Computer Programming

Harshita Maheshwari

What is Programming Language??

A Programming language is a set of rules that provides a way of telling a computer what operations to perform.

A programming language is a notational system for describing computation in a machine-readable and human-readable form.

English is a language. It has words, symbols and grammatical rules.

Programming language also has words, symbols and rules of grammar. Grammatical rules are called **Syntax**.

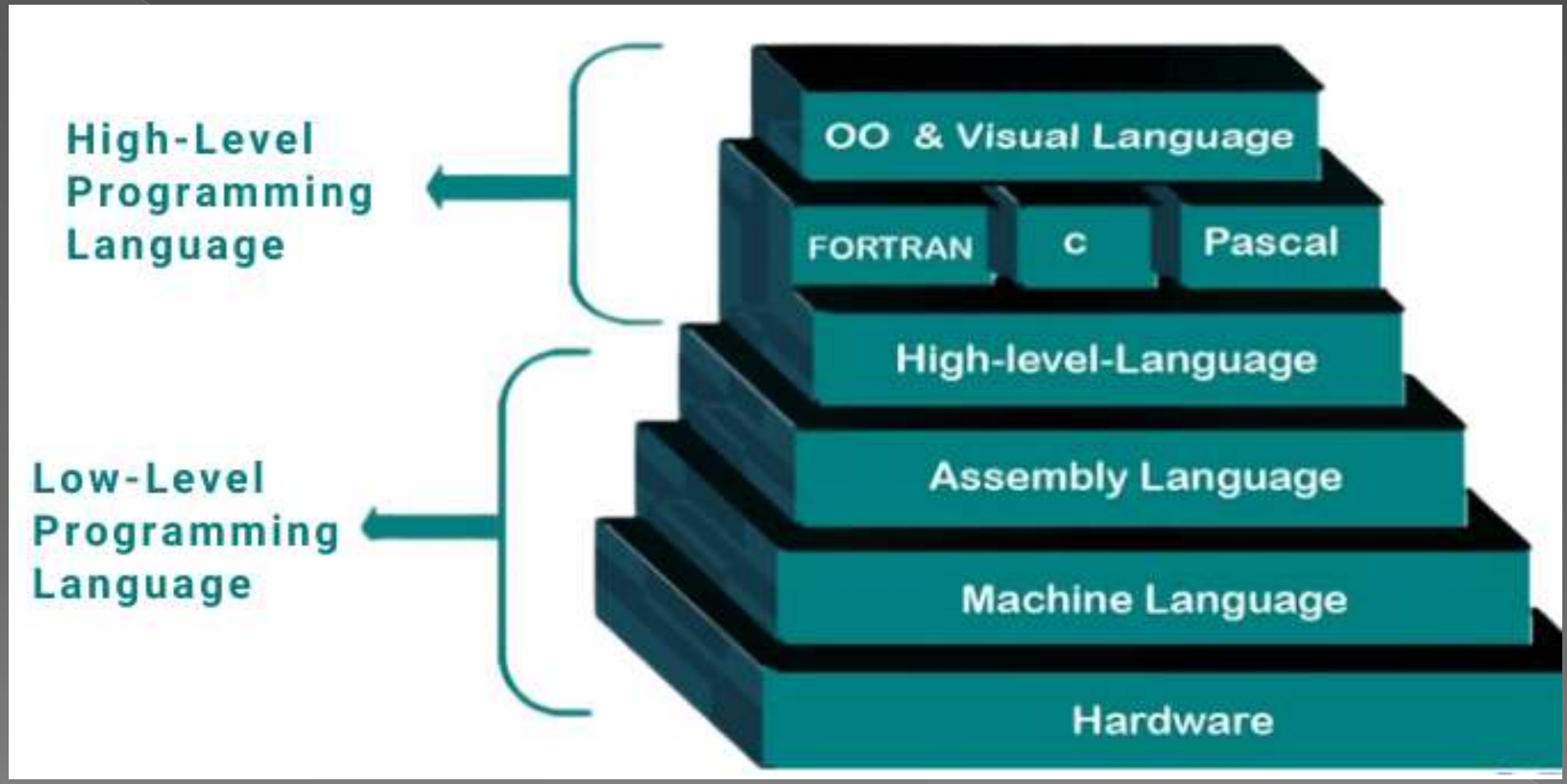
What is Program??

Program is something that is produced using a programming language.

A set of instructions to perform a particular job/task is called a program.

What is Programming??

- Programming is a Science – implements algorithm describe by mathematics and science.
- Programming is a Skill-requires design efforts
- Programming is an Engineering –requires tradeoffs between program speed, size, time and maintainability among many solutions.
- Programming is an Art – requires creativity and imagination


High-Level
Programming
Language
Low-Level
Programming
Language
OO & Visual Language
FORTRAN
C
Pascal
High-level-Language
Assembly Language
Machine Language
Hardware

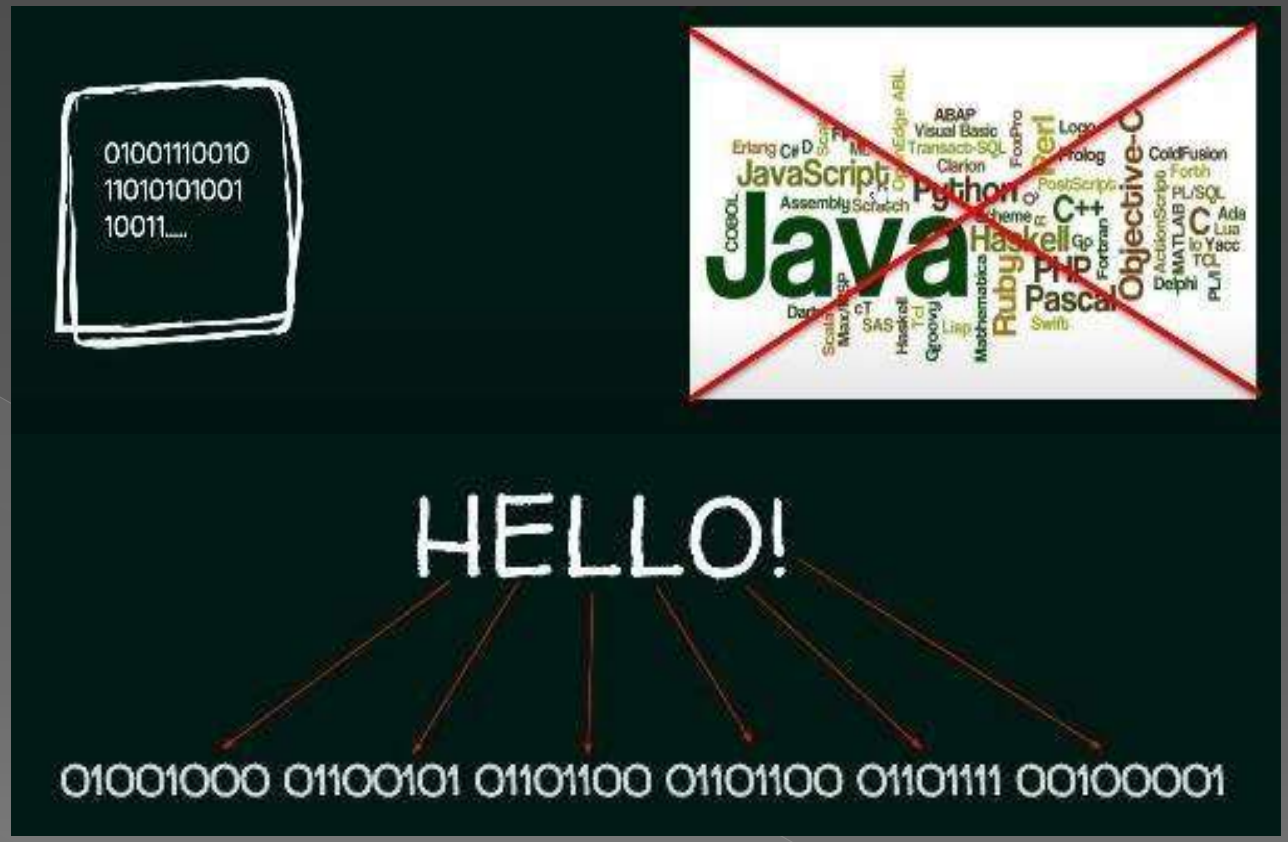
Machine Language

A sequence of 0s and 1s can be used to describe any physical operation that the computer performs.

The language that the computer understands is called the machine language.

Machine language is machine dependent as it is the only language the computer can understand.

Very efficient code but very difficult to write.



Assembly Language

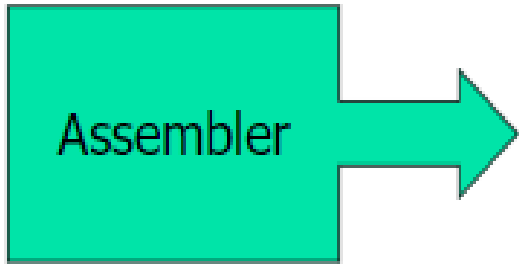
Symbolic operation codes replaced binary operation codes.

Assembly language programs needed to be "assembled" for execution by the computer.

Each assembly language instruction is translated into one machine language instruction.

Very efficient code and easier to write.

```
LD Ax, 9
LD Bx, 10
ADD Ax,Bx
LD (100),Ax
JMP Bx
HLT
```




```
100 0 00 11
11 000 0 0
1 00 0 111
```

In the above program:

The line number one loads register Ax with the value, 9.

The line number two loads register Bx with the value, 10.

The line number three adds the value of register Bx to the value of register Ax.

The line number four stores the value of register Ax in the main memory location, 100.

The line number five uses JMP to jump to register Bx to transfer the control to register Bx.

The line number six stops the program execution.

High-Level Language

A High-Level Language is an English-like language.

It allowed users to write in familiar notation, rather than numbers or abbreviations.

Most High-level languages are not Machine Dependent.

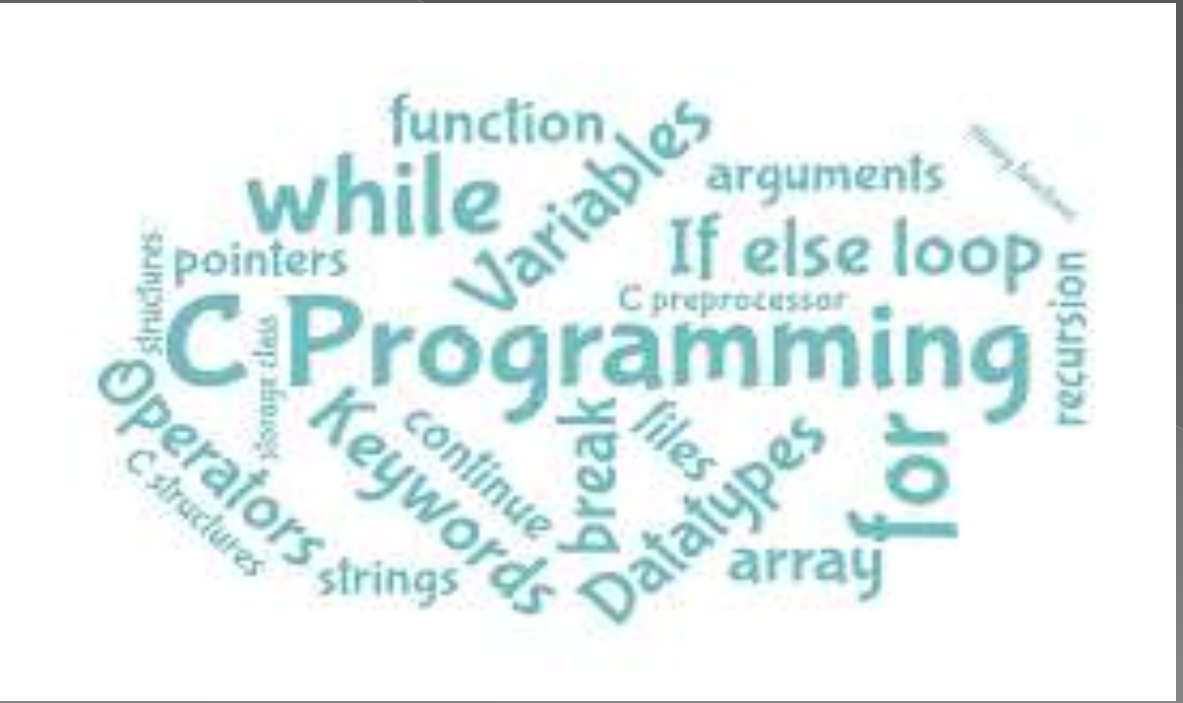
Translator for High-level languages is either a **Compiler** or an **Interpreter**.

Examples : FORTRON, COBOL. BASIC, C and C++ etc.

Programming Paradigms

- Procedural /Imperative Programming
- Object – Oriented Programming
- Logic / Declarative Programming
- Functional / Applicative Programming

C Programming



Introduction

- General- purpose
- Procedural Programming
- Structural Programming
- Middle level
- Portable
- Machine-independent
- Simple and easy
- Top-down approach

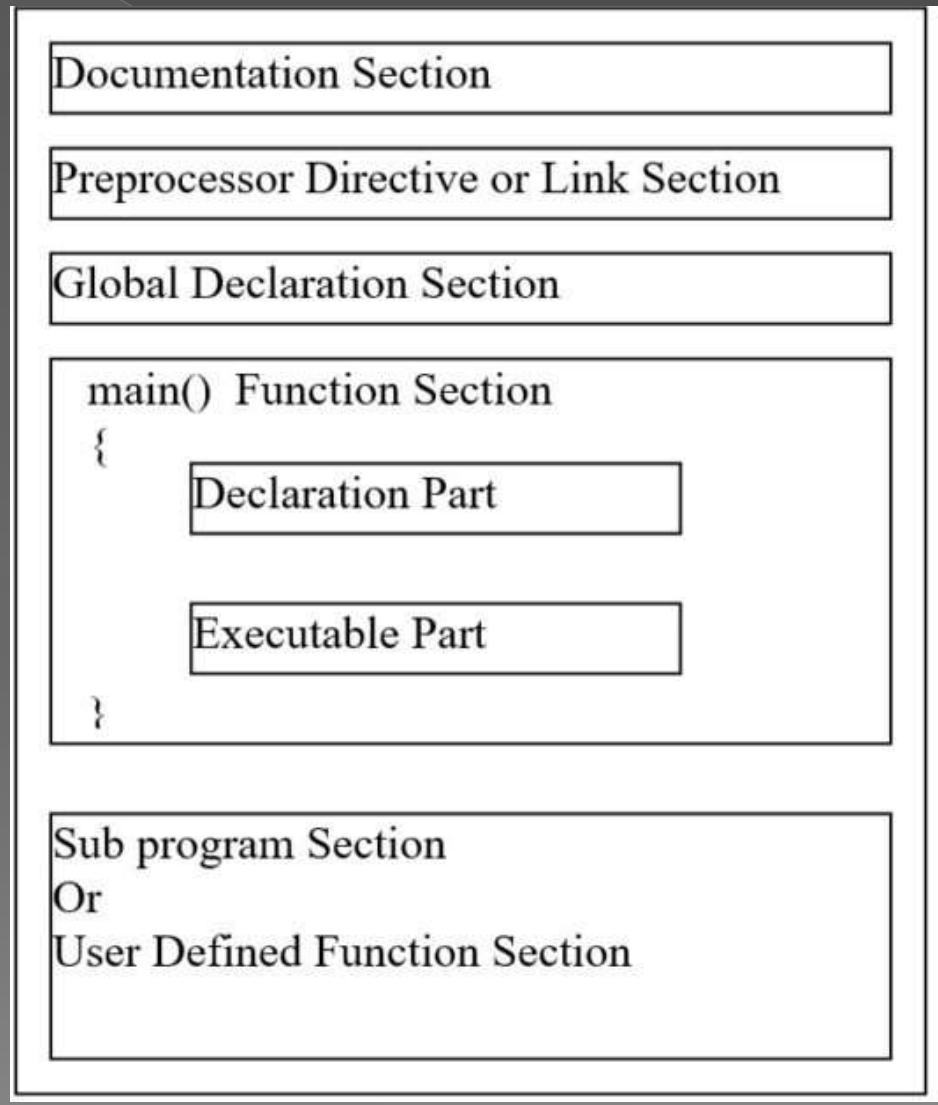
Developed by Dennis Ritchie at the Bell Laboratories in 1972.

C was created from ‘ALGOL’, ‘BCPL’ and ‘B’ programming languages.

Why C??

- C is important to build programming skills
- C covers basic features of all programming language
- C is the most popular language for hardware dependent programming.
- Unix OS is developed in C
- Oracle is written in C
- MYSQL is written in C
- Almost every device driver is written in C
- Core libraries of android are written in C
- Major part of Web Browser is developed in C
- C is World's most popular programming language

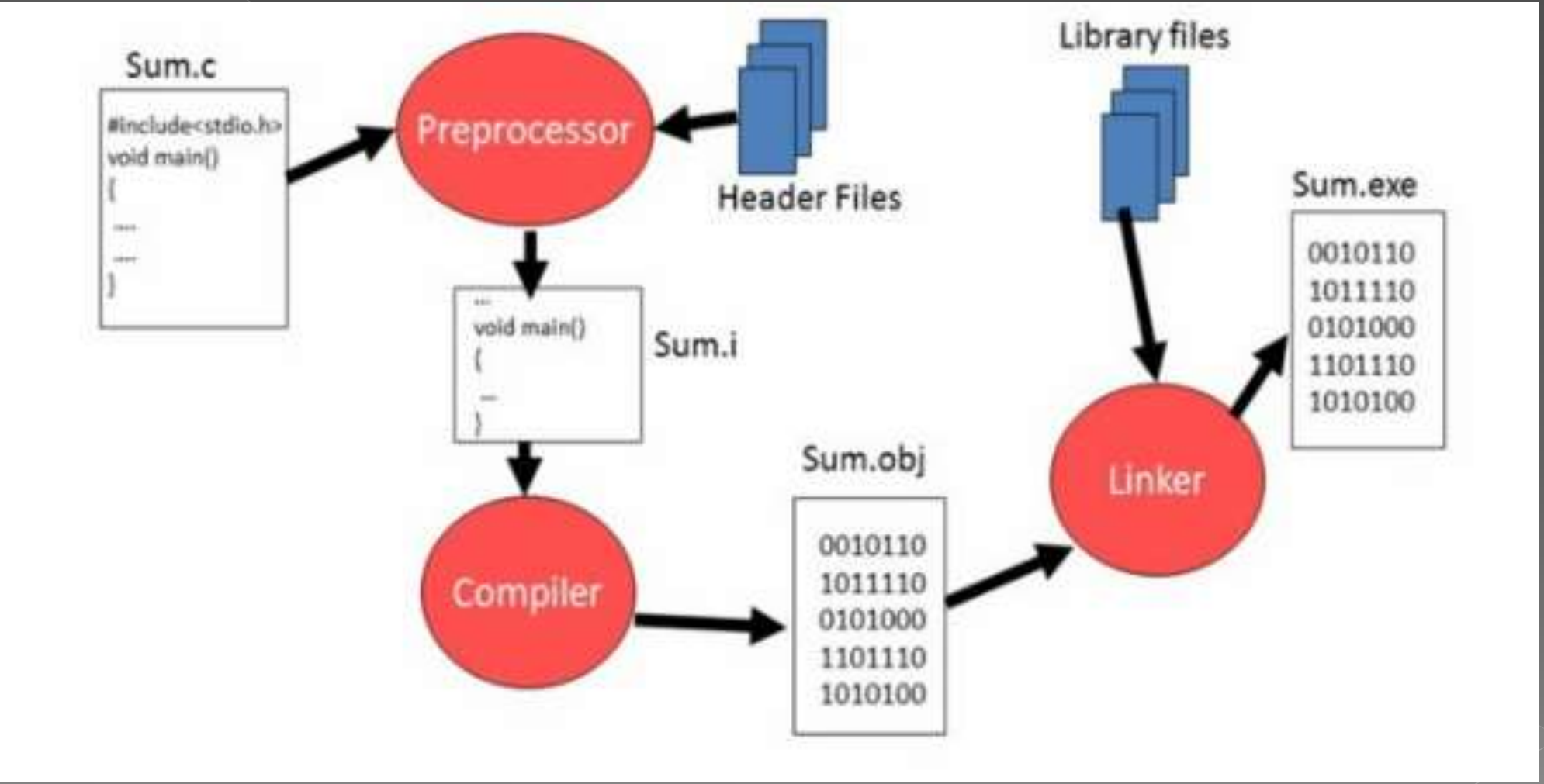
Structure of C Program



Example Program

```
/*program to print a message*/

#include<stdio.h>

main()

{

        printf("This is C program\n");

}
```

Compilation and Execution Process



Header Files

- The files that are specified in the include section is called as header file
- These are precompiled files that has some functions defined in them
- We can call those functions in our program by supplying parameters
- Header file is given an extension .h
- C Source file is given an extension .c

main()

- This is the entry point of a program
- When a file is executed, the start point is the main function
- From main function the flow goes as per the programmers choice.
- There may or may not be other functions written by user in a program
- Main function is compulsory for any c program

Compilation and Execution

Type a program and Save it

Compile the program – This will generate an exe file (executable)

Run the program (Actually the exe created out of compilation will run and not the .c file)

In different compiler we have different option for compiling and running.

Comments

Single line comment

// (double slash)

Multi line comment

/*....

