

# C Programming

- ▶ Introduction
- ▶ Merits of C-Language
- ▶ Characteristics of C-Language
- ▶ Application areas of C

# Introduction

- ▶ 'C' is a programming language developed by Dennis Ritchie in the 1972 at Bell Laboratories, USA.
- ▶ C is a powerful, flexible, portable and elegantly structured programming language.
- ▶ C is undoubtedly the most widely used general purpose language today in operating system and embedded system development.
- ▶ C considered to be a middle level language since it has the features of both the low level and the high level languages.
- ▶ Because it was designed for relatively good programming efficiency as compared to low level language and relatively good machine efficiency as compared to high level language.

# Merits of C

- ▶ C is a general purpose and structured programming language.
- ▶ It is standardized and system independent.
- ▶ It has limited data types with great efficiency.
- ▶ C contains a powerful instruction set for manipulating of data and modern methods of coding loops.
- ▶ It was designed to allow the users to add functions to the language. About 150 functions are available in built for use by the C programmer.

# Characteristics

- ▶ Simple and Clear.
- ▶ Structure approach
- ▶ Portable
- ▶ Case Sensitive
- ▶ Lesser keywords
- ▶ Easily available compiler
- ▶ Modular
- ▶ Easy error detection
- ▶ System programming
- ▶ Compact Code
- ▶ Availability of recursive functions.
- ▶ Availability of memory management

# Application Areas of C

- ▶ System programming that includes writing software for operating systems, language compilers, language translators, assemblers, text editors, linkers, loaders, device drivers etc.
- ▶ Network software that is network drivers to implement different communication protocols.
- ▶ Graphics programming that includes writing software for graphical user interface (GUI), scientific visualization and presentation graphics.
- ▶ Modern programs and database.

# Character Set

The character set of C are grouped into the following categories:

1. Letters (Uppercase A ..... Z, Lowercase a ..... z)
2. Digits (All decimal digits 0 ..... 9)
3. Special characters (., :, %, &, #, <, >, +, -, =, \*, ! etc)
4. White spaces (blank space, horizontal tab, carriage return, new line, form feed)

# Tokens

- ❑ The smallest individual units in the C program are known as C tokens.
- ❑ C has six types of tokens and program are written using these tokens and the syntax of the languages.

## Some C Tokens are:

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

# Keywords

- ❑ Keywords are the words which have been assigned specific meaning in the context of C Language programs.
- ❑ All keywords have fixed meaning and these meanings cannot be changed.
- ❑ Keywords serve as basic building blocks for program statements.
- ❑ There are 32 keywords in C languages which are given below:

<b>auto</b>	<b>continue</b>	<b>enum</b>	<b>if</b>	<b>short</b>	<b>switch</b>	<b>volatile</b>
break	default	extern	int	signed	typedef	while
case	do	float	long	sizeof	union	Char
double	for	register	static	unsigned		



# Identifiers

- ❑ Identifiers refer to the names of variables, functions, arrays, structures, unions and enumerations.
- ❑ These are user-defined names and consists of a sequence of letters and digits.
- ❑ Both uppercase and lowercase letters are permitted, although lowercase letters are commonly used.
- ❑ The underscore character is also permitted in identifiers.
- ❑ Eg:- `int num, char name[20];`
- ❑ `Void message(); struct emp {.....}`

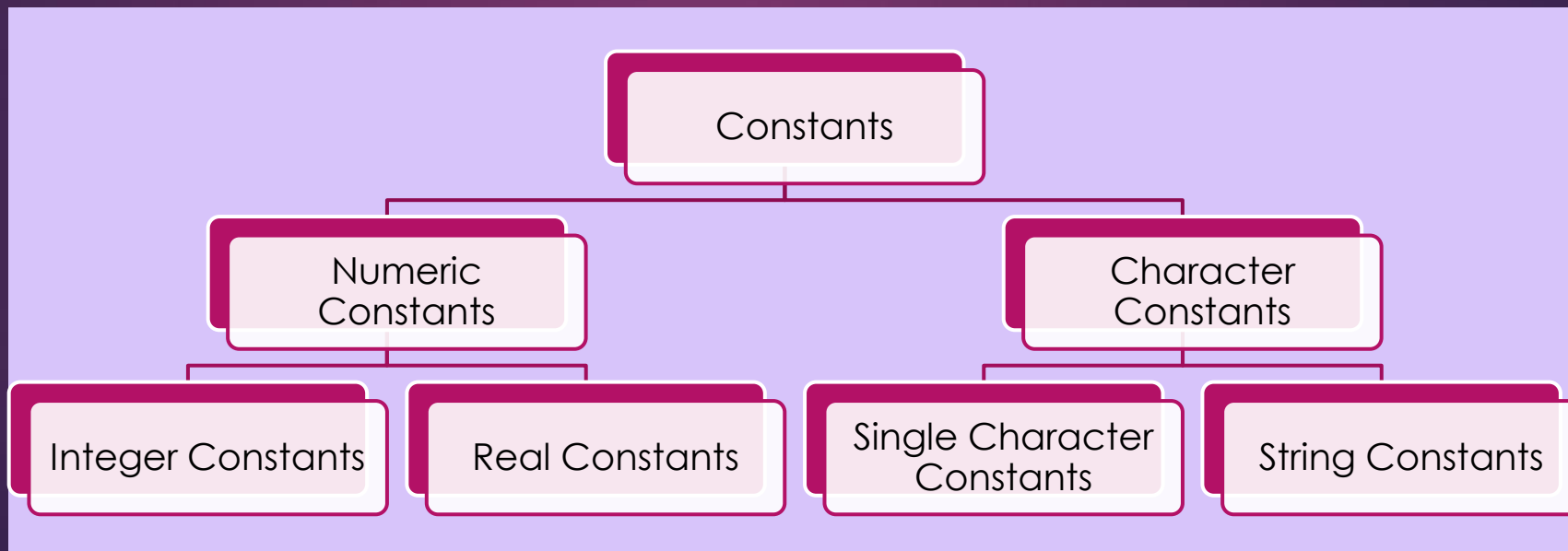
# Rules for Identifiers

- ❑ First character must be an alphabet ( or underscore)
- ❑ Must consist of only letter, digits or underscore..
- ❑ Only first 31 characters are significant (by ANSI).
- ❑ Can not use a keyword.
- ❑ Must not contain white space.

Invalid	Valid
1number total-amount std name	number1 total_amount std_name

# Constants

- ❑ Constants refer to fixed values that do not change during the execution of a program.
- ❑ C supports several types of constants as written below:-



# Integers

- ❑ An integer constant refers to a sequence of digits.
- ❑ There are three types of integers, namely,
  - ❑ Decimal Integers  
(All positive / negative combination of 0-9 digits numbers without decimal point)
  - ❑ Octal Integers  
(All positive / negative combination of 0-7 digits numbers, without decimal point)
  - ❑ Hexadecimal Integer  
(All positive / negative combination of 0-9 and A,B,C,D,E, F digits numbers without decimal point)

## Examples:-

- ❑ Decimal Integers :- 328, -893, 0, 87514, +56
- ❑ Octal Integers :- 037, 0, 0654, 021
- ❑ Hexadecimal Integers :- x037, X0654

# Real

- ❑ Real constants are shown in decimal notation, having a whole number followed by a decimal point and the fractional part.
- ❑ It is possible to omit digits before the decimal point or digits after the decimal point.
- ❑ A real number may also be expressed in exponential (or scientific notation).

215.65             $\rightarrow$     2.1565e2 or 2.1565 multiply by  $10^2$ .

6589.0             $\rightarrow$     0.6589e4 or 0.6589 multiply by  $10^4$ .

-0.00000012     $\rightarrow$     -1.2E-7 or -1.2 multiply by  $10^{-7}$

# Single

- ❑ A single character constant (or simply character constant) contains a single character enclosed within a pair of single quote marks.

'5', 'C', ':'

Note that the character constant '5' is not a same as number 5.

- ❑ Character constants have integer values known as ASCII values. For example, the statement.

```
printf("%d", 'a');
```

Would print the number 97, the ASCII value of the letter a. Similarly, the statement.

```
printf("%c", '97');
```

# String

- ❑ A string constant is a sequence of characters enclosed in double quotes.
- ❑ The character may be letters (small or capital), numbers, special characters and blank space.
- ❑ Examples are:-  
"Hello", "1989", "Very Good", "5+7", "N", "#.....@"

# Escaping Sequence

Constants	Meaning
\a	Audible alert
\b	Back space
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\'	Single quote
\"	Double quote
\?	Question mark
\\	Back slash



# Variable

- ❑ A variable is a data name that may be used to store a data value.
- ❑ A variable may take different values at different times during execution.
- ❑ A specific location or address in the memory is allocated for each variable and the value of the variable is stored in that location.
- ❑ Examples are:-

`Int emp_age, char choice, float total_amt`

- ❑ Rules for constructing variable name:-
- ❑ Variable name may be a combination of alphabets, digits or underscores.
- ❑ ANSI standard recognizes a length of 31 characters.
- ❑ First character must be an alphabet.
- ❑ No commas or blank spaces are allowed.
- ❑ It should not be a keyword.

