operators like one's complement, right-shift, left-shift, bitwise AND, OR, and XOR.
- ③ The one's complement converts all zeros in its operand to 1s and all 1s to 0s.
- ④ The right-shift and left-shift operators are useful in eliminating bits from a number—either from the left or from the right.
- ⑤ The bitwise AND operator is useful in testing whether a bit is on/off and in putting off a particular bit.
- ⑥ The bitwise OR operator is used to turn on a particular bit.
- ⑦ The XOR operator works almost same as the OR operator except one minor variation.



Operators And Expression: Precedence And Associativity Of Operators

Precedence And Associativity Of Operators

■ Precedence of operators

- If more than one operators are involved in an expression then, C has predefined rule of priority of operators. This rule of priority of operators is called **operator precedence**.
- In C, precedence of arithmetic operators(*,%,/,+,-) is higher than relational operators(==,!>,<,>=,<=) and precedence of relational operator is higher than logical operators(&&, || and !). Suppose an expression:

```
(a>b+c&&d)
```

This expression is equivalent to:

```
((a>(b+c))&&d)
```

i.e., (b+c) executes first

then, (a>(b+c)) executes

then, (a>(b+c))&&d) executes

Precedence And Associativity Of Operators

Associativity of operators

Associativity indicates in which order two operators of same precedence(priority) executes. Let us suppose an expression:

a==b !=c

Here, operators == and != have same precedence. The associativity of both == and != is left to right, i.e., the expression in left is executed first and execution take pale towards right. Thus, a==b!=c equivalent to :

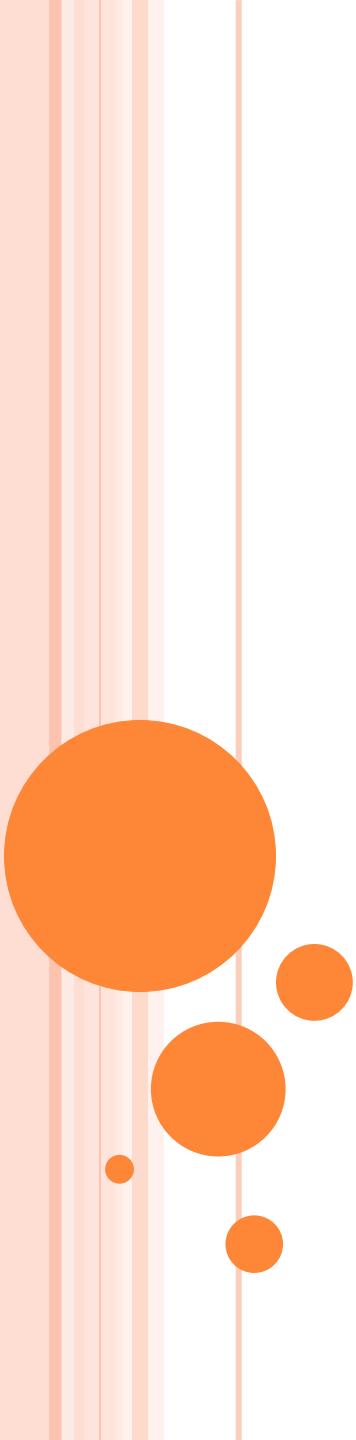
(a==b) !=c

The expression a=b=c is parsed as a=(b=c), and not as (a=b)=c because of right-to-left associativity.

Precedence	Operator	Description	Associativity
1	()	Function call	Left-to-Right
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(type)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10		Bitwise OR (inclusive or)	
11	&&	Logical AND	
12		Logical OR	
13	? :	Ternary conditional	Right-to-Left
14	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= =	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

The following table lists the precedence and associativity of C operators.

Operators are listed top to bottom, in descending precedence.



C - DECISION MAKING CONTROL STATEMENT AND BRANCHING

C - DECISION MAKING

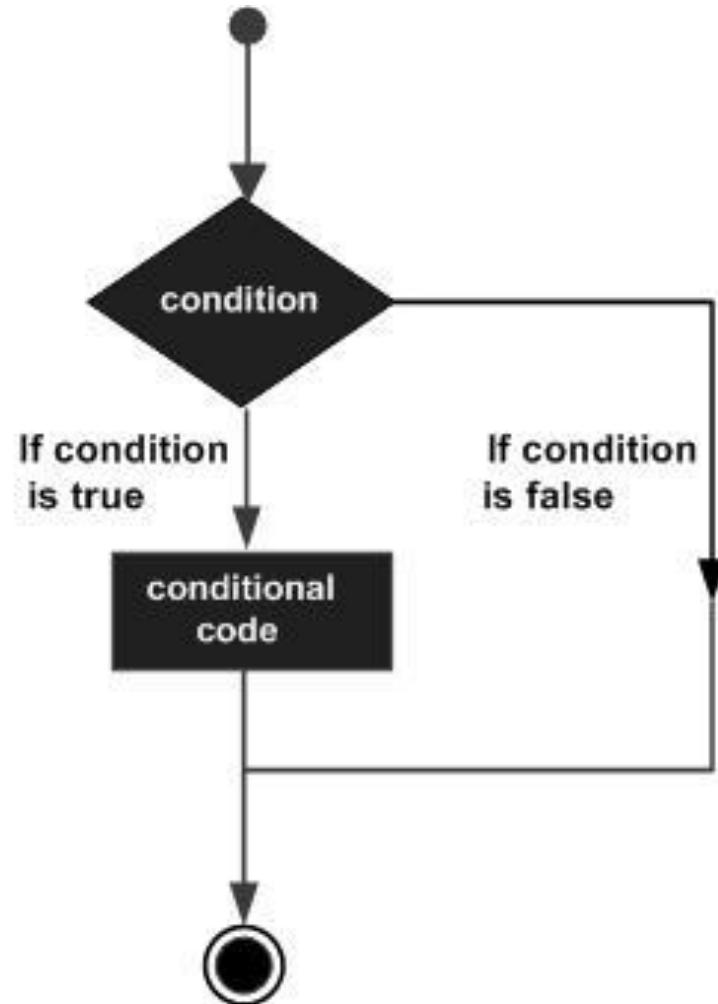
- Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.



C - DECISION MAKING CONT.

Following is the general form of a typical decision making structure found in most of the programming languages:

C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.



TYPES OF CONTROL STATEMENTS

C language provides following types of decision making statements.
Click the following links to check their detail.

Statement	Description
1. if statement	An if statement consists of a Boolean expression followed by one or more statements.
2. if...else statement	An if statement can be followed by an optional else statement , which executes when the Boolean expression is false.
3. nested if statements	You can use one if or else if statement inside another if or else if statement(s).
4. if...else if..else statement	An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement.
5. switch case -statement	A switch statement allows a variable to be tested for equality against a list of values.
6. nested switch statements	You can use one switch statement inside another switch statement(s).
7. Ternary operator ? :	? : can be used to replace if...else statements
8. goto statement	The goto statement transfers control to a label.

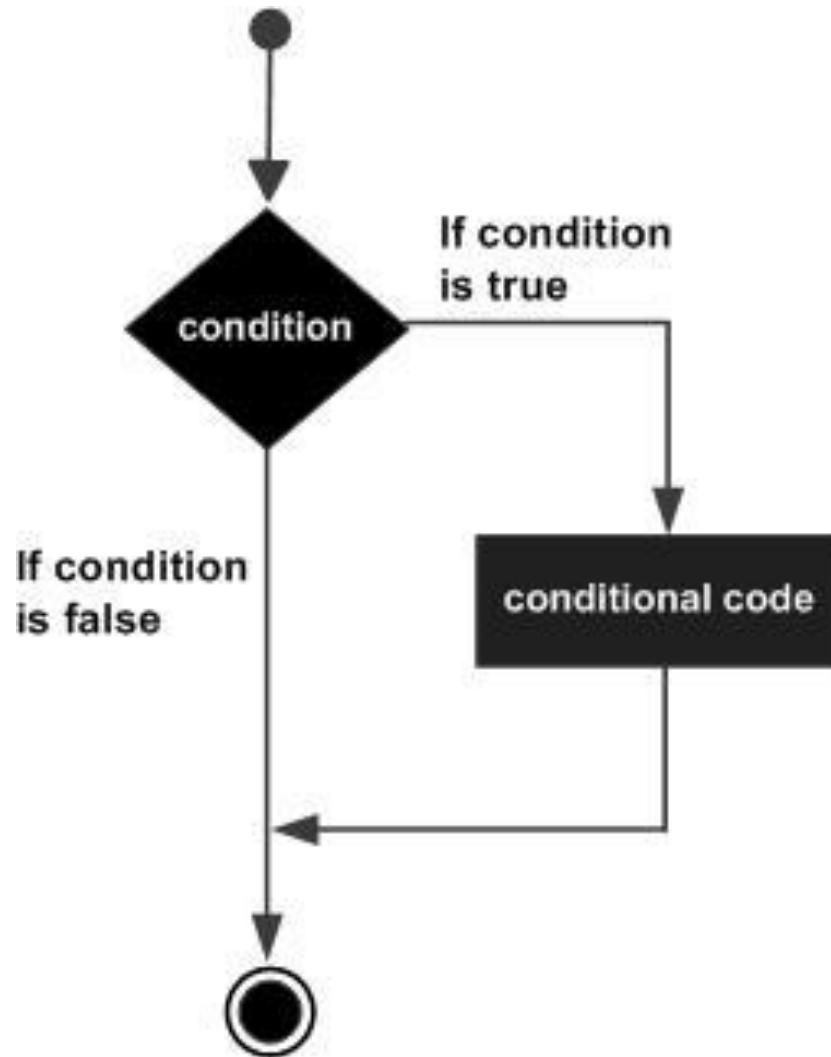
1. IF STATEMENT

- An **if** statement consists of a Boolean expression followed by one or more statements.
- **Syntax:**

```
if(boolean_expression)
{
    /* statement(s) will execute if the boolean expression is true */
}
```

- If the expression evaluates to **true**, then the block of code inside the if statement will be executed.
- If expression evaluates to **false**, then the first set of code after the end of the if statement will be executed.
- C language assumes any **non-zero** and **non-null** values as **true** and if it is either zero or null, then it is assumed as **false** value.

IF FLOW DIAGRAM



CODE EXAMPLE

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 10;

    /* check the boolean condition using if statement */
    if( a < 20 )
    {
        /* if condition is true then print the following */
        printf("a is less than 20\n");
    }
    printf("value of a is : %d\n", a);

    return 0;
}
```

```
a is less than 20;
value of a is : 10
```

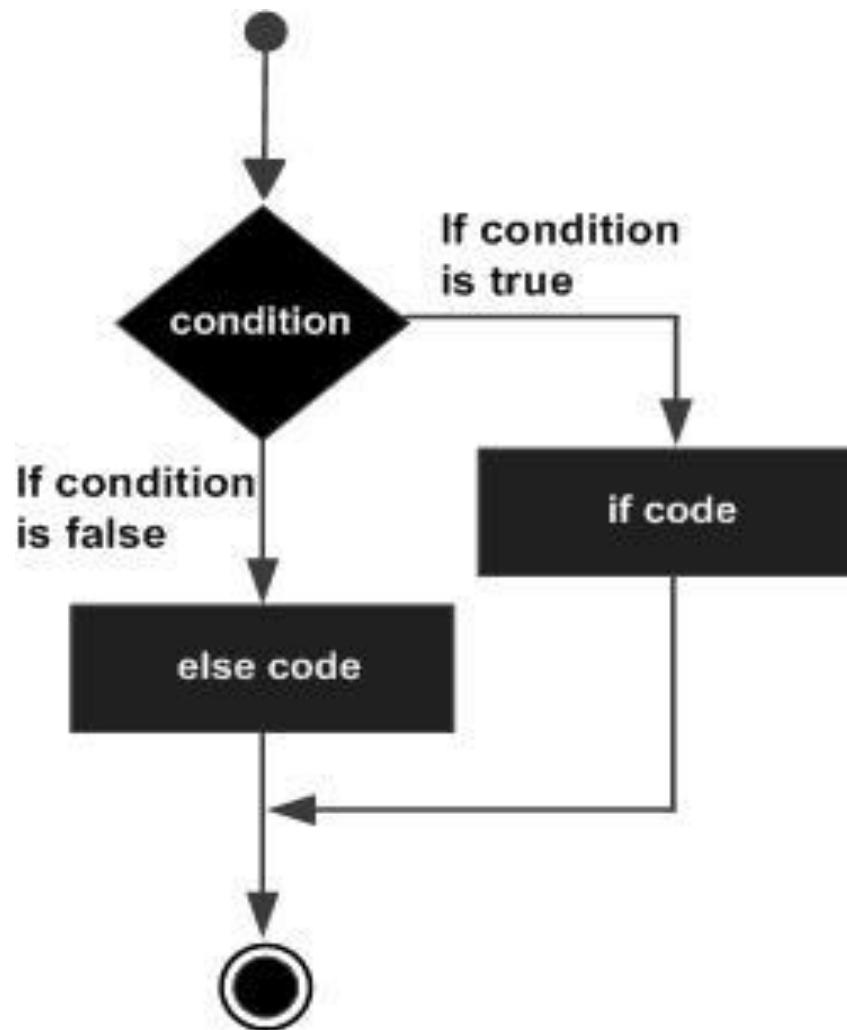
2. IF...ELSE STATEMENT

- An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.
- **Syntax:**

```
if(boolean_expression)
{
    /* statement(s) will execute if the boolean expression is true */
}
else
{
    /* statement(s) will execute if the boolean expression is false */
}
```

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

IF...ELSE FLOW DIAGRAM



CODE EXAMPLE : IF ELSE

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 100;

    /* check the boolean condition */
    if( a < 20 )
    {
        /* if condition is true then print the following */
        printf("a is less than 20\n" );
    }
    else
    {
        /* if condition is false then print the following */
        printf("a is not less than 20\n" );
    }
    printf("value of a is : %d\n", a);

    return 0;
}
```

```
a is not less than 20;
value of a is : 100
```

3. THE IF...ELSE IF...ELSE STATEMENT

- An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.
- When using if , else if , else statements there are few points to keep in mind:
 - An if can have zero or one else's and it must come after any else if's.
 - An if can have zero to many else if's and they must come before the else.
 - Once an else if succeeds, none of the remaining else if's or else's will be tested.



THE IF...ELSE IF...ELSE SYNTAX.

```
if(boolean_expression 1)
{
    /* Executes when the boolean expression 1 is true */
}
else if( boolean_expression 2)
{
    /* Executes when the boolean expression 2 is true */
}
else if( boolean_expression 3)
{
    /* Executes when the boolean expression 3 is true */
}
else
{
    /* executes when the none of the above condition is true */
}
```

CODE EXAMPLE: IF...ELSE IF...ELSE

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 100;

    /* check the boolean condition */
    if( a == 10 )
    {
        /* if condition is true then print the following */
        printf("Value of a is 10\n" );
    }
    else if( a == 20 )
    {
        /* if else if condition is true */
        printf("Value of a is 20\n" );
    }
    else if( a == 30 )
    {
        /* if else if condition is true */
        printf("Value of a is 30\n" );
    }
    else
    {
        /* if none of the conditions is true */
        printf("None of the values is matching\n" );
    }
    printf("Exact value of a is: %d\n", a );

    return 0;
}
```



4. C - NESTED IF STATEMENTS

- It is always legal in C to **nest** if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).
- Syntax:

```
if( boolean_expression 1)
{
    /* Executes when the boolean expression 1 is true */
    if(boolean_expression 2)
    {
        /* Executes when the boolean expression 2 is true */
    }
}
```

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

CODE EXAMPLE: NEST ELSE IF...ELSE

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 100;
    int b = 200;

    /* check the boolean condition */
    if( a == 100 )
    {
        /* if condition is true then check the following */
        if( b == 200 )
        {
            /* if condition is true then print the following */
            printf("Value of a is 100 and b is 200\n");
        }
    }
    printf("Exact value of a is : %d\n", a );
    printf("Exact value of b is : %d\n", b );

    return 0;
}
```

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```



5. C - SWITCH STATEMENT

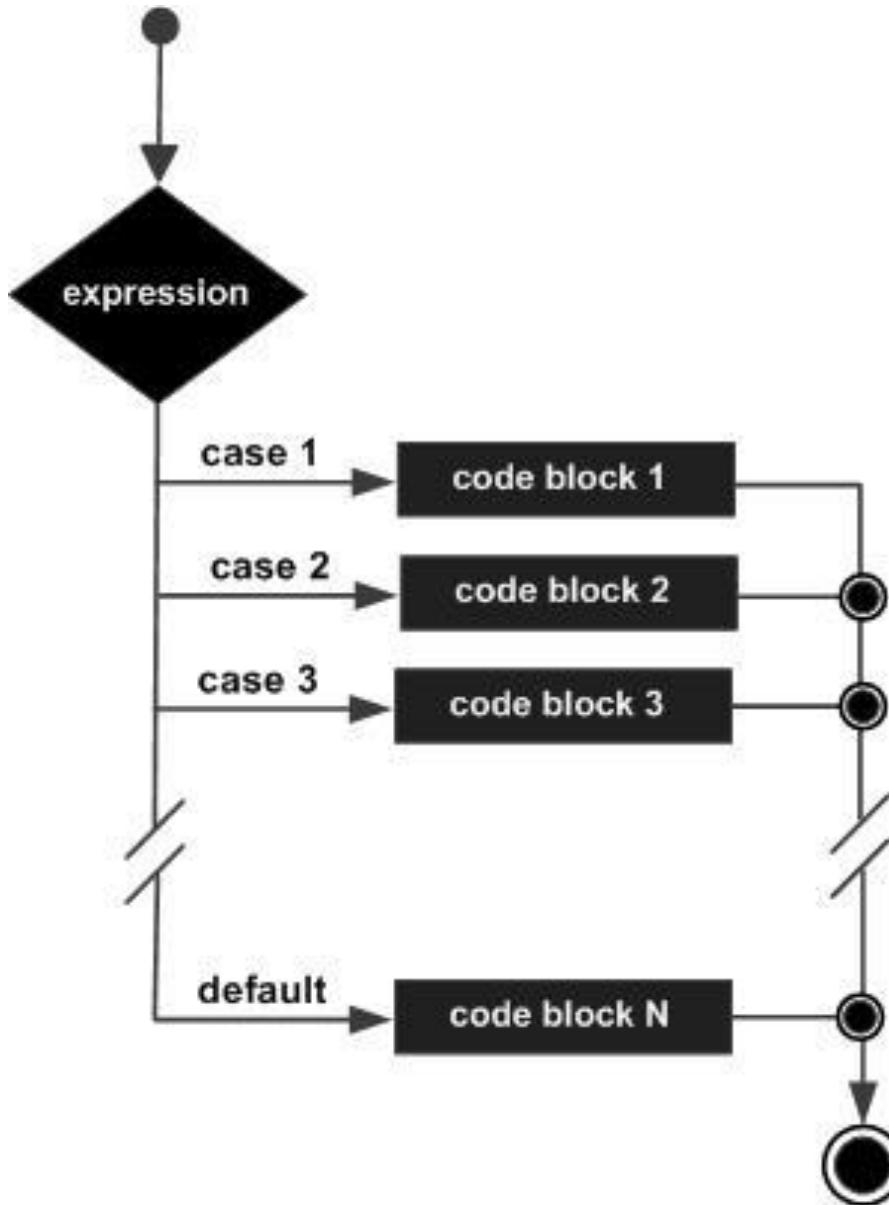
- A **switch** statement allows a variable to be tested for equality against a list of values.
- Each value is called a case, and the variable being switched on is checked for each **switch case**.
- Syntax:

```
switch(expression) {  
    case constant-expression :  
        statement(s);  
        break; /* optional */  
    case constant-expression :  
        statement(s);  
        break; /* optional */  
  
    /* you can have any number of case statements */  
    default : /* Optional */  
        statement(s);  
}
```

RULES APPLY TO A SWITCH STATEMENT:

- The **expression** used in a **switch** statement must have an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
- When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a **break**. If no **break** appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch.
- The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

SWITCH-CASE FLOW DIAGRAM



CODE EXAMPLE: SWITCH-CASE

```
int main ()
{
    /* local variable definition */
    char grade = 'B';

    switch(grade)
    {
        case 'A' :
            printf("Excellent!\n" );
            break;
        case 'B' :
        case 'C' :
            printf("Well done\n" );
            break;
        case 'D' :
            printf("You passed\n" );
            break;
        case 'F' :
            printf("Better try again\n" );
            break;
        default :
            printf("Invalid grade\n" );
    }
    printf("Your grade is %c\n", grade );

    return 0;
}
```



6. C - NESTED SWITCH STATEMENTS

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

```
switch(ch1) {  
    case 'A':  
        printf("This A is part of outer switch" );  
        switch(ch2) {  
            case 'A':  
                printf("This A is part of inner switch" );  
                break;  
            case 'B': /* case code */  
        }  
        break;  
    case 'B': /* case code */  
}
```

CODE EXAMPLE: NESTED SWITCH-CASE

```
int main ()
{
    /* local variable definition */
    int a = 100;
    int b = 200;

    switch(a) {
        case 100:
            printf("This is part of outer switch\n", a );
            switch(b) {
                case 200:
                    printf("This is part of inner switch\n", a );
                }
        }
    printf("Exact value of a is : %d\n", a );
    printf("Exact value of b is : %d\n", b );

    return 0;
}
```

This is part of outer switch
This is part of inner switch
Exact value of a is : 100
Exact value of b is : 200



7. THE ? : OPERATOR

- We have covered **conditional operator** ?: previously which can be used to replace **if...else** statements. It has the following general form:

```
Exp1 ? Exp2 : Exp3;
```

- Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.
- The ?: is called a ternary operator because it requires three operands.
- The value of a ? expression is determined like this:
 - Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression.
 - If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

THE ?: OPERATOR CONT.