.....*/

C Character Set

1) Letters A to Z in Capital letters. a to z in Small letters.
2) Digits All Decimal 0 to 9
3) Special Characters-

Symbol	Meaning	Symbol	Meaning
~	Tilde	`	Apostrophe
!	Exclamation mark	-	Minus sign
#	Number sign	=	Equal to sign
$	Dollar sign	{	Left brace
%	Percent sign	}	Right brace
^	Caret	[	Left bracket
&	Ampersand	]	Right bracket
*	Asterisk	:	Colon
(	Lest parenthesis	"	Quotation mark
)	Right parenthesis	;	Semicolon
_	Underscore	<	Opening angle bracket
+	Plus sign	>	Closing angle bracket
|	Vertical bar	?	Question mark
\	Backslash	,	Comma
`	Apostrophe	.	Period
-	Minus sign	/	Slash

Escape Sequence

Escape Sequence	Description
\t	Inserts a tab in the text at this point.
\b	Inserts a backspace in the text at this point.
\n	Inserts a newline in the text at this point.
\r	Inserts a carriage return in the text at this point. use \r to move to the start of the line and overwrite the existing text.
\f	Inserts a form feed in the text at this point. \f is used for page break. You cannot see any effect in the console. But when you use this character constant in your file then you can see the difference. (in cmd abc > xyz.doc)
\'	Inserts a single quote character in the text at this point.
\"	Inserts a double quote character in the text at this point.
\\	Inserts a backslash character in the text at this point.

Delimiters

Delimiters	Symbols	Use
Colon	:	Useful for label
Semicolon	;	Terminates statements
Parenthesis	()	Used in Expression and function
Square brackets	[]	Used for array declaration
Curly braces	{}	Scope of statement
Hash	#	Preprocessor directive
Comma	,	Variable seperator
Null Character	\0	In end of string

Keywords

Keywords are the reserved words (can not change the meaning) in programming. There are 32 keywords.

C is case sensitive language. All keywords must be written in 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

Identifiers

Identifier refers to name given to entities such as variables, functions, structures etc.

Identifiers must be unique. They are created to give a unique name to an entity to identify it.

Identifiers must be different from keywords.

You cannot use int as an identifier because int is a keyword.

Identifier is like a word

Instruction is like a sentence

Variable

- A variable is a name that may be used to store a data value.
- we can change value of a variable during execution of a program.
- The variable name is the usual way to reference the stored value.
- Hold Data Temporary.

Rules to define variable name :-

1.Variable name must be upto 31 characters.

2.Variable name must not start with a digit.

3.Variable name can consist of alphabets, digits and special symbols like underscore _.

4. Blank or spaces are not allowed in variable name.

5.Keywords are not allowed as variable name.

6.The Variable names may be a combination of uppercase and lowercase characters.

Ex: suM and sum are not the same

Variable declaration

The declaration of variables should be done in the declaration part of the program.

The variables must be declared before they are used in the program.

Declaration provides two things

1.Compiler obtains the variable name

2.It tells to the compiler data types of the variable being declared and helps in allocating the memory.

Syntax : Data_type variable_name; // declaration

Data_type variable_name=value; //declaration and initialization

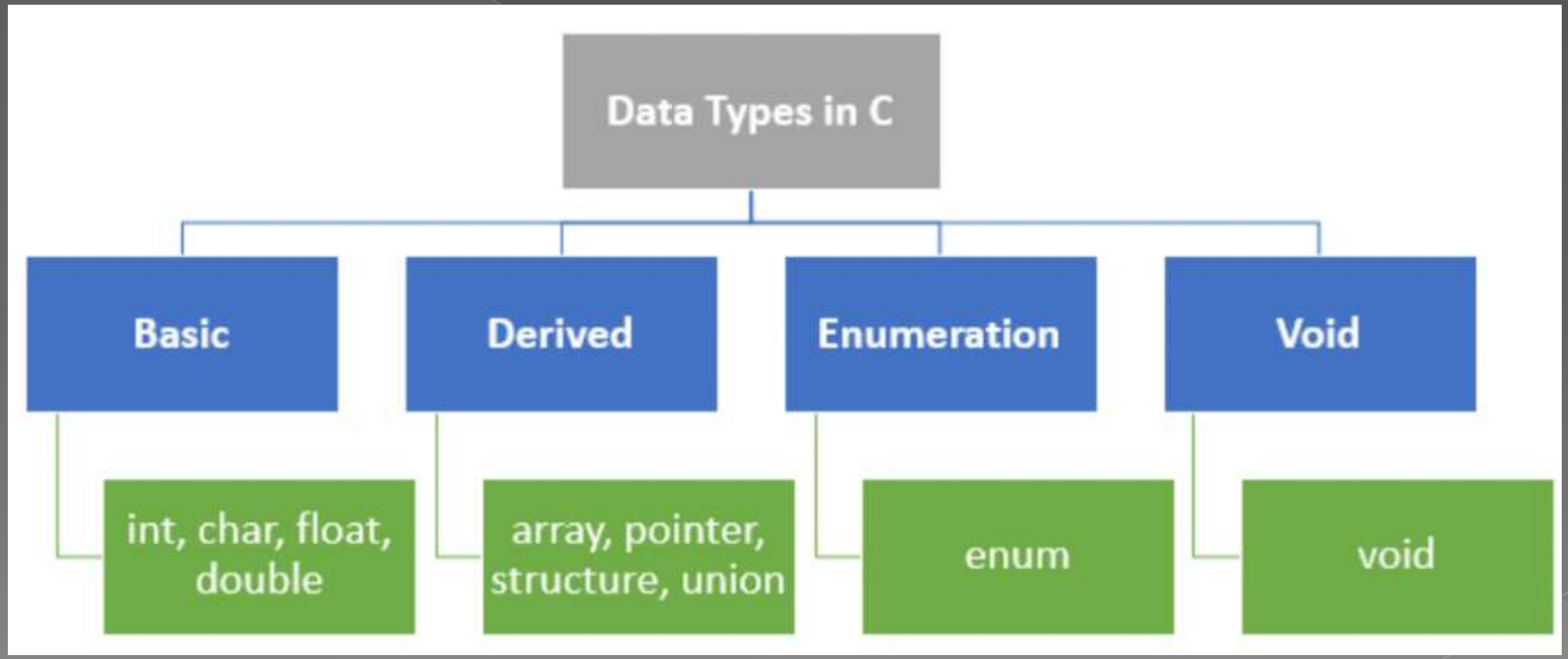
Example : int age;

char m;

int a,b,c;

Data Types

Data types specify how we enter data into our programs and what type of data we enter.



Size & Range of Data Types

Type	Size (bytes)	Format Specifier
`int`	at least 2, usually 4	`%d` , `%i`
`char`	1	`%c`
`float`	4	`%f`
`double`	8	`%lf`
`short int`	2 usually	`%hd`
`unsigned int`	at least 2, usually 4	`%u`
`long int`	at least 4, usually 8	`%ld` , `%li`
`long long int`	at least 8	`%lld` , `%lli`
`unsigned long int`	at least 4	`%lu`
`unsigned long long int`	at least 8	`%llu`
`signed char`	1	`%c`
`unsigned char`	1	`%c`
`long double`	at least 10, usually 12 or 16	`%Lf`

Input & Output

printf():- It is a library function. It resides in the library with header file stdio.h. It outputs data to the screen.

```
Eg:        printf("Hello");
           printf("Num=%d",num);
           printf("N1=%d\n N2=%d",n1,n2);
```

scanf():- It is a library function. It resides in the library with header file stdio.h. It reads the input data from the console.

```
Eg:   scanf("%d",&n);
      scanf("%d%d",&n1,&n2);
```

Constant

- constants are fixed values it cannot be changed during the execution of the program.
- These fixed values are also called literals. It can be any kind of data type.
- In C program we can define constants in two ways as shown below:

  1. Using #define preprocessor directive

  2. Using a const keyword

#define preprocessor directive is used to declare an alias name for existing variable or any value. We can use this to declare a constant as shown below:

Syntax: #define identifierName value

Ex: #define PI 3.14

identifierName: It is the name given to constant.

value: This refers to any value assigned to identifierName.

Using const keyword to define constants is as simple as defining variables, the difference is you will have to precede the definition with a const keyword.

Syntax: const data_type variable_name=value;

Ex: const float PI=3.14;

Types of Constant

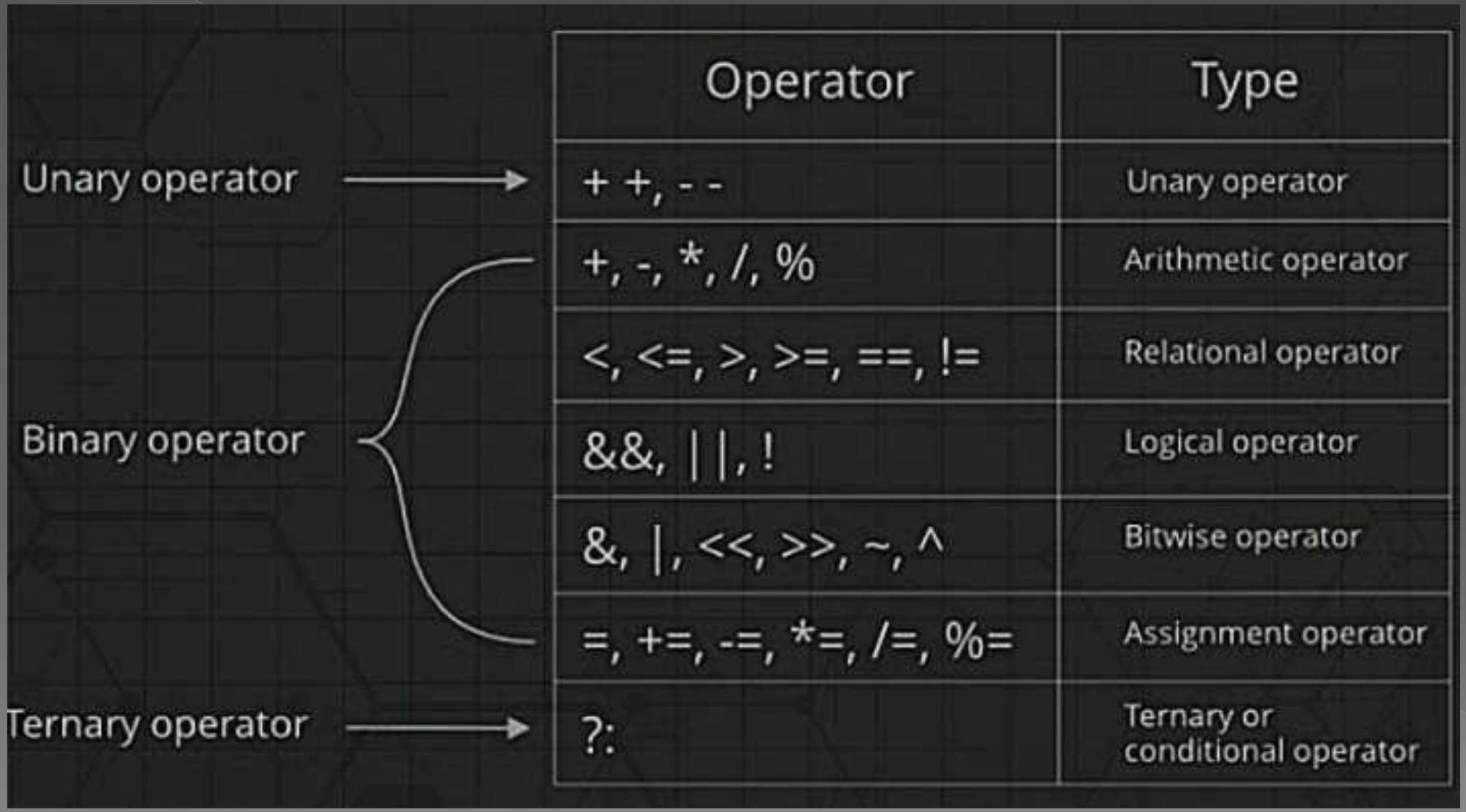
Constant type	data type (Example)
Integer constants	int (53, 762, -478 etc) unsigned int (5000u, 1000U etc) long int, long long int (483,647 2,147,483,680)
Real or Floating point constants	float (10.456789) doule (600.123456789)
Octal constant	int (Example: 013 /*starts with 0 */)
Hexadecimal constant	int (Example: 0x90 /*starts with 0x*/)
character constants	char (Example: 'A', 'B', 'C')
string constants	char (Example: "ABCD", "Hai")

Operators

Operators are symbols that is used to perform some mathematical or logical operations.

Types of operators

- Arithmetic operators
- Assignment operators
- Relational operators
- Logical operators
- Bitwise operators
- Conditional/Ternary operators
- Increment& Decrement operators
- Misc/Special operators


Operator
Type
Unary operator
+ +, - -
Unary operator
Binary operator
+, -, *, /, %
Arithmetic operator
<, <=, >, >=, ==, !=
Relational operator
&&, ||, !
Logical operator
&, |, <<, >>, ~, ^
Bitwise operator
=, +=, -=, *=, /=, %=
Assignment operator
Ternary operator
?:
Ternary or conditional operator

Operator	Description	Associativity
() [] . ->++ --	Parentheses or function call Brackets or array subscript Dot or Member selection operator Arrow operator Postfix increment/decrement	left to right
++ -- + - ! ~ (type) * & sizeof	Prefix increment/decrement Unary plus and minus not operator and bitwise complement type cast Indirection or dereference operator Address of operator Determine size in bytes	right to left
* / %	Multiplication, division and modulus	left to right
+ -	Addition and subtraction	left to right
<< >>	Bitwise left shift and right shift	left to right
< <= > >=	relational less than/less than equal to relational greater than/greater than or equal to	left to right
== !=	Relational equal to or not equal to	left to right
&&	Bitwise AND	left to right
^	Bitwise exclusive OR	left to right
|	Bitwise inclusive OR	left to right
&&	Logical AND	left to right
||	Logical OR	left to right
? :	Ternary operator	right to left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment operator Addition/subtraction assignment Multiplication/division assignment Modulus and bitwise assignment Bitwise exclusive/inclusive OR assignment	right to left
,	comma operator	left to right

Typecasting

A type cast is basically a conversion from one type to another.

1. Implicit: Done by the compiler on its own
2. Explicit : manually

```
int x = 10;
char y = 'a';
 x = x + y;                // y implicitly converted to int. ASCII value of 'a' is 97
float z = x + 1.0;         // x is implicitly converted to float

double x = 1.2;
 int sum = (int)x + 1;      // Explicit conversion from double to int
```

Decision Control Statements

Decision making statements

Decision making statements are used to perform operation on some condition.

It is used to identify the condition of the operation is true or false.

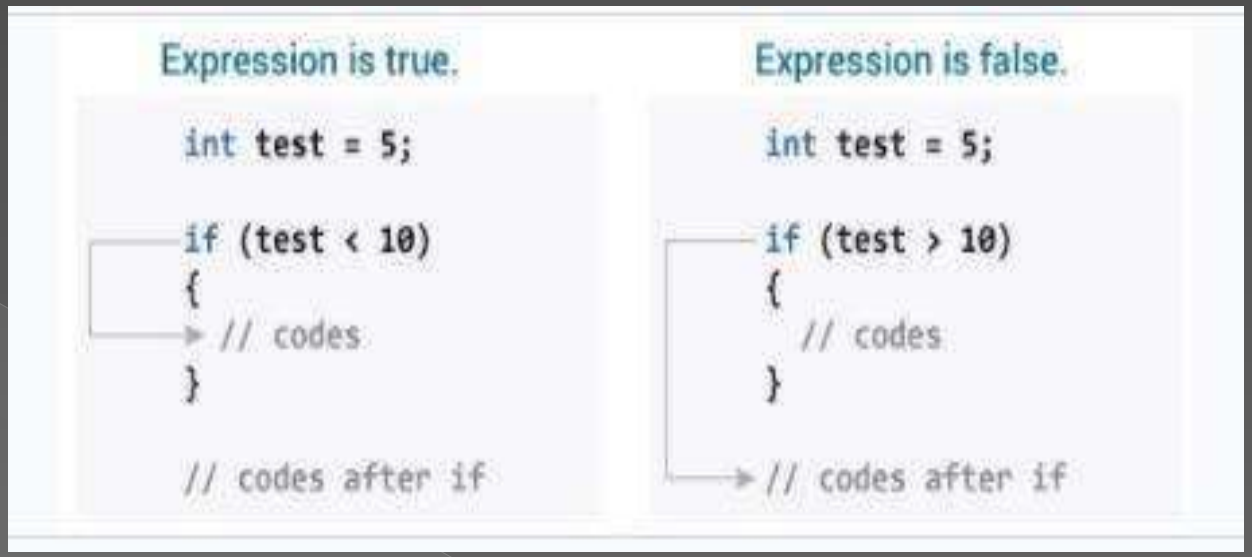
These statements are also called decision controlled statements.

- Here we have listed the decision making statements as follows:
  - Simple if
  - if else
  - Else if ladder
  - Nested if

Simple if

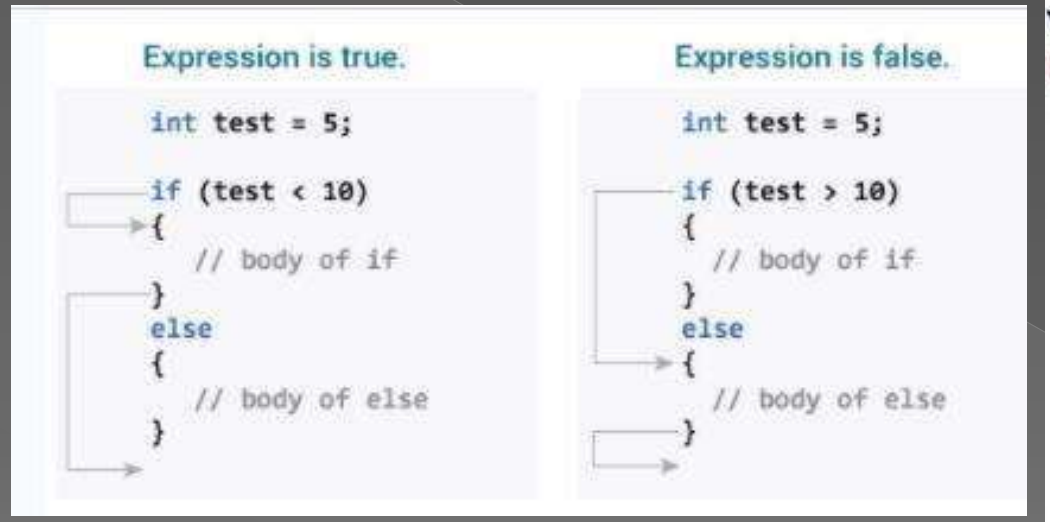
Syntax for simple if

```
if(condition)
{
Statements;
}
```




```
#include<stdio.h>  int main()
{
        int num;
        printf("\nEnter number"); scand("%d",&num);  if(num>0)
        printf("\nIt is positive number");
        return 0;
}
```

If..else




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

void main()
{
    int num1,num2;
    printf("Enter 1st number\n");
    scanf("%d",&num1);
    printf("Enter 2nd number\n");
    scanf("%d",&num2);
    if(num1>num2)
    {
        printf("The greastest number is %d",num1);
    }
    else
    {
        printf("The greastest number is %d",num2);
    }
   getch();
}
```

Else if ladder

The if...else statement executes two different codes depending upon whether the test expression is true or false. Sometimes, a choice has to be made from more than 2 possibilities.

The if...else ladder allows you to check between multiple test expressions and execute different statements.

Syntax:

```
if (test expression1) {
   // statement(s)
}
else if(test expression2) {
   // statement(s)
}
else if (test expression3) {
   // statement(s)
}
.
.
else {
   // statement(s)
}
```

Nested if

It is possible to include an if...else statement inside the body of another if...else statement.

Syntax:

```
if(condition){
        if(condition)
            statement;
        else
            statement;
  }
```

```c
#include <stdio.h>
int main() {
    int number1, number2;
    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);

    if (number1 >= number2) {
      if (number1 == number2) {
        printf("Result: %d = %d",number1,number2);
      }
      else {
        printf("Result: %d > %d", number1, number2);
      }
    }
    else {
        printf("Result: %d < %d",number1, number2);
    }

    return 0;
}
```

Loops

Loop

Loop is a block of statement that performs set of instructions.

In loops Repeating particular portion of the program either a specified number of time or until a particular no of condition is being satisfied.

There are two types of loops in c –

Entry Controlled /pre-tested - While loop ,for loop

Exit Controlled /post-tested - do while loop

While loop

The syntax of the while loop is:

initialization while(condition)

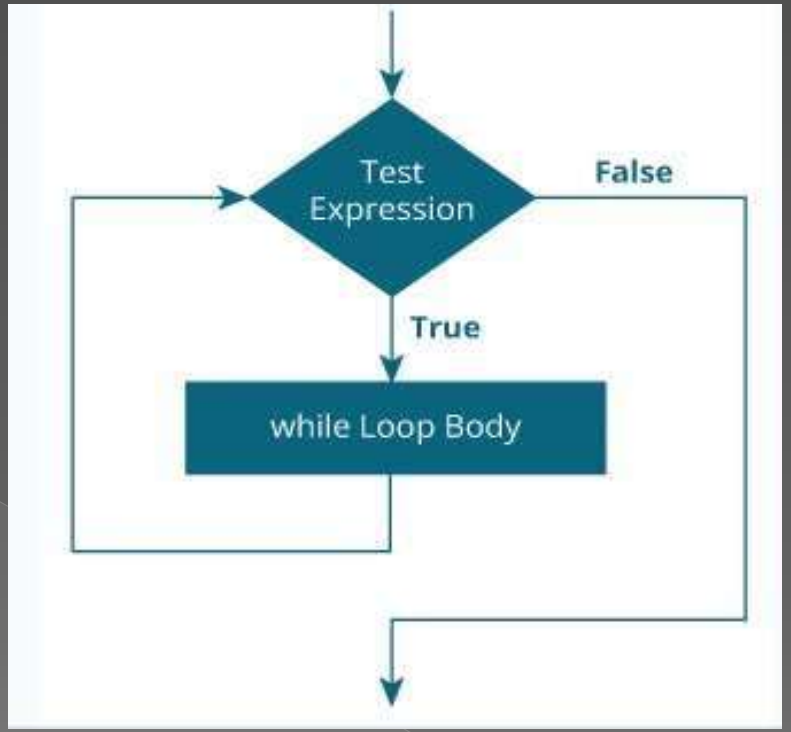
{

Body of the loop; update stmt;

}

For Ex-