# Variable Declaration and Initialization

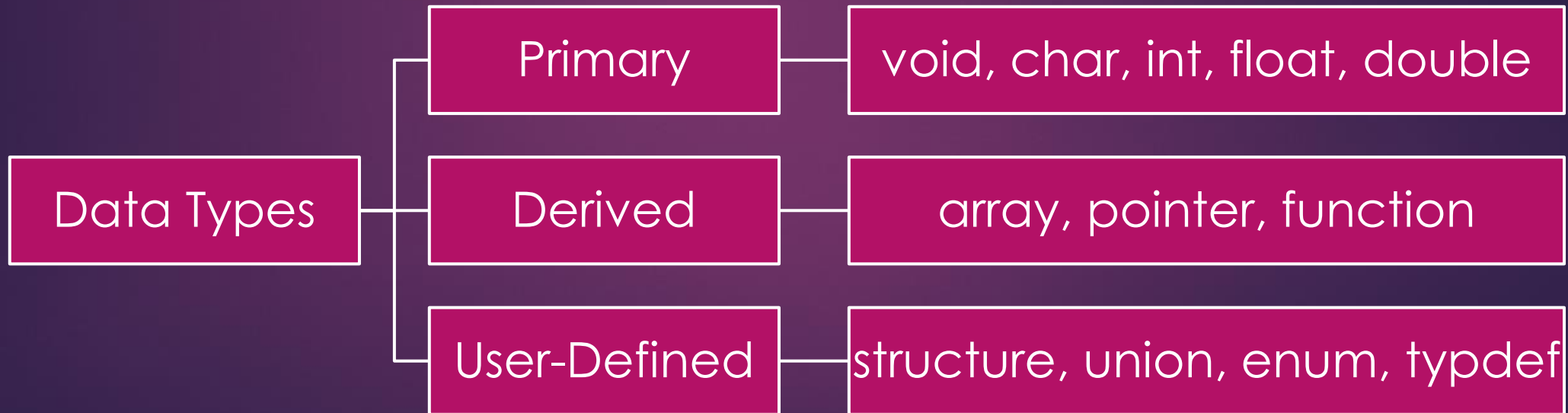
- ❑ Variable Declaration:-
- ❑ Syntax:-
- ❑ `<data_type> <variablename>;`
- ❑ `<data type> <list of variable>;`
- ❑ Eg:-
- ❑ `int number;`
- ❑ `float a, b, c;`
- ❑ Initialization of variable :- To give the first value at the time of declaration.
- ❑ Syntax:-
- ❑ `<data_type> <var_name> = <value>;`
- ❑ Eg :-
- ❑ `Int num = 20;`

# Data Types and Modifiers

- ❑ Data type is used to declare variables or functions of different types.
- ❑ The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.
- ❑ C language is rich in its data types. The variety of data types available allow the programmer to select the type appropriate to the needs of the application as well as the machine.

# Data Types

- ❑ ANSI C supports three classes of data types:
  - 1) Primary (or fundamental or built-in) data types.
  - 2) Derived data types
  - 3) User-defined data types



# Data Types

- ❑ All C language compilers support five fundamental data types:

1. char	Used to store a single character belonging to the defined character set of C language.
2. int	Used to store signed integer, positive or negative.
3. float	used to store real numbers with single precision (six digits after decimal points).
4. double	Stores real numbers with double precision, that is twice the storage space required by float.
5. void	Used to specify empty set containing no value.

# Data Types

- ❑ The following table shows the data types, storage space, format character and range of data type.

Data type	Storage Space	Formatting Specifiers	Range
char	1 byte	%c	ASCII character set
int	2 byte	%d	-32768 to +32767
float	4 byte	%f	$3.4 \times 10^{38}$ to $+3.4 \times 10^{38}$
double	8 byte	%f	$1.7 \times 10^{308}$ to $+1.7 \times 10^{308}$
void	0 byte	-	-

# Modifiers / Qualifiers

- ❑ The basic data types may have modifiers preceding them to indicate special properties of the variable being declared.
- ❑ These modifiers change the meaning of the basic data types to suit the specific needs.
- ❑ These modifiers are unsigned, signed, long and short.
- ❑ It is also possible to give these modifiers in combination, that is unsigned long int.

# Primary Data Types

## Integral Type

### Integer

#### Signed

int  
short int  
long int

#### Unsigned

unsigned int  
unsigned short int  
unsigned long int

### Character

#### Char

Signed char  
Unsigned char

## Floating Point Type

float  
double  
long double

**void**



# Data Types with Modifiers

Data type	Size	Range
Char or signed char	8	-128 to +127
Unsigned char	8	0 to 255
Int or signed int	16	-32768 to +32767
Unsigned int	16	0 to 65535
Short int or signed short int	8	-128 to +127
Unsigned short int	8	0 to 255
Long int or signed long int	32	-2147483648 to +2147483648
Unsigned long int	32	0 to 4294967295
Float	32	3.4E-38 to 3.4E+38
Double	64	1.7E-308 to 1.7E+308
Long double	80	3.4E-4932 to 3.4E+4932

# Operators

- ❑ An operator is a symbol that tells the computer to perform certain mathematical or logical manipulation.
- ❑ Operators are used in programs to manipulate data and variables. They usually form a part of the mathematical or logical expressions.

# Operators

❑ C operators can be classified into a number of categories:

- |                         |                          |
|-------------------------|--------------------------|
| 1) Arithmetic Operators | 5) Increment & Decrement |
| 2) Relational Operators | 6) Conditional Operator  |
| 3) Logical Operators    | 7) Bitwise Operators     |
| 4) Assignment Operators | 8) Special Operators     |

# Arithmetic Operators

- Arithmetic operators are the operators which are used to perform various arithmetic calculations.

Operator	Meaning	Example
+	Addition or unary plus	$5+5 = 10$ , $6+7 = 13$
-	Subtraction or unary minus	$5-5 = 0$ , $45-30 = 15$
*	Multiplication	$5*5 = 25$ , $8 * 3 = 24$
/	Division	$12/6 = 2$ , $10/3 = 3.333333$ (simple di) $5/5 = 1$ , $10/3 = 3$ (integer division)
%	Modulo division (Remainder after integer division)	$12\%7 = 5$ , $5\%3 = 2$ , $4\%7=4$ (Modulo division %) $= a-(a/b)*b$ where, a is numerator, b is denominator and (a/b) is an integer division.

# Relation Operators

- ❑ Relational operators are used to compare two operands to see whether they are equal to each other, unequal, or one is greater or lesser than the other.
- ❑ The operands can be variables, constants or expressions and the result is a numerical value. (true-1, false)

Operator	Meaning	Example
==	Equal to	5==5 -> 1 true, 4==5 -> 0 false
!=	Not equal to	5!=5 ->0 false, 4!=5 -> 1 true
<	Less than	5<6 -> 1 true, 6<5 -> 0 false
<=	Less than or equal to	5<=5 -> 1 true, 6<=5 -> 0 false
>	Greater than	5>6 -> 0 false, 6>5 -> 0 false
>=	Greater than equal to	5>=5 ->1 true, 6>=5 -> 1 true, 5>=6 ->0 false

# Logical Operators

- Logical operators are used to combine two or more relational expressions.

Operator	Meaning	Example
&&	Logical AND	$9 > 3 \ \&\& \ 9 > 5 \rightarrow 1$ true, $4 == 5 \rightarrow 0$ false
	Logical OR	$5 > 8 \    \ 5 > 4 \rightarrow 1$ true, $4 != 5 \rightarrow 1$ true
!	Logical NOT	$!(5 < 6) \rightarrow 0$ true, $!(5 > 6) \rightarrow 1$ true

- The result of logical AND (&&) will be true only if both operands are true.
- The result of a logical OR(||) operation will be true if either operand or both are true.
- Logical NOT(!) is used for reversing the value of the expressions

I	II	&&	
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

# Assignment Operators

- ❑ Assignment operator is used to assign the result of an expression to a variable. The most commonly used assignment operator is (=).
- ❑ An expression with assignment operator is of the following form:
  - ❑ **Variable = expression;**
  - ❑ **Example :-**
    - ❑ `Amt = 3000;`
    - ❑ `Age = 25;`
    - ❑ `Roll_number = 228289;`
    - ❑ `Name[20] = 'Canvas'`
    - ❑ `Si = (p*t*r)/100;`

# Assignment Operators (Shorthand)

- ❑ In addition, C has a set of 'shorthand' assignment operators of the form **v op=exp**;
- ❑ Where, **v** is a variable, **exp** is an expression and **op** is a c arithmetic operator.