- ?: operator can be used to replace if-else statements, which have the following form:

```
if(condition) {  
    var = X;  
}else{  
    var = Y;  
}
```

- For example, consider the following code:

```
if(y < 10){  
    var = 30;  
}else{  
    var = 40;  
}
```

- Above code can be rewritten like this:

```
var = (y < 10) ? 30 : 40;
```

Here, x is assigned the value of 30 if y is less than 10 and 40 if it is not. You can try the following example:

8. THE GOTO STATEMENT

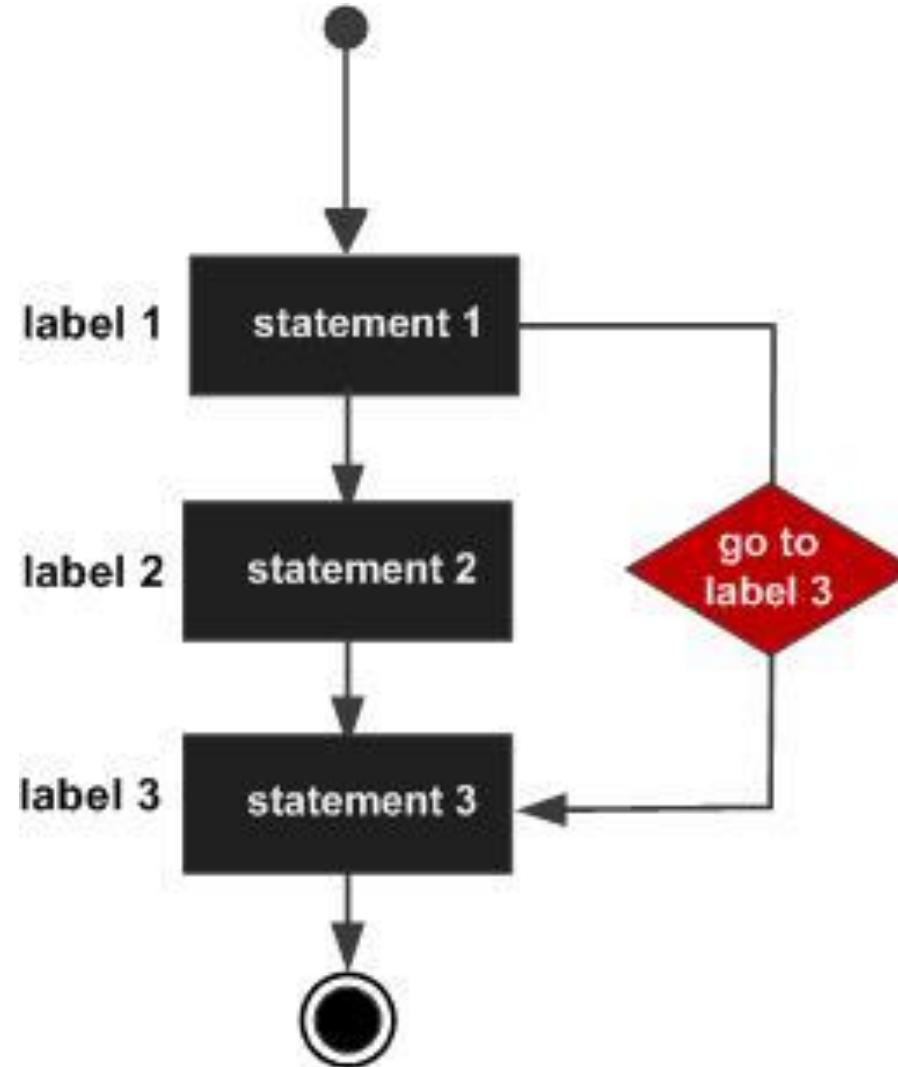
- A **goto** statement in C language provides an unconditional jump from the goto to a labeled statement in the same function.
- The given label must reside in the same function.
- **Syntax:**

```
goto label;  
...  
label: statement;
```

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

NOTE: Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten so that it doesn't need the goto.

GOTO STATEMENT FLOW DIAGRAM



CODE EXAMPLE: GOTO STATEMENT

```
int main ()
{
    /* local variable definition */
    int a = 10;

    /* do loop execution */
LOOP:do
{
    if( a == 15)
    {
        /* skip the iteration */
        a = a + 1;
        goto LOOP;
    }
    printf("value of a: %d\n", a);
    a++;

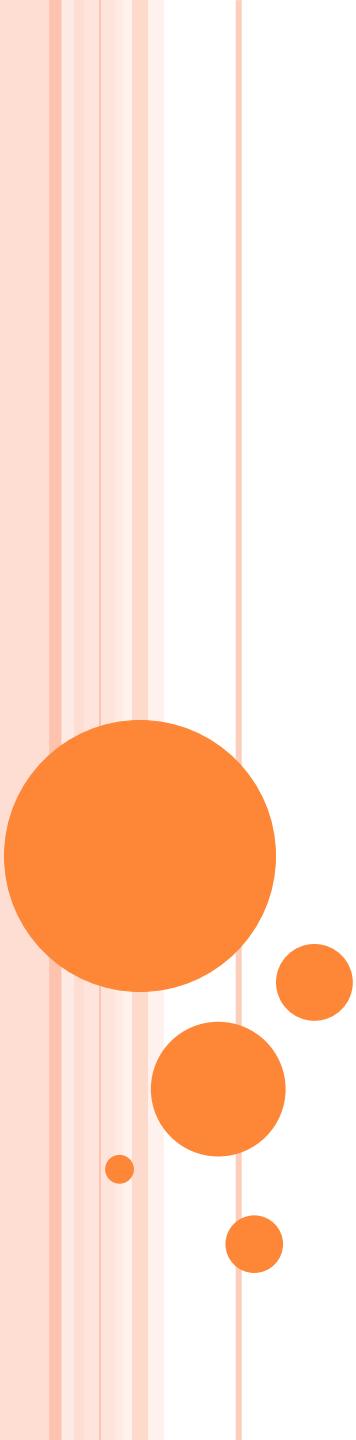
}while( a < 20 );

return 0;
}
```

Output:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```





C - DECISION MAKING CONTROL STATEMENT AND BRANCHING

C - DECISION MAKING

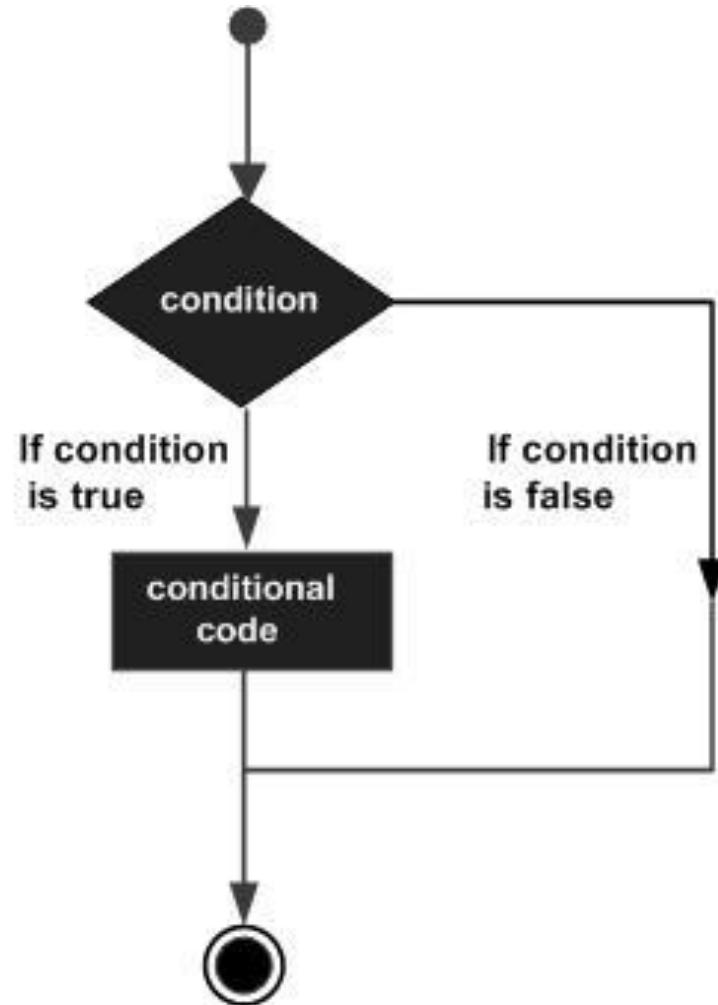
- Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.



C - DECISION MAKING CONT.

Following is the general form of a typical decision making structure found in most of the programming languages:

C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.



TYPES OF CONTROL STATEMENTS

C language provides following types of decision making statements.
Click the following links to check their detail.

Statement	Description
1. if statement	An if statement consists of a Boolean expression followed by one or more statements.
2. if...else statement	An if statement can be followed by an optional else statement , which executes when the Boolean expression is false.
3. nested if statements	You can use one if or else if statement inside another if or else if statement(s).
4. if...else if..else statement	An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement.
5. switch case -statement	A switch statement allows a variable to be tested for equality against a list of values.
6. nested switch statements	You can use one switch statement inside another switch statement(s).
7. Ternary operator ? :	? : can be used to replace if...else statements
8. goto statement	The goto statement transfers control to a label.

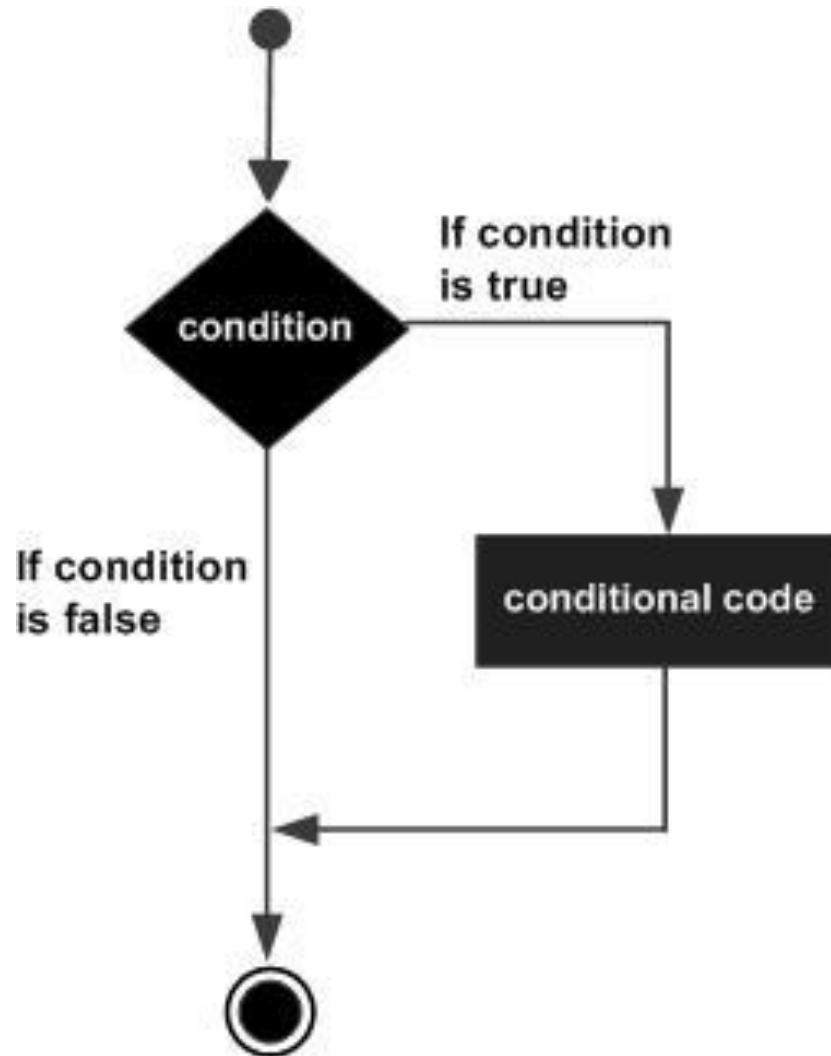
1. IF STATEMENT

- An **if** statement consists of a Boolean expression followed by one or more statements.
- **Syntax:**

```
if(boolean_expression)
{
    /* statement(s) will execute if the boolean expression is true */
}
```

- If the expression evaluates to **true**, then the block of code inside the if statement will be executed.
- If expression evaluates to **false**, then the first set of code after the end of the if statement will be executed.
- C language assumes any **non-zero** and **non-null** values as **true** and if it is either zero or null, then it is assumed as **false** value.

IF FLOW DIAGRAM



CODE EXAMPLE

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 10;

    /* check the boolean condition using if statement */
    if( a < 20 )
    {
        /* if condition is true then print the following */
        printf("a is less than 20\n");
    }
    printf("value of a is : %d\n", a);

    return 0;
}
```

```
a is less than 20;
value of a is : 10
```

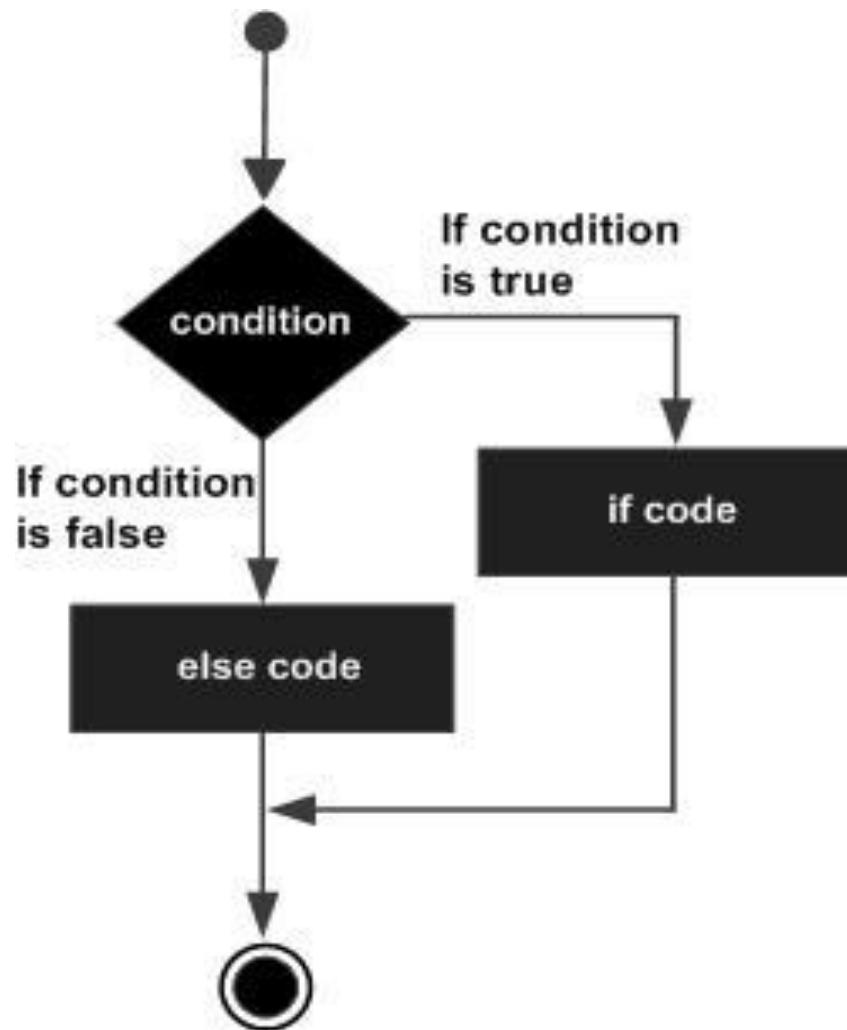
2. IF...ELSE STATEMENT

- An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.
- **Syntax:**

```
if(boolean_expression)
{
    /* statement(s) will execute if the boolean expression is true */
}
else
{
    /* statement(s) will execute if the boolean expression is false */
}
```

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

IF...ELSE FLOW DIAGRAM



CODE EXAMPLE : IF ELSE

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 100;

    /* check the boolean condition */
    if( a < 20 )
    {
        /* if condition is true then print the following */
        printf("a is less than 20\n" );
    }
    else
    {
        /* if condition is false then print the following */
        printf("a is not less than 20\n" );
    }
    printf("value of a is : %d\n", a);

    return 0;
}
```

```
a is not less than 20;
value of a is : 100
```

3. THE IF...ELSE IF...ELSE STATEMENT

- An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.
- When using if , else if , else statements there are few points to keep in mind:
 - An if can have zero or one else's and it must come after any else if's.
 - An if can have zero to many else if's and they must come before the else.
 - Once an else if succeeds, none of the remaining else if's or else's will be tested.



THE IF...ELSE IF...ELSE SYNTAX.

```
if(boolean_expression 1)
{
    /* Executes when the boolean expression 1 is true */
}
else if( boolean_expression 2)
{
    /* Executes when the boolean expression 2 is true */
}
else if( boolean_expression 3)
{
    /* Executes when the boolean expression 3 is true */
}
else
{
    /* executes when the none of the above condition is true */
}
```

CODE EXAMPLE: IF...ELSE IF...ELSE

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 100;

    /* check the boolean condition */
    if( a == 10 )
    {
        /* if condition is true then print the following */
        printf("Value of a is 10\n" );
    }
    else if( a == 20 )
    {
        /* if else if condition is true */
        printf("Value of a is 20\n" );
    }
    else if( a == 30 )
    {
        /* if else if condition is true */
        printf("Value of a is 30\n" );
    }
    else
    {
        /* if none of the conditions is true */
        printf("None of the values is matching\n" );
    }
    printf("Exact value of a is: %d\n", a );

    return 0;
}
```



4. C - NESTED IF STATEMENTS

- It is always legal in C to **nest** if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).
- Syntax:

```
if( boolean_expression 1)
{
    /* Executes when the boolean expression 1 is true */
    if(boolean_expression 2)
    {
        /* Executes when the boolean expression 2 is true */
    }
}
```

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

CODE EXAMPLE: NEST ELSE IF...ELSE

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 100;
    int b = 200;

    /* check the boolean condition */
    if( a == 100 )
    {
        /* if condition is true then check the following */
        if( b == 200 )
        {
            /* if condition is true then print the following */
            printf("Value of a is 100 and b is 200\n");
        }
    }
    printf("Exact value of a is : %d\n", a );
    printf("Exact value of b is : %d\n", b );

    return 0;
}
```

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```



5. C - SWITCH STATEMENT

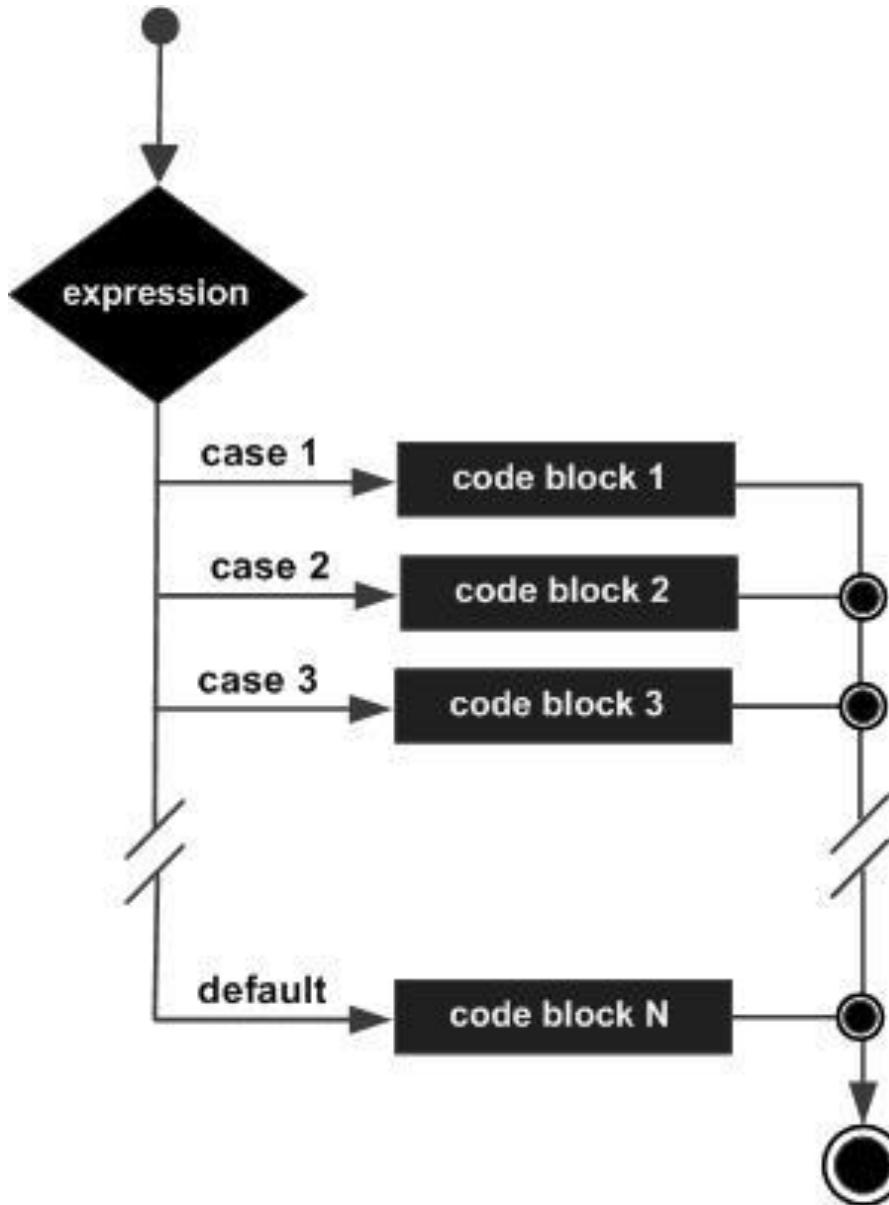
- A **switch** statement allows a variable to be tested for equality against a list of values.
- Each value is called a case, and the variable being switched on is checked for each **switch case**.
- Syntax:

```
switch(expression) {  
    case constant-expression :  
        statement(s);  
        break; /* optional */  
    case constant-expression :  
        statement(s);  
        break; /* optional */  
  
    /* you can have any number of case statements */  
    default : /* Optional */  
        statement(s);  
}
```

RULES APPLY TO A SWITCH STATEMENT:

- The **expression** used in a **switch** statement must have an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
- When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a **break**. If no **break** appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch.
- The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

SWITCH-CASE FLOW DIAGRAM



CODE EXAMPLE: SWITCH-CASE

```
int main ()
{
    /* local variable definition */
    char grade = 'B';

    switch(grade)
    {
        case 'A' :
            printf("Excellent!\n" );
            break;
        case 'B' :
        case 'C' :
            printf("Well done\n" );
            break;
        case 'D' :
            printf("You passed\n" );
            break;
        case 'F' :
            printf("Better try again\n" );
            break;
        default :
            printf("Invalid grade\n" );
    }
    printf("Your grade is %c\n", grade );

    return 0;
}
```



6. C - NESTED SWITCH STATEMENTS

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

```
switch(ch1) {  
    case 'A':  
        printf("This A is part of outer switch" );  
        switch(ch2) {  
            case 'A':  
                printf("This A is part of inner switch" );  
                break;  
            case 'B': /* case code */  
        }  
        break;  
    case 'B': /* case code */  
}
```

CODE EXAMPLE: NESTED SWITCH-CASE

```
int main ()
{
    /* local variable definition */
    int a = 100;
    int b = 200;

    switch(a) {
        case 100:
            printf("This is part of outer switch\n", a );
            switch(b) {
                case 200:
                    printf("This is part of inner switch\n", a );
                }
        }
    printf("Exact value of a is : %d\n", a );
    printf("Exact value of b is : %d\n", b );

    return 0;
}
```

This is part of outer switch
This is part of inner switch
Exact value of a is : 100
Exact value of b is : 200



7. THE ? : OPERATOR

- We have covered **conditional operator** ?: previously which can be used to replace **if...else** statements. It has the following general form:

```
Exp1 ? Exp2 : Exp3;
```

- Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.
- The ?: is called a ternary operator because it requires three operands.
- The value of a ? expression is determined like this:
 - Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression.
 - If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

THE ?: OPERATOR CONT.