```c
#include <stdio.h>
int main()
{
    int i = 1;

    while (i <= 5)
    {
        printf("%d\n", i);
        ++i;
    }

    return 0;
}
```

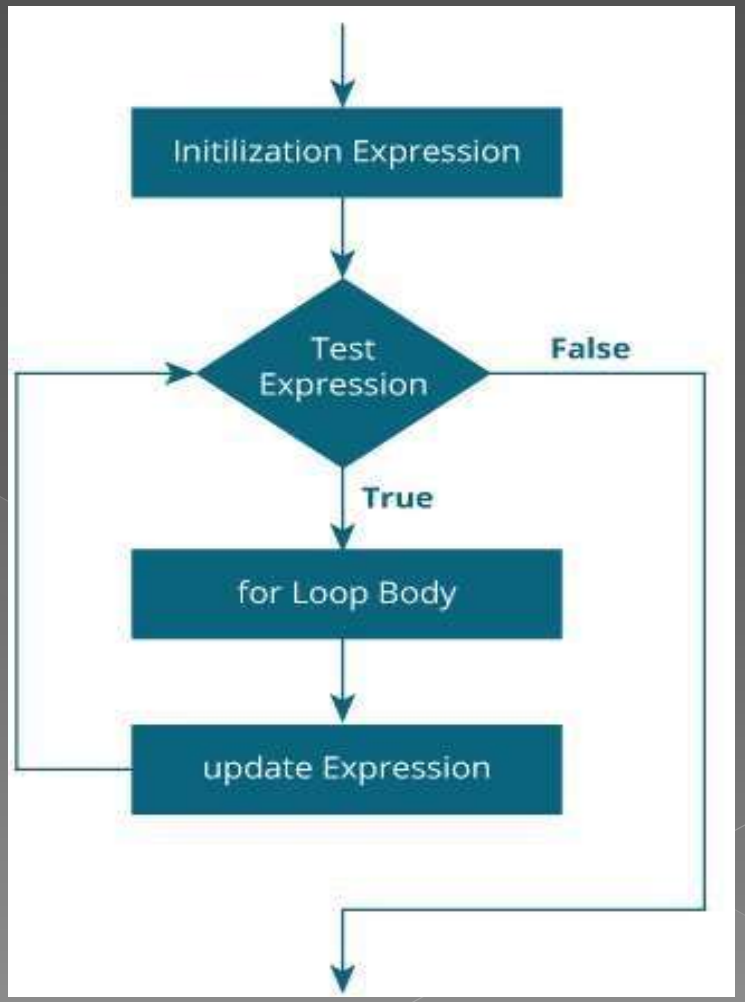


for loop

The syntax of the for loop is:

```
for (initializationStatement; testExpression; updateStatement)
{
// statements inside the body of loop
}
```

```c
// Print numbers from 1 to 10
#include <stdio.h>

int main() {
  int i;

  for (i = 1; i < 11; ++i)
  {
    printf("%d ", i);
  }
  return 0;
}
```

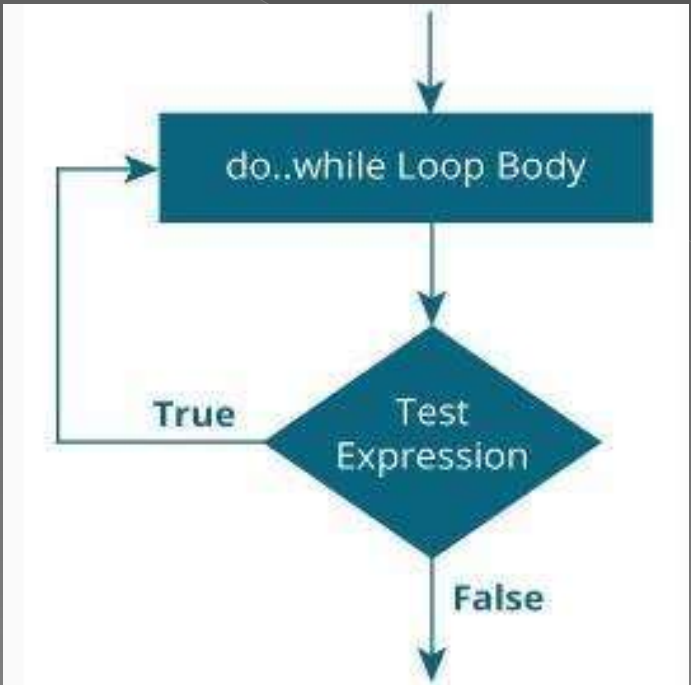


do..while loop

The do..while loop is similar to the while loop with one important difference. The body of do...while loop is executed at least once. Only then, the test expression is evaluated.

The syntax of the do...while loop is:

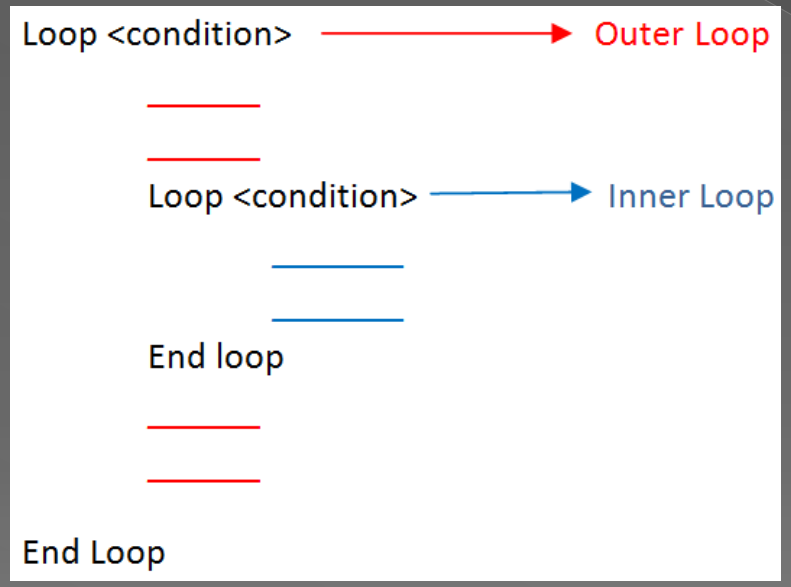
```
do
{
Body of the loop;
}while(condition);
```

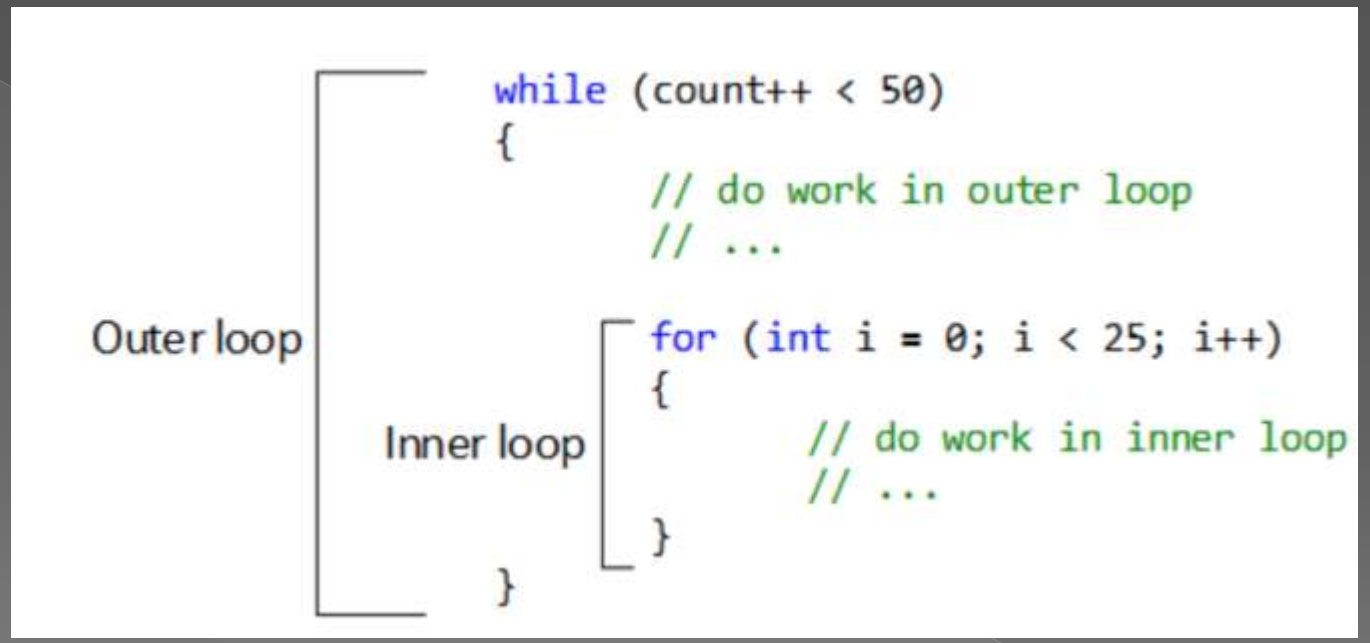



```c
#include <stdio.h>
int main()
{
    double number, sum = 0;

    // the body of the loop is executed at least once
    do
    {
        printf("Enter a number: ");
        scanf("%lf", &number);
        sum += number;
    }
    while(number != 0.0);

    printf("Sum = %.2lf",sum);

    return 0;
}
```

Nested loops

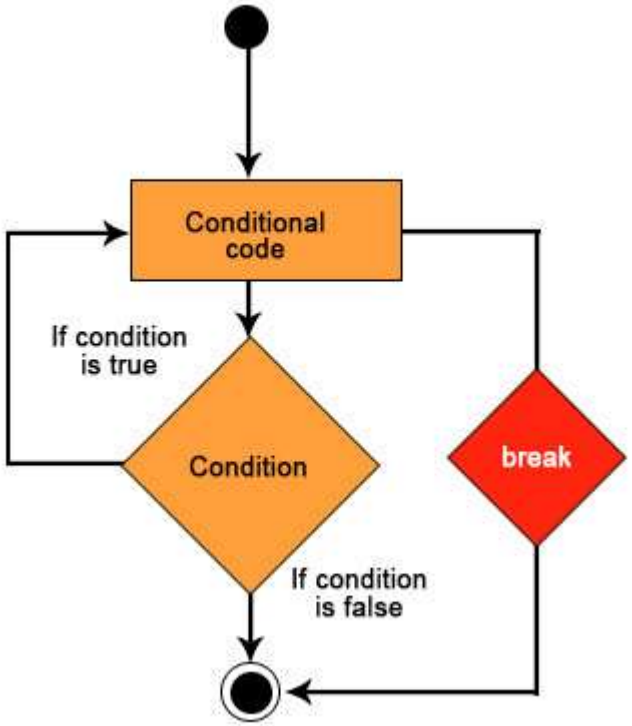
Allows the looping of statements inside another loop. The nesting level can be defined at n times. You can define any type of loop inside another loop;





break

The break is a keyword in C which is used to bring the program control out of the loop. The break statement is used inside loops or switch statement. The break statement breaks the loop one by one, i.e., in the case of nested loops, it breaks the inner loop first and then proceeds to outer loops.



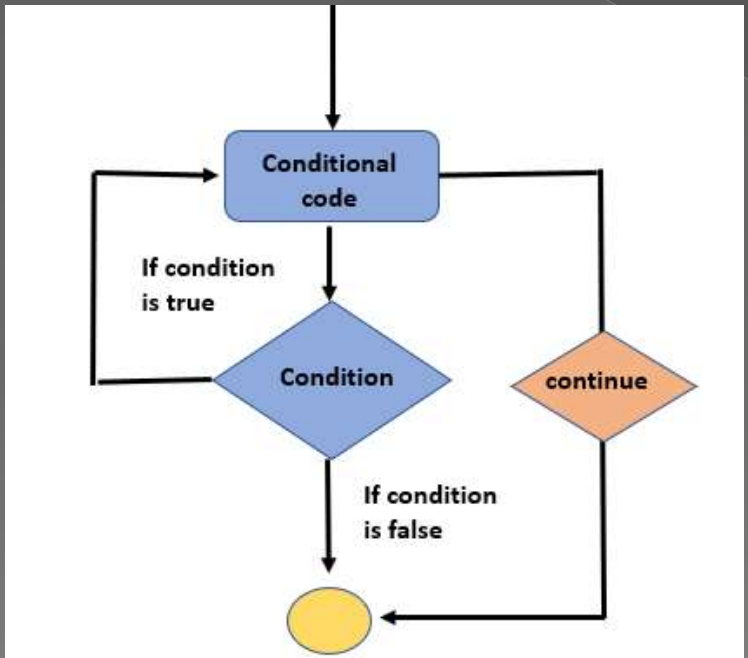

```
while (testExpression) {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
```

```
do {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
while (testExpression);
```

```
for (init; testExpression; update) {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
```

continue

The continue statement is used to bring the program control to the beginning of the loop. The continue statement skips some lines of code inside the loop and continues with the next iteration.




```
while (testExpression) {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
```

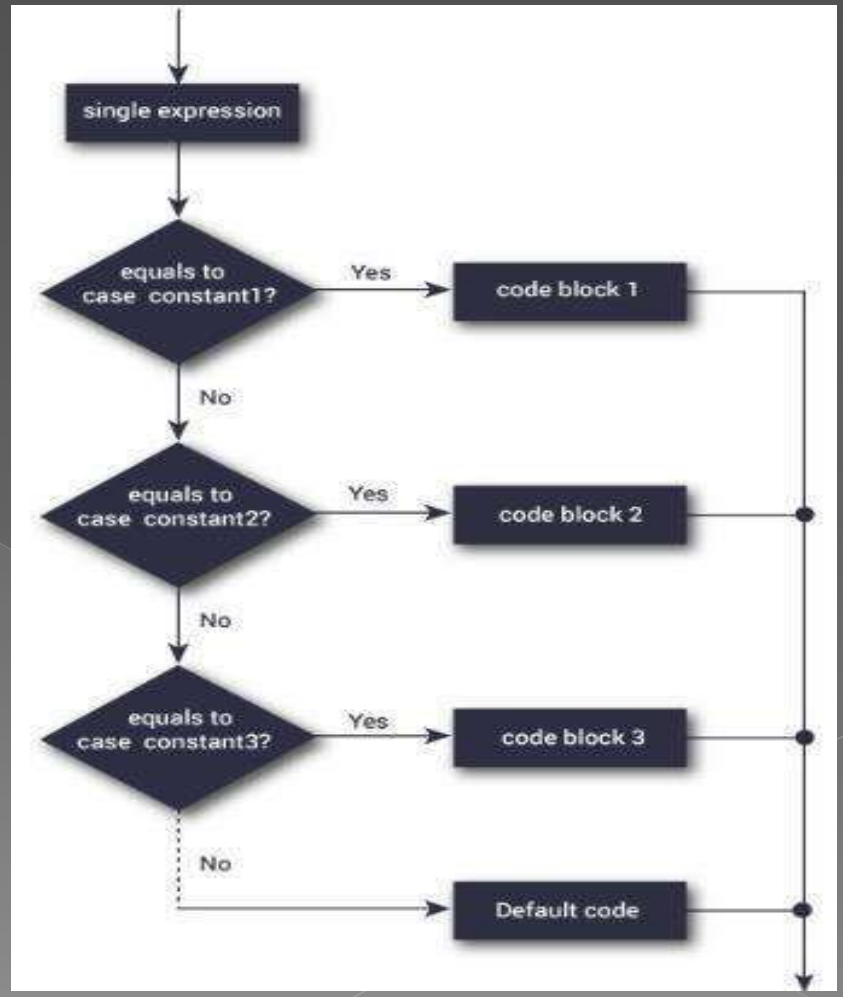
```
do {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
while (testExpression);
```

```
for (init; testExpression; update) {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
```

switch

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.

```
switch (expression)
{
    case constant1:
      // statements
      break;