Operator	Simple Statement	Shorthand
+=	n = n+5 -> 13	n+=5 ->13
-=	n = n-5 -> 3	n-=5 ->3
*=	n = n*2 -> 16	n*=2->16
/=	n = n/4 -> 2	n/=4->2
%=	n = n%5 -> 3	N%=5 -> 3

Where n is an integer variable, with value 8.



# Increment & Decrement Operators

- ❑ C has two very useful operators (++) and (--) called increment and decrement operators respectively.
- ❑ These operators are called unary operators as they require only one operand and should necessarily be a variable not a constant.
- ❑ The increment operator (++) adds one to the operand while the decrement operators (--) subtracts one from the operand.
- ❑ These operators may be used either before or after the operand. When they are used before the operand, it is termed as prefix, while when used after the operand, they termed as postfix form of operators.

# Increment & Decrement Operators

- ❑ `++a;` or `a++;`
- ❑ `--a;` or `a--;`
- ❑ `++a;` is equivalent to `a=a+1;` (or `a+=1;`)
- ❑ `--a;` is equivalent to `a=a-1;` (or `a-=1;`)
- ❑ While `++a` and `a++` means the same thing when they form statements independently,
- ❑ They behave differently when they are used in expressions on the right-hand side of an assignment statement.

# Increment & Decrement Operators

- ❑ Consider the following:-

```
n = 6;
```

```
y = ++n;
```

- ❑ In this case, the value of **y** and **n** would be **7**.

- ❑ Suppose if, we write the above statement as:

```
n = 6;
```

```
y = n++;
```

- ❑ Then, the value of **y** would be **6** and **n** would be **7**.

- ❑ Cause of postfix form, assignment(=) operator gets the higher priority than **++**, so that **6** will assign to **y** first, than **++** operator evaluates and adds **1** to **n** and finally **y** would be **6** and **n** gets **7**.

# Conditional / Ternary Operator

- ❑ A ternary operator is one which contains three operands. The only ternary operator available in C language is conditional operator pair “?:”. It is of the form.

`exp1 ? exp2 : exp3;`

Where `exp1`, `exp2` and `exp3` are expressions.

- ❑ This operator works as follows: **exp1** is evaluated first. If the result is true then **exp2** is executed, otherwise **exp3** is executed.

- ❑ **Example:-**

`a = 20;`

`b = 30;`

`x = (a > b) ? a : b;`

In this expression value of b will be assign to x.

# Bitwise Operators

- ❑ Bitwise operators are used for manipulation of data at bit level. These operators are used for testing the bits, or shifting them right or left. Bitwise operators may not

Operator	Meaning
&	Bitwise logical AND
	Bitwise logical OR
^	Bitwise logical XOR
<<	Left shift
>>	Right shift
~	One's complement

# Special Operators

- ❑ C language supports some special operators such as:
  - 1) comma operator,
  - 2) sizeof operator,
  - 3) pointer operator (& and \*) and
  - 4) member selection operator (. And ->)

## 1) Comma operator:

- ❑ This operator is used to link the related expressions together.
- ❑ Example:

```
int val, x, y;
```

```
val = (x=10, y=6, x+y);
```

It assigns 10 to x first, then 6 to y, finally sum, x+y to val.

# Special Operators

## 2) Sizeof operator:

- ❑ This operator is a compile time operator and when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable, a constant or a data type qualifier.

- ❑ Example:

`s = sizeof(sum);`

-> size of sum will assign to s in bytes.

`n = sizeof(long int);`

-> 4 will assign to n in bytes.

`k = sizeof(456);`

-> 2 will assign to k in bytes.

- ❑ The sizeof operator is normally used to determine length of array and structures when their sizes are not known to the programmer.
- ❑ It is also used to allocate the memory space dynamically to variables during execution of program.

# Basic Structure of C Program

Document Section		
Link Section		
Definition Section		
Global Declaration Section		
main() function section		
{ Declaration part; Execution part; }		
Subprogram Section		
	Function 1	(User Defined Function)
	Function 2	
	.....	
	Function n	



# Documentation Section

Document Section		
Link Section		
Definition Section		
Global Declaration Section		
main() function section		
{ Declaration part; Execution part; }		
Subprogram Section		
	Function 1	(User Defined Function)
	Function 2	
	.....	
	Function n	

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

Example:

```
/* This is my first C Program */
```

```
// Simple Interest Program
```

```
/* Program to find greater number */
```

# Link Section

Document Section		
Link Section		
Definition Section		
Global Declaration Section		
main() function section		
{ Declaration part; Execution part; }		
Subprogram Section		
	Function 1	(User Defined Function)
	Function 2	
	.....	
	Function n	

#include<.....>

Example:

#include<stdio.h>

#include<conio.h>

#include<math.h>

# Definition Section

Document Section		
Link Section		
Definition Section		
Global Declaration Section		
main() function section		
{ Declaration part; Execution part; }		
Subprogram Section		
	Function 1	(User Defined Function)
	Function 2	
	.....	
	Function n	

#define cost value;

Example:

#define PI 3.14

#define SIZE 20

#define FIX 100

# Global Declaration Section

Document Section		
Link Section		
Definition Section		
Global Declaration Section		
main() function section		
{ Declaration part; Execution part; }		
Subprogram Section		
	Function 1	(User Defined Function)
	Function 2	
	.....	
	Function n	

Data\_type variable;

## Example:

```
int counter;  
char choice;  
float net_amount;
```

## User defined function declaration

```
Int sum(int a, int b);  
Void message();
```

# main() Function Section

Document Section		
Link Section		
Definition Section		
Global Declaration Section		
main() function section		
{ Declaration part; Execution part; }		
Subprogram Section		
	Function 1	(User Defined Function)
	Function 2	
	.....	
	Function n	

**Example:**

```
main()  
{  
    int n;  
    n = 10;  
    printf("Number=%d",n);  
    return 0;  
}
```

# Sub-program Section

Document Section		
Link Section		
Definition Section		
Global Declaration Section		
main() function section		
{ Declaration part; Execution part; }		
Subprogram Section		
	Function 1	(User Defined Function)
	Function 2	
	.....	
	Function n	

**Example:**

```
void message()  
{  
    printf("This is user-defined function");  
}
```

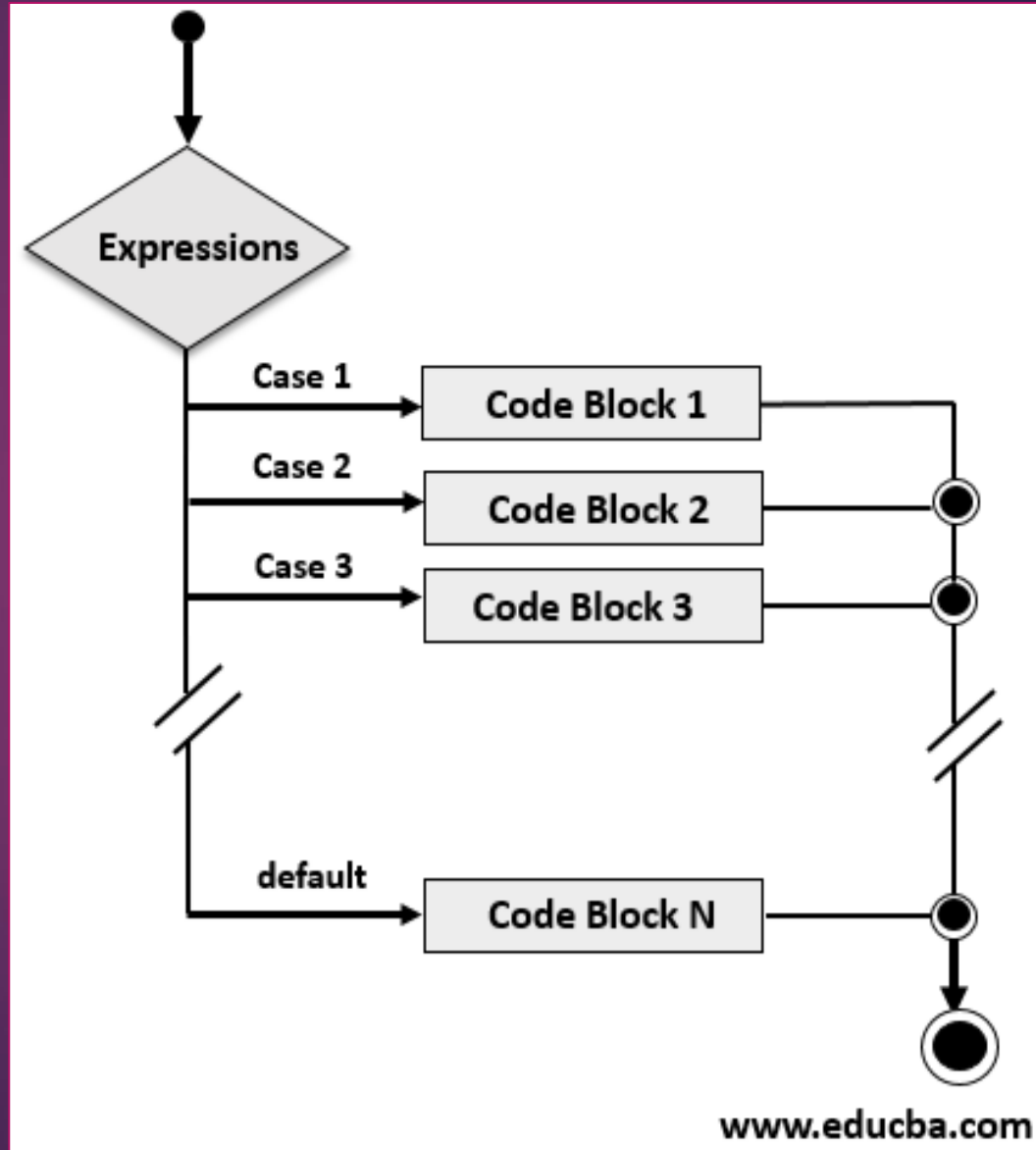
```
int sum(int a, intb)  
{  
    return (a+b);  
}
```