- ?: operator can be used to replace if-else statements, which have the following form:

```
if(condition) {  
    var = X;  
}else{  
    var = Y;  
}
```

- For example, consider the following code:

```
if(y < 10){  
    var = 30;  
}else{  
    var = 40;  
}
```

- Above code can be rewritten like this:

```
var = (y < 10) ? 30 : 40;
```

Here, x is assigned the value of 30 if y is less than 10 and 40 if it is not. You can try the following example:

8. THE GOTO STATEMENT

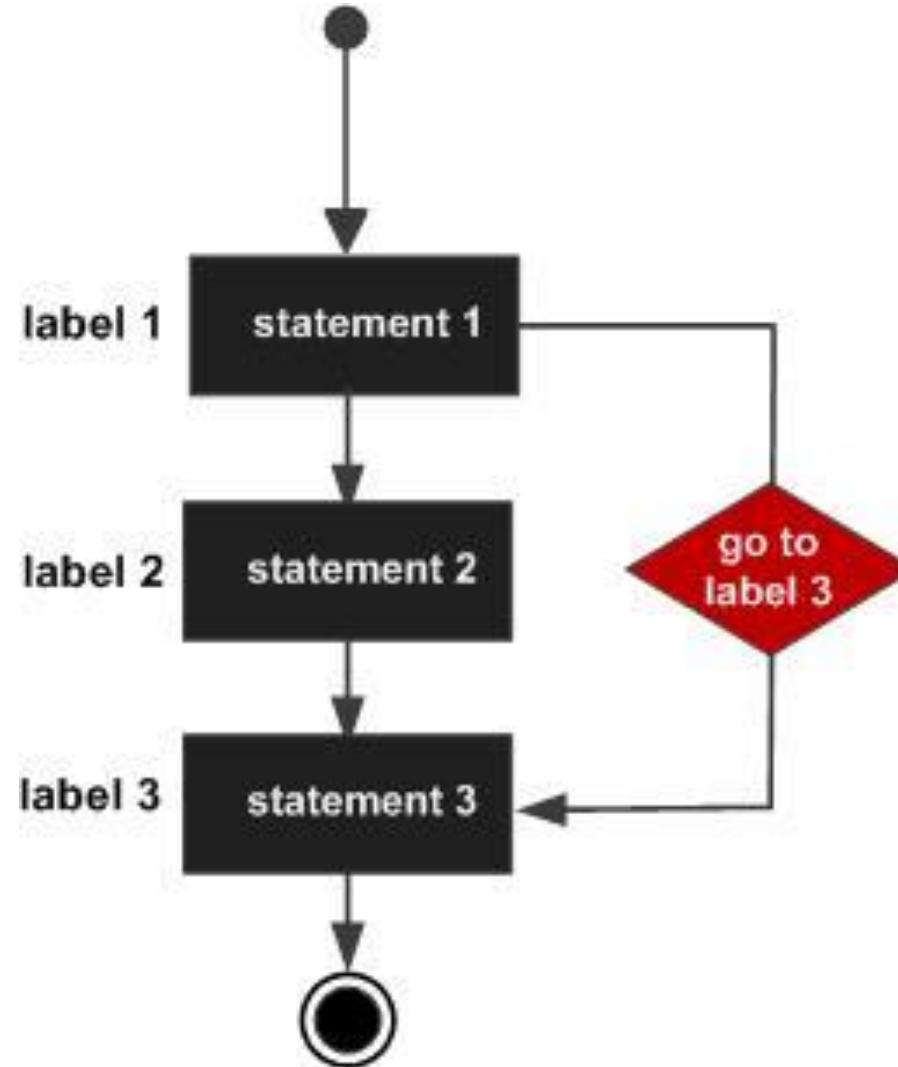
- A **goto** statement in C language provides an unconditional jump from the goto to a labeled statement in the same function.
- The given label must reside in the same function.
- **Syntax:**

```
goto label;  
...  
label: statement;
```

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

NOTE: Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten so that it doesn't need the goto.

GOTO STATEMENT FLOW DIAGRAM



CODE EXAMPLE: GOTO STATEMENT

```
int main ()
{
    /* local variable definition */
    int a = 10;

    /* do loop execution */
LOOP:do
{
    if( a == 15)
    {
        /* skip the iteration */
        a = a + 1;
        goto LOOP;
    }
    printf("value of a: %d\n", a);
    a++;

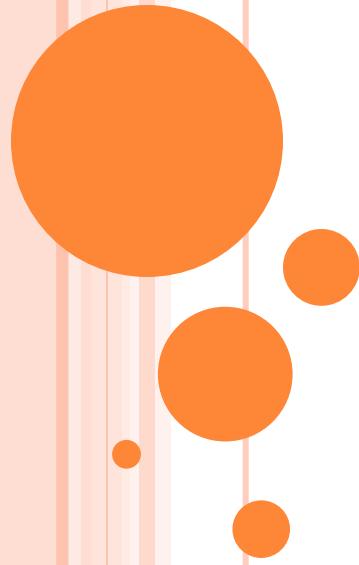
}while( a < 20 );

return 0;
}
```

Output:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

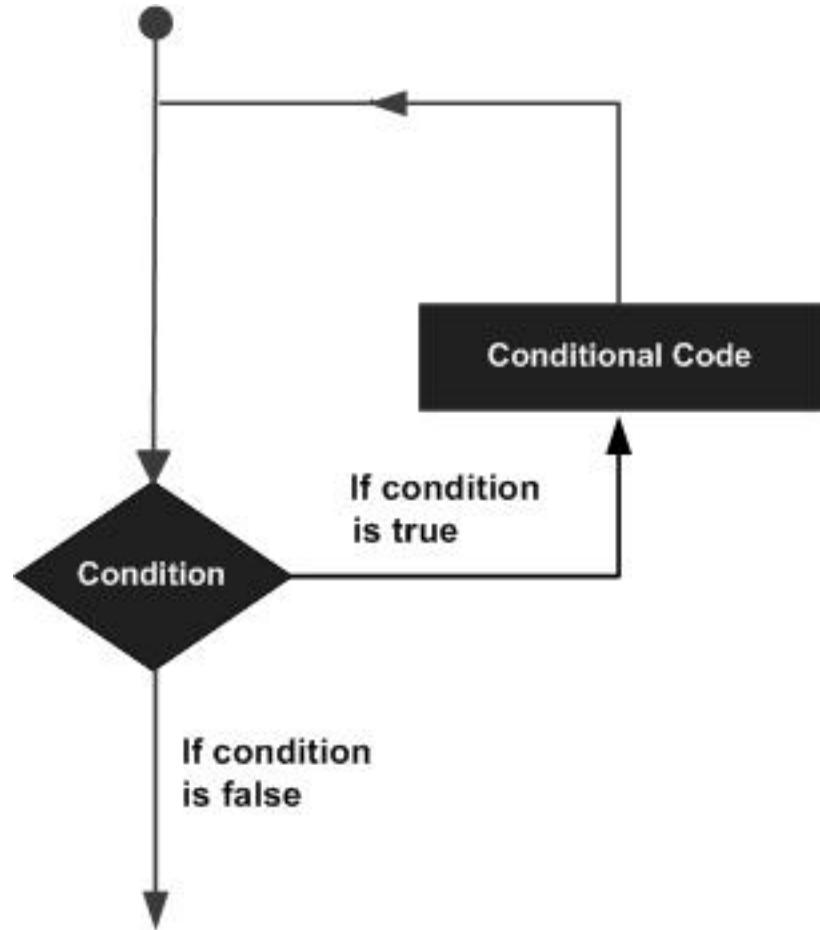




C – LOOP

C – LOOP

- There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.
- A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:



TYPES OF LOOP

C programming language provides the following types of loop to handle looping requirements.

Statement	Description
1. for loop	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
2. while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
3. do...while loop	Like a while statement, except that it tests the condition at the end of the loop body
4. nested loop	You can use one or more loop inside any another while, for or do..while loop.

FOR LOOP

- A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.
- **Syntax:**

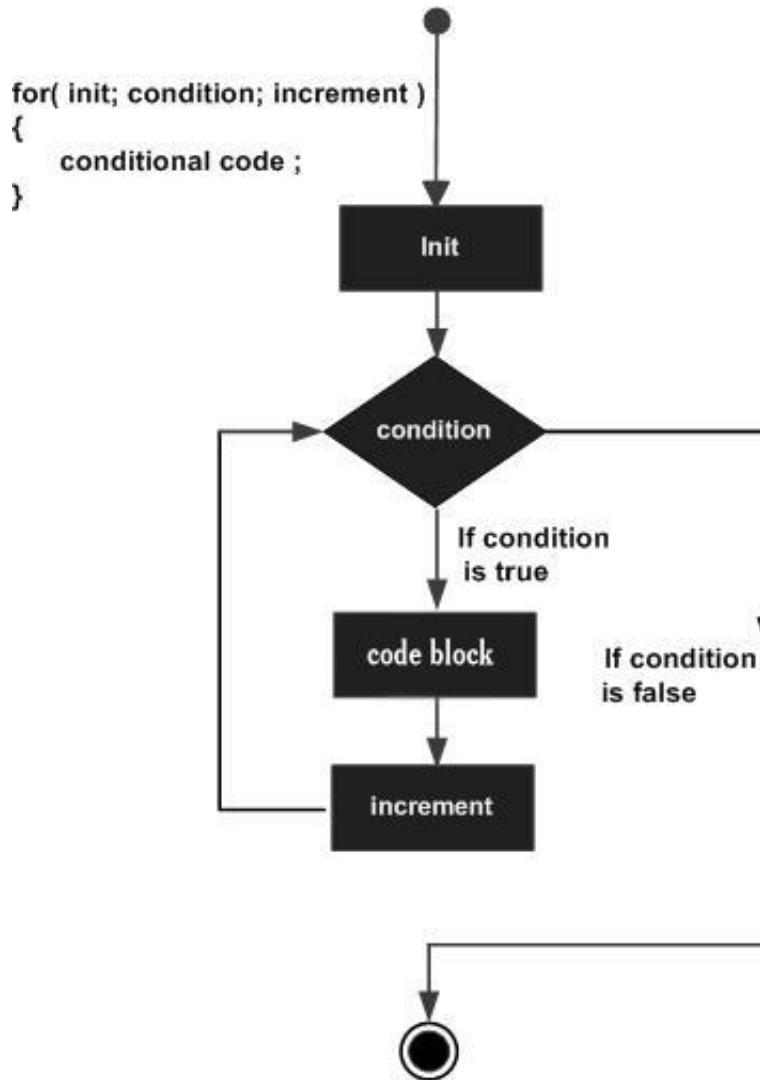
```
for ( init; condition; increment )  
{  
    statement(s);  
}
```



FLOW OF CONTROL IN A FOR LOOP

- The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

FLOW DIAGRAM : FOR LOOP



CODE EXAMPLE

```
#include <stdio.h>

int main ()
{
    /* for loop execution */
    for( int a = 10; a < 20; a = a + 1 )
    {
        printf("value of a: %d\n", a);
    }

    return 0;
}
```

Output:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

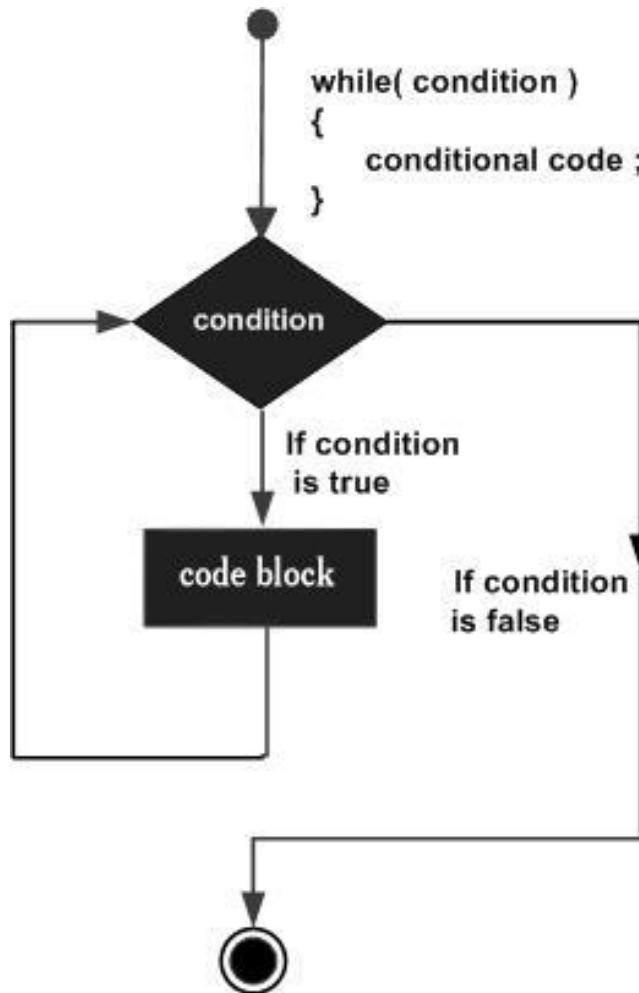
2. WHILE LOOP

- A **while** loop statement repeatedly executes a target statement as long as a given condition is true.
- **Syntax:**

```
while(condition)
{
    statement(s);
}
```

- Here, statement(s) may be a single statement or a block of statements.
- The condition may be any expression, and true is any nonzero value. The loop iterates while the condition is true.
- When the condition becomes false, program control passes to the line immediately following the loop.

FLOW DIAGRAM: WHILE LOOP



Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

CODE EXAMPLE : IF ELSE

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 10;

    /* while loop execution */
    while( a < 20 )
    {
        printf("value of a: %d\n", a);
        a++;
    }

    return 0;
}
```

Output:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

3. DO..WHILE LOOP

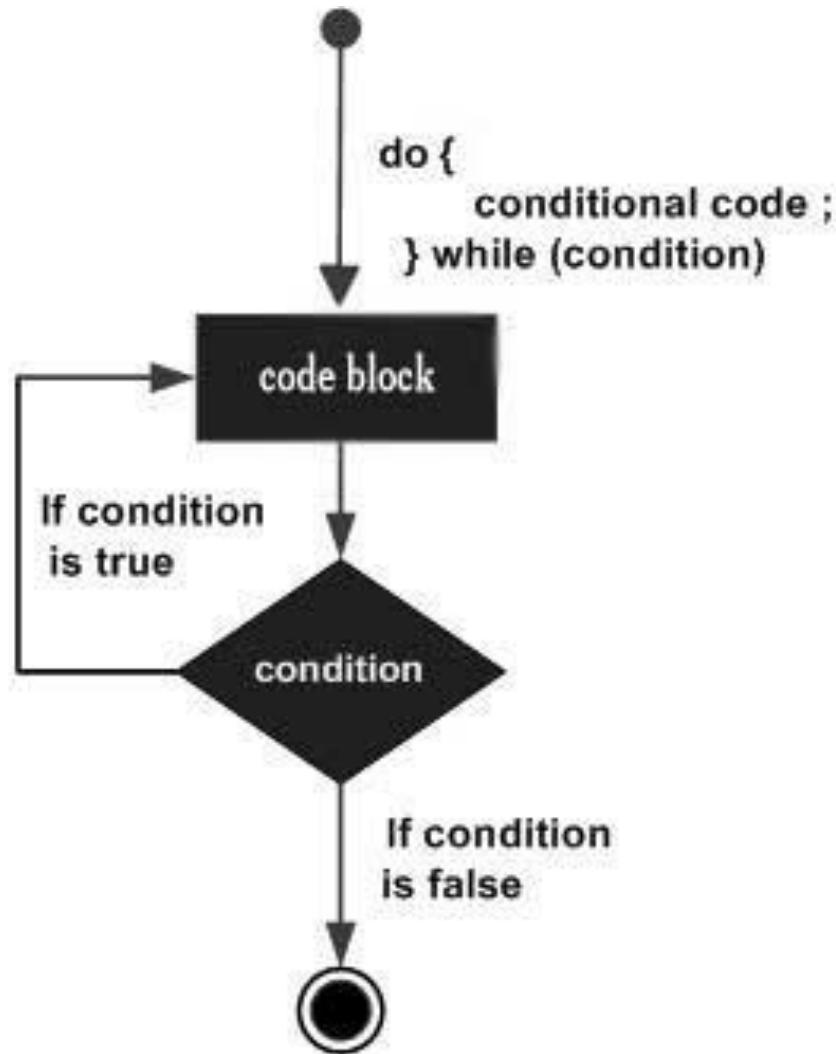
- Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop in checks its condition at the bottom of the loop.
- A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.
- Syntax:

```
do
{
    statement(s);

}while( condition );
```

- Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.
- If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

FLOW DIAGRAM: Do..WHILE LOOP



CODE EXAMPLE: Do...WHILE

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 10;

    /* do loop execution */
    do
    {
        printf("value of a: %d\n", a);
        a = a + 1;
    }while( a < 20 );

    return 0;
}
```

Output:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

NESTED LOOPS IN C

- It is allow to use one loop inside another loop.
- The syntax for a **nested for loop** statement

```
for ( init; condition; increment )
{
    for ( init; condition; increment )
    {
        statement(s);
    }
    statement(s);
}
```

- The syntax for a **nested while loop** statement

```
while(condition)
{
    while(condition)
    {
        statement(s);
    }
    statement(s);
}
```



NESTED LOOPS IN C

- The syntax for a **nested do...while loop**

```
do
{
    statement(s);
    do
    {
        statement(s);
    }while( condition );

}while( condition );
```

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



CODE EXAMPLE NESTED FOR LOOP

```
#include <stdio.h>

void main ()
{
    int i, j;

    for(i=0; i<5; i++) /*outer loop*/
    {
        printf("\nI is %d\n", i);
        for(j=0; j < 5; j++) /*inner loop*/
        {
            printf("J=%d, ", j);
        }
    }
}
```

I is 0
J=0, J=1, J=2, J=3, J=4,
I is 1
J=0, J=1, J=2, J=3, J=4,
I is 2
J=0, J=1, J=2, J=3, J=4,
I is 3
J=0, J=1, J=2, J=3, J=4,
I is 4
J=0, J=1, J=2, J=3, J=4,



CODE EXAMPLE NESTED FOR LOOP

```
#include <stdio.h>

void main ()
{
    int i, j;
    for(i=0; i<5; i++) /*outer loop*/
    {
        for(j=0; j<5; j++) /*inner loop*/
        {
            printf("*");
        }
        printf("\n");
    }
}
```

* * * * *

* * * * *

* * * * *

* * * * *

* * * * *



LOOP CONTROL STATEMENTS:

Loop control statements change execution from its normal sequence. C supports the following control statements.

Control Statement	Description
break statement	Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.



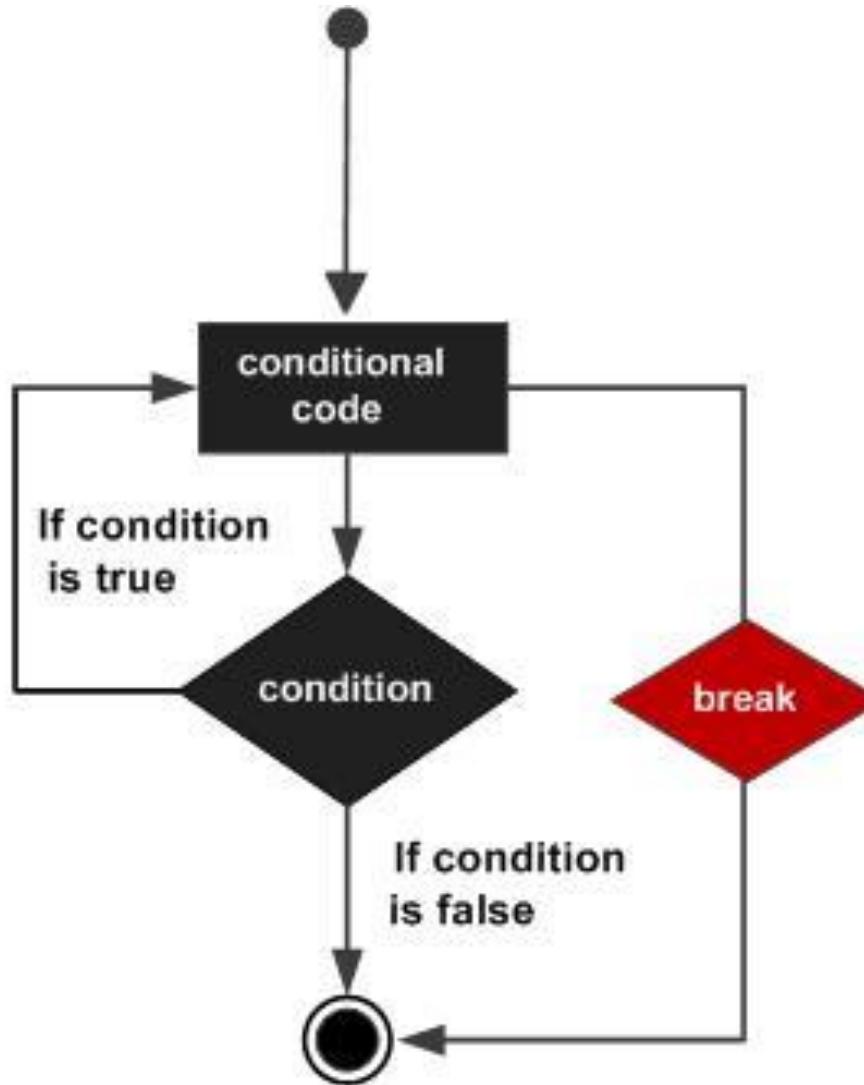
BREAK STATEMENT

- The **break** statement in C programming language has the following two usages:
 - When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
 - It can be used to terminate a case in the **switch** statement (we already saw).
- If you are using nested loops (i.e., one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.
- **Syntax:**

break;



FLOW DIAGRAM: BREAK STATEMENT



CODE EXAMPLE: BREAK

```
#include <stdio.h>

int main ()
{
    int i;
    for(i=0;i<6;i++)
    {
        if(i==3)
        {
            /*terminate the loop using break statement*/
            break;
        }
        printf("Value of i: %d\n", i);
    }
}
```

Value of i: 0
Value of i: 1
Value of i: 2

BREAK

```
while (test expression) {  
    statement/s  
    if (test expression) {  
        break;  
    }  
    statement/s  
}
```

```
do {  
    statement/s  
    if (test expression) {  
        break;  
    }  
    statement/s  
}  
while (test expression);
```

```
for (intial expression; test expression; update expression) {  
    statement/s  
    if (test expression) {  
        break;  
    }  
    statements/  
}
```

NOTE: The break statement may also be used inside body of else statement.