    case constant2:
      // statements
      break;
    .
    .
    .
    default:
      // default statements
}
```



Some important keywords:

1) **Break**: This keyword is used to stop the execution inside a switch block. It helps to terminate the switch block and break out of it.
2) **Default**: This keyword is used to specify the set of statements to execute if there is no case match.

Important Points About Switch Case Statements:

1) The switch expression must be of an integer or character type (integral constant).
2) The case value must be an integer or character constant.
3) The case value can be used only inside the switch statement.
4) Duplicate case values are not allowed.
5) The default and break statement is optional.
6) The break statement is used inside the switch to terminate a statement sequence.
7) Nesting of switch statements is allowed

Function

Function

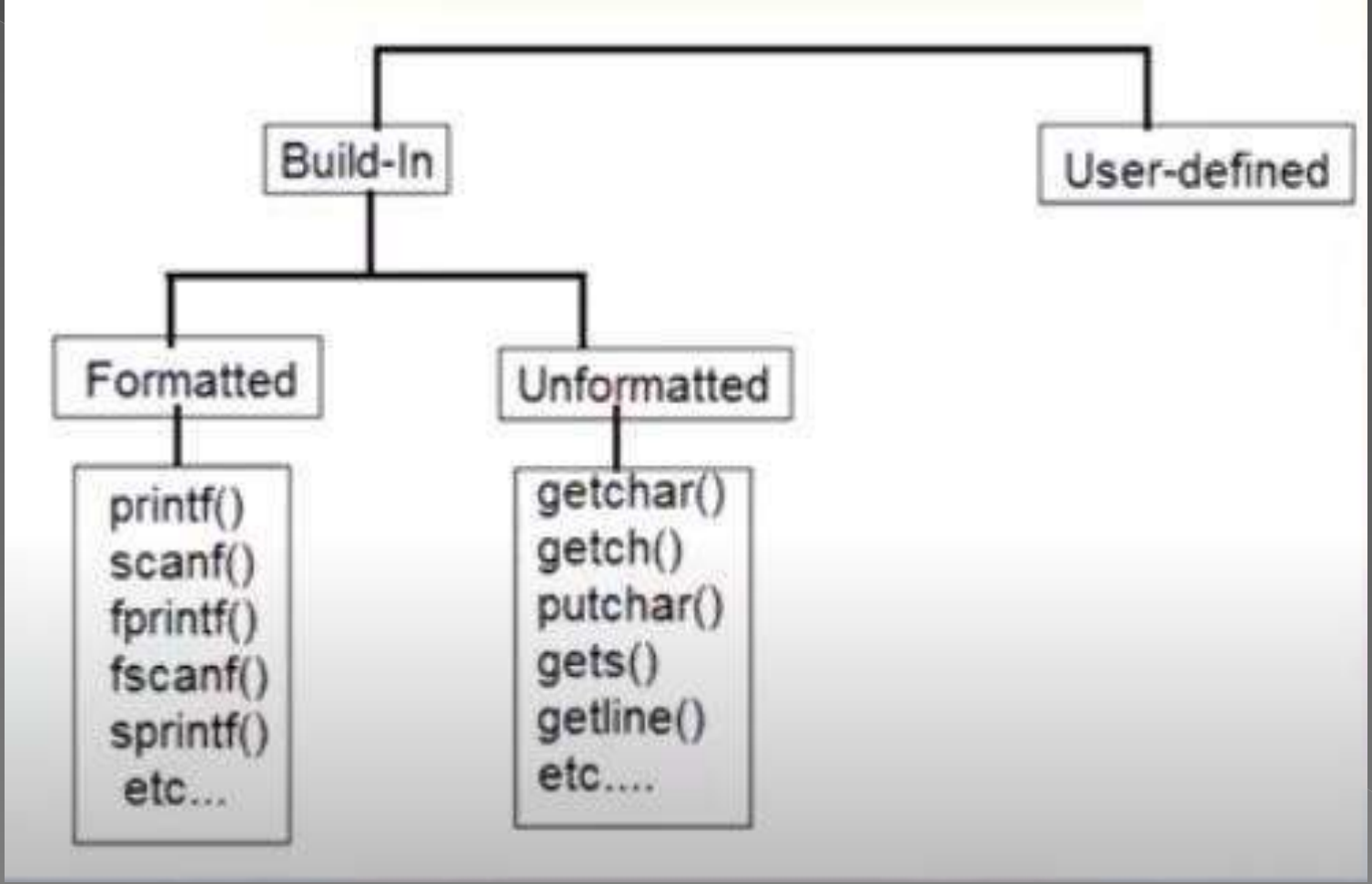
A function is a block of code that performs a specific task.

Function can be used especially when you want to repeat a set of code/re-use throughout the program and at different points.

Function provide re-usability and modularity.

Once written block of code can be used anywhere in the program without re-writing.

Types of Function



Elements of Function

Three program elements involved in using a user-defined function

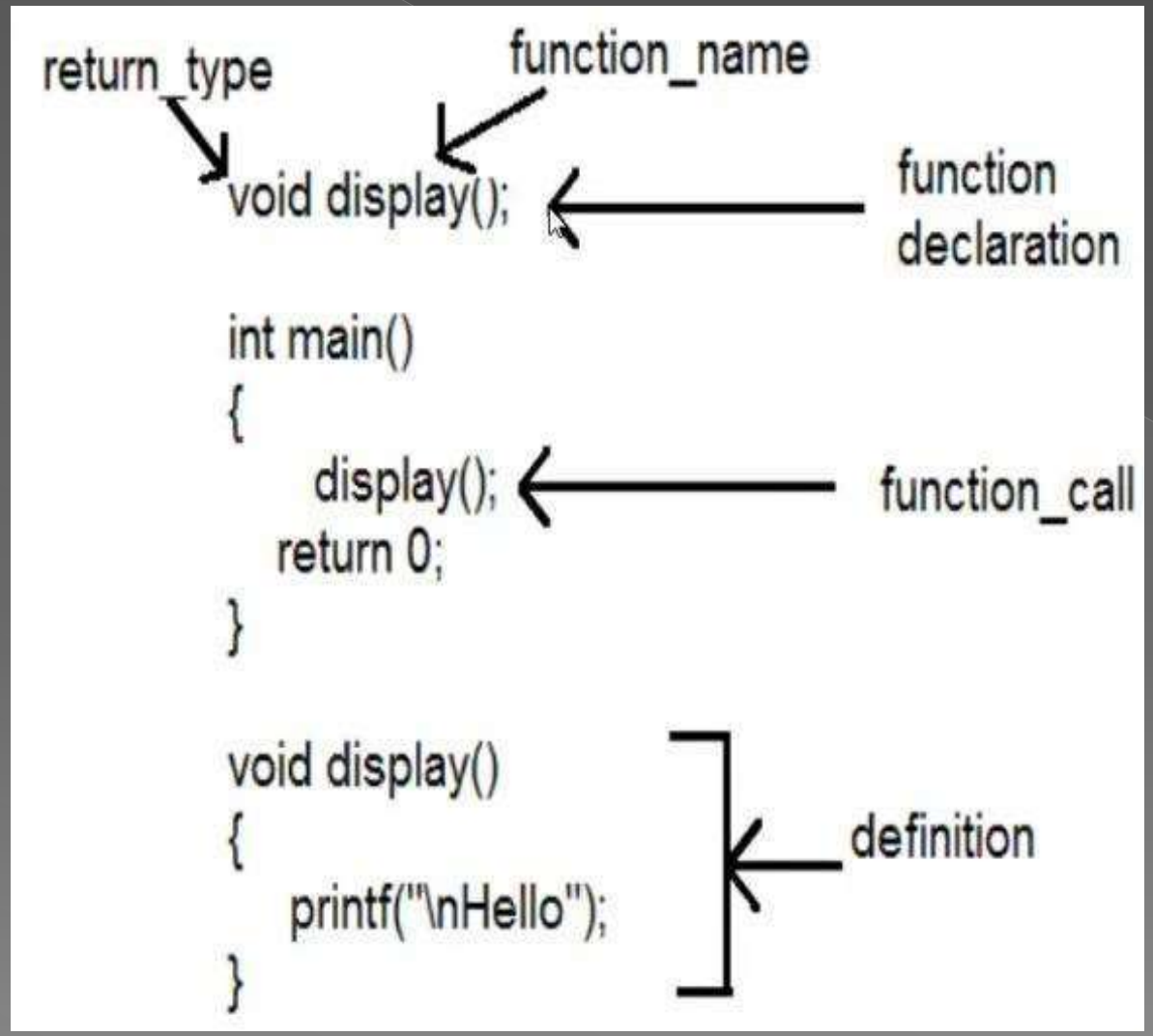
1)Function prototype/Function declaration.

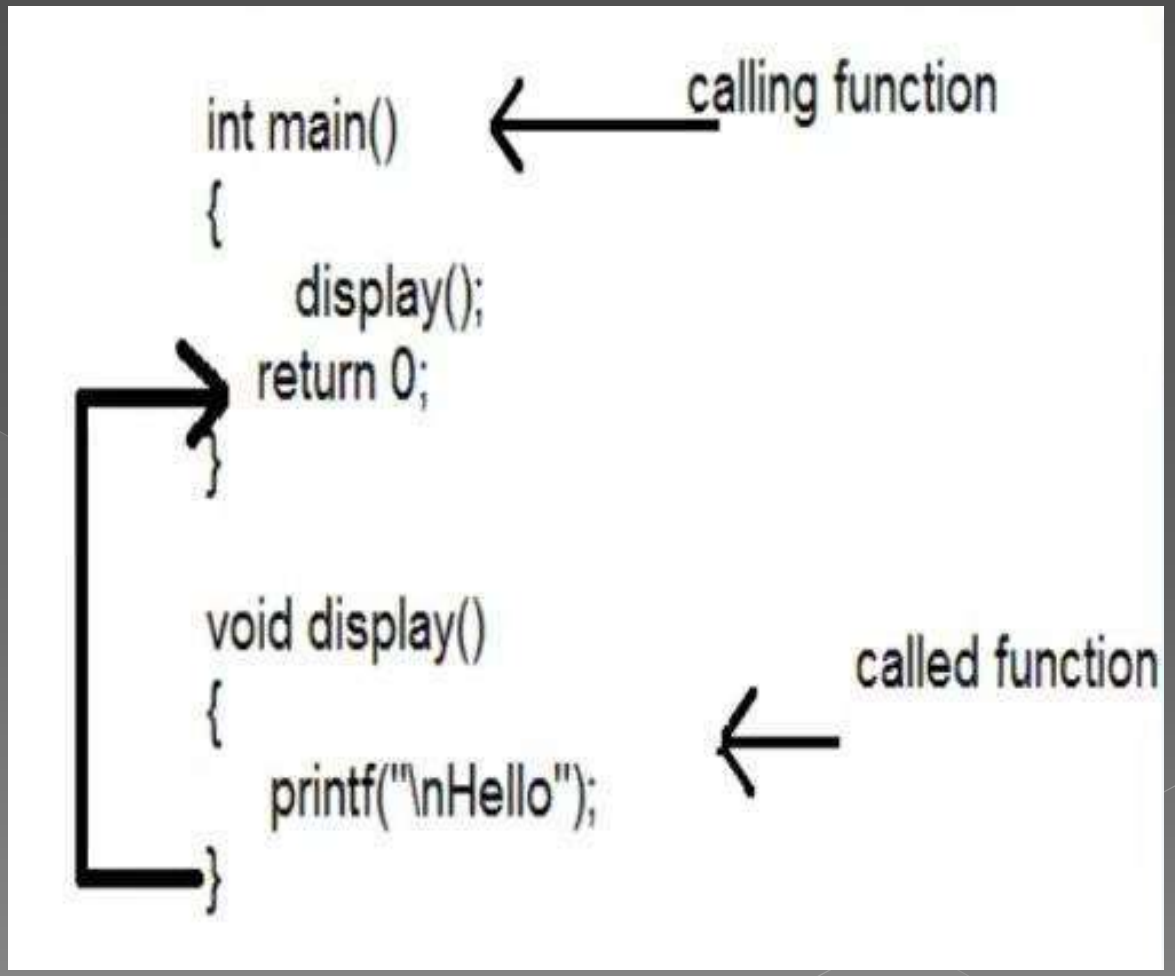
2)Function definition

3)Call to the function

C function aspects	Syntax
Function declaration	return_type function_name (argument list);
Function call	function_name (argument_list)
Function definition	return_type function_name (argument list) {function body;}

Example


return_type
function_name
void display();
function
declaration
int main()
{
display();
function_call
return 0;
}
void display()
{
printf("\nHello");
}
definition


int main()
calling function
{
display();
return 0;
}
void display()
{
printf("\nHello");
}
called function

Types of User Defined Function

1. No arguments passed and no return value
2. No arguments passed but a return value
3. Argument passed but no return value
4. Argument passed and a return value

No arguments passed and no return value

```
void addNumbers();

int main()
{
        addNumbers();
        return 0;
}

void addNumbers()
{
        int num1=2,num2=4;
        printf("Sum=%d\n",num1+num2);
}
```

No arguments passed but a return value

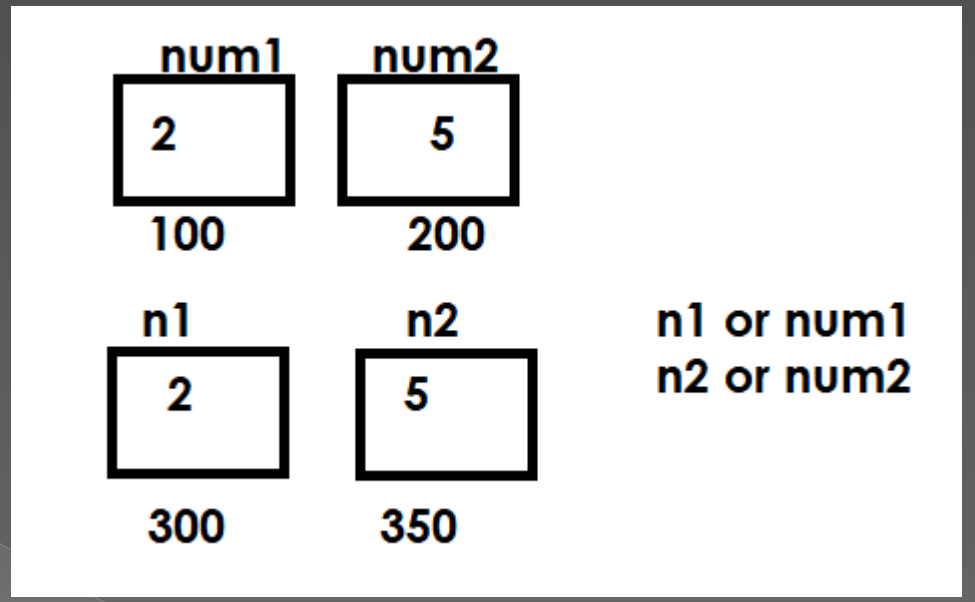
```
int addNumbers();

int main()
{
        int result;
        result=addNumbers();
        printf("Sum=%d\n",result);
        return 0;
}

int addNumbers()
{
        int num1=2,num2=4;
        return num1+num2;
}
```

return statement

- The return statement returns the flow of the execution to the function from where it is called.
- This statement does not mandatorily need any conditional statements. As soon as the statement is executed, the flow of the program stops immediately and returns the control from where it was called.
- The return statement may or may not return anything for a void function, but for a non-void function, a return value must be returned.

Argument passed but no return value

```
void addNumbers(int,int);

int main()
{
    int num1=2,num2=5;
    addNumbers(num1,num2);   <== actual parameter
    return 0;
}

void addNumbers(int n1,int n2)   <== formal parameter
{
    printf("Sum=%d\n",n1+n2);
}
```



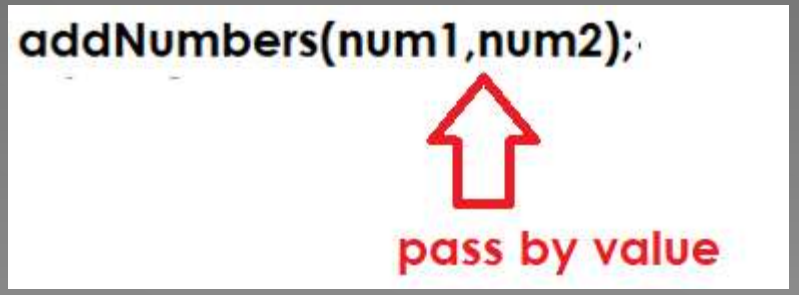
call by value

- In this method value passed to the function locally stored to the parameter/ argument in stack. In other words it copies the value of an argument into the formal parameter of the function.
- Memory is differing from both actual and formal parameters. Because it copy the value of actual parameter to formal parameter.
- The value of actual parameter cannot be changed by the formal parameter. such that original value is not modified.

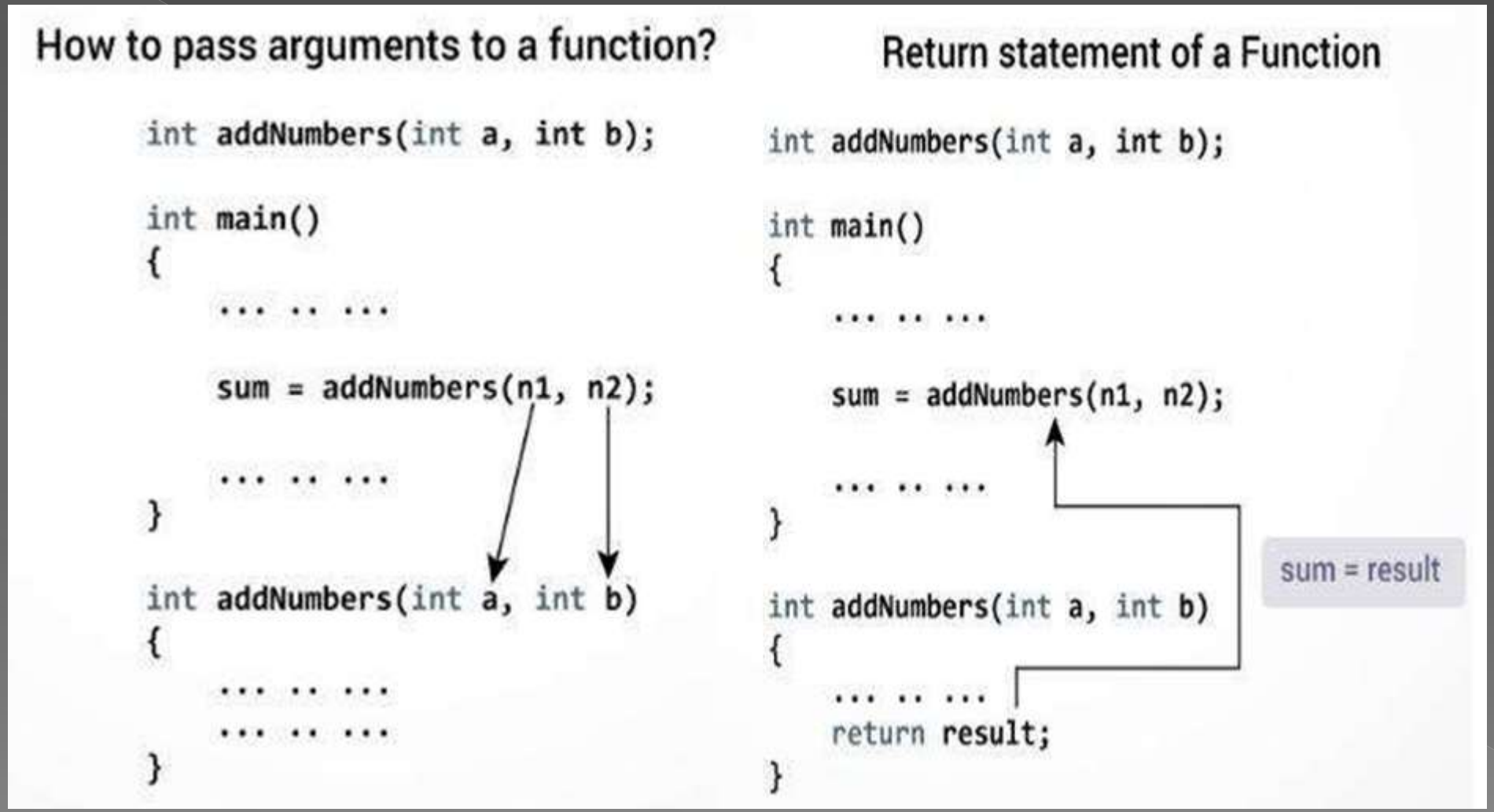
Note: - By default C uses the pass by value method to pass the arguments.

Actual parameter: It is used in function call.

Formal parameter: It is used in function definition



Argument passed and a return value



Recursion

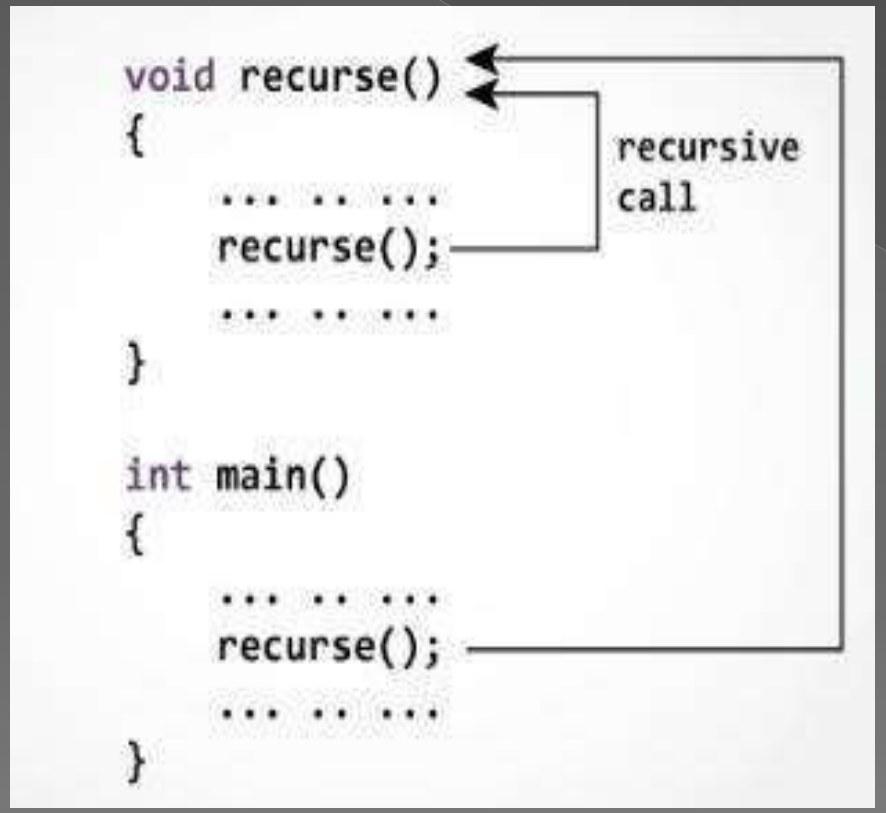
Functions call itself is called as recursive function. This will avoid shifting control to main() repeatedly.

It repeatedly performs the process. Just like looping statement it repeats same code.

Without using iteration statements (while, for) this kind of process will be more helpful for us.

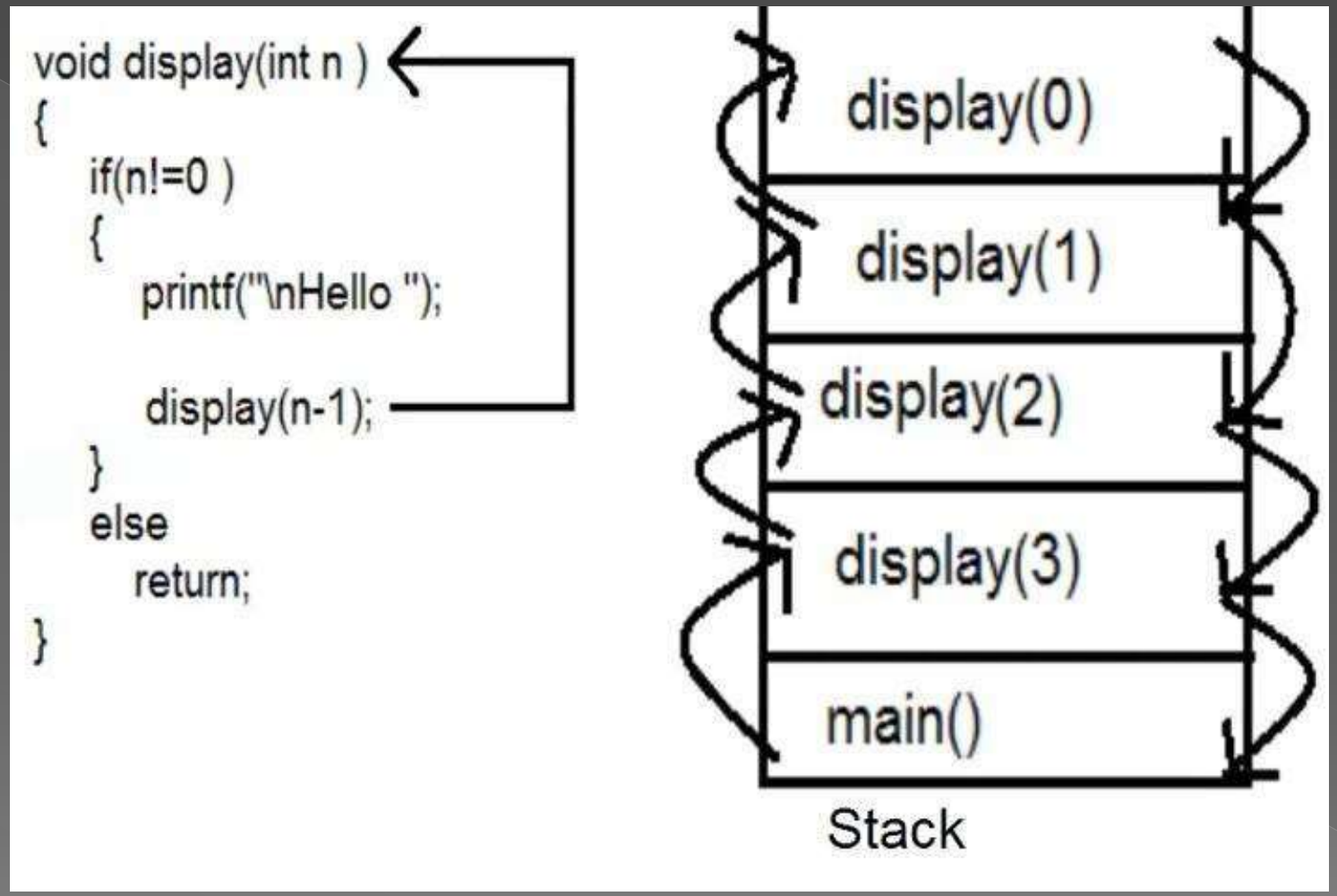
When we using recursive function, we have to define some exit condition from the function to prevent infinite function call.

How recursion works

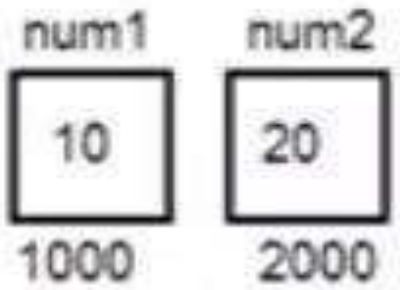


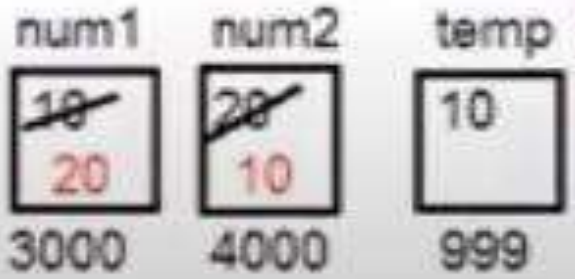