# Switch Case Statement

- ❑ The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.
- ❑ 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 case.

# Switch Case Statement

```
Syntax,  
Switch (<exp>  
{  
    case <val-1>:  
        code-block-1;  
        Break;  
    case <val-1>:  
        code-block-1;  
        Break;  
    .....  
    .....  
    .....  
    case <val-n>:  
        code-block-n;  
        Break;  
    default:  
        code-block;  
        Break;  
}
```





# Decision Making and Looping

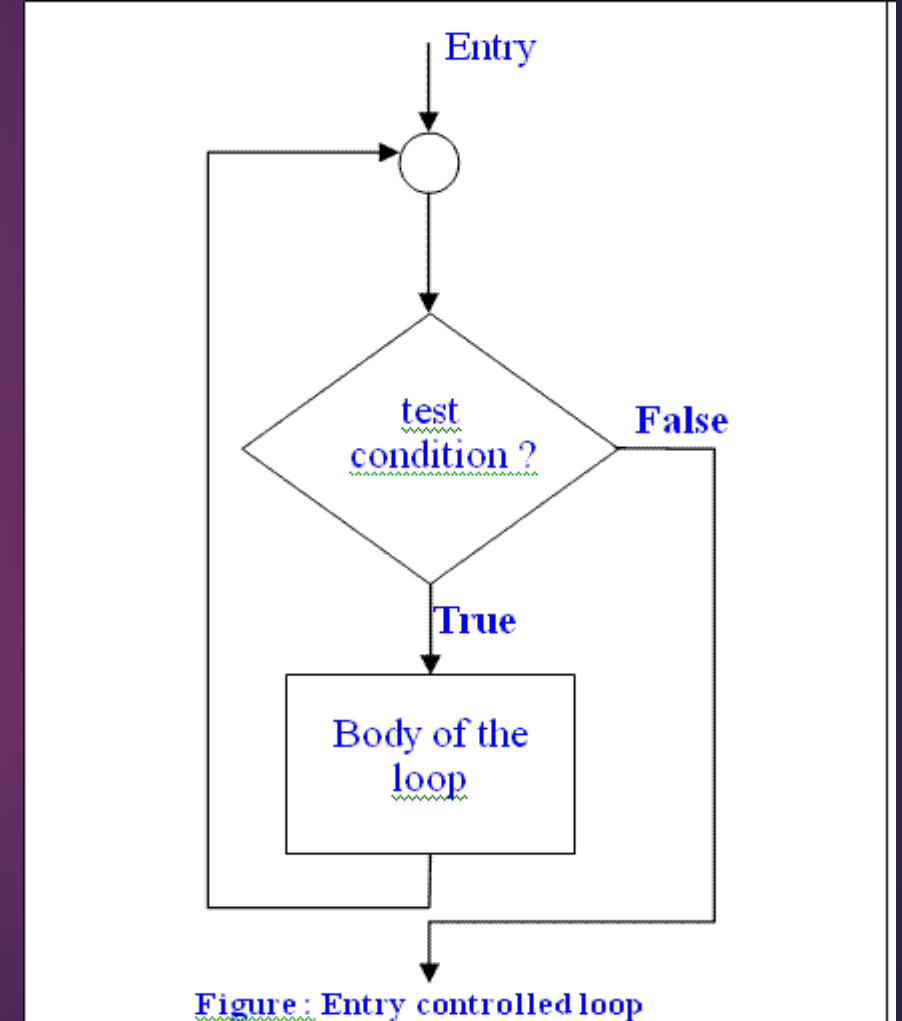
- ❑ The process of repeatedly executing a block of statements is known as looping.
- ❑ The statements in the block may be executed any number of times, from zero to n number.
- ❑ In looping, a sequence of statements are executed until some conditions for the termination of the loop are satisfied.

# Decision Making and Looping

- ❑ A loop consists of two segments:-
  - 1) Control statement
  - 2) Body of the loop
- ❑ The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of loop.
- ❑ Depending on the position of the control statement in the loop, a loop may be classified either **entry controlled loop** or **exit-controlled loop**.

# Decision Making and Looping

- ❑ **Entry Controlled Loop**
- ❑ In entry-controlled loop, the control conditions are tested before the start of the loop execution.
- ❑ If the conditions are not satisfied, then the body of the loop will not be executed.



# Decision Making and Looping

## ❑ Exit Controlled Loop

- ❑ In exit-controlled loop, the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time.

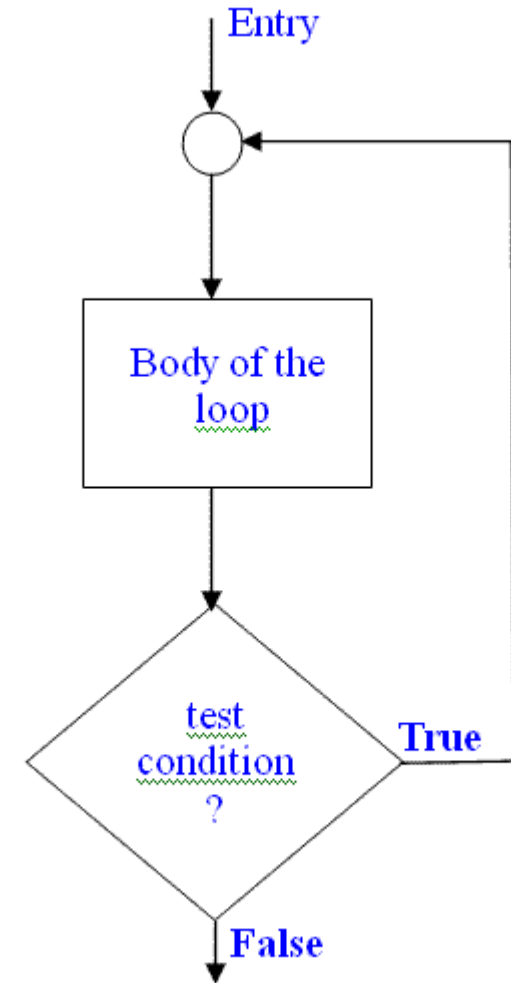


Figure : Exit controlled loop

# Decision Making and Looping

- ❑ Generally, a looping process includes the following four steps:-
  - 1) Setting and initialization of loop counter.
  - 2) Test specified condition for execution of loop.
  - 3) Execution of the statements in the loop.
  - 4) Increment / decrement of loop counter.

# Decision Making and Looping

□ C language provides three constructs for performing loop operations:-

- 1) while loop
- 2) do while loop
- 3) for loop

# while loop

## □ Syntax:-

```
<initialization>;  
while(<test_condition>)  
{  
    <body of the loop>  
}
```

# do while loop

## □ Syntax:-

```
<initialization>;
```

```
do
```

```
{
```

```
    <body of the loop>
```

```
}
```

```
while(<test_condition>;
```



# for loop

## □ Syntax:-

```
for (<intiliazation>;<test_condition>;<increment / decrement>)  
{  
    <body of the loop>  
}
```

# Input / Output Functions

- ❑ C language provides us console us console input / output functions that allow us to :-
  - 1) Read the input from the keyboard by the user and
  - 2) Display the output at the user screen.
- ❑ Input / Output functions in C programming falls into two categories :-
  - 1) Formatted input/output (I/O) functions.
  - 2) Unformatted input/output (I/O) functions.