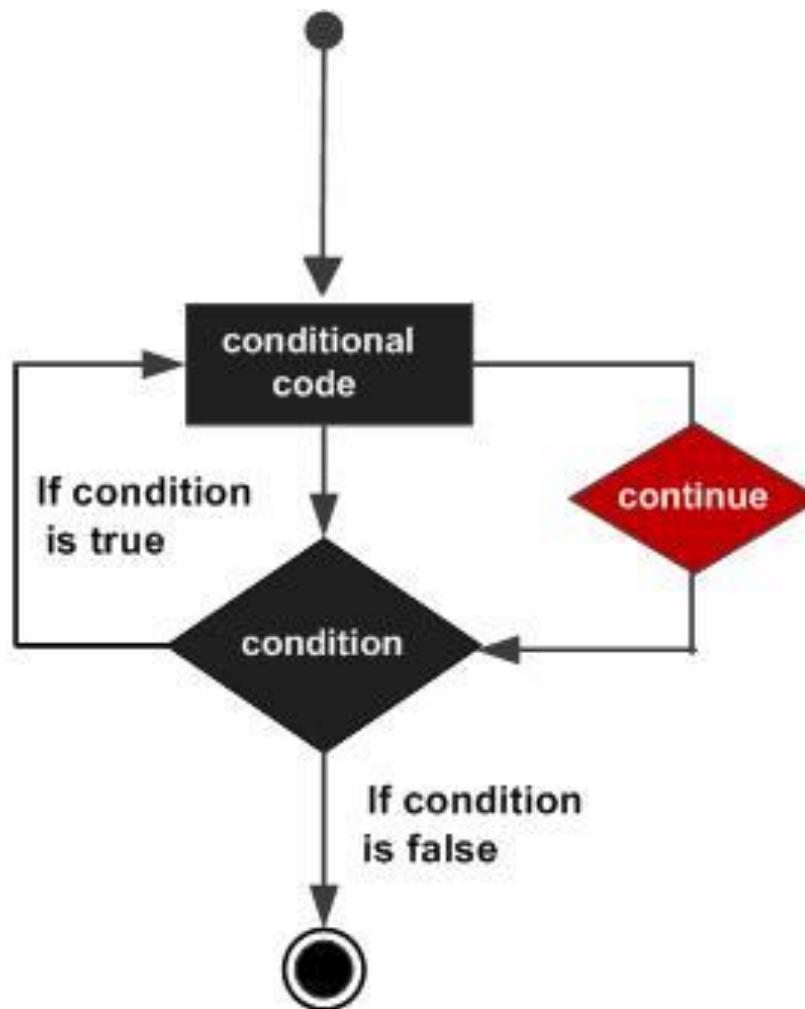
CONTINUE STATEMENT

- The **continue** statement works somewhat like the **break** statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between.
- For the **for** loop, **continue** statement causes the conditional test and increment portions of the loop to execute. For the **while** and **do...while** loops, **continue** statement causes the program control passes to the conditional tests.
- **Syntax:**

continue;



FLOW DIAGRAM: CONTINUE



CODE EXAMPLE: CONTINUE

```
#include <stdio.h>

int main ()
{
    int i;
    for(i=0;i<6;i++)
    {
        if( i == 3)
        {
            /* skip the iteration */
            continue;
        }
        printf("value of i: %d\n", i);
    }
    return 0;
}
```

value of i: 0
value of i: 1
value of i: 2
value of i: 4
value of i: 5

Notice
that 3 is
not there!

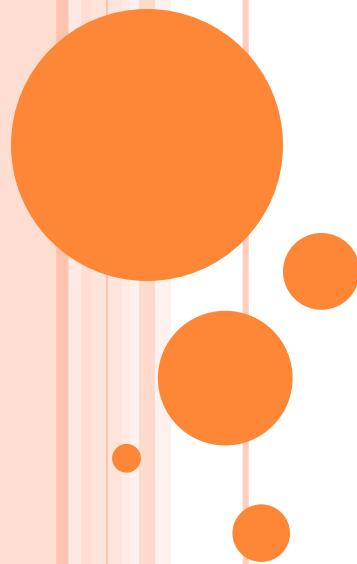
CONTINUE

```
→ while (test expression) {  
    statement/s  
    if (test expression) {  
        → continue;  
    }  
    statement/s  
}
```

```
do {  
    statement/s  
    if (test expression) {  
        → continue;  
    }  
    statement/s  
}  
→ while (test expression);
```

```
→ for (initial expression; test expression; update expression) {  
    statement/s  
    if (test expression) {  
        → continue;  
    }  
    statements/  
}
```

NOTE: The continue statement may also be used inside body of else statement.



C – ARRAY

C – ARRAY

- C programming language provides a data structure called **the array**, which can store a fixed-size sequential collection of elements of the same type.
- An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.
- Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.
- A specific element in an array is accessed by an **index**.



ARRAY

- All arrays consist of contiguous memory locations.
 - The lowest address corresponds to the first element and the highest address to the last element.

Array Name: a
Array Length: n

The diagram illustrates an array structure with n elements. The top row, labeled "Index:", shows numerical indices from 0 to $n-1$. The bottom row, labeled "Elements:", shows the corresponding array elements $a[0], a[1], a[2], a[3], \dots, a[n-1]$. Each element is enclosed in a box. Two arrows point upwards from the labels "First Element" and "Last Element" to the boxes containing $a[0]$ and $a[n-1]$, respectively.

DECLARING ARRAYS

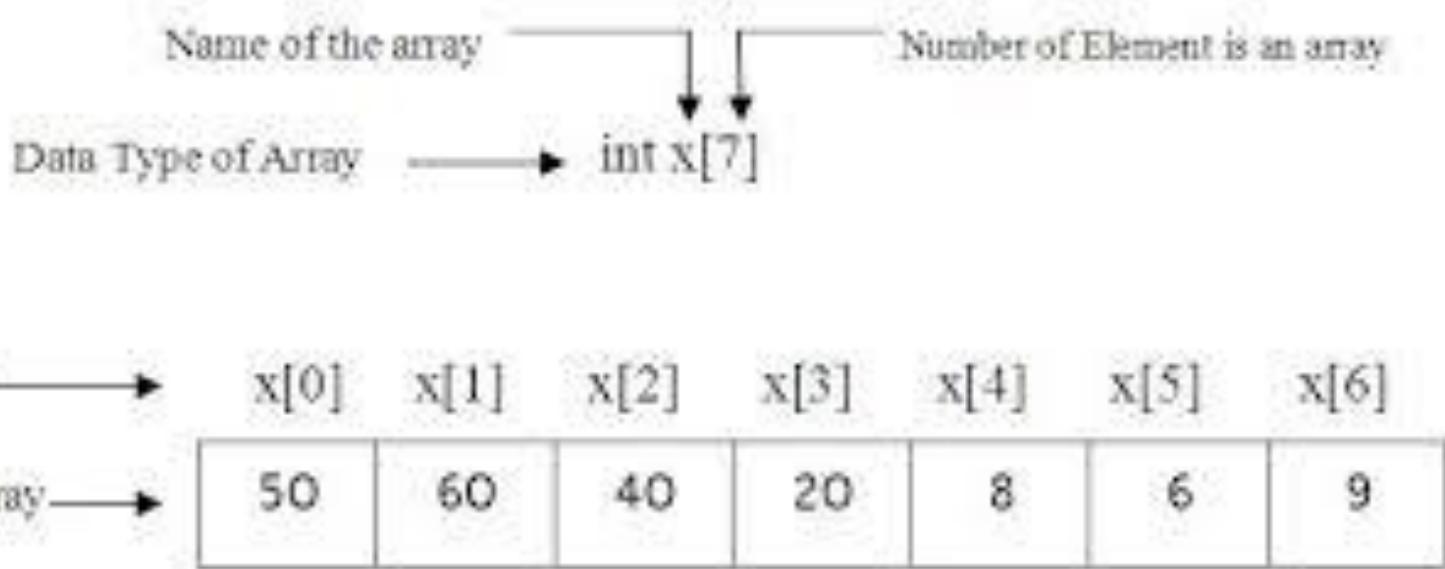
- To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows :

```
type arrayName [ arraySize ];
```

- This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and type can be any valid C data type.
- For example, to declare a 10-element array called balance of type double, use this statement:

```
double balance[10];
```

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



INITIALIZING ARRAYS

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

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

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

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

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



INITIALIZING ARRAYS CONT.

- Following is an example to assign a single element of the array:

```
balance[4] = 50.0;
```

- The above statement assigns element number 5th in the array with a value of 50.0.
- All arrays have 0 as the index of their first element which is also called base index and last index of an array will be total size of the array minus 1.
- Following is the pictorial representation of the same array we discussed above:

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0



ACCESSING ARRAY ELEMENTS

- An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array.
- For example:

```
double salary = balance[9];
```

- The above statement will take 10th element from the array and assign the value to salary variable.



CODE EXAMPLE: ARRAY

- Following is an example which will use all the above mentioned three concepts declaration, assignment and accessing arrays:

```
#include <stdio.h>

int main ()
{
    int n[ 10 ]; /* n is an array of 10 integers */
    int i,j;

    /* initialize elements of array n to 0 */
    for ( i = 0; i < 10; i++ )
    {
        n[ i ] = i + 100; /* set element at location i to i + 100 */
    }

    /* output each array element's value */
    for ( j = 0; j < 10; j++ )
    {
        printf("Element[%d] = %d\n", j, n[j] );
    }

    return 0;
}
```

Element[0]	=	100
Element[1]	=	101
Element[2]	=	102
Element[3]	=	103
Element[4]	=	104
Element[5]	=	105
Element[6]	=	106
Element[7]	=	107
Element[8]	=	108
Element[9]	=	109

TWO-DIMENSIONAL ARRAYS

- A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y you would write as follows:

```
type arrayName [ x ][ y ];
```

- A two-dimensional array can be think as a table which will have x number of rows and y number of columns.
- A 2-dimensional array **a**, which contains three rows and four columns can be shown as below:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

- Thus, every element in array a is identified by an element name of the form **a[i][j]**, where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

INITIALIZING TWO-DIMENSIONAL ARRAYS:

- Multidimensional arrays may be initialized by specifying bracketed values for each row.
- Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {  
    {0, 1, 2, 3}, /* initializers for row indexed by 0 */  
    {4, 5, 6, 7}, /* initializers for row indexed by 1 */  
    {8, 9, 10, 11} /* initializers for row indexed by 2 */  
};
```

- The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```



ACCESSING TWO-DIMENSIONAL ARRAY ELEMENTS

- An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

```
int val = a[2][3];
```

- The above statement will take 4th element from the 3rd row of the array. You can verify it in the above diagram.



ACCESSING TWO-DIMENSIONAL ARRAY ELEMENTS

- Let us check below program where we have used nested loop to handle a two dimensional array:

```
#include <stdio.h>

int main ()
{
    /* an array with 5 rows and 2 columns*/
    int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6}, {4,8} };
    int i, j;

    /* output each array element's value */
    for ( i = 0; i < 5; i++ )
    {
        for ( j = 0; j < 2; j++ )
        {
            printf("a[%d][%d] = %d\n", i,j, a[i][j] );
        }
    }
    return 0;
}
```

a[0][0]:	0
a[0][1]:	0
a[1][0]:	1
a[1][1]:	2
a[2][0]:	2
a[2][1]:	4
a[3][0]:	3
a[3][1]:	6
a[4][0]:	4
a[4][1]:	8

MULTI-DIMENSIONAL ARRAYS IN C

- C programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration:

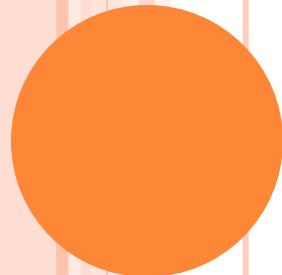
```
type name[size1][size2]...[sizeN];
```

- For example, the following declaration creates a three dimensional 5 . 10 . 4 integer array:

```
int threedim[5][10][4];
```

- As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.





C - FUNCTION

OUTLINE

- What is C function?
- Uses of C functions
- C function declaration, function call and definition with example program
- How to call C functions in a program?
 - Call by value
 - Call by reference
- C function arguments and return values
 - C function with arguments and with return value
 - C function with arguments and without return value
 - C function without arguments and without return value
 - C function without arguments and with return value
- Types of C functions
 - Library functions in C
 - User defined functions in C
 - Creating/Adding user defined function in C library
- Command line arguments in C
- Variable length arguments in C



WHAT IS C FUNCTION?

- A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**.
- You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task.
- A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.
- The C standard library provides numerous built-in functions that your program can call. For example, function **printf()** to print output in the console.
- A function is known with various names like a *method* or a *sub-routine* or a *procedure*, etc.

C - FUNCTIONS

- Most languages allow you to create functions of some sort.
- Functions are used to break up large programs into named sections.
- You have already been using a function which is the *main* function.
- Functions are often used when the same piece of code has to run multiple times.
- In this case you can put this piece of code in a function and give that function a name. When the piece of code is required you just have to call the function by its name. (So you only have to type the piece of code once).



C - FUNCTIONS

- In the example below we declare a function with the name MyPrint.
- The only thing that this function does is to print the sentence: “Printing from a function”.
- If we want to use the function we just have to call MyPrint() and the printf statement will be executed.

```
#include<stdio.h>

void MyPrint()
{
    printf("Printing from a function.\n");
}

void main()
{
    MyPrint();
}
```



DEFINING A FUNCTION

- The general form of a function definition is as follows:

```
return_type function_name( parameter list )
{
    body of the function
}
```

- A function definition in C language consists of
 - a *function header* and
 - a *function body*.



PARTS OF A FUNCTION

- Here are all the parts of a function:

- **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as *actual parameter* or *argument*. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Return Type:** A function may return a value. The **return type** is the data type of the value the function returns. If you don't want to return a result from a function, you can use void return type. Some functions perform the desired operations without returning a value. In this case, the return type is the keyword **void**.
- **Function Body:** The function body contains a collection of statements that define what the function does.

EXAMPLE: FUNCTION

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

```
/* function returning the max between two numbers */
int max(int num1, int num2)
{
    /* local variable declaration */
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
```

FUNCTION DECLARATIONS

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

```
return_type function_name( parameter list );
```

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

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

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

```
int max(int, int);
```

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

CALLING A FUNCTION

- While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.
- When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its *return* statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.
- To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

CALLING A FUNCTION

Understanding Flow

```
void main()
{
int num;

num = square(4);

printf("%d",num)
}
```

```
int square(int n1)
{
    int x = n1 * n1;
    return(x);
}
```

1. Firstly Operating System will call our main function.
2. When control comes inside main function , execution of main starts (i.e execution of C program starts)
3. Consider Line 4 :

```
num = square(4);
```

4. We have called a function square(4). [See : [How to call a function ?](#)].
5. We have passed “4” as parameter to function.

Note : Calling a function halts execution of the current function , it will execute called function , after execution control returned back to the calling function.

6. Function will return 16 to the calling function.(i.e main)
7. Returned value will be copied into variable.
8. printf will gets executed.
9. main function ends.
10. C program terminates.

CODE EXAMPLE: CALLING A FUNCTION

```
#include <stdio.h>

/* function returning the max between two numbers */
int max(int num1, int num2)
{
    /* local variable declaration */
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}

void main ()
{
    /* local variable definition */
    int a = 100;
    int b = 200;
    int ret;
    /* calling a function to get max value */
    ret = max(a, b);
    printf( "Max value is : %d\n", ret );
}
```



CODE EXAMPLE: CALLING A FUNCTION

```
#include <stdio.h>

/* function declaration/function prototype */
int max(int num1, int num2);

void main ()
{
    /* local variable definition */
    int a = 100;
    int b = 200;
    int ret;
    /* calling a function to get max value */
    ret = max(a, b);
    printf( "Max value is : %d\n", ret );
}

/* function returning the max between two numbers */
int max(int num1, int num2)
{
    /* local variable declaration */
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}
```

A C program with
function *declaration*/
function *prototype*.

FUNCTION ARGUMENTS

- C functions can accept an unlimited number of parameters.
- If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the ***formal parameters*** of the function.
- The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.



GLOBAL AND LOCAL VARIABLES

- Local variable:
 - A local variable is a variable that is declared inside a function.
 - A local variable can only be used in the function where it is declared.
- Global variable:
 - A global variable is a variable that is declared outside **all** functions.
 - A global variable can be used in all functions.
- See the following example(see the next slide)



GLOBAL AND LOCAL VARIABLES

```
#include <stdio.h>

// Global variables
int A;
int B;

int Add()
{
    return A + B;
}

void main()
{
    int answer; // Local variable
    A = 5;
    B = 7;
    answer = Add();
    printf("%d\n", answer);
    return 0;
}
```

- As you can see two global variables are declared, **A** and **B**. These variables can be used in **main()** and **Add()**.
- The local variable **answer** can only be used in **main()**.

TYPE OF FUNCTION CALL

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

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by reference	This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

CALL BY VALUE

- The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function.
- In this case, changes made to the parameter inside the function have no effect on the argument.
- By default, C programming language uses *call by value* method to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.



CALL BY VALUE

- Consider the function **swap()** definition as follows.

```
/* function definition to swap the values */
void swap(int x, int y)
{
    int temp;
    temp = x; /* save the value of x */
    x = y;    /* put y into x */
    y = temp; /* put temp into y */
}
```

CALL BY VALUE

- Now, let us call the function **swap()** by passing actual values as in the following example:

```
#include <stdio.h>

/* function definition to swap the values */
void swap(int x, int y)
{
    int temp;
    temp = x; /* save the value of x */
    x = y;    /* put y into x */
    y = temp; /* put temp into y */
}

void main ()
{
    /* local variable definition */
    int a = 100;
    int b = 200;

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

    /* calling a function to swap the values */
    swap(a, b);

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

Output:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

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

CALL BY REFERENCE

- The **call by reference** method of passing arguments to a function copies the address of an argument into the *formal parameter*.
- Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.
- To pass the value by call by reference, argument pointers are passed to the functions.



FUNCTION CALL BY REFERENCE

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

```
/* function definition to swap the values */
void swap(int *x, int *y)
{
    int temp;
    temp = *x; /* save the value of x */
    *x = *y;    /* put y into x */
    *y = temp; /* put temp into y */
}
```

CALL BY REFERENCE

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

```
#include <stdio.h>

/* function definition to swap the values */
void swap(int *x, int *y)
{
    int temp;
    temp = *x; /* save the value of x */
    *x = *y;    /* put y into x */
    *y = temp; /* put temp into y */
}

void main ()
{
    /* local variable definition */
    int a = 100;
    int b = 200;

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

    /* calling a function to swap the values.
     * &a indicates pointer to a ie. address of variable a and
     * &b indicates pointer to b ie. address of variable b.
     */
    swap(&a, &b);

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

Output:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

Which shows that the change has reflected outside of the function as well unlike call by value where changes does not reflect outside of the function.

RECURSION

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

```
void recursion()
{
    recursion(); /* function calls itself */
}

int main()
{
    recursion();
}
```

- The C programming language supports recursion, i.e., a function to call itself.
- But while using recursion, programmers need to be careful to define an exit condition (base condition) from the function, otherwise it will go in infinite loop.
- Recursive function are very useful to solve many mathematical problems like to calculate factorial of a number, generating Fibonacci series, etc.

NUMBER FACTORIAL

- Following is an example, which calculates factorial for a given number using a recursive function:

```
#include <stdio.h>

int factorial(unsigned int i)
{
    if(i <= 1)
    {
        return 1;
    }
    return i * factorial(i - 1);
}

int main()
{
    int i = 15;
    printf("Factorial of %d is %d\n", i, factorial(i));
    return 0;
}
```

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

```
Factorial of 15 is 2004310016
```

from main()

rec (int x)

{

 int f;

 if (x == 1)

 return (1);

 else

 f = x * rec (x - 1);

 return (f);

}

to main()

rec (int x)

{

 int f;

 if (x == 1)

 return (1);

 else

 f = x * rec (x - 1);

 return (f);

}

rec (int x)

{

 int f;

 if (x == 1)

 return (1);

 else

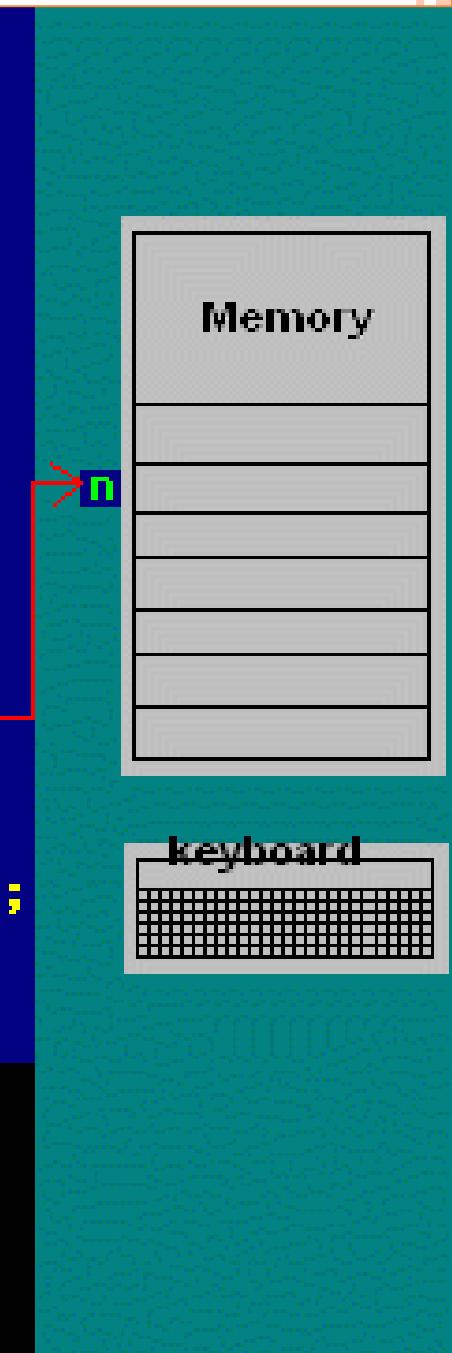
 f = x * rec (x - 1);

 return (f);

}