```
void  display(int);

int main()
{
    display(3);
}
```

```
void display(int n)
{
    if(n!=0)
    {
        printf("\nHello  ");
        display(n-1);
    }
    else
        return;
}
```


void display(int n)
{
if(n!=0)
{
printf("\nHello ");
display(n-1);
}
else
return;
}
display(0)
display(1)
display(2)
display(3)
main()
Stack

```
void Swap(int,int);

int main()
{
   int num1=10,num2=20;
   printf("Before swapping:%d  %d",num1,num2);
   Swap(num1,num2);
   printf("After swapping:%d  %d ",num1,num2);

   return 0;
}
void Swap(int num1,int num2)
{
    int temp;
    temp = num1;
    num1 = num2;
    num2 = temp;
}
```





Pointers

Pointer

- **Pointer is a variable whose value is the address of another variable.**
- Pointers will always allocate its own separate memory space.
- Pointer variable size is depend upon processor for example for a 32 bit computer the pointer size can be 4 bytes for a 64 bit computer the pointer size can be 8 bytes.

Syntax: type *var-name;

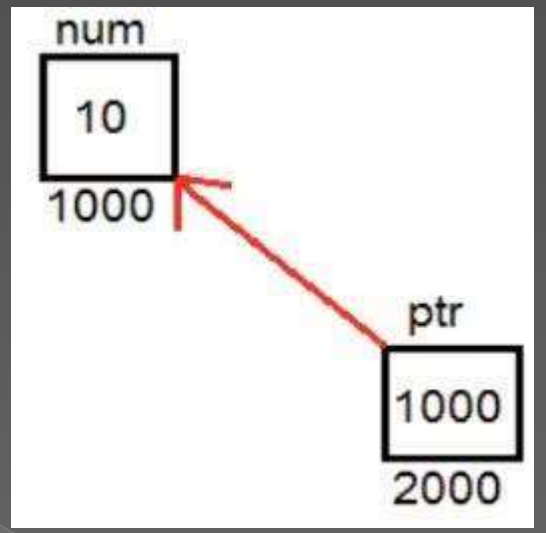
- * is called as indirection operator.
- * is used to declare a pointer and also dereference a pointer.

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

```
Int i = 1 , *ip ; //pointer declaration

ip = &i ; //pointer assignment

*ip = 3 ;  //pointer assignment
```

How to use pointers??

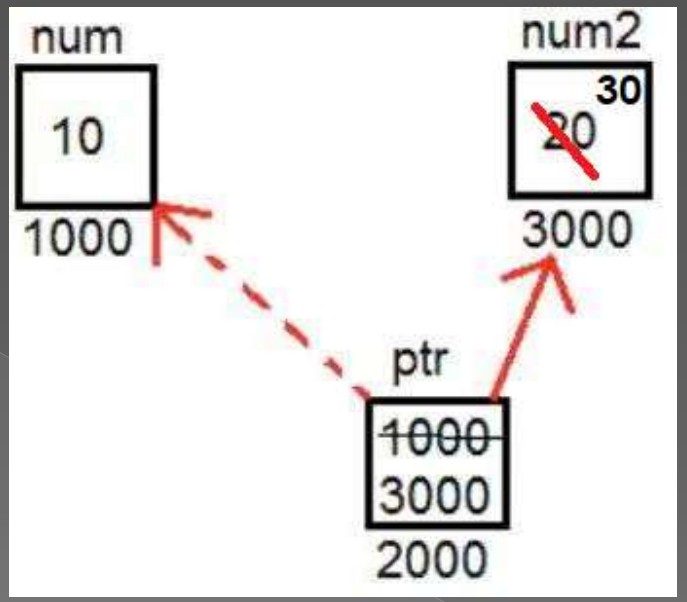
```
int num=10;
int *ptr;
ptr=&num;

printf("num=%d",num);//10
printf("num=%d",&num);//1000
printf("\n ptr=%d",ptr); //1000
printf("\n &ptr=%d",&ptr); //2000
printf("\n *ptr=%d",*ptr); // 10  (*ptr => *(1000)=>10)
```



Note:-

int pointer can hold the address of only int variable.

Pointers can hold address of different variable but only one at a time.




```
int num=10;
int *ptr=&num;
int num2=20;
ptr =&num2;
*ptr=30; // dereferencing (num2 value  will become 30 now)
float f=10.20;
ptr = &f; ?????
```

Pointer Conversions

One type of pointer can be converted to another type of pointer.

```
int main() {
double x=100.1, y;
int *p;
p= (int *) &x; //explicit type conversion
y= *p;
}
```

Types of pointers

- **NULL pointer :**

  A Null Pointer is a pointer that does not point to any memory location.

- **Wild Pointer**

  A pointer which has not been initialized to anything (not even NULL) is known as wild pointer.
  int *ptr;

- **Void Pointer**

  a pointer that points to some data location in storage, which doesn't have any specific type.

- **Dangling Pointer**

  A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer.

NULL Pointer

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an 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.

```
#include <stdio.h>
 int main ()
{
    int *ptr = NULL;
    printf("The value of ptr is : %x\n", ptr );
    return 0;
}
```

Void /generic Pointer

- Void refers to the type. Basically the type of data that it points to is can be any. If we assign address of char data type to void pointer it will become char Pointer, if int data type then int pointer and so on.

Important Points:

- void pointers cannot be dereferenced. It can however be done using typecasting the void pointer
- Pointer arithmetic is not possible on pointers of void due to lack of concrete value and thus size.

```
int x = 4;
float y = 5.5;
void *ptr;
ptr = &x;
printf("Integer variable is = %d", *( (int*) ptr) );
// (int*)ptr - does type casting of void
// *((int*)ptr) dereferences the typecasted

ptr = &y; // void pointer is now float
printf("\nFloat variable is= %f", *( (float*) ptr) );
```

Pointer Arithmetic

Arithmetic operations that can be used on pointers are-

+ , - , ++ , -- , += and -=.

Ex-

```
int var, *ptr_var;
ptr_var = &var;
var = 500;
ptr_var++ ;
```

Let var be a integer type variable having the value 500 and stored at the address 1000.

Then ptr_var as the value 1000 stored in it. Since integers are 4 bytes long, after the expression "ptr var++;" ptr_var will have the value as 1004 and not 1001,

Each time a pointer is incremented, it points to the memory location o he next element of its base type.

Each time it is decremented it points to the location of the previous element,

All other pointers will increase or decrease depending on the length of the datatype they are pointing to.

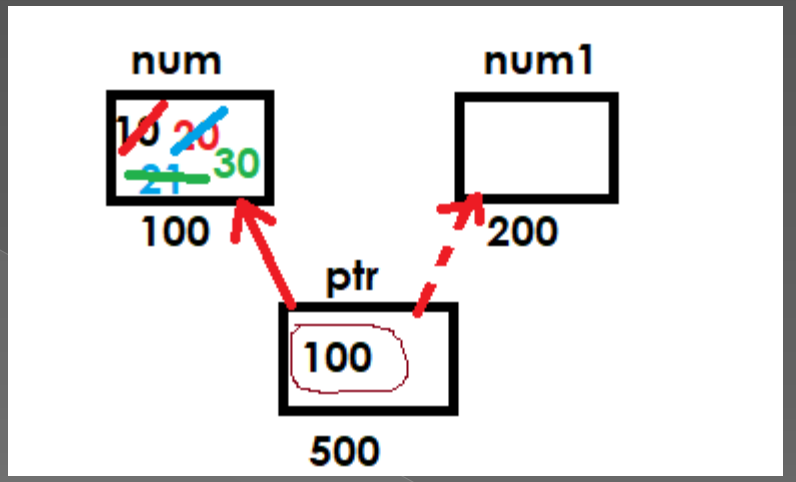
*ptr_var=*ptr_var+10; add 10 to *ptr_var ➔ 510

Arithmetic Rules

- You cannot multiply or divide pointers.
- You cannot add or subtract two pointers.
- You cannot apply bitwise operators to them.
- You cannot add or subtract type float or double to or from pointers.

Constant pointer

A constant pointer in C cannot change the address of the variable to which it is pointing, i.e., the address will remain constant.

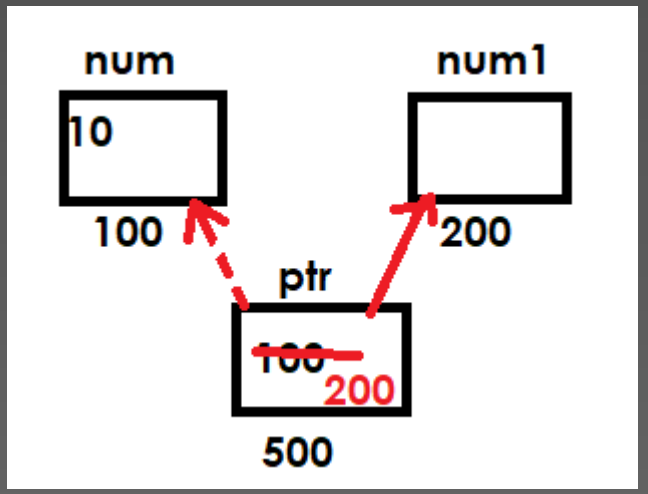



```
int num=10,num1;
int *const ptr = &num;

num=20;
num++;
*ptr=30; //work
```

```
ptr=&num1;  // error
```

Pointer to Constant

A pointer to constant is a pointer through which the value of the variable that the pointer points cannot be changed. The address of these pointers can be changed.




```
int num=10,num1;
const int *ptr = &num; or int const *ptr = &num;

num=20;                                   *ptr=30;    // error
num++;
ptr=&num1; //work
```

Pointer Comparison

- You can compare two pointers in a relational expression, example:

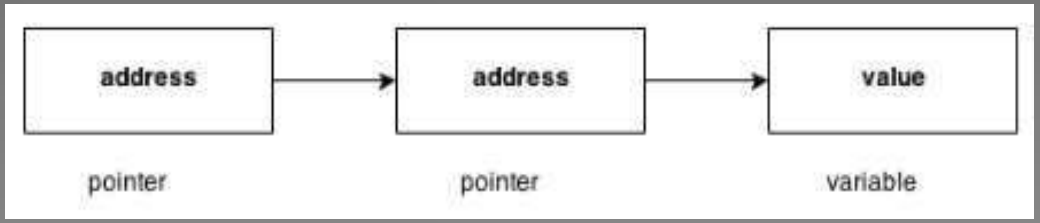
```
if(p<q)
printf("p points to lower memory than q \n");
```

- Pointer comparison are useful only when two pointers point to a common object such as an array.

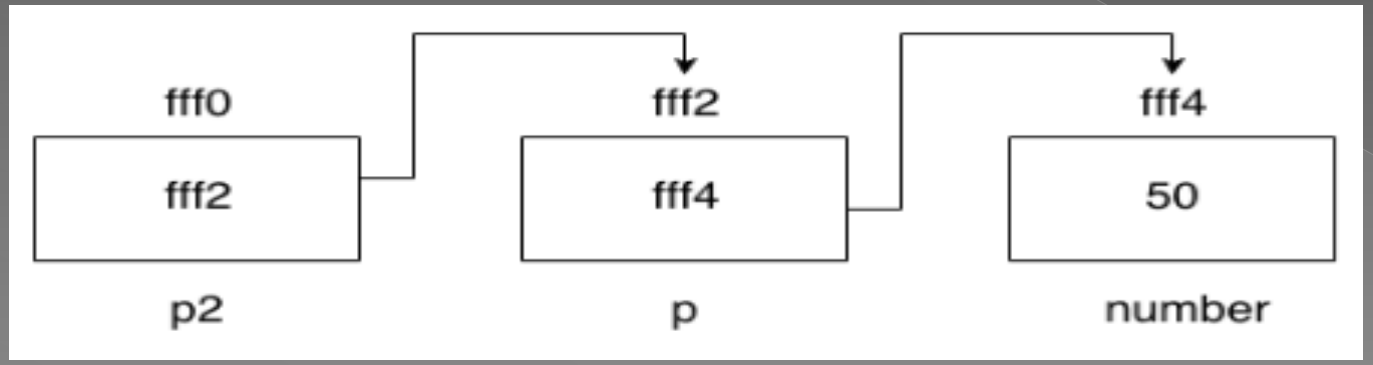
Pointer to Pointer (Double Pointer)

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.

Syntax: datatype **pointervariable;



```
int number= 50;
int *p;
int **p2;
p = &number;
p2 = &p;
printf("Number= %d\n", number);
printf("Number using single pointer = %d\n", *p);
printf("Number using double pointer = %d\n", **p2);
```



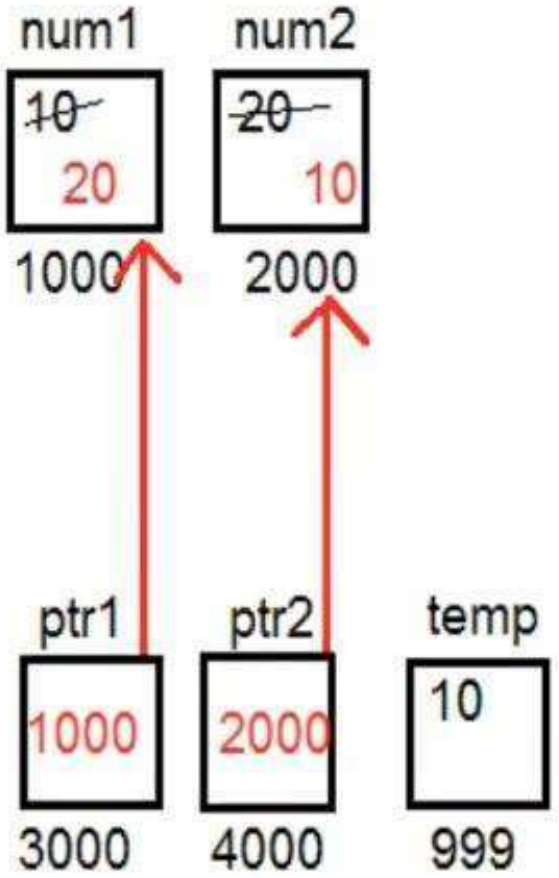
Pointer as Function Arguments

When pointers are passed to a function :
The address of the data item is passed and thus the function can freely access the contents of that address from within the function.

In this way, function arguments permit data-items to be altered in the calling routine and the function.

When the arguments are pointers or arrays, a call by address is made to the function as opposed to a call by value for the variable arguments.

```
void Swap(int *,int *);

int main()
{
   int num1=10,num2=20;
   .....................
  Swap(&num1,&num2);
   .....................
}

void Swap(int *ptr1,int *ptr2)
{
    int temp;
    temp = *ptr1;
    *ptr1 = *ptr2;
    *ptr2 = temp;
}
```



Call by Address

- In call by address it will pass the address of the (variable) actual parameter.
- So it will affect original value that means it can be modified by the formal parameter.
- Thus it will change the value; it can affect inside the function as well outside the function values.
- Address operator (&) is used in actual parameter to call the function.
- Address is used for both formal and actual parameter. Same memory is used for actual and formal parameter. This address is used to access and change the value of the variable.

Difference between Call by value and Call by Address

Call by Value	Call by Address
A way of passing arguments to a function by copying the actual values of arguments into formal parameters of the function.	A way of passing arguments to a function by copying the address of the arguments to the formal parameters of the function.
Values of the actual parameters copy to the formal parameters of the function	Address of the actual parameters copy to the formal parameters of the function.
Does not affect the original value	Affects the original values.
At most one value at a time can be returned with an explicit return statement.	Multiple values can be returned to calling function and no explicit return stmt. Required.

Function Pointer

- We can also create a pointer pointing to a function.
- The code of a function always resides in memory, which means that the function has some address. We can get the address of memory by using the function pointer.

Syntax : return type (*ptr_name)(type1, type2…);

```c
#include<stdio.h>
void printhello();
void (*funp)();
int main()
{
        funp=printhello; // funp=&printhello;
        funp();          //(*funp)();
        return 0;
}
void printhello()
{
        printf("Hello");
}
```

```c
#include<stdio.h>
int add(int,int);
int (*funp)(int,int);
int main()
{
        int a=5,b=10,res;
        funp=add;       // funp=&add;
        res=funp(a,b);  // res=(*funp)(a,b);
        printf("%d",res);
        return 0;
}
int add(int a,int b)
{
        return a+b;
}
```

Function returning a Pointer

C allows to return a pointer from a function.

```
#include<stdio.h>
int* fun();
int main()
{
        int *p;
        p=fun();
        printf("%d",p); // address of v variable
        return 0;
}
int* fun()
{
        int v=10;
        return &v;
}
```

Note:-it is not recommended to return the address of a local variable outside the function as it goes out of scope after function returns.

Why are Pointers Used??

- To return more than one value from a function
- To pass arrays & strings more conveniently from one function to another
- To manipulate arrays more easily by moving pointers to them, Instead of moving the arrays themselves
- To allocate memory and access it (Dynamic Memory Allocation)
- To create complex data structures such as Linked List, Where one data structure must contain references to other data structures

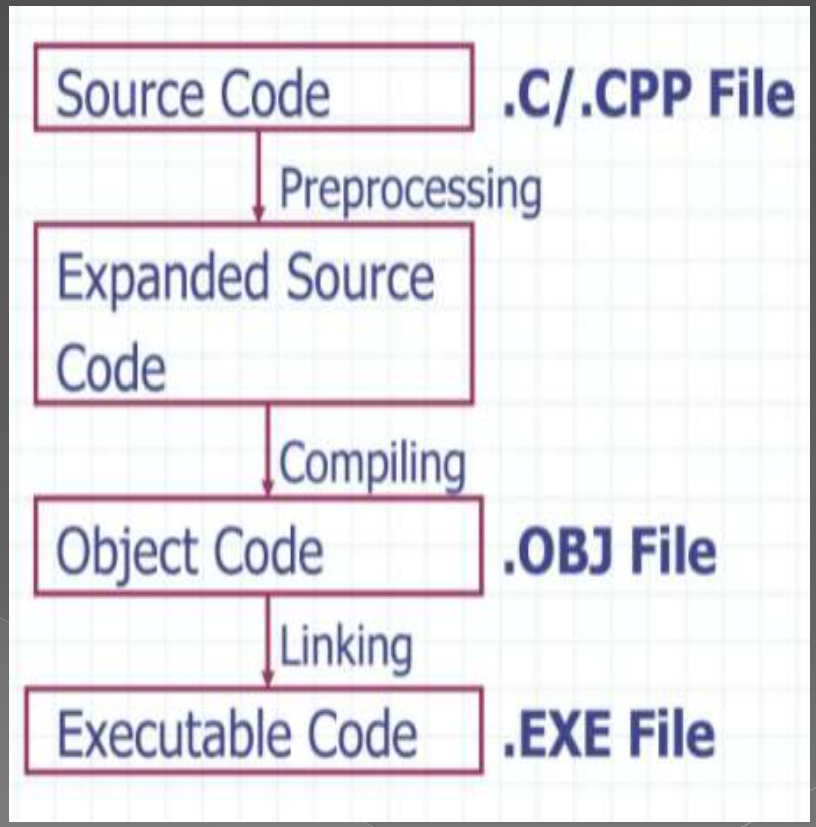
Advantages

- A pointer enables us to access a variable that is defined outside the function.
- Pointers reduce the length and complexity of a program.
- They increase the execution speed.
- The use of a pointer array to character strings results in saving of data storage space in memory.
- The function pointer can be used to call a function
- Pointer arrays give a convenient method for storing strings
- Many of the 'C' Built-in functions that work with strings use Pointers
- It provides a way of accessing a variable without referring to the variable directly

Pre-Processor Directives

Introduction

- Always start with # symbol.
- It is resolved before compilation and generates expanded source code.
- Text replacement/substitution.
- The directives can be placed anywhere in a program but most often placed at the beginning of the program, before the function definition.



Types

1. File Inclusion Directive
2. Simple pre-processor /macro
3. Function macro (having parameter)
4. Conditional compilation
5. Miscellaneous

Header File Inclusion

#include directive:

Puts copy of file in place of directive

Two forms

#include <filename>:

For standard library header files

Searches predesignated directories

#include "filename"

Searches in current directory

Normally used for programmer-defined files

Simple Macro

#define directive:

Constants represented as symbols
When program compiled, all occurrences replaced
Takes no memory
Name not be seen by debugger (only replacement text)
Do not have specific data type (Not type-safe)

Syntax:- #define macro_name macro_definition

Eg: #define PI 3.14 // Note:-No semicolon
Calling Macro- printf("%f", PI); //macro call

OR

area= radius *radius *PI;

printf("%f", 3.14f); //after replacement

area= radius *radius *3.14f; //after replacement

Function Macro

```
#define Square(x)  x*x
int main(){
        ..............
        printf("%d", Square(2)); //printf("%d",2*2);
       printf("%f",Square((2-1)); //printf("%f",2-1*2-1);
      printf("%f",Square(2.1f)); //printf("%f",2.1*2.1);
}
```

#undef directive:-

Undefines symbolic constant or macro

Can later be redefined

Syntax: #undef Square

Multiline macros

By using a the "\" to indicate a line continuation, we can write our macros across multiple lines to make them a bit more readable.
For instance, we could rewrite swap as