# Formatted Input / Output Functions

- ❑ Formatted I/O functions allow to supply input or display output in user's desired format.
- ❑ Formatted input and output functions contain format specifier in their syntax. **Eg. %d, %f, %s, %c**
- ❑ They are used for storing data more user friendly.
- ❑ They are used with all data types.
- ❑ printf() and scanf() are examples for formatted input and output functions.

# Unformatted Input / Output Functions

- ❑ These functions do not allow to supply input or display output in user desired format.
- ❑ Unformatted input and output functions do not contain format specifier in their syntax.
- ❑ They are used for storing data more compactly.
- ❑ They are used mainly for character and string data types.
- ❑ **Examples:-**  
`getch(), getche(), getch(),`  
`putch(), putchar(), gets(), puts()`

# printf() and scanf()

- ❑ The printf() and scanf() functions are used for output and input respectively in C language. Both functions are inbuilt library functions, defined in stdio.h (header file);
- ❑ **Syntax :- printf("format string", argument\_list);**
- ❑ Format string is also known as control string and can be a constant message or format specifier that starts with % sign or can contain both.
- ❑ Argument list can be one or more comma separated variable list. \*can be optional

# scanf()

- ❑ **Syntax :-** `scanf("format string", argument_list);`
- ❑ Format string is also known as control string and has format specifier that starts with % sign.
- ❑ Argument list can be one or more comma separated variable list with address of operator or ampersand.
- ❑ Examples:-
- ❑ `printf("Hello C Languages World");`
- ❑ `printf("Amount=%d", amt);`
- ❑ `scanf("%d", &age);`

# Unformatted Input / Output Functions

- 1) Character Input / Output Functions
- 2) String Input / Output Functions
  - ❑ `#include <conio.h>`
  - ❑ Console input / output header file

# Character Input Functions

- ❑ **getch(), getche(), getchar()**
- ❑ `getch()` – Reads a single character from the user at the console, without echoing it.
- ❑ `getche()` – Reads a single character from the user at the console, and echoing it.
- ❑ `getchar()` – Reads a single character from the user at the console, and echoing it but needs an Enter key to be pressed at the end.



# Character Output Functions

- ❑ **putch(), putchar()**
- ❑ `putch()` – Displays a single character value at the console.
- ❑ `putchar()` – Displays a single character value at the console.

# Character Output Functions

## ❑ **putch(), putchar()**

- ❑ **putch()** – Displays a single character value at the console.
- ❑ **putchar()** – Displays a single character value at the console.

# Unformatted Input / Output Functions

## ❑ String Input Functions

- ❑ `gets()` – Reads a single string entered by the user at the console.

## ❑ String Output Functions

- ❑ `puts()` – Displays a single string's value at the console.

# Unformatted Input / Output Functions

## ❑ Syntax:-

- ❑ `<char_var> = getch();`
- ❑ `<char_var> = getche();`
- ❑ `<char_var> = getchar();`
- ❑ `putch(<char_var>);`
- ❑ `putchar(<char_var>);`
- ❑ -----
- ❑ `gets(<str_var>);`
- ❑ `puts(<str_var>);`

# Array

- ❑ An array is a variable that can store multiple values.
- ❑ An array is a group of data items of same data type that share a common name.
- ❑ An array is a collective name given to a group of similar quantities. Each member in the group is referred to by its position in the group.
- ❑ Arrays are allotted the memory in a strictly contiguous fashion.

# Types of Array

## ❑ Single / One Dimensional Array

The simplest array is one dimensional array which is simply a list of variables of same data type.

28	26	46	30	44
----	----	----	----	----

## ❑ Double / Two Dimensional Array

An array of one dimensional arrays is called a double dimensional array.

11	23	56	19
55	46	12	99
61	30	74	18

# Declaration of Array

## ❑ Single / One Dimensional Array

### Syntax,

```
data_type array_name [size];
```

### Example,

```
int Age[5];
```

Age[0]	Age[1]	Age[2]	Age[3]	Age[4]
28	26	46	30	44
62345	62347	62349	62351	62353

# Declaration of Array

## ❑ Double / Two Dimensional Array

### Syntax,

```
data_type array_name [row][col];
```

### Example,

```
int mat[3][3];
```

mat[0][0]	mat[0][1]	mat[0][2]	mat[0][3]
mat[1][0]	mat[1][1]	mat[1][2]	mat[1][3]
mat[2][0]	mat[2][1]	mat[2][2]	mat[2][3]

11	23	56	19
55	46	12	99
61	30	74	18



# Memory Allotment in Array

m[0][0]	m[0][1]	m[0][2]	m[0][3]
m[1][0]	m[1][1]	m[1][2]	m[1][3]
m[2][0]	m[2][1]	m[2][2]	m[2][3]

## □ Row Major Order

m[0][0]	m[0][1]	m[0][2]	m[0][3]	m[1][0]	m[1][1]	m[1][2]	m[1][3]	m[2][0]	m[2][1]	m[2][2]	m[2][3]
6210	6212	6214	6216	6218	6220	6222	6224	6226	6228	6230	6232

## □ Column Major Order

m[0][0]	m[1][0]	m[2][0]	m[0][1]	m[1][1]	m[2][1]	m[0][2]	m[1][2]	m[2][2]	m[0][3]	m[1][3]	m[2][3]
6210	6212	6214	6216	6218	6220	6222	6224	6226	6228	6230	6232

# Access Array Elements

- ❑ **To store in specific location**

```
ar[4] = 65;
```

- ❑ **For Displaying**

```
printf("%d", ar[4]);
```

- ❑ **For Calculations**

```
ar[0] = ar[0]+ar[4]+5;
```

# String Function

- ❑ **Strings are defined as an array of characters.**
- ❑ Examples :- “India”, “10:30”, “abc@gmail.com”
- ❑ strlen()
- ❑ strcpy()
- ❑ strcmp()
- ❑ strcat()
- ❑ strupper()
- ❑ strlower()
- ❑ strrev()

# String Function

- ❑ `strlen()` :- calculates the length of a string.

Syntax:- `<int_var> = strlen(<str>);`

- ❑ `strupr()` :- converts lower case letters in a string to upper case.

Syntax:- `strupr(<str>);`

- ❑ `strlwr()`:- converts upper case letters in a string to lower case.

Syntax:- `strlwr(<str>);`

# String Function

❑ `strrev()` :- Reverses a string.

**Syntax :- `strrev(<str>);`**

❑ `strcpy()` :- Copies one string into another.

**Syntax :- `strcpy(<dest_str>, <src_str>);`**

❑ `strcmp()` :- Compares one string to another.

**Syntax :- `strcmp(<str1>, <str2>);`**

❑ `strcat()` :- Appends one string to another.

**Syntax :- `strcat(<dest_str>, <src_str>);`**

# User Defined Function

## ❑ Definition:

Function is basically a set of statements that takes inputs, perform some computation and produces output.

## ❑ Syntax:

```
<return_type> <function_name>(set_of_inputs);
```

# Why Functions

There are two important reasons of why we are using functions:

- ❑ **Reusability**

Once the function is defined, it can be reused over and over again.

- ❑ **Abstraction**

If you're just using the function in your program then you don't have to worry about how it works inside.

**Example,**

scanf function

# Function Declaration / Prototype

Function declaration (also called **function prototype**) means declaring the properties of a function to the compiler.

**Example,**

```
int fun(int a, char c);
```

**Properties:**

Name of function : fun

Return type of function : int

Number of parameters : 2

Type of parameter 1 : int

Type of parameter 2 : char




# Important Takeaway

- ❑ It is not necessary to put the name of the parameters in function prototype.

**Example,**

```
int fun(int var1, char var2);
```

Not necessary  
to mention  
these names



- ❑ Not necessary but it is preferred to declare the function before using it.

# Function Definition

❑ Function definition consists of block of code which is capable of performing some specific task.

❑ **Example,**

```
int add(int a, int b)
{
    int sum;
    sum = a + b;
    return sum;
}
```

# How Function Works

```
int add(int, int);
```

Don't forget to mention the prototype of a function.

**Recall:** There is no need to mention names of the parameters.

```
int main()
```

```
{
```

```
    int m = 20, n = 30, sum;
```

```
    sum = add(m, n);
```

```
    printf("Sum is %d", sum);
```

```
}
```

```
int add(int a, int b);
```

```
{
```

```
    return (a+b);
```

```
}
```

This is the way we call a function.

**Note:** while calling a function the return type of the function.

Also we should not mention the data types of the arguments.

This is the way how we define a function.

**Note:** it is important to mention both data type and name of parameters.

# Parameters vs Argument

- ❑ Parameter is a variable in the declaration and definition of the function.
- ❑ Argument is the actual value of the parameter that gets passed to the function.
- ❑ **Parameter** is also called **Formal Parameter** and **Argument** is also called as **Actual Parameter**.

# Functions – call by value and call by reference

## ❑ Actual Parameters:

The parameters passed to a function.