```
#include <stdio.h>
#include <conio.h>
long int fact(int x)
{
    long int f;
    if(x==1)
        return(x);
    else
        f=x*fact(x-1);
    return(f);
}

int main()
{
    int n;
    clrscr();
    printf("Enter a number\n");
    scanf("%d",&n);
    printf("Factorial of %d is %ld",n,fact(n));
    getch();
    return 0;
}
```



FIBONACCI SERIES

- Following is another example, which generates Fibonacci series for a given number using a recursive function:

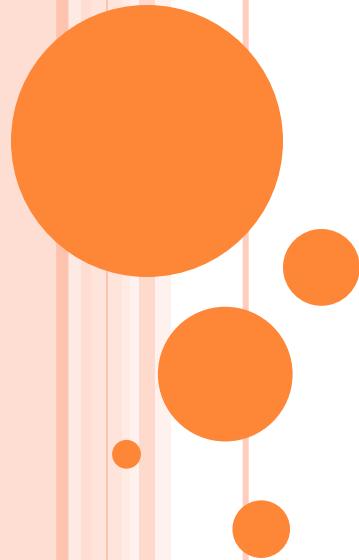
```
#include <stdio.h>

int fibonaci(int i)
{
    if(i == 0)
    {
        return 0;
    }
    if(i == 1)
    {
        return 1;
    }
    return fibonaci(i-1) + fibonaci(i-2);
}

int main()
{
    int i;
    for (i = 0; i < 10; i++)
    {
        printf("%d\t", fibonaci(i));
    }
    return 0;
}
```

Try

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



C – POINTER

C – POINTER

- If you want to be proficient in the writing of code in the C programming language, you must have a thorough working knowledge of how to use pointers.
- Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer.
- As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which will print the address of the variables defined:

C – POINTER

```
#include <stdio.h>

int main ()
{
    int var1;
    char var2[10];

    printf("Address of var1 variable: %x\n", &var1 );
    printf("Address of var2 variable: %x\n", &var2 );

    return 0;
}
```

Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6

So you understood what is memory address and how to access it, so base of the concept is over. Now let us see what is a pointer.

WHAT IS POINTER?

- A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address.
- The general form of a pointer variable declaration is:

*dataType *var_name;*

- Here,
 - **dataType** is the pointer's base type; it must be a valid C data type(i.e., int, float, char etc).
 - **var_name** is the name of the pointer variable.
 - The asterisk * you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer.

POINTER

- Following are the valid pointer declaration:

```
int    *ip;      /* pointer to an integer */
double *dp;      /* pointer to a double */
float  *fp;      /* pointer to a float */
char   *ch;      /* pointer to a character */
```

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

HOW TO USE POINTERS?

- There are few important operations, which we will do with the help of pointers very frequently.
 - ① we define a pointer variable
 - ② assign the address of a variable to a pointer and
 - ③ finally access the value at the address available in the pointer variable by dereferencing.
- Dereferencing is done by using unary operator * that returns the value of the variable located at the address specified by its operand.
- Dereferencing a pointer means getting the value stored in the memory at the address which the pointer “points” to.
- The * is the value-at-address operator, also called the indirection operator. It is used both when declaring a pointer and when dereferencing a pointer.



HOW TO USE POINTERS?

- Following example makes use of these operations:

```
#include <stdio.h>

int main ()
{
    int var = 20;      /* actual variable declaration */
    int *ip;          /* pointer variable declaration */

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

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

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

    /* access the value using the pointer */
    printf("Value of *ip variable: %d\n", *ip );

    return 0;
}
```

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

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

HOW TO USE POINTERS?

- the & is the address-of operator and is used to reference the memory address of a variable.
- By using the & operator in front of a variable name we can retrieve the memory address-of that variable. It is best to read this operator as address-of operator.
- Following code shows some common notations for the value-at-address (*) and address-of (&) operators.

```
// declare an variable ptr which holds the value-at-address of an int type
int *ptr;
// declare assign an int the literal value of 1
int val = 1;
// assign to ptr the address-of the val variable
ptr = &val;
// dereference and get the value-at-address stored in ptr
int deref = *ptr;
printf("%d\n", deref);
```

NULL POINTERS IN C

- It is always a good practice to assign a NULL value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.
- The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:

```
#include <stdio.h>

void main ()
{
    int *ptr = NULL;

    printf("The value of ptr is : %x\n", ptr );
}
```

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

The value of ptr is 0

```
#include<conio.h>
#include<stdio.h>
int main()
{
    int *ptr1,*ptr2,a,b;
    clrscr();
    printf("Enter two numbers\n");
    scanf("%d%d",&a,&b);
    printf("Given numbers are %d and %d\n",a,b);
    ptr1=&a;
    ptr2=&b;
    printf("Address of a is %x and that of b is %x\n",ptr1,ptr2);
    printf("Sum of %d and %d is %d\n",a,b,*ptr1+*ptr2);
    getch();
    return 0;
}
```

Variable name ----->

	MEMORY BLOCKS	
--	---------------	--

Address ----->

NULL POINTERS IN C

- On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.
- To check for a null pointer you can use an if statement as follows:

```
if(ptr)      /* succeeds if p is not null */  
if(!ptr)    /* succeeds if p is null */
```



C POINTERS IN DETAIL

- Pointers have many but easy concepts and they are very important to C programming. There are following few important pointer concepts which should be clear to a C programmer:

Concept	Description
C - Pointer arithmetic	There are four arithmetic operators that can be used on pointers: ++, --, +, -
C - Array of pointers	You can define arrays to hold a number of pointers.
C - Pointer to pointer	C allows you to have pointer on a pointer and so on.
Passing pointers to functions in C	Passing an argument by reference or by address both enable the passed argument to be changed in the calling function by the called function.
Return pointer from functions in C	C allows a function to return a pointer to local variable, static variable and dynamically allocated memory as well.



C - POINTER ARITHMETIC

- C pointer is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can a numeric value.
- There are four arithmetic operators that can be used on pointers: `++`, `--`, `+`, and `-`.
- To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer:

`++ptr`

- Now, after the above operation, the **ptr** will point to the location 1004 because each time **ptr** is incremented, it will point to the next integer location which is 4 bytes next to the current location.
- If **ptr** points to a character whose address is 1000, then above operation will point to the location 1001 because next character will be available at 1001.

INCREMENTING A POINTER

- We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer.
- The following program increments the variable pointer to access each succeeding element of the array (see next slide).



INCREMENTING A POINTER

```
#include <stdio.h>

const int MAX = 3;

int main ()
{
    int var[] = {10, 100, 200};
    int i, *ptr;

    /* let us have array address in pointer */
    ptr = var;
    for ( i = 0; i < MAX; i++)
    {

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

        /* move to the next location */
        ptr++;
    }
    return 0;
}
```

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

DECREMENTING A POINTER

- The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below:

```
#include <stdio.h>

const int MAX = 3;

int main ()
{
    int var[] = {10, 100, 200};
    int i, *ptr;

    /* let us have array address in pointer */
    ptr = &var[MAX-1];
    for ( i = MAX; i > 0; i--)
    {

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

        /* move to the previous location */
        ptr--;
    }
    return 0;
}
```

```
Address of var[3] = bfedbcd8
Value of var[3] = 200
Address of var[2] = bfedbcd4
Value of var[2] = 100
Address of var[1] = bfedbcd0
Value of var[1] = 10
```

POINTER * AND ++

Expression

*p++ or * (p++)

(*p) ++

*++p or * (++p)

++*p or ++ (*p)

Meaning

Value of expression is *p before increment; increment p later

Value of expression is *p before increment; increment *p later

Increment p first; value of expression is *p after increment

Increment *p first; value of expression is *p after increment



POINTER COMPARISONS

- Pointers may be compared by using relational operators, such as ==, <, and >.
- If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.
- The following program modifies the previous example one by incrementing the variable pointer as long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1] (see next slide).



POINTER COMPARISONS

```
#include <stdio.h>

const int MAX = 3;

int main ()
{
    int var[] = {10, 100, 200};
    int i, *ptr;

    /* let us have address of the first element in pointer */
    ptr = var;
    i = 0;
    while ( ptr <= &var[MAX - 1] )
    {

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

        /* point to the previous location */
        ptr++;
        i++;
    }
    return 0;
}
```

```
Address of var[0] = bfdbcb20
Value of var[0] = 10
Address of var[1] = bfdbcb24
Value of var[1] = 100
Address of var[2] = bfdbcb28
Value of var[2] = 200
```

ARRAY OF POINTERS

- Before we understand the concept of arrays of pointers, let us consider the following example, which makes use of an array of 3 integers:

```
#include <stdio.h>

const int MAX = 3;

int main ()
{
    int var[] = {10, 100, 200};
    int i;

    for (i = 0; i < MAX; i++)
    {
        printf("Value of var[%d] = %d\n", i, var[i]);
    }
    return 0;
}
```

Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200

ARRAY OF POINTERS

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

```
int *ptr[ ];
```

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



ARRAY OF POINTERS

- Following example makes use of three integers, which will be stored in an array of pointers as follows:

```
#include <stdio.h>

const int MAX = 3;

int main ()
{
    int var[] = {10, 100, 200};
    int i, *ptr[MAX];

    for ( i = 0; i < MAX; i++)
    {
        ptr[i] = &var[i]; /* assign the address of integer. */
    }
    for ( i = 0; i < MAX; i++)
    {
        printf("Value of var[%d] = %d\n", i, *ptr[i] );
    }
    return 0;
}
```

Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200



ARRAY OF POINTERS

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

```
#include <stdio.h>

const int MAX = 4;

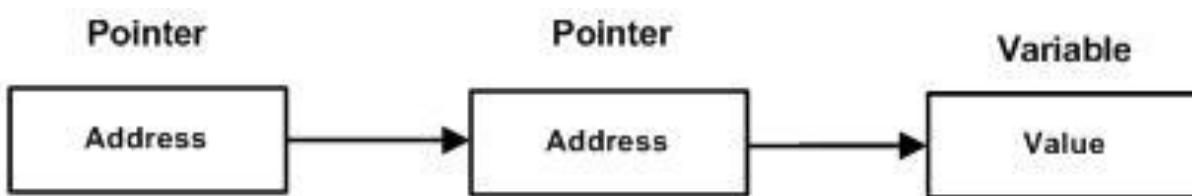
int main ()
{
    char *names[] = {
        "Zara Ali",
        "Hina Ali",
        "Nuha Ali",
        "Sara Ali",
    };
    int i = 0;

    for ( i = 0; i < MAX; i++)
    {
        printf("Value of names[%d] = %s\n", i, names[i] );
    }
    return 0;
}
```

```
Value of names[0] = Zara Ali
Value of names[1] = Hina Ali
Value of names[2] = Nuha Ali
Value of names[3] = Sara Ali
```

POINTER TO POINTER

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



- A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, following is the declaration to declare a pointer to a pointer of type *int*:

```
int ***var;
```

POINTER TO POINTER

- When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example:

```
#include <stdio.h>

int main ()
{
    int var;
    int *ptr;
    int **pptr;

    var = 3000;

    /* take the address of var */
    ptr = &var;

    /* take the address of ptr using address of operator & */
    pptr = &ptr;

    /* take the value using pptr */
    printf("Value of var = %d\n", var );
    printf("Value available at *ptr = %d\n", *ptr );
    printf("Value available at **pptr = %d\n", **pptr);

    return 0;
}
```

Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000

PASSING POINTERS TO FUNCTIONS

- C programming language allows you to pass a pointer to a function. To do so, simply declare the function parameter as a pointer type.
- Following is a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function:

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

void getSeconds(unsigned long *par);

int main ()
{
    unsigned long sec;

    getSeconds( &sec );

    /* print the actual value */
    printf("Number of seconds: %ld\n", sec );

    return 0;
}

void getSeconds(unsigned long *par)
{
    /* get the current number of seconds */
    *par = time( NULL );
    return;
}
```

Number of seconds :1294450468

PASSING POINTERS TO FUNCTIONS

- The function, which can accept a pointer, can also accept an array as shown in the following example:

Average value is: 214.40000

```
#include <stdio.h>

/* function declaration */
double getAverage(int *arr, int size);

int main ()
{
    /* an int array with 5 elements */
    int balance[5] = {1000, 2, 3, 17, 50};
    double avg;

    /* pass pointer to the array as an argument */
    avg = getAverage( balance, 5 ) ;

    /* output the returned value */
    printf("Average value is: %f\n", avg );

    return 0;
}

double getAverage(int *arr, int size)
{
    int     i, sum = 0;
    double avg;

    for (i = 0; i < size; ++i)
    {
        sum += arr[i];
    }

    avg = (double)sum / size;

    return avg;
}
```

RETURN POINTER FROM FUNCTIONS

- C allows us to return a pointer from a function. To do so, you would have to declare a function returning a pointer as in the following example:

```
int * myFunction()  
{  
    ;  
    ;  
    ;  
}
```

- Second point to remember is that, it is not good idea to return the address of a local variable to outside of the function so you would have to define the local variable as **static** variable.



RETURN POINTER FROM FUNCTIONS

- Consider the following function which will generate 10 random numbers and return them using an array name which represents a pointer i.e., address of first array element.

```
1523198053
1187214107
1108300978
430494959
1421301276
930971084
123250484
106932140
1604461820
149169022
*(p + [0]) : 1523198053
*(p + [1]) : 1187214107
*(p + [2]) : 1108300978
*(p + [3]) : 430494959
*(p + [4]) : 1421301276
*(p + [5]) : 930971084
*(p + [6]) : 123250484
*(p + [7]) : 106932140
*(p + [8]) : 1604461820
*(p + [9]) : 149169022
```

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

/* function to generate and retrun random numbers. */
int * getRandom( )
{
    static int r[10];
    int i;

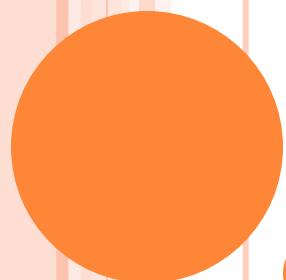
    /* set the seed */
    srand( (unsigned)time( NULL ) );
    for ( i = 0; i < 10; ++i)
    {
        r[i] = rand();
        printf("%d\n", r[i] );
    }

    return r;
}

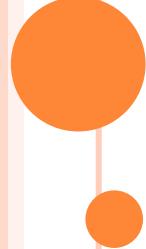
/* main function to call above defined function */
int main ( )
{
    /* a pointer to an int */
    int *p;
    int i;

    p = getRandom();
    for ( i = 0; i < 10; i++ )
    {
        printf("*(p + [%d]) : %d\n", i, *(p + i) );
    }

    return 0;
}
```



C - STRING



C - STRING

- strings are arrays of chars. String literals are words surrounded by double quotation marks.

"This is a static string"

- The string in C programming language is actually a one-dimensional array of characters which is terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.
- A string can be declared as a **character array** or with a **string pointer**.
- The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Or

```
char greeting[] = "Hello";
```

Or

```
char *greeting = "Hello";
```



C - STRING

- Following is the memory presentation of above defined string in C/C++:

Index	0	1	2	3	4	5
Variable	H	e	I	I	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

- It's important to remember that there will be an extra character on the end of a string, literally a '\0' character, just like there is always a period at the end of a sentence. Since this string terminator is unprintable, it is not counted as a letter, but it still takes up a space. Technically, in a fifty char array you could only hold 49 letters and one null character at the end to terminate the string.
- Actually, you do not place the *null* character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array.

C - STRING

- Let us try to print above mentioned string:

```
#include <stdio.h>

int main ()
{
    char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

    printf("Greeting message: %s\n", greeting );

    return 0;
}
```

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

```
Greeting message: Hello
```

- Note: %s is used to print a string.



STRING POINTER

- String pointers are declared as a pointer to a char.
- When there is a value assigned to the string pointer the NULL is put at the end automatically.
- Take a look at this example:

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

void main()
{
    char *ptr_mystring;

    ptr_mystring = "HELLO";

    printf("%s\n", ptr_mystring);

}
```

STRING POINTER

- It is not possible to read, with scanf(), a string with a string pointer. You have to use a character array and a pointer. See this example:

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

void main()
{
    char my_array[10];
    char *ptr_section2;

    printf("Type hello and press enter\n");
    scanf("%s", my_array);
    ptr_section2 = my_array;
    printf("%s\n", ptr_section2);
}
```

READING A LINE OF TEXT

gets() and puts() are two string functions to take string input from user and display string respectively

```
int main(){
    char name[30];
    printf("Enter name: ");
    gets(name);      //Function to read string from user.
    printf("Name: ");
    puts(name);      //Function to display string.
    return 0;
}
```

STRING RELATED OPERATIONS

- Find the Frequency of Characters in a String
- Find the Number of Vowels, Consonants, Digits and White space in a String
- Reverse a String by Passing it to Function
- Find the Length of a String
- Concatenate Two Strings
- Copy a String
- Remove all Characters in a String except alphabet
- Sort a string in alphabetic order
- Sort Elements in Lexicographical Order (Dictionary Order)
- Change Decimal to Hexadecimal Number
- Convert Binary Number to Decimal



FIND THE FREQUENCY OF CHARACTERS

```
#include <stdio.h>
int main(){
    char c[1000],ch;
    int i,count=0;
    printf("Enter a string: ");
    gets(c);
    printf("Enter a character to find frequency: ");
    scanf("%c",&ch);
    for(i=0;c[i]!='\0';++i)
    {
        if(ch==c[i])
            ++count;
    }
    printf("Frequency of %c = %d", ch, count);
    return 0;
}
```

```
Enter a string: This website is awesome.
Enter a frequency to find frequency: e
Frequency of e = 4
```



C PROGRAM TO FIND FREQUENCY OF CHARACTERS IN A STRING

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

int main()
{
    char string[100];
    int c = 0, count[26] = {0};

    printf("Enter a string\n");
    gets(string);

    while ( string[c] != '\0' )
    {
        /* Considering characters from 'a' to 'z' only */

        if ( string[c] >= 'a' && string[c] <= 'z' )
            count[string[c]-'a']++;

        c++;
    }

    for ( c = 0 ; c < 26 ; c++ )
    {
        if( count[c] != 0 )
            printf("%c occurs %d times in the entered
string.\n",c+'a',count[c]);
    }

    return 0;
}
```

This program computes frequency of characters in a string i.e. which character is present how many times in a string.

For example in the string "code" each of the character 'c', 'o', 'd', and 'e' has occurred one time.

Only *lower case alphabets* are considered, other characters (uppercase and special characters) are ignored. You can easily modify this program to handle uppercase and special symbols.



FIND NUMBER OF VOWELS, CONSONANTS, DIGITS AND WHITE SPACE CHARACTER

```
#include<stdio.h>

int main(){
    char line[150];
    int i,v,c,ch,d,s,o;
    o=v=c=ch=d=s=0;
    printf("Enter a line of string:\n");
    gets(line);
    for(i=0;line[i]!='\0';++i)
    {
        if(line[i]=='a' || line[i]=='e' || line[i]=='i' || line[i]=='o' || line[i]=='u' ||
line[i]=='A' || line[i]=='E' || line[i]=='I' || line[i]=='O' || line[i]=='U')
            ++v;
        else if((line[i]>='a'&& line[i]<='z') || (line[i]>='A'&& line[i]<='Z'))
            ++c;
        else if(line[i]>='0'&&c<='9')
            ++d;
        else if (line[i]==' ')
            ++s;
    }
    printf("Vowels: %d",v);
    printf("\nConsonants: %d",c);
    printf("\nDigits: %d",d);
    printf("\nWhite spaces: %d",s);
    return 0;
}
```

Output

```
Enter a line of string:
This program is easy 2 understand
Vowels: 9
Consonants: 18
Digits: 1
White spaces: 5
```

REVERSE STRING

```
#include<stdio.h>
#include<string.h>
void Reverse(char str[]);
int main(){
    char str[100];
    printf("Enter a string to reverse: ");
    gets(str);
    Reverse(str);
    printf("Reversed string: ");
    puts(str);
    return 0;
}
void Reverse(char str[]){
    int i,j;
    char temp[100];
    for(i=strlen(str)-1,j=0; i+1!=0; --i,++j)
    {
        temp[j]=str[i];
    }
    temp[j]='\0';
    strcpy(str,temp);
}
```

To solve this problem, two standard library functions **strlen()** and **strcpy()** are used to calculate length and to copy string respectively.

```
Enter a string to reverse: zimargorp
Reversed string: programiz
```

CALCULATED LENGTH OF A STRING WITHOUT USING STRLEN() FUNCTION

You can use standard library function `strlen()` to find the length of a string but, this program computes the length of a string manually without using `strlen()` function.

```
#include <stdio.h>
int main()
{
    char s[1000], i;
    printf("Enter a string: ");
    scanf("%s", s);
    for(i=0; s[i]!='\0'; ++i);
    printf("Length of string: %d", i);
    return 0;
}
```

Enter a string: Programiz
Length of string: 9



CONCATENATE TWO STRINGS MANUALLY

You can concatenate two strings using standard library function **strcat()** , this program concatenates two strings manually without using strcat() function.

```
#include <stdio.h>
int main()
{
    char s1[100], s2[100], i, j;
    printf("Enter first string: ");
    scanf("%s", s1);
    printf("Enter second string: ");
    scanf("%s", s2);
    for(i=0; s1[i]!='\0'; ++i); /* i contains length of string s1. */
    for(j=0; s2[j]!='\0'; ++j, ++i)
    {
        s1[i]=s2[j];
    }
    s1[i]='\0';
    printf("After concatenation: %s", s1);
    return 0;
}
```

```
Enter first string: lol
Enter second string: :)
After concatenation: lol:)
```

COPY STRING MANUALLY

```
#include <stdio.h>
int main()
{
    char s1[100], s2[100], i;
    printf("Enter string s1: ");
    scanf("%s", s1);
    for(i=0; s1[i]!='\0'; ++i)
    {
        s2[i]=s1[i];
    }
    s2[i]='\0';
    printf("String s2: %s", s2);
    return 0;
}
```

```
Enter String s1: programiz
String s2: programiz
```

You can use the `strcpy()` function to copy the content of one string to another but, this program copies the content of one string to another manually without using `strcpy()` function.