```
#define SWAP(a, b) {                \
                   int temp = b;    \
                   b = a;           \
                   a = temp;        \
}
```

Conditional Macro

1. #ifdef
2. #ifndef
3. #if
4. #elif
5. #else
6. #endif

```c
#include<stdio.h>
#define PI 3.14
int main(){
    #ifdef PI
        printf("%f",PI);
    #else
        printf("PI not defind");
    #endif
    return 0;
}
```

```
#include<stdio.h>
#define n -2
int main()
{
        #if(n<0)
                printf("n is negative..");
        #else
                printf("n is positive");
        #endif
                printf("\nvalue of n=%d",n);
                return 0;
}
```

Difference between Normal Function and Function Macro

Function	Macro
Multi-line/ complex logic	Single line logic
Works by shifting control between function calls	Works on text replacement/ substitution
Type-safe	Not type-safe
Use number of lines in code only once i.e for definition	Every macro call is replaced with definition so number of lines in code is increased.

Storage Classes

Storage Classes

Every variable in C programming has two properties: type and storage class.

Storage class tells-

1.where the variable is stored

2. initial value of the variable

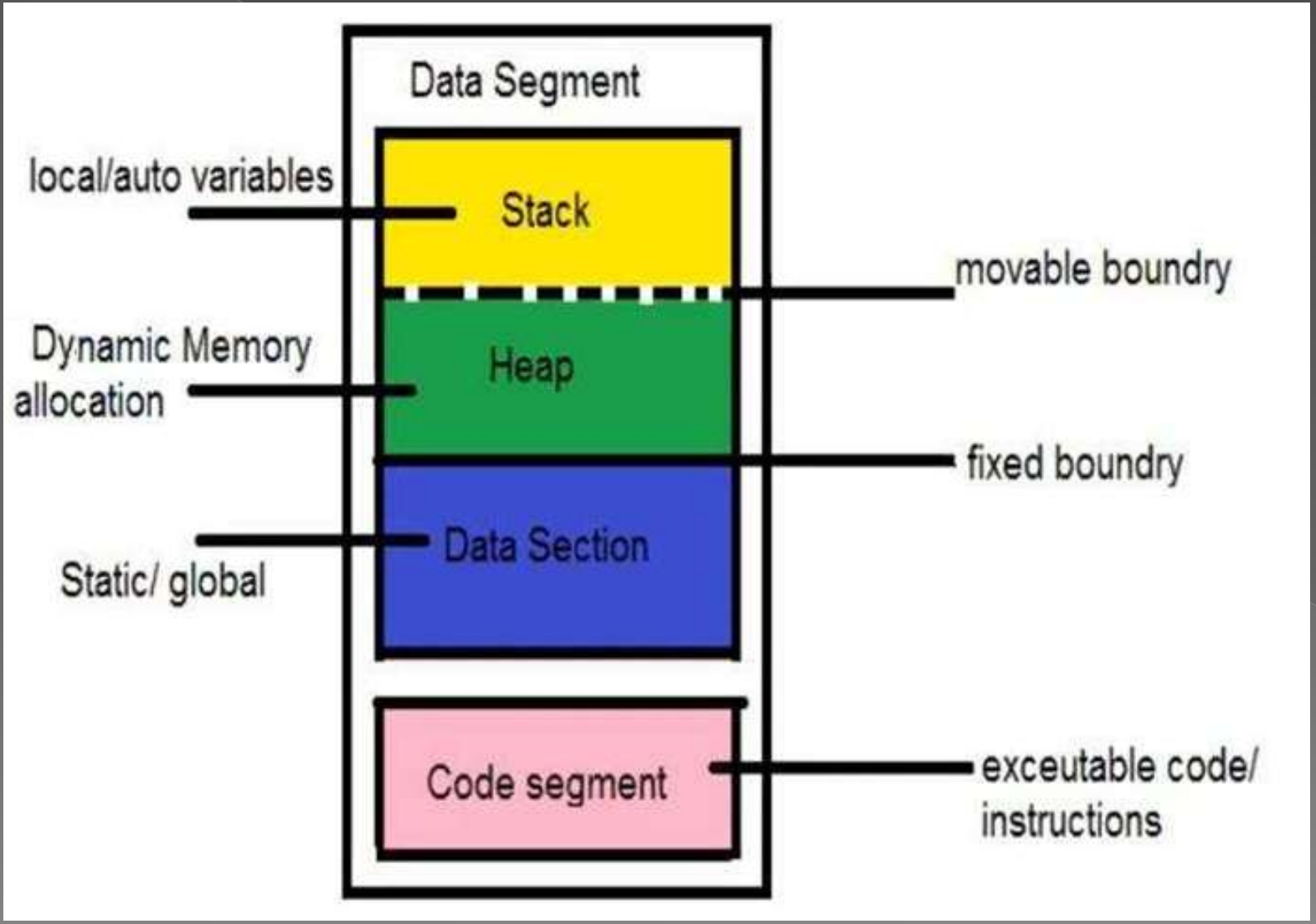
3. Scope of the variable (which a variable is accessed) and

4. lifetime of a variable.

There are 4 types of storage class:

1. automatic
2. external
3. static
4. register

Memory Allocation



1. Static Memory Allocation
2. Dynamic Memory Allocation

Static memory allocation is an allocation technique which allocates a fixed amount of memory during compile time and the operating system internally uses a data structure known as Stack to manage this.

Automatic Storage Class

- Automatic storage class Variable is stored in memory.
- Default value is garbage value
- Scope is local to the block where it is declared
- Life is, with in the block in where it is declared
- Automatic variable can be declared using the keyword auto

Eg: auto int a;

- By default all variables are automatic

  int a; same as auto int a;

Example 1:

```
main()
{
auto int i=10;
printf("%d",i);
}
```

Output: 10

Example 2:

```
auto int i;
```

Output: 1002

In example 1, i value is initialised to 10.So,output is 10.
In example 2, i value is not initialized. So,compiler reads i value is a garbage value.

Register Storage Class

- Variable is stored in a register instead of RAM.
- When a variable stored in register, it access high speed
- The no of registers are limited in a processor, if the registers are not free it will convert to automatic variable
- The mostly length of the registers are 2 bytes, so we cannot store a value greater than its size, if it so it will covert to automatic variable
- Register variable can be declared using the keyword register

Eg: register int a;

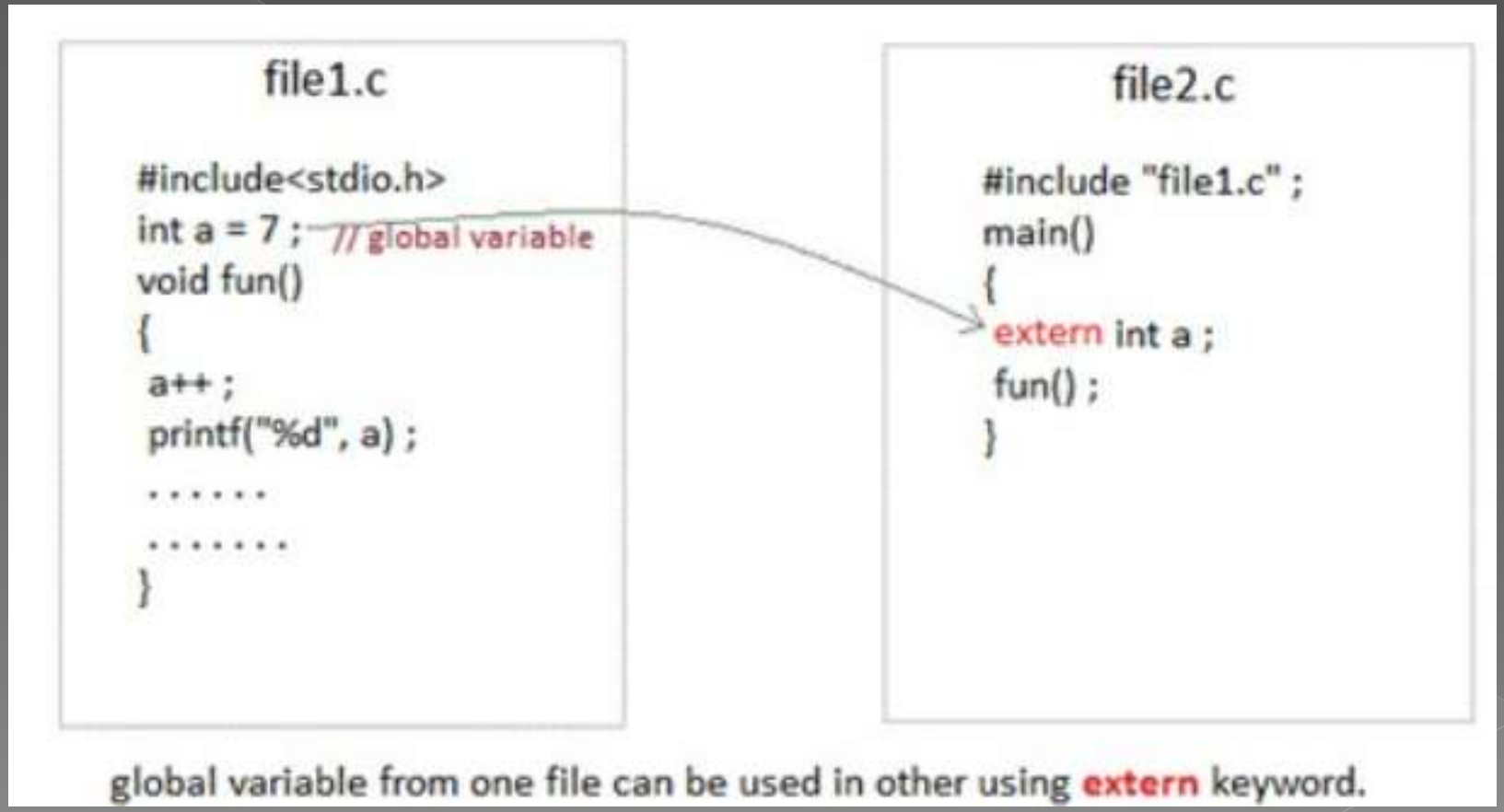
Extern Storage Class

- Default value is 0
- Scope is end of the program
- Life is, with in the block in end of the program
- external variable can be declared using the keyword extern
- extern is used to give a reference of a global variable that is visible to ALL the program files.

```
extern int a=10;
void main()
{
  void show();
    printf("a= %d",a);
    show();
}
void show()
{
    printf("%d",a);
}
Output   a=10    10
```

External Storage Class

The extern keyword is used before a variable to inform the compiler that the variable is declared some where else.



Static Storage Class

- Default value is 0
- Scope local to the block.
- Life is, value of the variable persists between different function calls.
- static variable can be declared using the keyword static

Eg: static int a;

- static can also be defined within a function.
- The variable cannot reinitialized when the function is called.
- This inside a function static variable retains its value during various calls.

Static Storage Class Example

```
main()
{
    void show();
    show();
    show();
}
void show() {
static int a=10;
printf("%d",a);
a++;
}
```

o/p will be 10 11

second time a=10; will not execute

Storage class	Auto	Static	Register	Extern
Default Initial Value	Garbage value	0 (Zero)	Garbage	0(Zero)
Location	RAM	RAM	CPU registers	RAM
Scope	Local to thevariable where the variable is defined	Local to the variable where the variable is defined	Local to the variable where the variable is defined	Entire Program
Life	As long as the control is within the block where the variable is defined	As long as the program is under execution	As long as the control is within the block where the variable is defined	As long as the program is under execution

Array

What is an Array ??

- An array is a collection/group of a fixed number of components wherein all of the components are of the same type referred to a same name.
- Array is a derived Datatype.

For Example : To store values- 5, 10, 15, 20, and 25.
Previously we would declare five variables: int iNum1, iNum2, iNum3, iNum4, iNum5;
By using array, int aiNum[5];

- Arrays can take any type (including the primitive data types)
- Array is a constant pointer.
- Like any other instances, arrays must be declared before use.
- Array always allocate single memory but can store multiple values all together at same time.

Types of Array

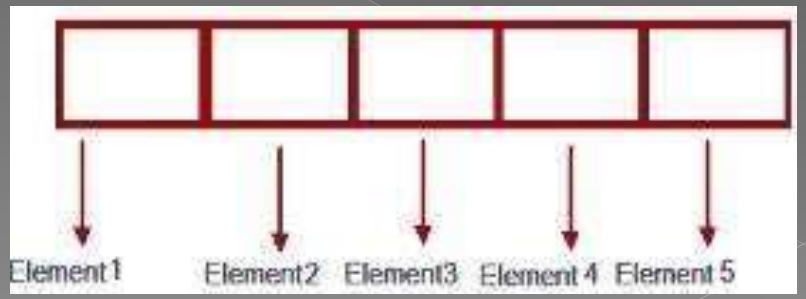
- One Dimensional Array
- Two Dimensional Array
- Multi Dimensional Array

One Dimensional Array

A variable which represent the list of items using only one index (subscript) is called one-dimensional array.

Syntax:- Datatype array_name[size];

For Example , if we want to represent a set of five numbers say(35,40,20,57,19), by an array variable number, then number is declared as follows int arr[5] ;



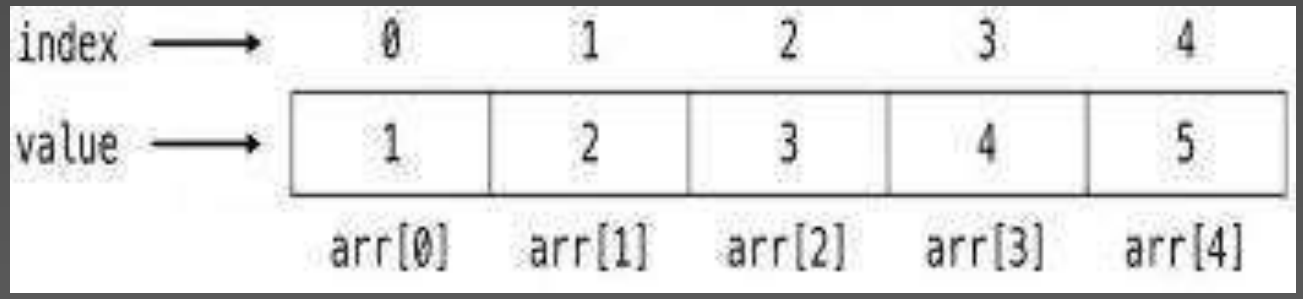
Declare and initialize -

int arr[5]= {35,40,20,57,19};

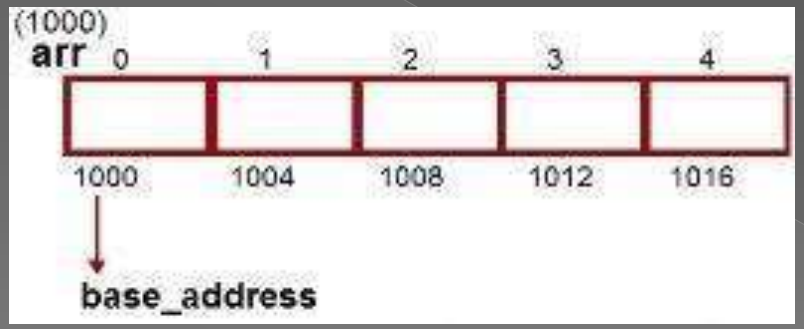
int arr[]={35,40,20,57,19};

For Ex - int arr[5]={1,2,3,4,5};

- Array index starts with zero.
- To access array element-
  printf("%d", arr[0]);



- To Assign value – arr[0]=20;
- Memory Allocation-



- Array name always store the base address i.e address of index zero.
- Array size can not be in negative.

Array Evaluation

Array name internally works as a pointer.

For Eg:- int arr[5]={1,2,3,4,5};

arr[2] => access index 2 value

*(arr+2)

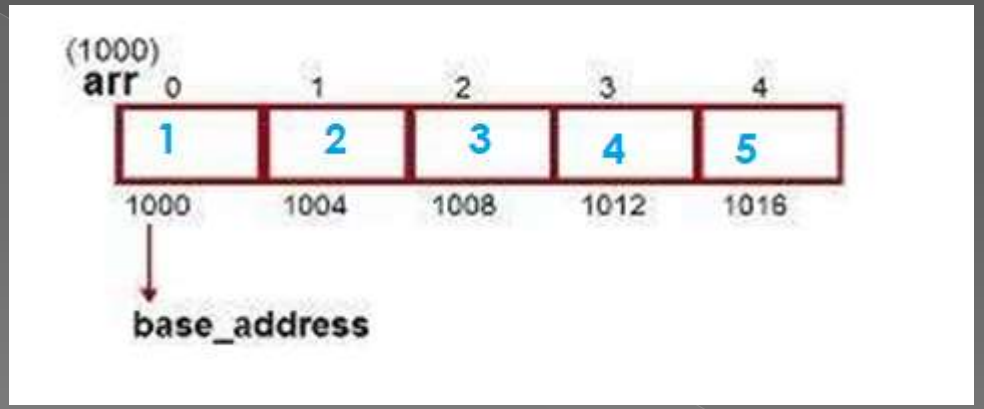
*(1000+2)

*(1000+2*4)

*(1000+8)

*(1008) (value at 1008)

= 3



Ways to Access Array Element

For ex – int arr[5]={1,2,3,4,5};

To access value at index 2 we can write –

1. arr[2]
2. 2[arr]
3. *(arr+2)
4. *(2+arr)

Pointer to an Array

Base address: Base address i.e. address of the first element of the array (also allocated by the compiler).

```
int i;
int a[5] = {1, 2, 3, 4};
int *p = a;     // same as int*p = &a[0]

for (i = 0; i < 5; i++)
{
    printf("%d ", *p);
    p++;
}
```

```
#include<stdio.h>
#define Size 5
int main(){
        int arr[Size]={1,12,3,4,5};//assume base address is 100
                                   //100,104,108,112,116
        int *p,*q;
        p=&arr[1]; //(arr+1)
        q=&arr[4]; // (arr+4)
        printf("%d\n",p<q); //104<116 => true (1)
        printf("%d\n",*p<*q);  // 12<5 => false (0)
        printf("%d\n",p==q); // 104!=116 false(0)
        p+=3; // p=p+3 => &arr[4]
        printf("%d\n",p==q); //116==116 True(1)
        return 0;
}
```

Pass arrays to a function

Method 1: Single element of an array can be passed in similar manner as passing variable to a function.

```
void printdata(int);

int main()
{
int arr[5]={1,2,3,4,5};
printdata(arr[1]); //passing array element arr[1] only
return 0;
}

void printdata(int n)
{    printf("%d",n);  }
```

Pass an entire 1D arrays to a function

Method 2: While passing array as argument to the function, only the name of the array is passed. (i.e base address)

Note: - Array never pass by value..always pass by address.

```
void printdata(int[],int);
int main()
{
        int arr[5]={1,2,3,4,5};
        int i;
        printdata(arr,5);
return 0;
}
```

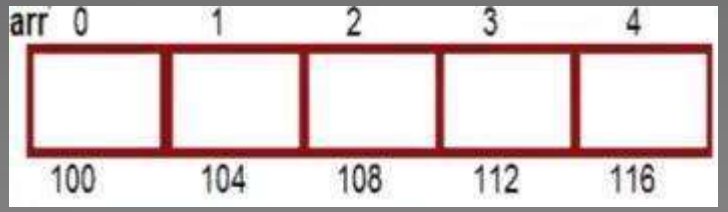
```
void printdata(int a[],int size)
{
    int i;
    for(i=0;i<size;i++)
        printf("%d",a[i]);
}
```

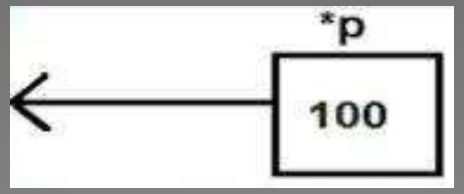
Pass an entire 1D arrays to a function

Method 3:

```
Void getdata(int *,int);

int main()
{
int arr[5];
getdata(arr,5);
 return 0;
}
```

```
void getdata(int *p,int size)
{
    int i;
    printf("enter values");
    for(i=0;i<size;i++)
          printf("%d",*(p+i));
}
```





Two Dimensional Array

2D Array is used to store m*n elements i.e multiple rows with columns.

2D array is collection of 1D array.

Eg:-

I need to store of 10 students each in 3 modules, so instead of declaring 10 1D array we can go for declaring single 2D array for all 10 students.

Declaration and Initialization

To declare 2D array, we merely use 2 set of square brackets.

- the first contains the number of rows.
- the second contains the number of columns.

Syntax- Data_type array_name[rowsize][colsize]

```
int arr[3][3];
int arr[3][3]={1,2,3,4,5,6,7,8,9};
                    OR
int arr[3][3]={{1,2,3},{4,5,6},{7,8,9}};
int arr[][3]={{1,2,3},{4,5,6}};
```

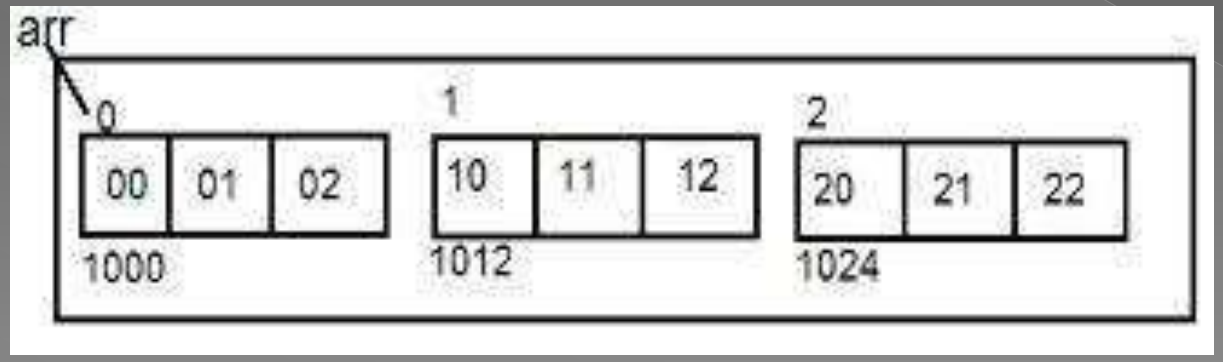
Memory Allocation

For Ex - int arr[3][3];
Here, we have 3 1ID array's and each of 3 element (columns)

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

arr

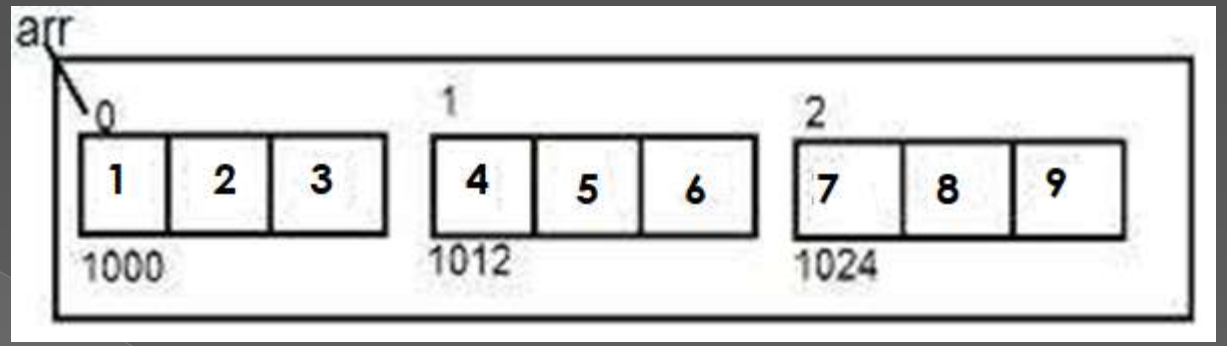
	Column 0	Column 1	Column 2
Row 0	1000	1004	1008
Row 1	1012	1016	1020
Row 2	1024	1028	1032



Access Array Elements

int arr[3][3]={{1,2,3},{4,5,6},{7,8,9}};

1. arr[1][2] // 5
2. *(*(arr+1)+2) //5
3. 2[1[arr]] //5
4. *(arr[1]+2) //5



arr = Base Address of 2D array

arr[0] = Base Address of 1st 1D array of 3 elements in 2D array

arr[1] = Base Address of 2nd 1D array of 3 elements in 2D array

arr[2] = Base Address of 3rd 1D array of 3 elements in 2D array

int arr[3][3]={{1,2,3},{4,5,6},{7,8,9}};

- arr[1][2] //row no=1 & col=2
- *(*(arr+1)+2)
- *(*(1000+1)+2)
- *(*(1000+1*3*4)+2) //3-col size and 4 is sizeof nit
- *(*(1000+12)+2)
- *(1012)+2) //inner * has been evaluated
- *(1012+2*4) //same as 1D array
- *(1012+8)
- *(1020)
- =6

Pass 2D arrays to a function

2D arrays are always passed by address.