## ❑ Formal Parameters:

The parameters received by a function.

```
add(m, n);
```

```
int add(int a, int b)
{
    return (a+b);
}
```

# Function Call by Value

- ❑ Function call by value is the default way of calling a function in C programming.
- ❑ **Actual parameters / Argument:**  
The parameters that appear in function calls. **Eg** :- sum(a,b);
- ❑ **Formal parameters / Parameters:**  
The parameters that appear in function declarations and definition.  
**Eg**-  
sum(int x, int y)  
sum(int x, int y)  
{  
.....  
.....  
}

# Function Call by Value

```
#include<stdio.h>
int sum(int a, int b);
void main()
{
    int x=10, y=20, s;
    s=sum(x,y);
    printf("Sum = %d", s);
}
void sum(int a, int b)
{
    return a+b;
}
```

## Calling Environment

Actual Parameter / Argument



## Called Environment

Formal Parameter



# Function Call by Value

- ❑ When we pass the actual parameters while calling a function then this is known as function call by value.
- ❑ In this case the values of actual parameters are copied to the formal parameters.
- ❑ Thus operations performed on the formal parameters don't reflect in the actual parameters.



# Function Call by Reference

- ❑ Here both actual and formal parameters refers to same memory location. Therefore, any changed made to the formal parameters will get reflected to the actual parameters.
- ❑ Here instead of passing values, we pass address.

# Structure

- ❑ A structure is a collection of variables referenced under one name providing a convenient means of keeping related information together.
- ❑ A structure is a custom data type which combines different data types to form a new user-defined data types.

# Structure

## ❑ Syntax:-

```
struct <structure_name>
{
    <data_type> member1;
    <data_type> member2;
    .....
    .....
};
```

## ❑ Example:-

```
struct emp
{
    char name [20];
    int code;
    float bs;
};
```

# Create Structure Variable

## ❑ Using Structure name:-

```
struct Emp  
{  
    char name[20];  
    int code;  
    Float bs;  
};  
struct Emp e1, e2;
```

Structure name is optional, we have not used any structure name here.

```
struct  
{  
    char  
    name[20];  
    int code;  
    Float bs;  
}s1;
```

# Accessing Structure members

- ❑ There are two ways to access structure

- ❑ Using **.(dot) operator**

Example,

```
struct Emp e1;  
e1.bs=20000;  
e1.code=1001;
```

- ❑ Using **-> (arrow) operator**

Example,

```
struct Emp e1;  
Struct Emp *p;  
p = &e1;  
p->bs=20000;  
p->code=1001;
```

# Array of Structure Variables

## ❑ Built-in Array

<data-type> <array-name> [size];

int arr[5];

## ❑ With Structure

struct <struct-type> <array-name> [<size>];

struct product pr[5];

# Structure & Functions

- ❑ Passing Structure as Function Argument
- ❑ Structure as Function Return Type

# Passing Structure as Function Argument

## ❑ Syntax,

```
<return-type> <fun-name>(<struct-arg>)  
{  
    .....  
    .....  
}
```

## ❑ Example,

```
void display(struct student st)  
{  
    .....  
    .....  
}
```



# Structure as function return type

## ❑ Syntax,

```
<struct-type> <fun-name>(<arg>)  
{  
    .....  
    return <struct-var>  
}
```

## ❑ Example,

```
struct student display()  
{  
    .....  
    return st;  
}
```

# Union

- ❑ A union is a user-defined type similar to structure but unlike structure, union members share a common memory space.
- ❑ You can define a union with many members, but only one member can contain a value at any given time.
- ❑ Unions provide an efficient way of using the same memory location for multiple-purpose.

## Example: Accessing Union Members

```
#include <stdio.h>

union Job {
    float salary;
    int workerNo;
} j;

int main() {
    j.salary = 12.3;
    // when j.workerNo is assigned a value,
    // j.salary will no longer hold 12.3
    j.workerNo = 100;
    printf("Salary = %.1f\n", j.salary);
    printf("Number of workers = %d", j.workerNo);
    return 0;
}
```

### Output

Salary = 0.0

Number of workers = 100

# Difference between Array & Structure

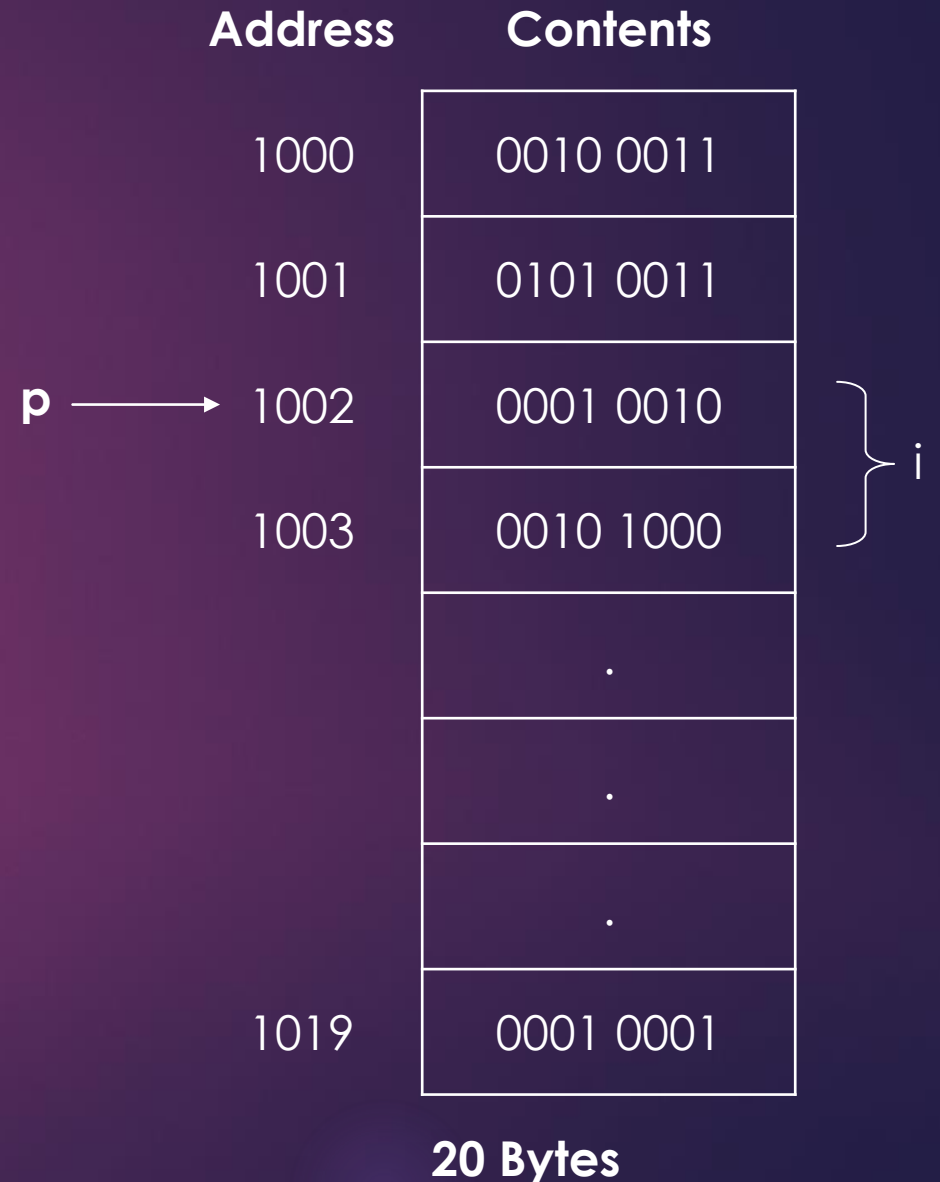
Array		Structure	
1	Array is a built-in data type	1	Structure is a derive data type
2	Array holds the group of same elements under a single name	2	Struture holds the group of different elements under a single name
3	We cannot have array of array	3	We can have array of structure
4	Memory occupied by an array is the multiple of no. of index.	4	Memory occupied structure is sum of individual data type.
5	Array itself is a constant pointer	5	Structure cannot be a pointer
6	Exampl,	6	Example,
	<pre>void main() {     int x[3]={5,7,6},i;     for (i=0;i&lt;3;i++)     {         printf("%d\t", x[i]);     } }</pre>		<pre>void main() {     struct sample {         int x;         float y;         char n[10]; };     sample s = {45,87.55,"abc"};     printf("\n%d, %f, %s", s.x, s.y, s.n); }</pre>

# Difference between Union & Structure

Union		Structure	
1	Memory occupied by union is of highest data type of all.	1	Memory occupied by structure is sum of individual data type.
2	Professionally specialized to interact with hardware	2	Cannot be used to interact with hardware in all aspects.
3	If any of the values of any element has been changed there is direct impact to the other element's values.	3	Every element value's are independent to each other.
4	Memory allocation is performed by sharing the memory with highest data type.	4	Memory allocation of every element is independent to each other thereby the memory allocation is sum of every element.