REMOVE CHARACTERS IN STRING EXCEPT ALPHABETS

```
#include<stdio.h>

int main(){
    char line[150];
    int i,j;
    printf("Enter a string: ");
    gets(line);
    for(i=0; line[i]!='\0'; ++i)
    {
        while (((line[i]>='a'&&line[i]<='z') || (line[i]>='A'&&line[i]<='Z' || line[i]=='\0')))
        {
            for(j=i;line[j]!='\0';++j)
            {
                line[j]=line[j+1];
            }
            line[j]='\0';
        }
    }
    printf("Output String: ");
    puts(line);
    return 0;
}
```

This program takes a string from user and *for* loop executed until all characters of string is checked. If any character inside a string is not a alphabet, all characters after it including null character is shifted by 1 position backwards.

```
Enter a string: p2'r'o@gram84iz./
Output String: programiz
```

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

void sort_string(char *s)
{
    int c, d = 0, length;
    char *pointer, *result, ch;
    length = strlen(s);
    result = (char*)malloc(length+1);
    pointer = s;
    for ( ch = 'a' ; ch <= 'z' ; ch++ )
    {
        for ( c = 0 ; c < length ; c++ )
        {
            if ( *pointer == ch )
            {
                *(result+d) = *pointer;
                d++;
            }
            pointer++;
        }
        pointer = s;
    }
    *(result+d) = '\0';
    strcpy(s, result);
    free(result);
}

void main()
{
    char string[100];
    printf("Enter some text\n");
    gets(string);
    sort_string(string);
    printf("%s\n", string);
}
```

SORT A STRING IN ALPHABETIC ORDER

C program to sort a string in alphabetic order: For example if user will enter a string "programming" then output will be "aggimmnoprr" or output string will contain characters in alphabetical order.

```
Enter some text
game
aegm
```

SORT ELEMENTS IN LEXICOGRAPHICAL ORDER (DICTIONARY ORDER)

```
#include<stdio.h>
#include <string.h>
int main(){
    int i,j;
    char str[10][50],temp[50];
    printf("Enter 10 words:\n");
    for(i=0;i<10;++i)
        gets(str[i]);
    for(i=0;i<9;++i)
        for(j=i+1;j<10 ;++j){
            if(strcmp(str[i],str[j])>0)
            {
                strcpy(temp,str[i]);
                strcpy(str[i],str[j]);
                strcpy(str[j],temp);
            }
        }
    printf("In lexicographical order: \n");
    for(i=0;i<10;++i)
        puts(str[i]);
}
return 0;
```

This program takes 10 words from user and sorts elements in lexicographical order. To perform this task, two dimensional string is used.

```
Enter 10 words:
fortran
java
perl
python
php
javascript
c
cpp
ruby
csharp

In lexicographical order:
c
cpp
csharp
fortran
java
javascript
perl
php
python
ruby
```

C LIBRARY FUNCTIONS

- C supports a wide range of functions that manipulate null-terminated strings:

S.no	String functions	Description
1	<code>strcat()</code>	Concatenates str2 at the end of str1.
2	<code>strncat()</code>	appends a portion of string to another
3	<code>strcpy()</code>	Copies str2 into str1
4	<code>strncpy()</code>	copies given number of characters of one string to another
5	<code>strlen()</code>	gives the length of str1.
6	<code>strcmp()</code>	Returns 0 if str1 is same as str2. Returns <0 if str1 < str2. Returns >0 if str1 > str2.
7	<code>strcmpi()</code>	Same as strcmp() function. But, this function negotiates case. "A" and "a" are treated as same.
8	<code>strchr()</code>	Returns pointer to first occurrence of char in str1.
9	<code> strrchr()</code>	last occurrence of given character in a string is found
10	<code>strstr()</code>	Returns pointer to first occurrence of str2 in str1.
11	<code> strrstr()</code>	Returns pointer to last occurrence of str2 in str1.
12	<code>strdup()</code>	duplicates the string
13	<code>strlwr()</code>	converts string to lowercase
14	<code>strupr()</code>	converts string to uppercase
15	<code>strrev()</code>	reverses the given string
16	<code>strset()</code>	sets all character in a string to given character
17	<code>strnset()</code>	It sets the portion of characters in a string to given character
18	<code> strtok()</code>	tokenizing given string using delimiter

Following example makes use of few of the above-mentioned functions:

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

int main ()
{
    char str1[12] = "Hello";
    char str2[12] = "World";
    char str3[12];
    int len ;

    /* copy str1 into str3 */
    strcpy(str3, str1);
    printf("strcpy( str3, str1) :  %s\n", str3 );

    /* concatenates str1 and str2 */
    strcat( str1, str2);
    printf("strcat( str1, str2):   %s\n", str1 );

    /* total length of str1 after concatenation */
    len = strlen(str1);
    printf("strlen(str1) :  %d\n", len );

    return 0;
}
```

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

```
strcpy( str3, str1) :  Hello
strcat( str1, str2):   HelloWorld
strlen(str1) :  10
```

STRCAT() FUNCTION

- strcat() function concatenates two given strings. It concatenates source string at the end of destination string.
- Syntax for strcat() function is given below.
 - `char * strcat (char * destination, const char * source);`
- Example :
- `strcat (str2, str1);` - str1 is concatenated at the end of str2.
`strcat (str1, str2);` - str2 is concatenated at the end of str1.
- As you know, each string in C is ended up with null character ('\0').
- In strcat() operation, null character of destination string is overwritten by source string's first character and null character is added at the end of new destination string which is created after strcat() operation.



EXAMPLE PROGRAM FOR STRCAT()

- In this program, two strings “is fun” and “C tutorial” are concatenated using strcat() function and result is displayed as “C tutorial is fun”.

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

int main( )
{
    char source[ ] = " is fun" ;
    char target[ ]= " C tutorial" ;

    printf( "\nSource string = %s", source ) ;
    printf( "\nTarget string = %s", target ) ;

    strcat( target, source ) ;

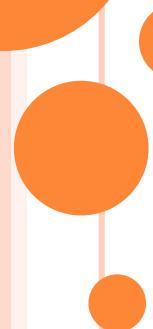
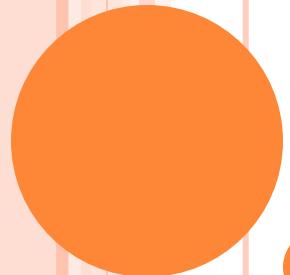
    printf( "\nTarget string after strcat( ) = %s", target );
}
```

Output:

Source string = is fun

Target string = C tutorial

Target string after strcat() = C tutorial is fun



C - STRUCTURES

C - STRUCTURES

- C arrays allow you to define type of variables that can hold several data items of the same kind but **structure** is another user defined data type available in C programming, which allows you to combine data items of different kinds.
- Structure is the collection of variables of different types under a single name for better handling.
- Structures are used to represent a record, Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:
 - Title
 - Author
 - Subject
 - Book ID



DEFINING A STRUCTURE

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

```
struct [structure tag]
{
    member definition;
    member definition;
    ...
    member definition;
} [one or more structure variables];
```

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

DEFINING A STRUCTURE

- Here is the way you would declare the Book structure:

Example 1

```
struct Books
{
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
} book;
```

Structure tag

Member definition

structure variables

Example 2

```
struct person
{
    char name[50];
    int cit_no;
    float salary;
};
```

- With the declaration of the structure you have created a new type, called **Books**.

STRUCTURE VARIABLE DECLARATION

- When a structure is defined, it creates a user-defined type but, no storage is allocated.
- For the above structure of person, variable can be declared as:

```
struct person
{
    char name[50];
    int cit_no;
    float salary;
};
```

Inside main function:

```
struct person p1, p2, p[20];
```

Another way of creating structure variable is:

```
struct person
{
    char name[50];
    int cit_no;
    float salary;
}p1 ,p2 ,p[20];
```

In both cases, 2 variables p_1 , p_2 and array p having 20 elements of type **struct person** are created.

DIFFERENCE BETWEEN C VARIABLE, ARRAY AND STRUCTURE

- A normal C variable can hold only one data of one data type at a time.
- An array can hold group of data of same data type.
- A structure can hold group of data of different data types
- Data types can be int, char, float, double and long double etc.

Datatype	C variable		C array		C structure	
	Syntax	Example	Syntax	Example	Syntax	Example
int	int a	a = 20	int a[3]	a[0] = 10 a[1] = 20 a[2] = 30 a[3] = '\0'	struct student { int a; char b[10]; }	a = 10 b = "Hello"
char	char b	b='Z'	char b[10]	b="Hello"		

BELOW TABLE EXPLAINS FOLLOWING CONCEPTS IN C STRUCTURE

- How to declare a C structure?
- How to initialize a C structure?
- How to access the members of a C structure?

Type	Using normal variable	Using pointer variable
Syntax	struct tag_name { data type var_name1; data type var_name2; data type var_name3; };	struct tag_name { data type var_name1; data type var_name2; data type var_name3; };
Example	struct student { int mark; char name[10]; float average; };	struct student { int mark; char name[10]; float average; };
Declaring structure variable	struct student report;	struct student *report, rep;
Initializing structure variable	struct student report = {100, "Mani", 99.5};	struct student rep = {100, "Mani", 99.5}; report = &rep;
Accessing structure members	report.mark report.name report.average	report -> mark report -> name report -> average

ACCESSING MEMBERS OF A STRUCTURE

- Structure can be accessed in 2 ways. They are,
 1. Using normal structure variable
 2. Using pointer variable
- There are two types of operators used for accessing members of a structure.
 - Member operator(.)
 - Structure pointer operator(->)
- Dot(.) operator is used to access the data using normal structure variable.
- Arrow (->) is used to access the data using pointer variable.
- We already have learnt how to access structure data using normal variable. So, we are showing here how to access structure data using pointer variable.

POINTERS TO STRUCTURES

You can define pointers to structures in very similar way as you define pointer to any other variable as follows:

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the & operator before the structure's name as follows:

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows:

```
struct_pointer->title;
```



EXAMPLE PROGRAM FOR C STRUCTURE

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

struct student
{
    int id;
    char name[20];
    float percentage;
};

int main()
{
    struct student std;

    std.id=1;
    strcpy(std.name, "Raju");
    std.percentage = 86.5;

    printf(" Id is: %d \n", std.id);
    printf(" Name is: %s \n", std.name);
    printf(" Percentage is: %f \n", std.percentage);
    return 0;
}
```

This program is used to store and access “id, name and percentage” for one student. We can also store and access these data for many students using array of structures.

Output:

Id is: 1

Name is: Raju

Percentage is: 86.500000

```

#include <stdio.h>
#include <string.h>

struct student {
    int id;
    char name[30];
    float percentage;
};

int main() {
    int i;
    struct student record[2];

    // 1st student's record
    record[0].id=1;
    strcpy(record[0].name, "Raju");
    record[0].percentage = 86.5;

    // 2nd student's record
    record[1].id=2;
    strcpy(record[1].name, "Surendren");
    record[1].percentage = 90.5;

    // 3rd student's record
    record[2].id=3;
    strcpy(record[2].name, "Thiyagu");
    record[2].percentage = 81.5;

    for(i=0; i<3; i++)
    {
        printf("      Records of STUDENT : %d \n", i+1);
        printf(" Id is: %d \n", record[i].id);
        printf(" Name is: %s \n", record[i].name);
        printf(" Percentage is: %f\n\n",record[i].percentage);
    }
    return 0;
}

```

EXAMPLE PROGRAM-ARRAY OF STRUCTURES

This program is used to store and access “id, name and percentage” for 3 students. Structure array is used in this program to store and display records for many students. You can store “n” number of students record by declaring structure variable as ‘struct student record[n]“, where n can be 1000 or 5000 etc.

Output:

Records of STUDENT : 1

Id is: 1

Name is: Raju

Percentage is: 86.500000

Records of STUDENT : 2

Id is: 2

Name is: Surendren

Percentage is: 90.500000

Records of STUDENT : 3

Id is: 3 Name is: Thiyagu

Percentage is: 81.500000



EXAMPLE PROGRAM OF STRUCTURE

Write a C program to add two distances entered by user. Measurement of distance should be in inch and feet.(Note: 12 inches = 1 foot)

```
#include <stdio.h>
struct Distance{
    int feet;
    float inch;
}d1,d2,sum;
int main(){
    printf("1st distance\n");
    printf("Enter feet: ");
    scanf("%d",&d1.feet); /* input of feet for structure variable d1 */
    printf("Enter inch: ");
    scanf("%f",&d1.inch); /* input of inch for structure variable d1 */
    printf("2nd distance\n");
    printf("Enter feet: ");
    scanf("%d",&d2.feet); /* input of feet for structure variable d2 */
    printf("Enter inch: ");
    scanf("%f",&d2.inch); /* input of inch for structure variable d2 */
    sum.feet=d1.feet+d2.feet;
    sum.inch=d1.inch+d2.inch;
    if (sum.inch>12){ //If inch is greater than 12, changing it to feet.
        ++sum.feet;
        sum.inch=sum.inch-12;
    }
    printf("Sum of distances=%d\'.1f\"",sum.feet,sum.inch);
/* printing sum of distance d1 and d2 */
    return 0;
}
```

PASSING STRUCTURE TO FUNCTION

- A structure can be passed to any function from main function or from any sub function.
- Structure definition will be available within the function only.
- It won't be available to other functions unless it is passed to those functions by value or by address(reference).
- Else, we have to declare structure variable as global variable. That means, structure variable should be declared outside the main function. So, this structure will be visible to all the functions in a C program.
- **Passing structure to function in C:** It can be done in below 3 ways.
 - ① Passing structure to a function by value
 - ② Passing structure to a function by address(reference)
 - ③ No need to pass a structure – Declare structure variable as global

EXAMPLE – PASSING STRUCTURE TO FUNCTION BY VALUE

- A structure variable can be passed to the function as an argument as normal variable.
- If structure is passed by value, change made in structure variable in function definition does not reflect in original structure variable in calling function.
- You would access structure variables in the similar way as you have accessed in the above example:

```
#include <stdio.h>

struct student{
    char name[50];
    int roll;
};

void Display(struct student stu){
    printf("\nName: %s",stu.name);
    printf("\nRoll: %d",stu.roll);
}

/* function prototype should be below to the structure
declaration otherwise compiler shows error */
void main(){
    struct student s1;
    printf("Enter student's name: ");
    scanf("%s",&s1.name);
    printf("Enter roll number:");
    scanf("%d",&s1.roll);
    // passing structure variable s1 as argument
    Display(s1);
}
```

Passing structure variable

Write a C program to create a structure student, containing name and roll. Ask user the name and roll of a student in main function. Pass this structure to a function and display the information in that function.

Output:

Enter student's name: Kevin
Enter roll number: 149
Name: Kevin
Roll: 149

EXAMPLE – PASSING STRUCTURE TO FUNCTION BY VALUE

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

struct student{
    int id;
    char name[20];
    float percentage;
};

void func(struct student record) {
    printf(" Id is: %d \n", record.id);
    printf(" Name is: %s \n", record.name);
    printf(" Percentage is: %f \n", record.percentage);
}

int main() {
    struct student record;

    record.id=1;
    strcpy(record.name, "Raju");
    record.percentage = 86.5;
    func(record);
    return 0;
}
```

In this program, the whole structure is passed to another function by value. It means the whole structure is passed to another function with all members and their values. So, this structure can be accessed from called function. This concept is very useful while writing very big programs in C.

Output:

Id is: 1

Name is: Raju

Percentage is: 86.500000

EXAMPLE – PASSING STRUCTURE TO FUNCTION BY VALUE

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

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

struct Books
{
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
};

/* function declaration */
void printBook( struct Books book );
int main( )
{
    struct Books Book1;           /* Declare Book1 of type Book */
    struct Books Book2;           /* Declare Book2 of type Book */

    /* book 1 specification */
    strcpy( Book1.title, "C Programming");
    strcpy( Book1.author, "Nuha Ali");
    strcpy( Book1.subject, "C Programming Tutorial");
    Book1.book_id = 6495407;

    /* book 2 specification */
    strcpy( Book2.title, "Telecom Billing");
    strcpy( Book2.author, "Zara Ali");
    strcpy( Book2.subject, "Telecom Billing Tutorial");
    Book2.book_id = 6495700;

    /* print Book1 info */
    printBook( Book1 );

    /* Print Book2 info */
    printBook( Book2 );

    return 0;
}
void printBook( struct Books book )
{
    printf( "Book title : %s\n", book.title);
    printf( "Book author : %s\n", book.author);
    printf( "Book subject : %s\n", book.subject);
    printf( "Book book_id : %d\n", book.book_id);
}
```

PASSING STRUCTURE TO FUNCTION BY ADDRESS/REFERENCE

- The **address** location of **structure** variable is passed to function while passing it by reference.
- If structure is passed by reference, change made in structure variable in function definition reflects in original structure variable in the calling function.
- Exercise: Write a C program to add two distances(feet-inch system) entered by user. To solve this program, make a structure. Pass two structure variable (containing distance in feet and inch) to add function by reference and display the result in main function without returning it.



```

#include <stdio.h>
struct distance{
    int feet;
    float inch;
};

void Add(struct distance d1, struct distance d2, struct distance *d3)
{
    /* Adding distances d1 and d2 and storing it in d3 */
    d3->feet=d1.feet+d2.feet;
    d3->inch=d1.inch+d2.inch;
    /* if inch is greater or equal to 12, converting it to feet. */
    if (d3->inch>=12) {
        d3->inch-=12;
        ++d3->feet;
    }
}

void main()
{
    struct distance dist1, dist2, dist3;
    printf("First distance\n");
    printf("Enter feet: ");
    scanf("%d",&dist1.feet);
    printf("Enter inch: ");
    scanf("%f",&dist1.inch);
    printf("Second distance\n");
    printf("Enter feet: ");
    scanf("%d",&dist2.feet);
    printf("Enter inch: ");
    scanf("%f",&dist2.inch);
    /*passing structure variables dist1 and dist2 by value whereas
    passing structure variable dist3 by reference */
    Add(dist1, dist2, &dist3);
    printf("\nSum of distances = %d'-%.1f\"", dist3.feet, dist3.inch);
}

```

EXAMPLE—PASSING STRUCTURE TO FUNCTION BY REFERENCE

Output:

First distance
 Enter feet: 12
 Enter inch: 6.8
 Second distance
 Enter feet: 5
 Enter inch: 7.5

Sum of distances = 18'-2.3"



Explanation of previous example

In the previous program, structure variables dist1 and dist2 are passed by value (because value of dist1 and dist2 does not need to be displayed in main function) and dist3 is passed by reference ,i.e, address of dist3 (`&dist3`) is passed as an argument.

Thus, the structure pointer variable d3 points to the address of dist3. If any change is made in d3 variable, effect of it is seen in dist3 variable in main function.



EXAMPLE—PASSING STRUCTURE TO FUNCTION BY ADDRESS/REFERENCE

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

struct student {
    int id;
    char name[20];
    float percentage;
};

void func(struct student *record) {
    printf(" Id is: %d \n", record->id);
    printf(" Name is: %s \n", record->name);
    printf(" Percentage is: %f \n", record->percentage);
}

int main() {
    struct student record;

    record.id=1;
    strcpy(record.name, "Raju");
    record.percentage = 86.5;

    func(&record); //Passing the address
    return 0;
}
```

Here the structure is passed to another function by address. It means only the address of the structure is passed to another function. The whole structure is not passed to another function with all members and their values. So, this structure can be accessed from called function by its address.

Output:

Id is: 1

Name is: Raju

Percentage is: 86.500000

EXAMPLE—PASSING STRUCTURE TO FUNCTION BY ADDRESS/REFERENCE

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

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

struct Books
{
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
};

/* function declaration */
void printBook( struct Books *book );
int main( )
{
    struct Books Book1;           /* Declare Book1 of type Book */
    struct Books Book2;           /* Declare Book2 of type Book */

    /* book 1 specification */
    strcpy( Book1.title, "C Programming");
    strcpy( Book1.author, "Nuha Ali");
    strcpy( Book1.subject, "C Programming Tutorial");
    Book1.book_id = 6495407;

    /* book 2 specification */
    strcpy( Book2.title, "Telecom Billing");
    strcpy( Book2.author, "Zara Ali");
    strcpy( Book2.subject, "Telecom Billing Tutorial");
    Book2.book_id = 6495700;

    /* print Book1 info by passing address of Book1 */
    printBook( &Book1 );

    /* print Book2 info by passing address of Book2 */
    printBook( &Book2 );

    return 0;
}
void printBook( struct Books *book )
{
    printf( "Book title : %s\n", book->title);
    printf( "Book author : %s\n", book->author);
    printf( "Book subject : %s\n", book->subject);
    printf( "Book book_id : %d\n", book->book_id);
}
```

EXAMPLE PROGRAM TO DECLARE A STRUCTURE VARIABLE AS GLOBAL

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

struct student {
    int id;
    char name[20];
    float percentage;
};

struct student record; // Global declaration of structure

void structure_demo() {
    printf(" Id is: %d \n", record.id);
    printf(" Name is: %s \n", record.name);
    printf(" Percentage is: %f \n", record.percentage);
}

int main() {
    record.id=1;
    strcpy(record.name, "Raju");
    record.percentage = 86.5;

    structure_demo();
    return 0;
}
```

Structure variables also can be declared as global variables as we declare other variables in C. So, When a structure variable is declared as global, then it is visible to all the functions in a program. In this scenario, we don't need to pass the structure to any function separately.

Output:

Id is: 1

Name is: Raju

Percentage is: 86.500000



COPY A STRUCTURE

- There are many methods to copy one structure to another structure in C.
 - We can copy using direct assignment of one structure to another structure or
 - we can use C inbuilt function “memcpy()” or
 - we can copy by individual structure members.

Output:

Records of STUDENT1 - record1 structure

Id : 1

Name : Raju

Percentage : 90.500000

Records of STUDENT1 – Direct copy from record1

Id : 1

Name : Raju

Percentage : 90.500000

Records of STUDENT1 – copied from record1 using memcpy

Id : 1

Name : Raju

Percentage : 90.500000

Records of STUDENT1 – Copied individual members from record1

Id : 1

Name : Raju

Percentage : 90.500000

```
#include <string.h>

struct student {
    int id;
    char name[30];
    float percentage;
};

int main() {
    int i;
    struct student record1 = {1, "Raju", 90.5};
    struct student record2, *record3, *ptr1, record4;
    printf("Records of STUDENT1 - record1 structure \n");
    printf(" Id : %d \n Name : %s\n Percentage : %f\n",
           record1.id, record1.name, record1.percentage);
    // 1st method to copy whole structure to another structure
    record2=record1;
    printf("\nRecords of STUDENT1 - Direct copy from " \
           "record1 \n");
    printf(" Id : %d \n Name : %s\n Percentage : %f\n",
           record2.id, record2.name, record2.percentage);
    // 2nd method to copy using memcpy function
    ptr1 = &record1;
    memcpy(record3, ptr1, sizeof(record1));
    printf("\nRecords of STUDENT1 - copied from record1 " \
           "using memcpy \n");
    printf(" Id : %d \n Name : %s\n Percentage : %f\n",
           record3->id, record3->name, record3->percentage);
    // 3rd method to copy by individual members
    printf("\nRecords of STUDENT1 - Copied individual " \
           "members from record1 \n");
    record4.id=record1.id;
    strcpy(record4.name, record1.name);
    record4.percentage = record1.percentage;
    printf(" Id : %d \n Name : %s\n Percentage : %f\n",
           record4.id, record4.name, record4.percentage);
}
```

KEYWORD TYPEDEF WHILE USING STRUCTURE

- Programmer generally use *typedef* while using structure in C language. For example:

```
typedef struct complex{  
    int imag;  
    float real;  
}comp;  
  
Inside main:  
comp c1,c2;
```

- Here, *typedef* keyword is used in creating a type *comp*(which is of type as **struct complex**). Then, two structure variables *c1* and *c2* are created by this *comp* type.

STRUCT MEMORY ALLOCATION

- How structure members are stored in memory?

- Always, contiguous(adjacent) memory locations are used to store structure members in memory. Consider below example to understand how memory is allocated for structures.