```
#include<stdio.h>
#define ROW 2
#define COL 3
int main(){
        int arr[ROW][COL],i,j;
        printf("Enter %d * %d elememts",ROW,COL);
        for(i=0;i<ROW;i++){
                for(j=0;j<COL;j++)
                        scanf("%d",&arr[i][j]);
        }
        display(arr);
        return 0;
}
```

```
void display(int a[ROW][COL]){
        printf("\n Array elements =\n");
        int i,j;
        for(i=0;i<ROW;i++){
                for(j=0;j<COL;j++)
                        printf("%d\t",a[i][j]);
                printf("\n");
        }
}
```

Pass 2D arrays to a function

```
#include<stdio.h>
#define ROW 2
#define COL 3
int main(){
        int arr[ROW][COL],i,j;
        printf("Enter %d * %d elememts",ROW,COL);
        for(i=0;i<ROW;i++){
                for(j=0;j<COL;j++)
                        scanf("%d",&arr[i][j]);
        }
        display(arr);
        return 0;
}
```

```
void display(int (*a)[COL]){
        printf("\n Array elements =\n");
        int i,j;
        for(i=0;i<ROW;i++){
                for(j=0;j<COL;j++)
                        printf("%d\t",a[i][j]);
                printf("\n");
        }
}
```

Pass 2D arrays to a function without using [ ]

Typecast 2D array into 1D array - by this we can fetch whole 2D array data by single pointer.

```
#include<stdio.h>
#define ROW 2
#define COL 3
int main(){
        int arr[ROW][COL],i,j;
        printf("Enter %d * %d elememts",ROW,COL);
        for(i=0;i<ROW;i++){
                for(j=0;j<COL;j++)
                        scanf("%d",&arr[i][j]);
        }
        print((int *)arr);
        return 0;
}
```

```
void print(int *a){
        printf("\n Array elements =\n");
        int i,j;
        for(i=0;i<ROW;i++){
                for(j=0;j<COL;j++)
                        printf("%d\t",*((a+i*COL))+j);
                printf("\n");
        }
}
```

Character array and strings

A string is a sequence of characters terminated with a null character \0 (Back-slash zero) .

Same as array of integer we can have array of characters. Each character within the string is stored within one element of array successively.

When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character \0 at the end by default.

Declaration and Initialization

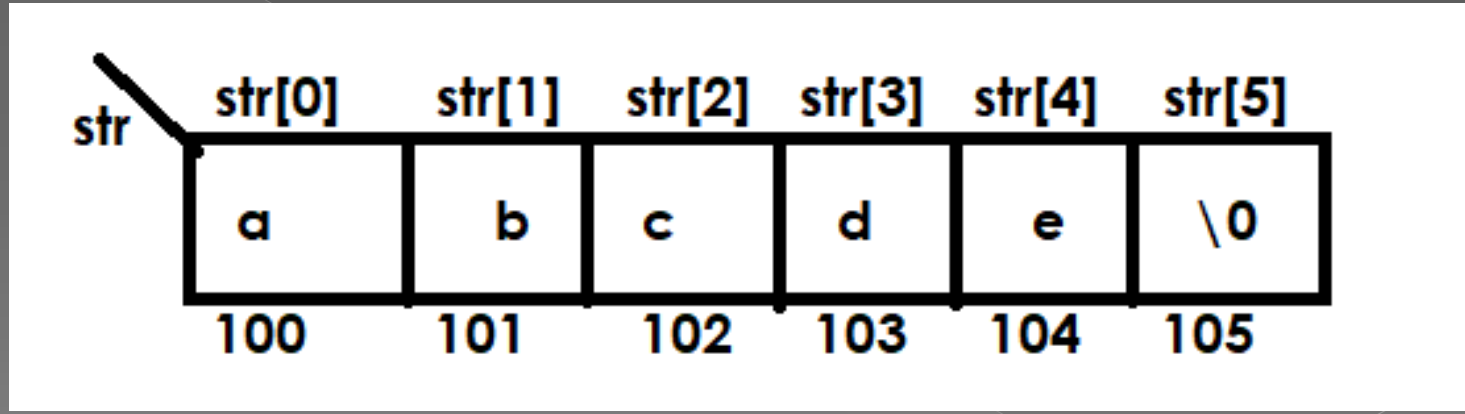
Character array:

```
char str[6]={'a','b','c','d','e','\0'};
char str[]={'a','b','c','d','e','\0'};
```

OR

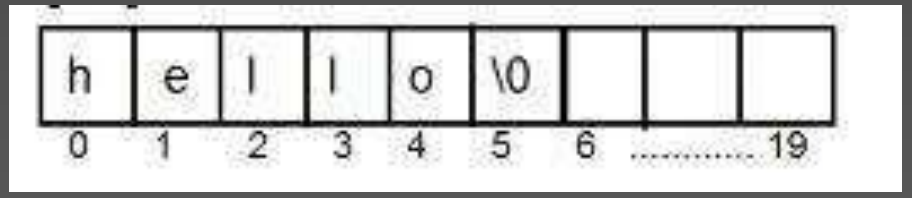
String:

```
char str[6]={"abcde"};
char str[6]="abcde";
char str[]={"abcde"};
```



Display String

char str[20]="hello"



Format specifier - %s

printf("%d",str); //base address

printf("%s",str); //hello

printf("%c",str[0]); // h

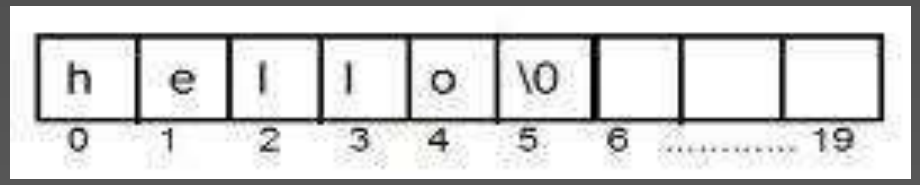
OR

puts(str); //hello

Accept String from User

char str[20];

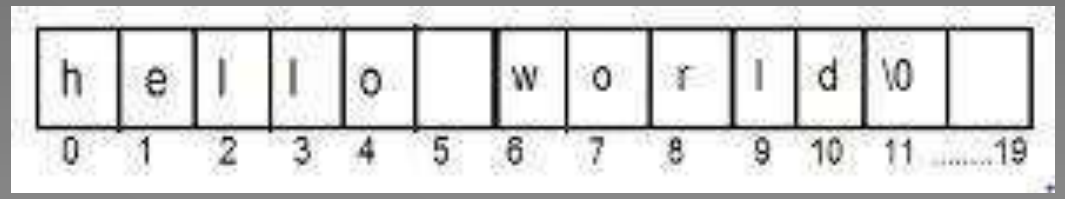
scanf("%s",str);



Note:- no & symbol in scanf. scanf accept string until whitespace or enter key pressed.

gets(str);

gets() accept string with space



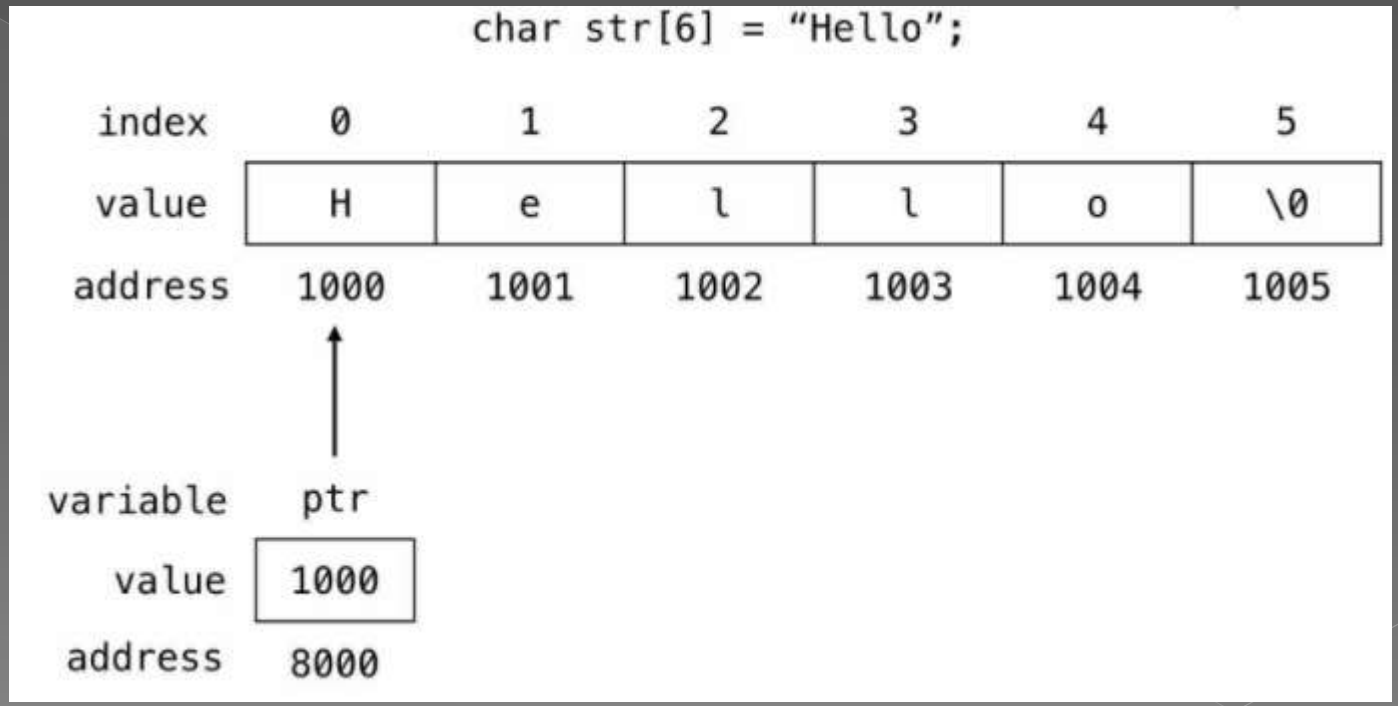
Pointers and Strings

Suppose we wish to store “hello”. We may either store it in a string or we may ask C complier to store it at some location in memory and assign the address of the string to char pointer.

```
char str[6 ]= “Hello”;
char  *ptr = “Hello”;
```

OR

```
char str[6 ]= “Hello”;
char *ptr=str;
```



Some points to remember –

1. We can not assign a string to another, whereas we can assign a char pointer to another char pointer.

```
char str1[]="Hello";
char str2[10];
char *s="Good";
char *q;
str2=str1; //error
q=s; //work
```

2. Once a string has been defined, it can not be initialized to another set of characters. Unlike strings, such an operation is perfectly valid with char pointer.

```
char str1[]="Hello";
char *s="Good";
str1="Bye"; //error
s="Bye"; //works
```

Passing String to Function

```c
#include <stdio.h>
void Display(char ch[]);
int main(){
    char c[50];
    printf("Enter string: ");
    gets(c);
    Display(c);       // Passing string c to function.
    return 0;
}
void Display(char ch[]){
    printf("String Output: ");
    puts(ch);
}
```

String Handling Functions - strcat()

There are various string handling functions define in string.h

strcat(str1,str2)
Function: Concatenates 2 strings (str2 is appended to str1).
Return value: The address of str1.

We can't use such statement to join 2 strings together: str1 = str1 + str2;
str1 and str2 both are address values;
str1 is an address constant, and it can't be assigned by any value again.

String Handling Functions -strcpy()

strcpy (str1, str2)
Function: Copies one string (str2) over another (str1).
Return value: The address of str1.

We can't use an assignment statement to assign any string to a character array.(Except the initialization)

```
char   str1[10] = "How ", str2[10];
str2 = "How ";
str1 = str2;
strcpy ( str1, str2 );
```

String Handling Functions-strcmp()

strcmp (str1, str2)
Function: Compares two strings. It compares the characters of st1 and str2 one by one. And it will stop when it finds the first different character or it encounters one null character.
Return value: When the function stops the comparison, the difference of the ASCII value between the current character of str1 and that of str2.

```
strcmp ( "A", "B" ) => 'A' – 'B' => -1
strcmp ( "a", "A" ) => 'a' – 'A' => 32
strcmp ( "ABC", "AB" )     => 'C' – '\0' = 67
strcmp ( "computer", "compare" )  => 'u' – 'a' = 20
strcmp ( "36", "3654" )     => '\0' – '5' = -53
```

String Handling Functions - strlen()

strlen (str)

Function: Return the length of the string (the number of the characters of str except the null character).

For these following declarations, what is the value of the function strlen(s)?

(1) char s[10] = {'A', '\0', 'B', 'C', '\0', 'D'}; 1 ("A")

(2) char s[ ]="\t\b\\\0will\n"; 3 ("\t\b\\")

(3) char s[ ]="\x69\082\n"; 1 ("i")

Array of String

Arrays of strings can be declared and handled in a similar manner to that described for 2-D Dimensional arrays.

Syntax: <data type><name of string>[number of strings] [number of characters];

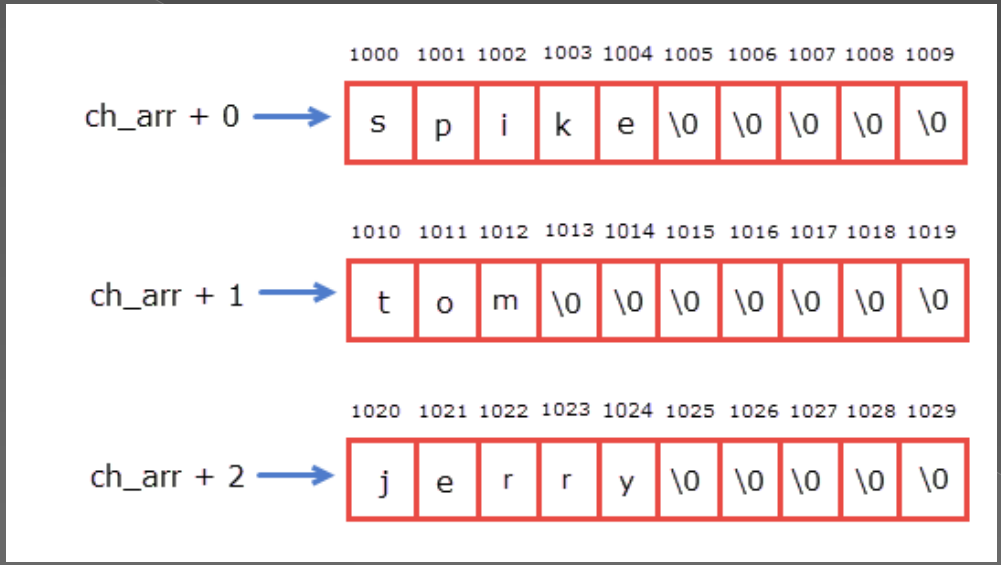
Ex- char ch_arr[3][10];
Here, ch_arr[3][10] means 3 names having 10 characters each.

```
char ch_arr[3][10] = {
                        {'s', 'p', 'i', 'k', 'e', '\0'},
                        {'t', 'o', 'm','\0'},
                        {'j', 'e', 'r', 'r', 'y','\0'}
                    };
```

```
char ch_arr[3][10] = {
                                "spike",
                                "tom",
                                "jerry"
                            };
```

The ch_arr is a pointer to an array of 10 characters or int(*)[10].

Therefore, if ch_arr points to address 1000 then ch_arr + 1 will point to address 1010.



From this, we can conclude that:
ch_arr + 0 points to the 0th string or 0th 1-D array.
ch_arr + 1 points to the 1st string or 1st 1-D array.
ch_arr + 2 points to the 2nd string or 2nd 1-D array.

*(ch_arr + 0) + 0 points to the 0th character of 0th 1-D array (i.e s)
*(ch_arr + 0) + 1 points to the 1st character of 0th 1-D array (i.e p)
*(ch_arr + 1) + 2 points to the 2nd character of 1st 1-D array (i.e m)

```
#include<stdio.h>
int main()
{
    int i;
    char ch_arr[3][10] = {"spike","tom","jerry"};
    for(i = 0; i < 3; i++)
    {
        printf("string = %s \t address = %u\n", ch_arr + i, ch_arr + i);
    }
    return 0;
}
```

```
char city[4][12] = {
  "Chennai",
  "Kolkata",
  "Mumbai",
  "New Delhi"
};
```

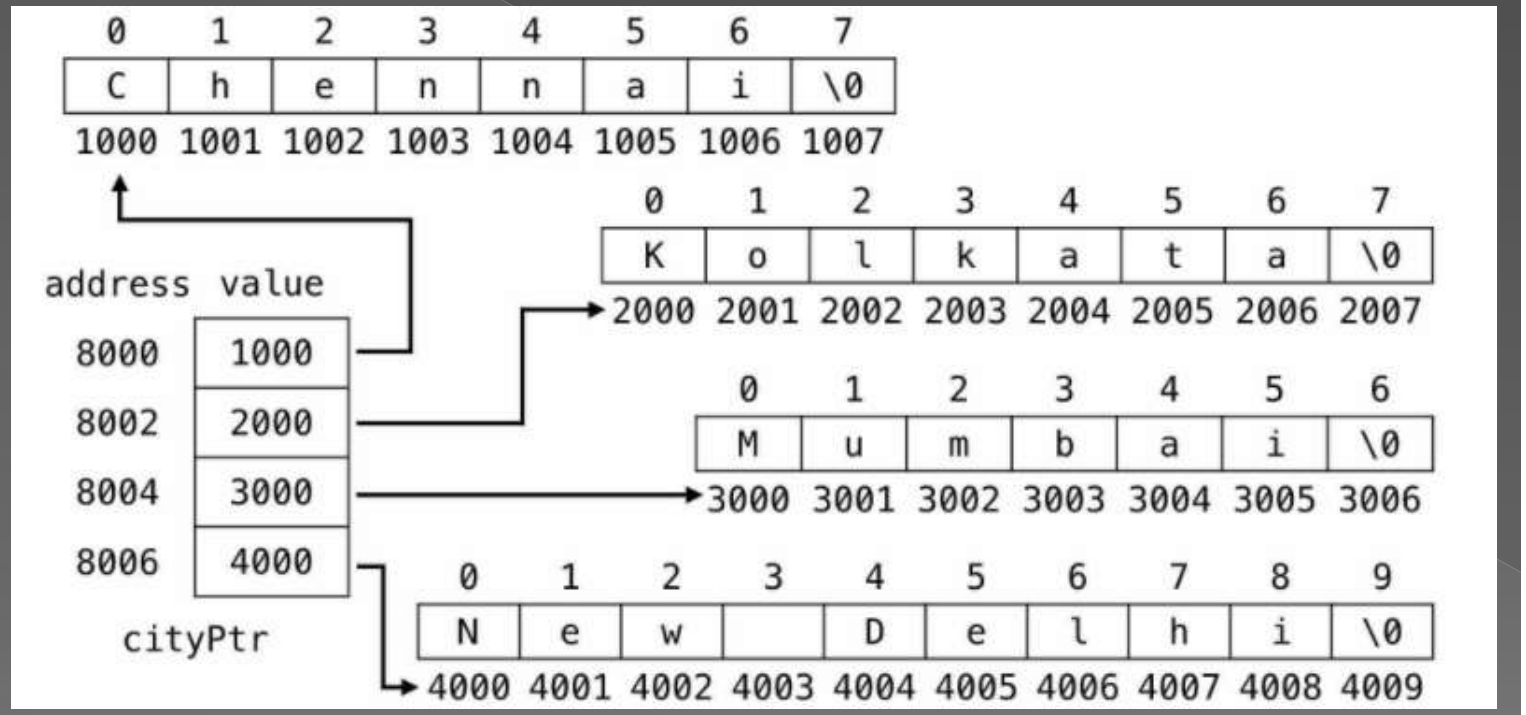
	0	1	2	3	4	5	6	7	8	9	10	11
0	C	h	e	n	n	a	i	\0				
1	K	o	l	k	a	t	a	\0				
2	M	u	m	b	a	i	\0					
3	N	e	w		D	e	l	h	i	\0		

The problem with this approach is that we are allocating 4x12 = 48 bytes memory to the city array and we are only using 33 bytes.

We can save those unused memory spaces by using pointers as shown below.