# Pointer

- ❑ Pointer is special variable that are used to store address rather than value.
- ❑ A pointer is a variable that stores the memory address of another variable as its value.

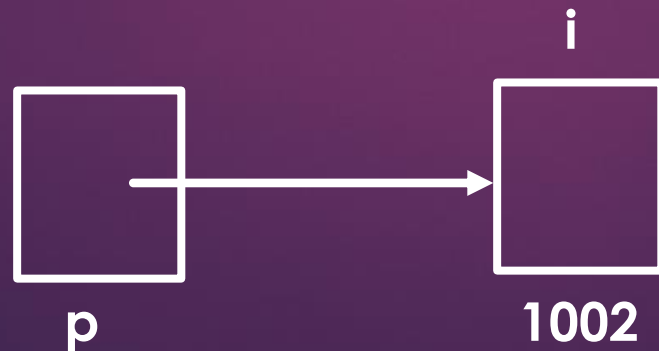


# Pointer

- ❑ Pointer is a special variable that is capable of storing some address.



- ❑ It points a memory location where the first byte is store.



# Declaration of pointer

## ❑ Syntax,

`data_type * pointer_name`

## ❑ Here,

Data type refers to the type of the value that the pointer will point to.

Eg,

`int *ptr;` -----> Points to integer value.

`char *ptr;` -----> Points to character value

`float * ptr;` -----> Points to float value.



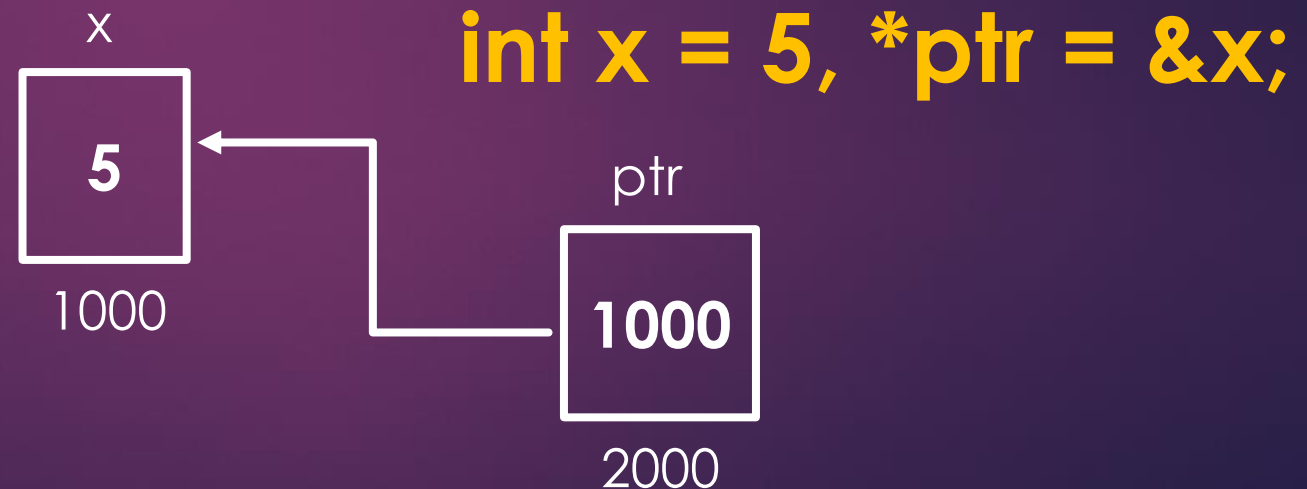
# Initialization of pointer (Need of AddressOf Operator)

- ❑ Simply declaring a pointer is not enough.
- ❑ It is important to initialize pointer before use.
- ❑ One way to initialize a pointer is to assign address of some variable.

❑ **Eg,**

```
int x = 5;  
int *ptr;  
ptr = &x;
```

& => Address of operator



# Value of operator

- Value of operator / indirection operator / dereference operator is an operator that is used to access the value stored at the location pointed by the pointer.

□ **Eg,**

```
int x = 5;
```

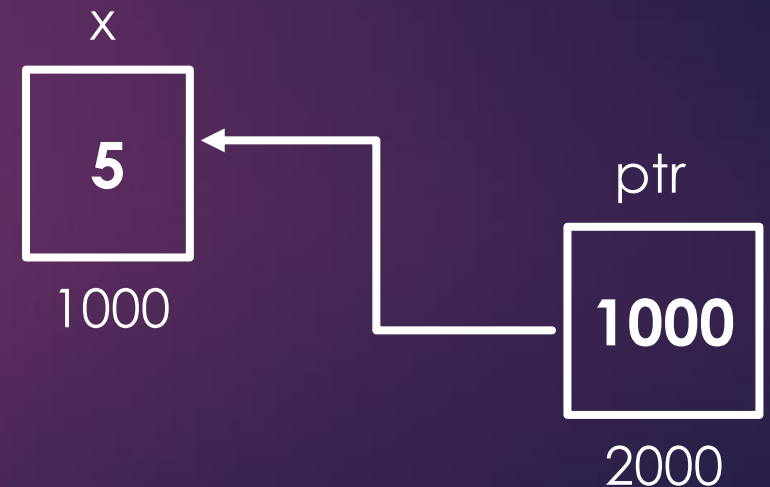
```
int *ptr;
```

```
ptr = &x;
```

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

## Value of Operator,

It says go to the address of object and take what is stored in the object.



## Modification

- We can also change the value of the object pointed by the pointer.

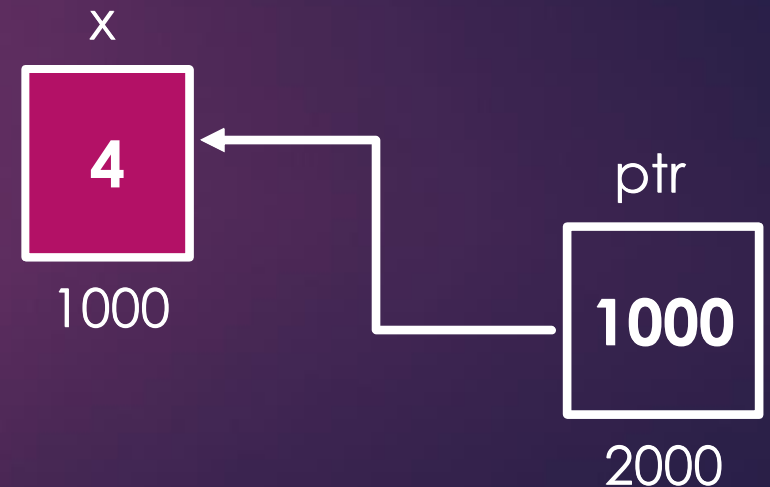
□ Eg,

```
int x = 10;
```

```
int *ptr = &x;
```

```
*ptr = 4;
```

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



# Caution

- ❑ Never apply the indirection operator to the uninitialized pointer.

- ❑ **Eg,**

```
int *ptr;
```

```
printf("%", *ptr);
```

- ❑ **Output,**

Undefined behavior.

# One more .....

- ❑ Assigning value to an uninitialized pointer is dangerous.

- ❑ **Eg,**

```
int *ptr;
```

```
*ptr = 1;
```

- ❑ **Output,**

Segmentation Fault (SIGSEGV)

**Segmentation** fault is caused by program trying to read or write an **illegal** memory.

# Assigning pointer to pointer

- ❑ Assigning value to an uninitialized pointer is dangerous.