```
#include <stdio.h>
struct student
{
    int id1;
    int id2;
    char a;
    char b;
    float percentage;
};

int main() {
    int i;
    struct student record1 = {1, 2, 'A', 'B', 90.5};

    printf("size of structure in bytes : %d\n", sizeof(record1));

    printf("\nAddress of id1      = %u", &record1.id1 );
    printf("\nAddress of id2      = %u", &record1.id2 );
    printf("\nAddress of a         = %u", &record1.a );
    printf("\nAddress of b         = %u", &record1.b );
    printf("\nAddress of percentage= %u", &record1.percentage);

    return 0;
}
```

Output:

size of structure in bytes : 16

Address of id1 = 675376768

Address of id2 = 675376772

Address of a = 675376776

Address of b = 675376777

Address of percentage = 675376780

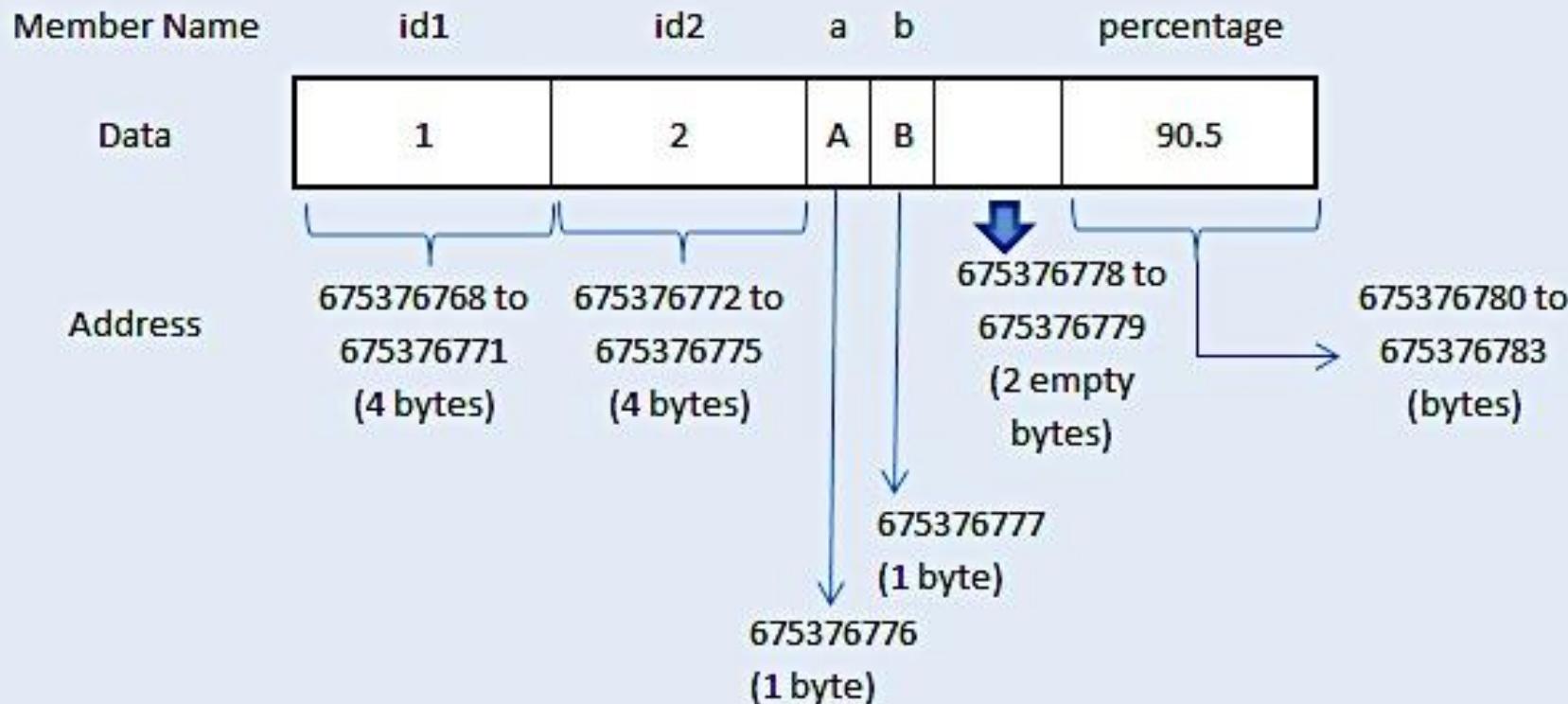
STRUCT MEMORY ALLOCATION

- There are 5 members declared for structure in above program.
In 32 bit compiler,
 - 4 bytes of memory is occupied by *int* datatype.
 - 1 byte of memory is occupied by *char* datatype and
 - 4 bytes of memory is occupied by *float* datatype.
- Please refer below table to know from where to where memory is allocated for each datatype in contiguous (adjacent) location in memory.

Datatype	Memory allocation in C (32 bit compiler)		
	From Address	To Address	Total bytes
int id1	675376768	675376771	4
int id2	675376772	675376775	4
char a	675376776		1
char b	675376777		1
Addresses 675376778 and 675376779 are left empty (Do you know why? Please see Structure padding topic below)			2
float percentage	675376780	675376783	4

STRUCT MEMORY ALLOCATION

- The pictorial representation of above structure memory allocation is given below.
- This diagram will help you to understand the memory allocation concept in C very easily.



STRUCTURE PADDING

- In order to align the data in memory, one or more empty bytes (addresses) are inserted (or left empty) between memory addresses which are allocated for other structure members while memory allocation. This concept is called *structure padding*.
- Architecture of a computer processor is such a way that it can read 1 word (4 byte in 32 bit processor) from memory at a time.
- To make use of this advantage of processor, data are always aligned as 4 bytes package which leads to insert empty addresses between other member's address.
- Because of this structure padding concept in C, size of the structure is always not same as what we think.



STRUCTURE PADDING

- For example, consider below structure that has 5 members.
- struct student

```
{  
    int id1;  
    int id2;  
    char a;  
    char b;  
    float percentage;  
};
```

- As per C concepts, int and float datatypes occupy 4 bytes each and char datatype occupies 1 byte for 32 bit processor. So, only 14 bytes ($4+4+1+1+4$) should be allocated for above structure.
- But, this is wrong. Do you know why?
 - Architecture of a computer processor is such a way that it can read 1 word from memory at a time.
 - 1 word is equal to 4 bytes for 32 bit processor and 8 bytes for 64 bit processor.
 - So, 32 bit processor always reads 4 bytes at a time and 64 bit processor always reads 8 bytes at a time.
 - This concept is very useful to increase the processor speed.
 - To make use of this advantage, memory is arranged as a group of 4 bytes in 32 bit processor and 8 bytes in 64 bit processor.

```

#include <string.h>
/* Below structure1 and structure2 are same.
   They differ only in member's alignment */
struct structure1 {
    int id1;
    int id2;
    char name;
    char c;
    float percentage;
};

struct structure2 {
    int id1;
    char name;
    int id2;
    char c;
    float percentage;
};

int main() {
    struct structure1 a;
    struct structure2 b;

    printf("size of structure1 in bytes : %d\n", sizeof(a));
    printf ("\n    Address of id1      = %u", &a.id1 );
    printf ("\n    Address of id2      = %u", &a.id2 );
    printf ("\n    Address of name      = %u", &a.name );
    printf ("\n    Address of c          = %u", &a.c );
    printf ("\n    Address of percentage = %u", &a.percentage );

    printf("\n\nsize of structure2 in bytes : %d\n", sizeof(b));
    printf ("\n    Address of id1      = %u", &b.id1 );
    printf ("\n    Address of name      = %u", &b.name );
    printf ("\n    Address of id2      = %u", &b.id2 );
    printf ("\n    Address of c          = %u", &b.c );
    printf ("\n    Address of percentage = %u", &b.percentage );
}

```

EXAMPLE PROGRAM FOR STRUCTURE PADDING

- This C program is compiled and executed in 32 bit compiler.
- Please check memory allocated for structure1 and structure2 of this program.

Output:

size of structure1 in bytes : 16

Address of id1 = 1297339856

Address of id2 = 1297339860

Address of name = 1297339864

Address of c = 1297339865

Address of percentage = 1297339868

size of structure2 in bytes : 20

Address of id1 = 1297339824

Address of name = 1297339828

Address of id2 = 1297339832

Address of c = 1297339836

Address of percentage = 1297339840

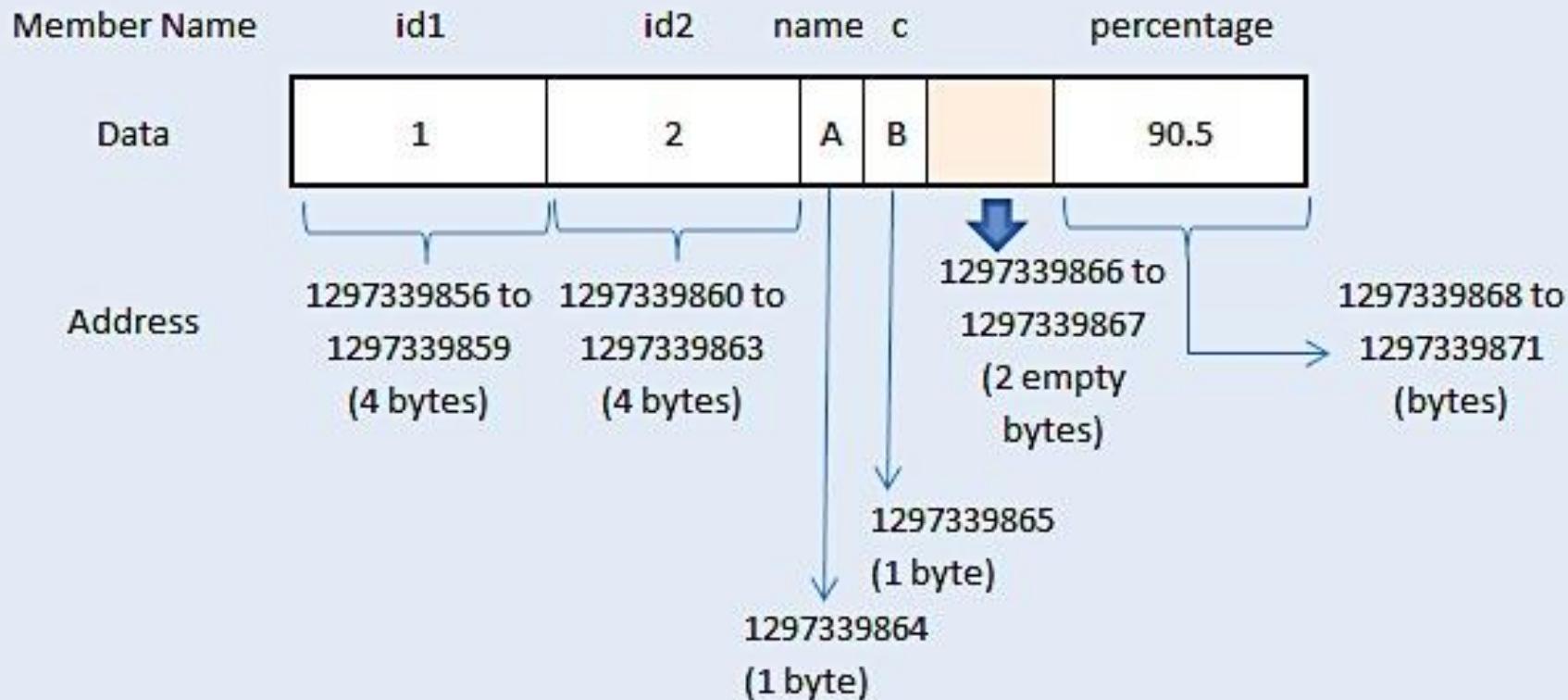
STRUCTURE PADDING ANALYSIS FOR PREVIOUS C PROGRAM

- Memory allocation for **structure1**:

- In above program, memory for structure1 is allocated sequentially for first 4 members.
- Whereas, memory for 5th member “percentage” is not allocated immediate next to the end of member “c”
- There are only 2 bytes remaining in the package of 4 bytes after memory allocated to member “c”.
- Range of this 4 byte package is from 1297339864 to 1297339867.
- Addresses 1297339864 and 1297339865 are used for members “name and c”. Addresses 1297339866 and 1297339867 only is available in this package.
- But, member “percentage” is datatype of float and requires 4 bytes. It can’t be stored in the same memory package as it requires 4 bytes. Only 2 bytes are free in that package.
- So, next 4 byte of memory package is chosen to store percentage data which is from 1297339868 to 1297339871.
- Because of this, memory 1297339866 and 1297339867 are not used by the program and those 2 bytes are left empty.
- So, size of structure1 is 16 bytes which is 2 bytes extra than what we think. Because, 2 bytes are left empty.

STRUCTURE PADDING ANALYSIS FOR PREVIOUS C PROGRAM

Memory allocation for **structure1**



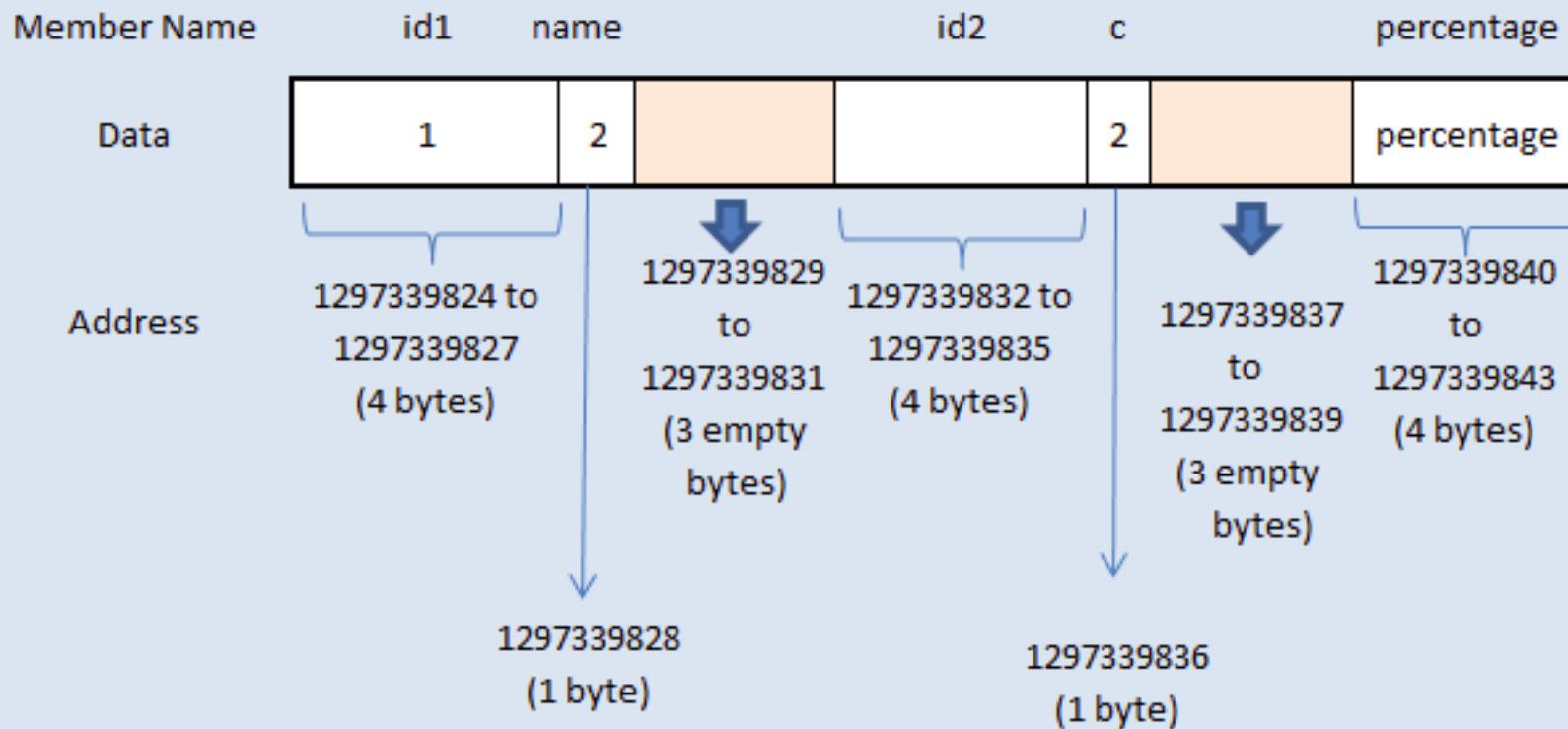
STRUCTURE PADDING ANALYSIS FOR PREVIOUS C PROGRAM

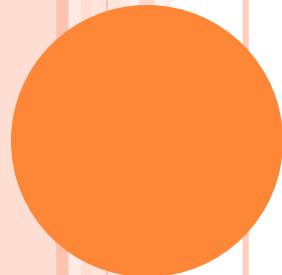
○ Memory allocation for structure2:

- Memory for structure2 is also allocated as same as above concept. Please note that structure1 and structure2 are same. But, they differ only in the order of the members declared inside the structure.
- 4 bytes of memory is allocated for 1st structure member “id1” which occupies whole 4 byte of memory package.
- Then, 2nd structure member “name” occupies only 1 byte of memory in next 4 byte package and remaining 3 bytes are left empty. Because, 3rd structure member “id2” of datatype integer requires whole 4 byte of memory in the package. But, this is not possible as only 3 bytes available in the package.
- So, next whole 4 byte package is used for structure member “id2”.
- Again, 4th structure member “c” occupies only 1 byte of memory in next 4 byte package and remaining 3 bytes are left empty.
- Because, 5th structure member “percentage” of datatype float requires whole 4 byte of memory in the package.
- But, this is also not possible as only 3 bytes available in the package. So, next whole 4 byte package is used for structure member “percentage”.
- So, size of structure2 is 20 bytes which is 6 bytes extra than what we think. Because, 6 bytes are left empty.

STRUCTURE PADDING ANALYSIS FOR PREVIOUS C PROGRAM

Memory allocation for **structure2**





C - TYPEDEF

Dr. Sheak Rashed Haider Noori

Assistant Professor

Department of Computer Science

C - TYPEDEF

- **Typedef** is a keyword that is used to give a new symbolic name for the existing name in a C program.
- This is same like defining alias for the commands. Consider the below structure.
- ```
struct student
{
 int mark [2];
 char name [10];
 float average;
}
```
- Variable for the above structure can be declared in two ways.
- 1<sup>st</sup> way :
  - struct student record; /\* for normal variable \*/
  - struct student \*record; /\* for pointer variable \*/
- 2<sup>nd</sup> way :
  - typedef struct student status;
- When we use “typedef” keyword before struct <tag\_name> like above, after that we can simply use type definition “status” in the C program to declare structure variable.
- Now, structure variable declaration will be, “status record”.
- This is equal to “struct student record”. Type definition for “struct student” is status. i.e. status = “struct student”

## C - TYPEDEF

- An alternative way for structure declaration using **typedef** in C:

- ```
typedef struct student
{
    int mark [2];
    char name [10];
    float average;
} status;
```

- To declare structure variable, we can use the below statements.

- ```
status record1; /* record 1 is structure variable */
status record2; /* record 2 is structure variable */
```



# EXAMPLE PROGRAM FOR C TYPEDEF

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

// Structure using typedef:
typedef struct student
{
 int id;
 char name[20];
 float percentage;
} status;

int main()
{
 status record;
 record.id=1;
 strcpy(record.name, "Raju");
 record.percentage = 86.5;
 printf(" Id is: %d \n", record.id);
 printf(" Name is: %s \n", record.name);
 printf(" Percentage is: %f \n", record.percentage);
 return 0;
}
```

## Output:

id is: 1

Name is: Raju

Percentage is: 86.500000

## C - TYPEDEF

- Typedef can be used to simplify the real commands as per our need.
- For example, consider below statement.
- `typedef long long int myLong;`
- In above statement, `myLong` is the type definition for the real C command “`long long int`”.
- We can use type definition `myLong` instead of using full command “`long long int`” in a C program once it is defined.