- ❑ **Eg,**

```
int *ptr;  
*ptr = 1;
```

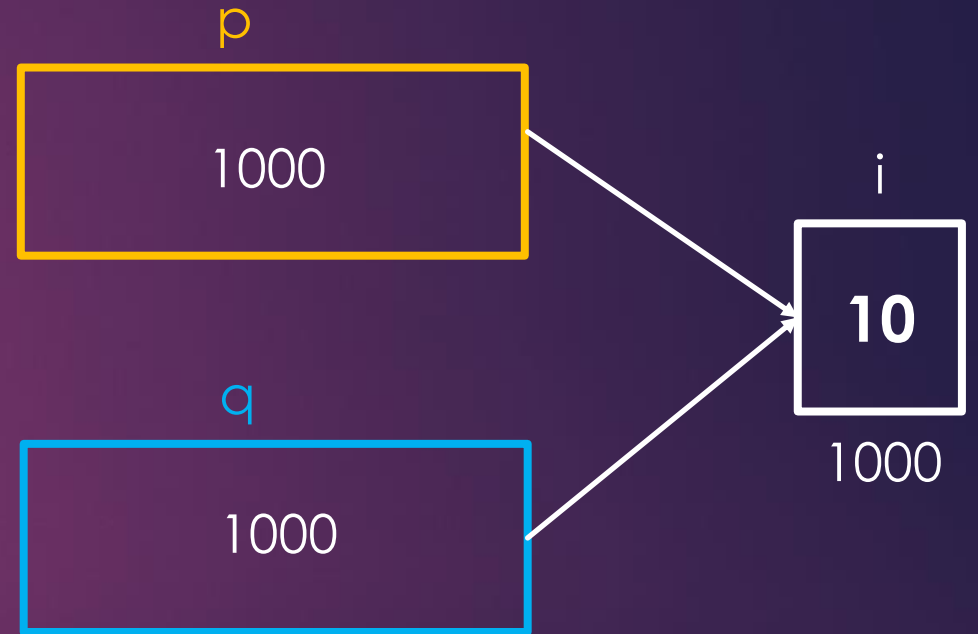
- ❑ **Output,**

Segmentation Fault (SIGSEGV)

**Segmentation** fault is caused by program trying to read or write an **illegal** memory.

# Assigning pointer to pointer

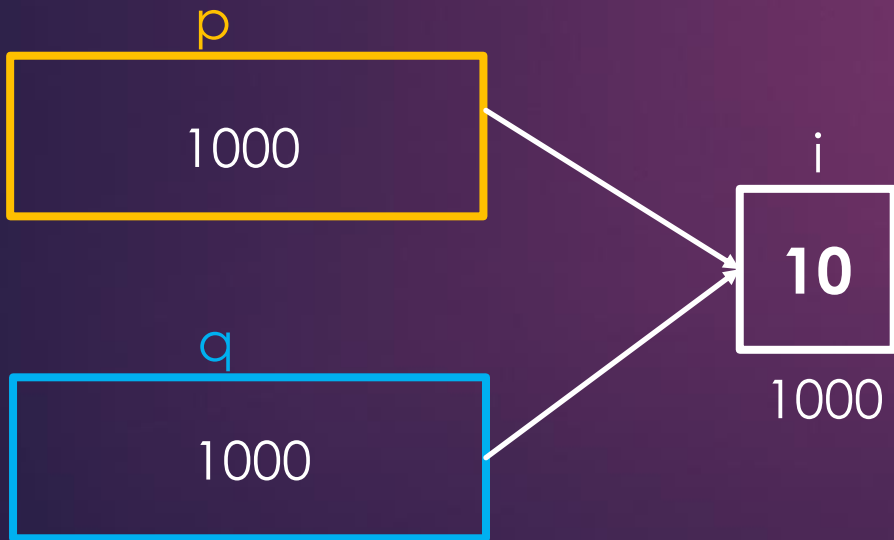
```
int i = 10;  
int *p, *q;  
p = &i;  
q = p;  
printf("%d %d"), *p, *q);
```



Output : 10 10

# Note: $p = q$ is not equal to $*p = *q$

```
int i = 10;  
int *p, *q;  
p = &i;  
q = p;  
printf("%d %d"), *p, *q);
```



```
int i = 10, j = 20;  
int *p, *q;  
p = &i;  
q = &j;  
*p = *q .....?
```



# Pointer to Structure

- ❑ Pointer is a special variable which can store memory address of other variable rather than value.
- ❑ In pointer to structure – declare a pointer variable of structure. **struct stud \*p;**
- ❑ Assign the memory address of a structure variable to pointer. **p = &s;**
- ❑ Use arrow (->) operator in place of dot (.) operator to access structure members. **p->roll = 1001;**

# Pointer to Structure

```
struct stud
{
    char sname[20];
    int roll;
    float marks;
};
```

```
main()
{
    struct stud s;
    struct stud *p;
    p = &s;
    p->roll = 1001;
};
```

# Why do we use pointer?

- ❑ Pointers can improve the efficiency of certain routine involving arrays.
- ❑ Pointers provide the means by which functions can modify their calling arguments.
- ❑ Pointer support dynamic allocation.
- ❑ Pointer provide support for dynamic data structures, such as linked lists and binary trees.

# Working with Files

- ❑ The task of storing data in the form of input or output produced by running C programs in data files, namely, a text file or a binary file for future reference and analysis.
- ❑ A file is a container in computer storage devices used for storing data.
- ❑ A file is nothing but a source of storing information permanently in the form of a sequence of bytes on a disk.
- ❑ The contents of a file are not volatile like the C compiler memory. The various operations available like creating a file, opening a file, reading a file or manipulating data inside a file is referred to as file handling.

# Files that can be handled in C

## ❑ **Text Files / Stream-oriented data files –**

The data is stored in the same manner as it appears on the screen. The I/O operations like buffering, data conversions, etc. take place automatically.

## ❑ **Binary Files / System-oriented data files –**

System-oriented data files are more closely associated with the OS and data stored in memory without converting into text format.

# The contents of a file can be handle in C languages by two different techniques:-

## ❑ Sequential File Access

Sequential Access to a data file means that the computer system reads or writes information to the file sequentially, starting from the beginning of the file and proceeding step by step.

## ❑ Random File Access

On the other hand, Random Access to a file means that the computer system can read or write information anywhere in the data file. This type of operation is also called “Direct Access” because the computer system knows where the data is stored (using Indexing) and hence goes “directly” and reads the data.

# File Operations

- 1) Creation of a new file.
- 2) Opening an existing file.
- 3) Reading data from a file.
- 4) Writing data in a file.
- 5) Closing a file.

# Steps for processing file

- 1) Declare a file pointer variable.
- 2) Open a file using `fopen()` function.
- 3) Process the file using the suitable function.
- 4) Close the file using `fclose()` function.



# File Operation Functions in C

<code>fopen()</code>	Creates a new file for use /Opens a new existing file for you.
<code>fclose()</code>	Closes a file which has been opened for use.
<code>getc()</code>	Reads a character from a file.
<code>putc()</code>	Writes a character to a file.
<code>fprintf()</code>	Writes a set of data values to a file.
<code>fscanf()</code>	Reads a set of data values from a file.
<code>getw()</code>	Reads a integer from a file.
<code>putw()</code>	Writes a integer to the file.
<code>fseek()</code>	Sets the position to a desired point in the file.
<code>ftell()</code>	Gives the current position in the file.
<code>rwind()</code>	Sets the position to the beginning of the file.
<code>fgets()</code>	Read string of characters from a file.
<code>fputs()</code>	Write string of characters to file.
<code>feof()</code>	Detects end-of-file marker in a file

# fopen function

❑ C fopen is a C library function used to open an existing file or create a new file.

❑ **Syntax,**

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

Or

```
FILE *filePointer;
```

```
filePointer = fopen("filePath", "mode");
```

Mode	Description
r	Opens an existing text file.
w	Opens a text file for writing if the file doesn't exist then a new file is created.
a	Opens a text file for appending (writing at the end of existing file) and create the file if it does not exist.
r+	Opens a text file for reading and writing.
w+	Open for reading and writing and create the file if it does not exist. If the file exists then make it blank.
a+	Open for reading and appending and create the file if it does not exist. The reading will start from the beginning, but writing can only be appended.

# C fopen is a C library function used to open an existing file or create a new file.

## □ Syntax,

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

Or

```
FILE *filePointer;
```

```
filePointer = fopen("filePath", "mode");
```

**/\* a program ask name, age & height of a person  
as user wants and store in to file 'data.txt'. \*/**

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

void main(){
    char choice;
    char name[40];
    int age;
    float height;

    FILE *fptr;
    fptr = fopen("data.txt", "w");
    do {
        fflush(stdin);
        printf("Enter Name \n");

        scanf("%s", name);
        printf("Enter Age \n");
        scanf("%d", &age);
        printf("Enter Height \n");
        scanf("%f", &height);
        printf("Any More [Y/N] :");
        fflush(stdin);
        scanf("%c", &choice);
        // choice = getch();
        fprintf(fptr, "%s\t %d\t %f\n", name, age,
height);
    } while (choice=='Y' || choice=='y');
    fclose(fptr);
    getch();
}
```

**/\* a program to display name, age & height of a person  
from previously created file 'data.txt'. \*/**

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main(){
    char choice;
    char name[40];
    int age;
    float height;

    FILE *fptr;
    fptr = fopen("data.txt", "r");
    printf("\nName\tAge\tHeight");

    while(fscanf(fptr, "%s %d %f", name, &age,
    &height)!=EOF){
        printf("\n%s\t%d\t%.2f", name, age,
height);
    }
    fclose(fptr);
    getch();
}
```

# Some operations on Data file

## ❑ Closing the file

Syntax :- `fclose (file_pointer);`

## ❑ Renaming the file

Syntax :- `rename("oldfilename", "newfilename");`

## ❑ Removing/deleting the file

Syntax :- `remove("filename");`

# Thank You

Have a good day, ahead 😊