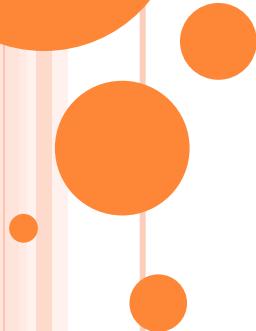
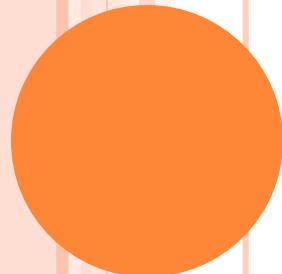
```
#include <stdio.h>
#include <limits.h>

int main()
{
 typedef long long int myLong;
 myLong a, b;

 printf("Storage size for long long int data type : %ld \n", sizeof(myLong));
 printf("Storage size for a variable : %ld \n", sizeof(a));
 return 0;
}
```

### Output:

Storage size for long long int data type : 8  
Storage size for a variable : 8



# C – FILE-IO

**Dr. Sheak Rashed Haider Noori**

**Assistant Professor**

**Department of Computer Science**

# C - FILE

- A file represents a sequence of bytes, does not matter if it is a text file or binary file.
- C programming language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices.
- This lesson will take you through important calls for the file management.



# OPENING FILES

- You can use the **fopen()** function to create a new file or to open an existing file, this call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream.
- Following is the prototype of this function call:

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

- Here,
  - **filename** is string literal, which you will use to name your file and
  - **mode** represent the access, which can have one of the following values(see the next slide).



# FILE ACCESS MODE

| Mode | Role                                                                                    | If the file already exists                                                                                                   | If the file does not exist                  |
|------|-----------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------|
| r    | Opens an existing file for reading only                                                 | it would be opened and can be read. After the file is opened, the user cannot add data to it                                 | the operation would fail                    |
| w    | Saves a new file                                                                        | the file's contents would be deleted and replaced by the new content                                                         | a new file is created and can be written to |
| a    | Opens an existing file, saves new file, or saves a existing file that has been modified | the file is opened and can be modified or updated. New information written to the file would be added to the end of the file | a new file is created and can be written to |
| r+   | Opens an existing file                                                                  | the file is opened and its existing data can be modified or updated                                                          | the operation would fail                    |
| w+   | Creates new file or saves an existing one                                               | the file is opened, its contents would be deleted and replaced with the new contents                                         | a new file is created and can be written to |
| a+   | Creates a new file or modifies an existing one                                          | it is opened and its contents can be updated. New information written to the file would be added to the end of the file      | a new file is created and can be written to |

# CLOSING A FILE

- To close a file, use the **fclose( )** function. The prototype of this function is:

```
int fclose(FILE *fp);
```

- The **fclose( )** function returns *zero* on success, or **EOF** if there is an error in closing the file.
- This function actually, flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file.
- The **EOF** is a constant defined in the header file **stdio.h**.
- There are various functions provide by C standard library to read and write a file character by character or in the form of a fixed length string. Let us see few of the in the next section.

# WRITING A FILE

- Following is the simplest function to write individual characters to a stream:

```
int fputc(int c, FILE *fp);
```

- The function **fputc()** writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise **EOF** if there is an error.
- You can use the following functions to write a null-terminated string to a stream:

```
int fputs(const char *s, FILE *fp);
```

- The function **fputs()** writes the string s to the output stream referenced by fp. It returns a non-negative value on success, otherwise **EOF** is returned in case of any error.
- You can use **int fprintf(FILE \*fp, const char \*format, ...)** function as well to write a string into a file.

# WRITING A FILE

- Try the following example:

```
#include <stdio.h>

main()
{
 FILE *fp;

 fp = fopen("/tmp/test.txt" , "w+");
 fprintf(fp, "This is testing for fprintf...\\n");
 fputs("This is testing for fputs...\\n", fp);
 fclose(fp);
}
```

When the above code is compiled and executed, it creates a new file *test.txt* in */tmp* directory and writes two lines using two different functions. Let us read this file in next section.

# READING A FILE

- Following is the simplest function to read a single character from a file:

```
int fgetc(FILE * fp);
```

- The **fgetc()** function reads a character from the input file referenced by **fp**. The return value is the character read, or in case of any error it returns **EOF**.
- The following functions allow you to read a string from a stream:

```
char *fgets(char *buf, int n, FILE *fp);
```

- The functions **fgets()** reads up to **n - 1** characters from the input stream referenced by **fp**. It copies the read string into the buffer **buf**, appending a **null** character to terminate the string.
- If this function encounters a newline character '\n' or the end of the file **EOF** before they have read the maximum number of characters, then it returns only the characters read up to that point including new line character.
- You can also use **int fscanf(FILE \*fp, const char \*format, ...)** function to read strings from a file but it stops reading after the first space character encounters.

# READING A FILE

```
#include <stdio.h>

main()
{
 FILE *fp;
 char buff[255];

 fp = fopen("/tmp/test.txt", "r");
 fscanf(fp, "%s", buff);
 printf("1 : %s\n", buff);

 fgets(buff, 255, (FILE*)fp);
 printf("2: %s\n", buff);

 fgets(buff, 255, (FILE*)fp);
 printf("3: %s\n", buff);
 fclose(fp);

}
```

When this code is compiled and executed, it reads the file created in previous section and produces the following result:

```
1 : This
2: is testing for fprintf...
3: This is testing for fputs...
```

Let's see a little more detail about what happened here. First fscanf() method read just **This** because after that it encountered a space, second call is for fgets() which read the remaining line till it encountered end of line. Finally last call fgets() read second line completely.

# READING A FILE

Suppose the input file consists of lines with a *username* and an *integer test score*, e.g.:

list.txt

-----

Foo 70

Bar 98

and that each username is no more than 8 characters long. We might use the files we opened above by copying each username and score from the input file to the output file.

In the process, we'll increase each score by 10 points for the output file:

```
char username[9]; /* One extra for nul char. */
int score;
...
while (fscanf(ifp, "%s %d", username, &score) != EOF) {
 fprintf(ofp, "%s %d\n", username, score+10);
}
```

The function fscanf(), like scanf(), normally returns the number of values it was able to read in. However, when it hits the end of the file, it returns the special value EOF.

So, testing the return value against EOF is one way to stop the loop.

## READING A FILE CONT.

The bad thing about testing against EOF is that if the file is not in the right format (e.g., a letter is found when a number is expected):

list.txt

-----

foo 70  
bar 98  
biz A+  
...

then fscanf() will not be able to read that line (since there is no integer to read) and it won't advance to the next line in the file. For this error, fscanf() will not return EOF (it's not at the end of the file)



## READING A FILE CONT.

Errors like that will at least mess up how the rest of the file is read. In some cases, they will cause an infinite loop.

One solution is to test against the number of values we expect to be read by `fscanf()` each time. Since our format is `"%s %d"`, we expect it to read in 2 values, so our condition could be:

```
while (fscanf(ifp, "%s %d", username, &score) == 2) {
 ...
```

Now, if we get 2 values, the loop continues. If we don't get 2 values, either because we are at the end of the file or some other problem occurred (e.g., it sees a letter when it is trying to read in a number with `%d`), then the loop will end.

## READING A FILE CONT.

Another way to test for end of file is with the library function feof(). It just takes a file pointer and returns a true/false value based on whether we are at the end of the file.

To use it in the above example, you would do:

```
while (!feof(ifp)) {
 if (fscanf(ifp, "%s %d", username, &score) != 2)
 break;
 fprintf(ofp, "%s %d", username, score+10);
}
```

Note that, like testing != EOF, it might cause an infinite loop if the format of the input file was not as expected. However, we can add code to make sure it reads in 2 values (as we've done above).

# BINARY I/O FUNCTIONS

- There are following two functions, which can be used for binary input and output:

```
size_t fread(void *ptr, size_t size_of_elements,
 size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
 size_t number_of_elements, FILE *a_file);
```

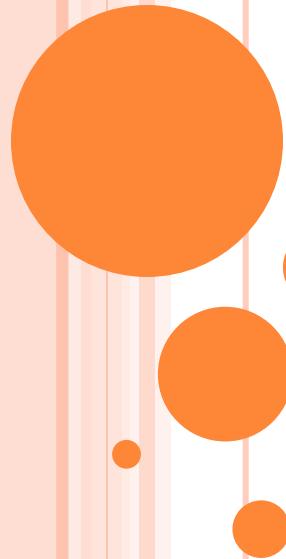
- Both of these functions should be used to read or write blocks of memories - usually arrays or structures.
- If you are going to handle binary files then you will use below mentioned access modes instead of the above mentioned:

```
"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"
```

## SEE ALSO

- [www.cs.bu.edu/teaching/c/file-io/intro/](http://www.cs.bu.edu/teaching/c/file-io/intro/)
- [www.codingunit.com/c-tutorial-file-io-using-text-files](http://www.codingunit.com/c-tutorial-file-io-using-text-files)
- [en.wikibooks.org/wiki/A\\_Little\\_C\\_Primer/C\\_File-IO\\_Through\\_Library\\_Functions](http://en.wikibooks.org/wiki/A_Little_C_Primer/C_File-IO_Through_Library_Functions)





# C - PREPROCESSORS

**Dr. Sheak Rashed Haider Noori**

**Assistant Professor**

**Department of Computer Science**

# C - PREPROCESSORS

- Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.
- Commands used in preprocessor are called **preprocessor directives** and they begin with “#” symbol.
- It must be the first nonblank character, and for readability.
- Below is the list of preprocessor directives that C language offers.

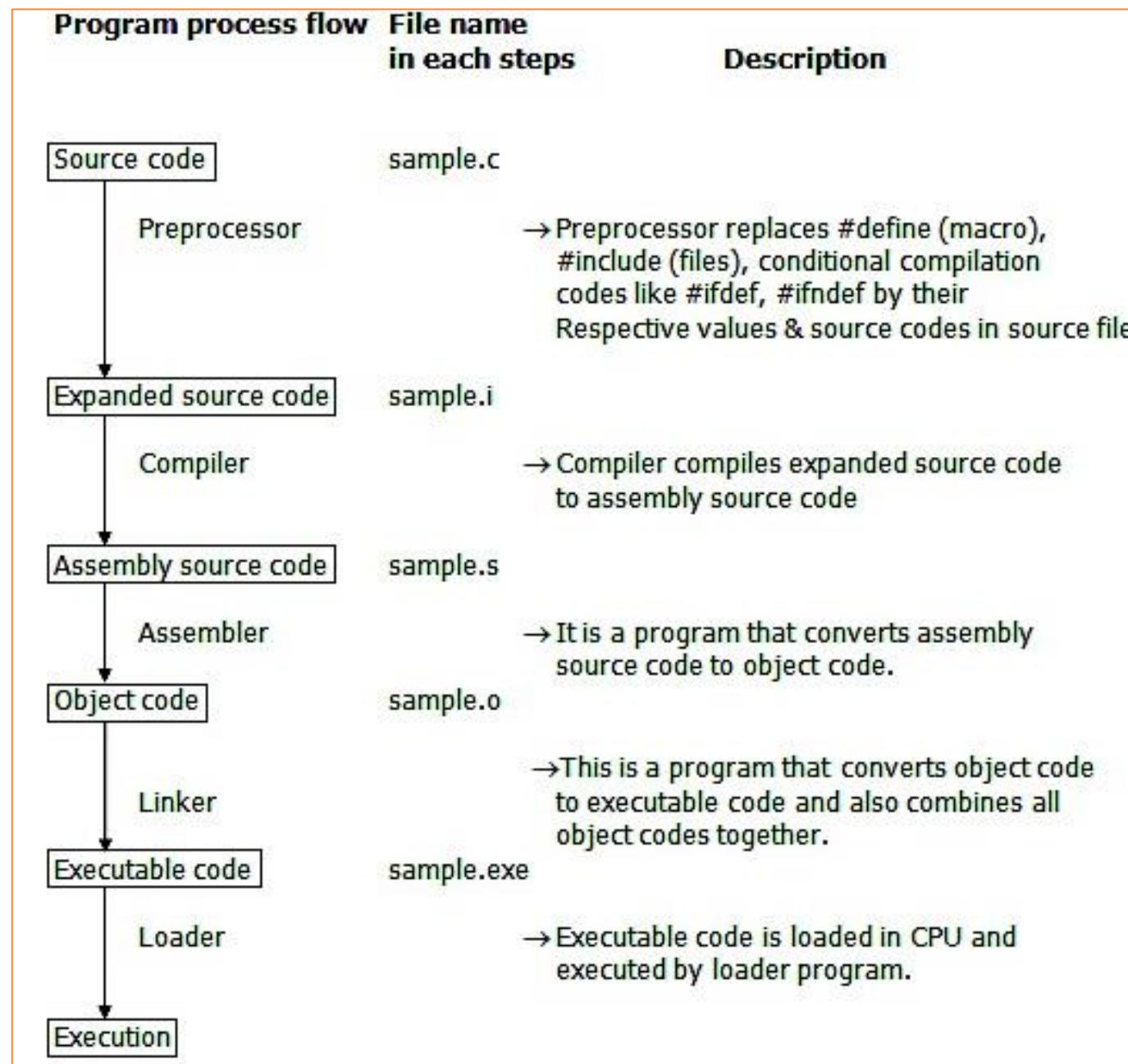
| S.no | Preprocessor            | Syntax                              | Description                                                                                                                           |
|------|-------------------------|-------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| 1    | Macro                   | #define                             | This macro defines constant value and can be any of the basic data types.                                                             |
| 2    | Header file inclusion   | #include <file_name>                | The source code of the file “file_name” is included in the main program at the specified place                                        |
| 3    | Conditional compilation | #ifdef, #endif, #if, #else, #ifndef | Set of commands are included or excluded in source program before compilation with respect to the condition                           |
| 4    | Other directives        | #undef, #pragma                     | #undef is used to undefine a defined macro variable. #Pragma is used to call a function before and after main function in a C program |

# IMPORTANT PREPROCESSOR DIRECTIVES

Following section lists down all important preprocessor directives:

| Directive | Description                                                          |
|-----------|----------------------------------------------------------------------|
| #define   | Substitutes a preprocessor macro                                     |
| #include  | Inserts a particular header from another file                        |
| #undef    | Undefines a preprocessor macro                                       |
| #ifdef    | Returns true if this macro is defined                                |
| #ifndef   | Returns true if this macro is not defined                            |
| #if       | Tests if a compile time condition is true                            |
| #else     | The alternative for #if                                              |
| #elif     | #else an #if in one statement                                        |
| #endif    | Ends preprocessor conditional                                        |
| #error    | Prints error message on stderr                                       |
| #pragma   | Issues special commands to the compiler, using a standardized method |

A program in C language involves into different processes. Below diagram will help you to understand all the processes that a C program comes across.



# PREPROCESSORS EXAMPLES

```
#define MAX_ARRAY_LENGTH 20
```

This directive tells the CPP to replace instances of `MAX_ARRAY_LENGTH` with 20. Use `#define` for constants to increase readability.

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

These directives tell the CPP to get `stdio.h` from **System Libraries** and add the text to the current source file. The next line tells CPP to get `myheader.h` from the local directory and add the content to the current source file.

```
#undef FILE_SIZE
#define FILE_SIZE 42
```

This tells the CPP to undefine existing `FILE_SIZE` and define it as 42.

```
#ifndef MESSAGE
#define MESSAGE "You wish!"
#endif
```

This tells the CPP to define `MESSAGE` only if `MESSAGE` isn't already defined.

```
#ifdef DEBUG
/* Your debugging statements here */
#endif
```

This tells the CPP to do the process the statements enclosed if `DEBUG` is defined. This is useful if you pass the `-DDEBUG` flag to `gcc` compiler at the time of compilation. This will define `DEBUG`, so you can turn debugging on and off on the fly during compilation.

# PREDEFINED MACROS

ANSI C defines a number of macros. Although each one is available for your use in programming, the predefined macros should not be directly modified.

| Macro               | Description                                                     |
|---------------------|-----------------------------------------------------------------|
| <code>_DATE_</code> | The current date as a character literal in "MMM DD YYYY" format |
| <code>_TIME_</code> | The current time as a character literal in "HH:MM:SS" format    |
| <code>_FILE_</code> | This contains the current filename as a string literal.         |
| <code>_LINE_</code> | This contains the current line number as a decimal constant.    |
| <code>_STDC_</code> | Defined as 1 when the compiler complies with the ANSI standard. |

```
#include <stdio.h>

main()
{
 printf("File :%s\n", __FILE__);
 printf("Date :%s\n", __DATE__);
 printf("Time :%s\n", __TIME__);
 printf("Line :%d\n", __LINE__);
 printf("ANSI :%d\n", __STDC__);
}
```

## Output

```
File :test.c
Date :Jun 2 2012
Time :03:36:24
Line :8
ANSI :1
```

# EXAMPLE PROGRAM FOR #DEFINE, #INCLUDE PREPROCESSORS IN C

- **#define** - This macro defines constant value and can be any of the basic data types.
- **#include <file\_name>** - The source code of the file “file\_name” is included in the main C program where “**#include <file\_name>**” is mentioned.

```
#include <stdio.h>

#define height 100
#define number 3.14
#define letter 'A'
#define letter_sequence "ABC"
#define backslash_char '\\'

void main()
{
 printf("value of height : %d \n", height);
 printf("value of number : %f \n", number);
 printf("value of letter : %c \n", letter);
 printf("value of letter_sequence : %s \n", letter_sequence);
 printf("value of backslash_char : %c \n", backslash_char);

}
```

## Output

```
value of height : 100
value of number : 3.140000
value of letter : A
value of letter_sequence : ABC
value of backslash_char : \
```

## EXAMPLE PROGRAM FOR #IFDEF, #ELSE AND #ENDIF

- “#ifdef” directive checks whether particular macro is defined or not. If it is defined, “If” clause statements are included in source file.
- Otherwise, “else” clause statements are included in source file for compilation and execution.

```
#include <stdio.h>
#define RAJU 100

int main()
{
 #ifdef RAJU
 printf("RAJU is defined. So, this line will be added in this C file\n");
 #else
 printf("RAJU is not defined\n");
 #endif
 return 0;
}
```

### Output

RAJU is defined. So, this line will be added in this C file

## EXAMPLE PROGRAM FOR #IFNDEF AND #ENDIF

- #ifndef exactly acts as reverse as #ifdef directive. If particular macro is not defined, “If” clause statements are included in source file.
- Otherwise, else clause statements are included in source file for compilation and execution.

```
#include <stdio.h>
#define RAJU 100

int main()
{
 #ifndef SELVA
 {
 printf("SELVA is not defined. So, now we are going to define here\n");
 #define SELVA 300
 }
 #else
 printf("SELVA is already defined in the program");

 #endif
 return 0;
}
```

### Output

SELVA is not defined. So, now we are going to define here

## EXAMPLE PROGRAM FOR #IF, #ELSE AND #ENDIF

- “If” clause statement is included in source file if given condition is true.
- Otherwise, else clause statement is included in source file for compilation and execution.

```
#include <stdio.h>
#define a 100

int main()
{
 #if (a==100)
 printf("This line will be added in this C file since a = 100\n");
 #else
 printf("This line will be added in this C file since a is not equal to 100\n");
 #endif
 return 0;
}
```

### Output

This line will be added in this C file since a = 100

## EXAMPLE PROGRAM FOR UNDEF

- This directive undefines existing macro in the program.

```
#include <stdio.h>

#define height 100
void main()
{
 printf("First defined value for height : %d\n",height);
 #undef height // undefining variable
 #define height 600 // redefining the same for new value
 printf("value of height after undef & redefine:%d",height);
}
```

### Output

First defined value for height : 100  
value of height after undef & redefine:600

# EXAMPLE PROGRAM FOR PRAGMA

```
#include <stdio.h>

void function1();
void function2();

#pragma startup function1
#pragma exit function2

int main()
{
 printf("\n Now we are in main function");
 return 0;
}

void function1()
{
 printf("nFunction1 is called before main function call");
}

void function2()
{
 printf("\nFunction2 is called just before end of main function");
}
```

## Output

Function1 is called before main function call

Now we are in main function

Function2 is called just before end of main function



# PREPROCESSOR OPERATORS

- The C preprocessor offers following operators to help you in creating macros:
- Macro Continuation (\)
  - A macro usually must be contained on a single line.
  - The macro continuation operator is used to continue a macro that is too long for a single line. For example:

```
#define message_for(a, b) \
 printf(#a " and " #b ": We love you!\n")
```



# PREPROCESSOR OPERATORS

## ○ Stringize (#)

- The stringize or number-sign operator ('#'), when used within a macro definition, converts a macro parameter into a string constant.
- This operator may be used only in a macro that has a specified argument or parameter list. For example:

```
#include <stdio.h>

#define message_for(a, b) \
 printf(#a " and " #b ": We love you!\n")

int main(void)
{
 message_for(Carole, Debra);
 return 0;
}
```

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

Carole and Debra: We love you!

# PREPROCESSOR OPERATORS

## ○ Token Pasting (##)

- The token-pasting operator (##) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token. For example:

```
#include <stdio.h>

#define tokenpaster(n) printf ("token" #n " = %d", token##n)

int main(void)
{
 int token34 = 40;

 tokenpaster(34);
 return 0;
}
```

- When the above code is compiled and executed, it produces the following result: **token34 = 40**
- How it happened, because this example results in the following actual output from the preprocessor:

```
printf ("token34 = %d", token34);
```

- This example shows the concatenation of token##n into token34 and here we have used both **stringize** and **token-pasting**.

# PREPROCESSOR OPERATORS

## ○ The defined() Operator

- The preprocessor **defined** operator is used in constant expressions to determine if an identifier is defined using `#define`. If the specified identifier is defined, the value is true (non-zero). If the symbol is not defined, the value is false (zero). The defined operator is specified as follows:

```
#include <stdio.h>

#if !defined (MESSAGE)
 #define MESSAGE "You wish!"
#endif

int main(void)
{
 printf("Here is the message: %s\n", MESSAGE);
 return 0;
}
```

- When the above code is compiled and executed, it produces the following result: **Here is the message: You wish!**



# PARAMETERIZED MACROS

## Parameterized Macros

- One of the powerful functions of the CPP is the ability to simulate functions using parameterized macros. For example, we might have some code to square a number as follows:

```
int square(int x) {
 return x * x;
}
```

- We can rewrite above code using a macro as follows:

```
#define square(x) ((x) * (x))
```



## Parameterized Macros

- Macros with arguments must be defined using the **#define** directive before they can be used. The argument list is enclosed in parentheses and must immediately follow the macro name. Spaces are not allowed between the macro name and open parenthesis. For example:

```
#include <stdio.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void)
{
 printf("Max between 20 and 10 is %d\n", MAX(10, 20));
 return 0;
}
```

- When the above code is compiled and executed, it produces the following result: **Max between 20 and 10 is 20**