```
char *cityPtr[4] = {
  "Chennai",
  "Kolkata",
  "Mumbai",
  "New Delhi"
};
```

	0	1	2	3	4	5	6	7	8	9
0	C	h	e	n	n	a	i	\0		
1	K	o	l	k	a	t	a	\0		
2	M	u	m	b	a	i	\0			
3	N	e	w		D	e	l	h	i	\0




```c
#include <stdio.h>
int main() {
   char *cityPtr[4] = {
    "Chennai",
    "Kolkata",
    "Mumbai",
    "New Delhi"
  };

  int r, c;
  for (r = 0; r < 4; r++) {
    c = 0;
    while(*(cityPtr[r] + c) != '\0') {
      printf("%c", *(cityPtr[r] + c));
      c++;
    }
    printf("\n");
  }
  return 0;
}
```

Pass Array of String to Function

```
#include <stdio.h>
void display(char **ptr , int count)
{
    int i=0;
    for(i=0;i<count;i++)
    {
        printf("\n Name [%d] : %s",i,ptr[i]);
    }
}
int main()
{
   char *name[4] = {"Ram" , "Shani" , "Kiran"};
   // Passing array of strings to Function
   display(name,3); // here 3 is the number of strings
   return 0;
}
```

Dynamic Memory Allocation

Memory Allocation

The blocks of information in a memory system is called memory allocation.

To allocate memory it is necessary to keep in information of available memory in the system. If memory management system finds sufficient free memory, it allocates only as much memory as needed, keeping the rest available to satisfy future request.

In memory allocation has two types. They are **static** and **dynamic** memory allocation.

Static memory allocation

- It is used to allocate the memory at compile time. Size is fixed when program is created.
- It can be allocate the memory faster than the dynamic allocation. More memory space is required.

Dynamic memory allocation

- It is used to allocate the memory at runtime. It can used to release the unwanted memory space during the program execution.
- It is used to modify the size of the previously allotted memory space.
- It allots memory space to array of elements and initialized to them 0.
- Less memory space is required. Slower than the static allocation.

Library Functions

It has 4 functions of stdlib.h header file.
When you allocate memory at run-time it will allocate space on Heap Memory section.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	deallocate the previously allocated space
realloc()	Change the size of previously allocated space

malloc()

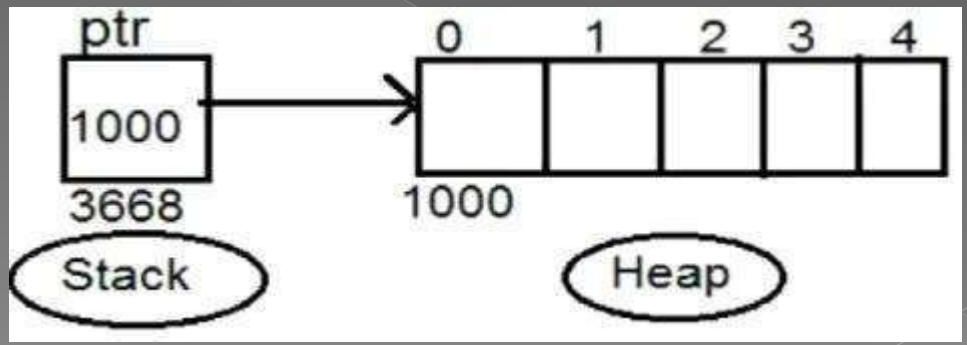
Syntax

ptr_name=(void*)malloc(size);

Eg:- int *ptr,n;

ptr= (int*)malloc(n*sizeof(int));

It initializes memory to garbage value.

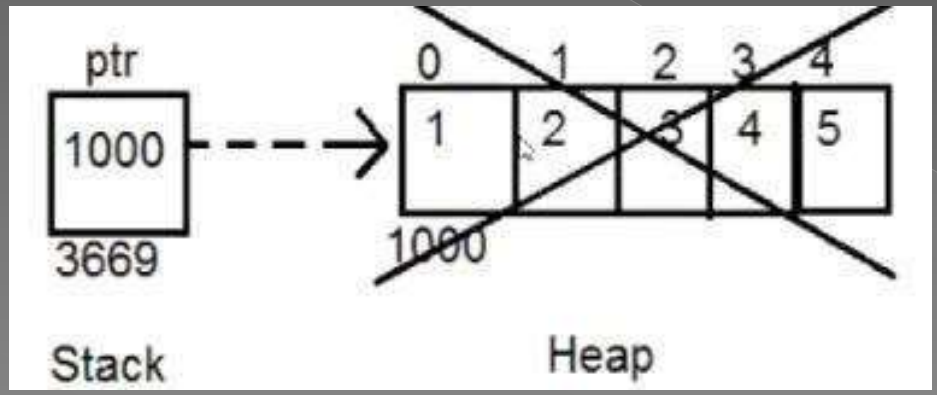


free()

- If we don not deallocate then it leads to memory leakage.
- It is important to release memory allocated on heap after you complete your use.

Syntax: free(pointer)

free(ptr);



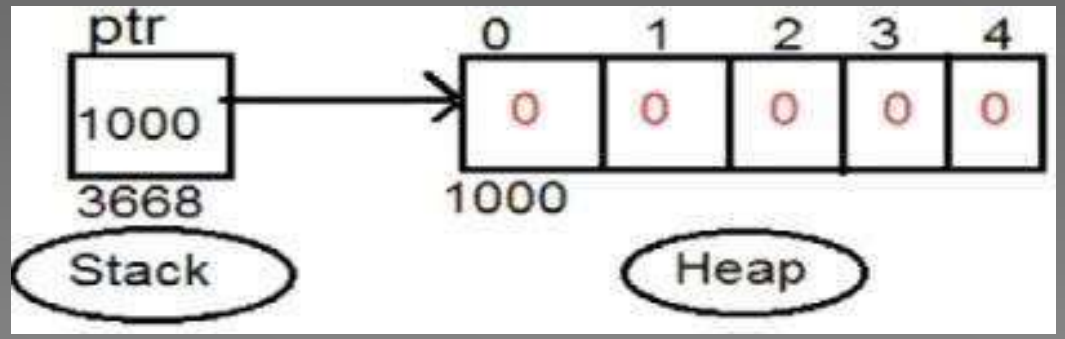
calloc()

Synatx:- (void*)calloc(n,element_size)

Eg:-

```
int *ptr,n;
ptr=(int*)calloc(n,sizeof(int));
```

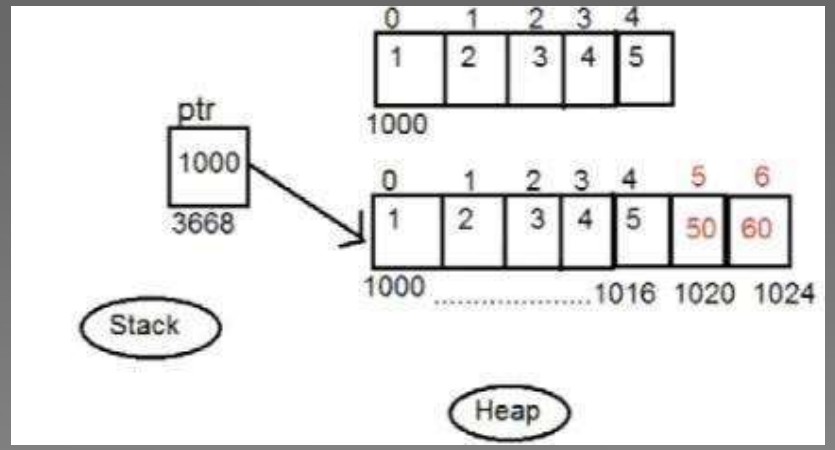
calloc() initalizes memory to zero (0).

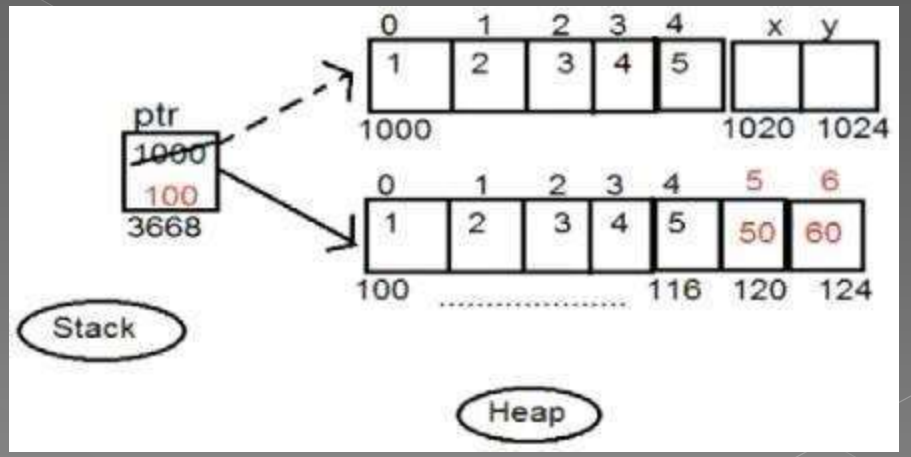


realloc()

- realloc( ) It is used to modify the size of allocated memory.
- If memory is not enough for malloc( ) and calloc( ), then use the realloc( ) function for the change memory size.
- Syntax:- (void*)realloc(pointer,new_size*sizeof(datatype));

Eg:- ptr=(int*)realloc(ptr,(n+new_size)*sizeof(int));

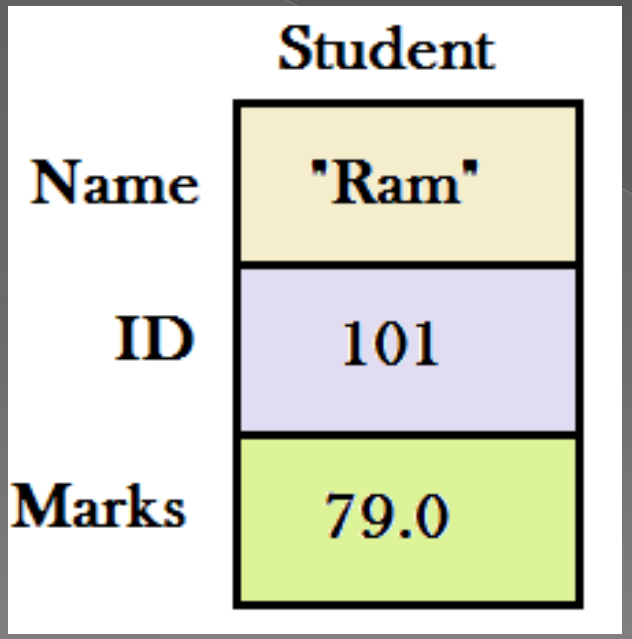




Structure
&
Union

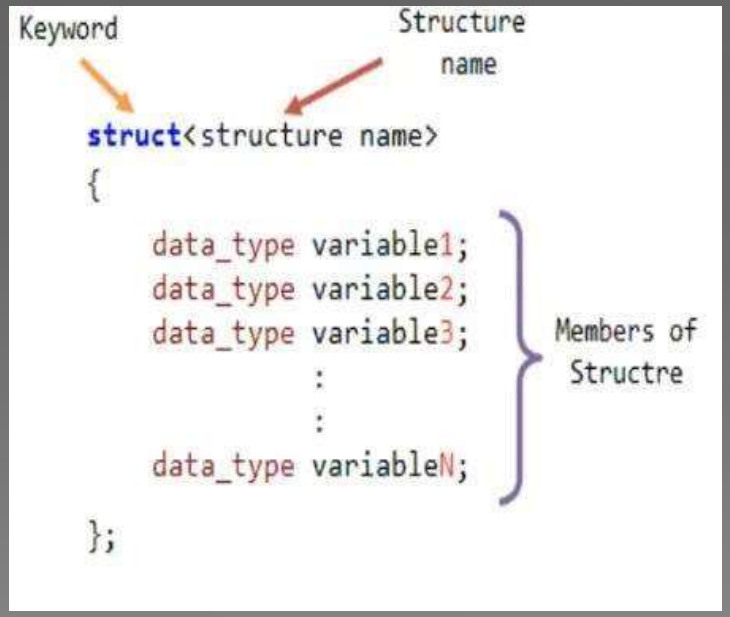
Need

Suppose I want to store a student details all together into a single block of information.



Structure

- Structure is a collection of different data-types or heterogeneous data under a single name.
- It is like a blueprint so when we declare structure never memory space is allocated at point of declaration.
- It allocate space when we use that structure.



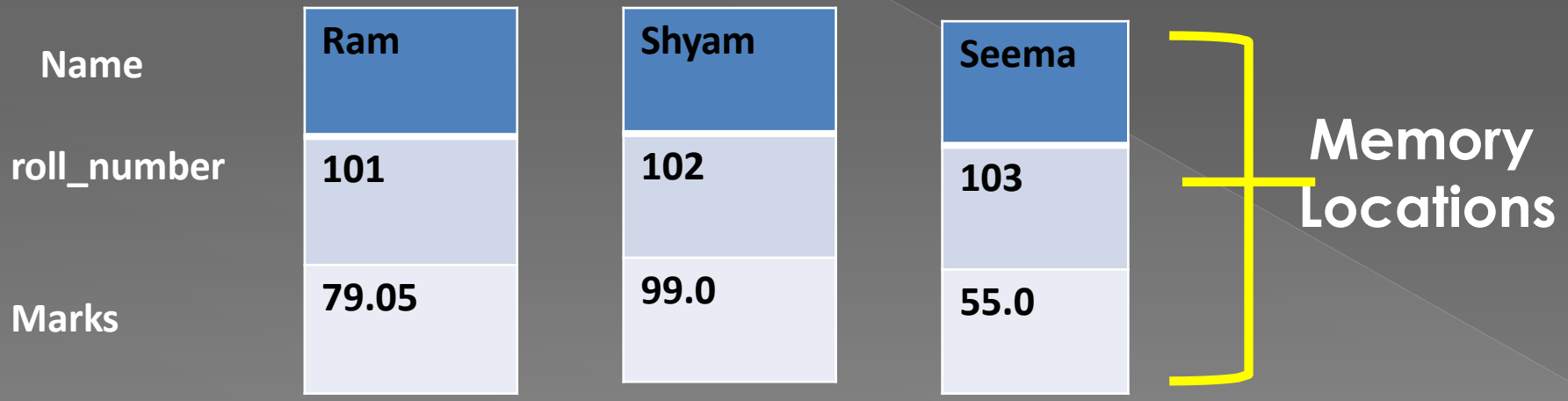
struct keyword

tag or structure tag

```
struct Student
{
   char name[20];
   int roll_number;
   float marks;
} ;
```

Members or fields of structure

Structure Variables



Declaring structure variable

```
struct Student
{
   char name[20];
   int roll_number;
   float marks;
   }st1,st2;

int main(){
struct Student s1,s2;

struct Student stud1={"Rekha",130,89.0f}; // Pre-Initialization
}
```

typedef

- If the structure name is not meaningful or if it is too long to use again-n-again then you can rename structure.

- This is done by using keyword "**typedef**".

```
typedef struct   point
{
    int x;
    int y;
 }Point;
```

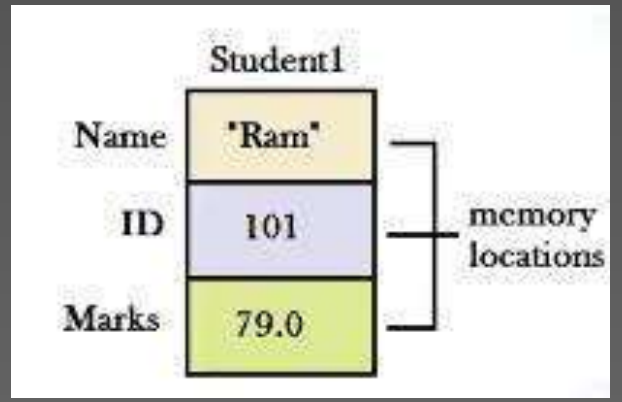
```
typedef struct
{
    double radius;
    int x;
    int y;
} Circle;
```

```
struct Student
{
    char name[20];
     int rolllno;
}
typedef struct Student st1;
```

Assign and Access structure variable

To access any element-
struct_var.element (using dot/period operator)
Eg-

```
typedef struct Student
{
char name[20]; int id ;float marks;
}stud;

int main(){
        stud Student1;
        strcpy(Student1.Name,"Ram");
        Student1.id=101;
        Student1.Marks=79.0;
        printf("%d %s %f",Student1.id, Student1.Name, Student1.Marks );
```

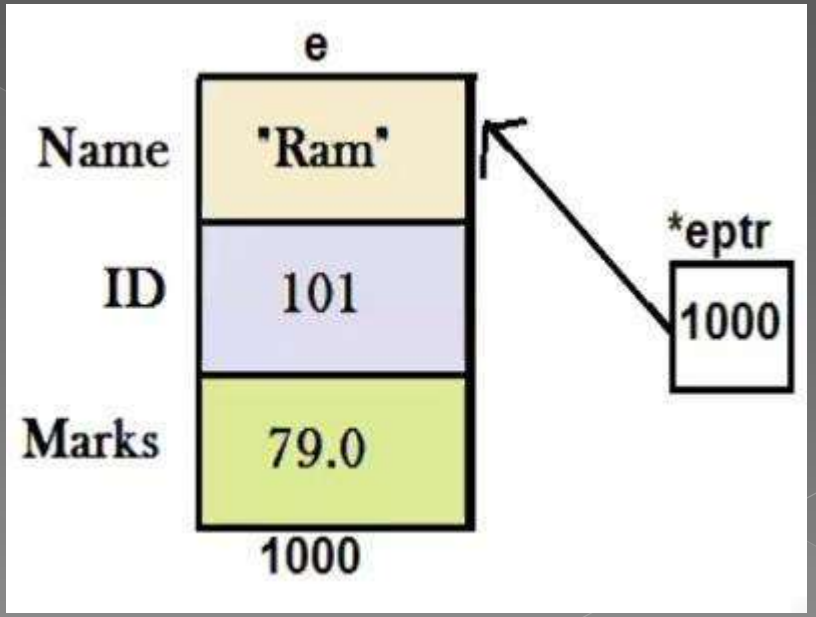


Coping one structure variable to other -
Student1=Student2;

Student1.id=Student2.id; //memberwise copy
strcpy(Student1.name,Student2.name);

Structure Pointer

```
int main(){
        stud e={“Ram”,101,79};
        stud *eptr;
        eptr=&e;
printf(“%s %d %f”,eptr->name, eptr->id,  eptr->marks)
```



Size of Structure

Sum of size for all elements

Eg:

Name=20

Id=4

Marks=4

Total=28bytes

Array of Structures

```
stud stud_arr[3];
for(i=0;i<3;i++)
    printf("%s%d%f",stud_arr[i].name,
                    stud_arr[i].id,
                    stud_arr[i].marks);
```

	0	1	2
Name	"Ram"	"Mohan"	"Rohan"
ID	101	102	103
Marks	79.0	99.0	55.0

Pass structure member in Function

```
typedef struct Student{
        int rollno;
        char name[20];
        float marks;
}stud;
void displayrollno(int);
int main(){
        stud st1={10,"Ram",50};
        displayrollno(st1.rollno); // displayrollno(10);
        return 0;
}

void displayrollno(int r){ // int r=st1.rollno ; r=10;
        printf("rollno=%d",r);
}
```

Pass by Value

```
typedef struct Student{
        int rollno;
        char name[20];
        float marks;
}stud;
void display(stud);
int main(){
        stud st1={10,"Ram",50};
        display(st1); // pass by value
        return 0;
}

void display(stud s){ //stud s=st1; // copy st1 struct var. into  s
        printf("\nRollno=%d,name=%s,marks=%.2f",s.rollno,s.name,s.marks);
}
```

Pass by Address

```
typedef struct Student{
        int rollno;
        char name[20];
        float marks;
}stud;
void display(stud);
void accept(stud *);
int main(){
        stud st1;
        accept(&st1); // pass by address
        display(st1); // pass by value
        return 0;
}
void accept(stud *sptr){ // stud *sptr=&st1;
        printf("enter rollno,name and marks");
        scanf("%d",&sptr->rollno); // 10
        fflush(stdin);
        gets(sptr->name);
        scanf("%f",&sptr->marks);
}
void display(stud s){ //stud s=st1; // copy st1 struct var. into  s
        printf("\nRollno=%d,name=%s,marks=%.2f",s.rollno,s.name,s.marks);
}
```

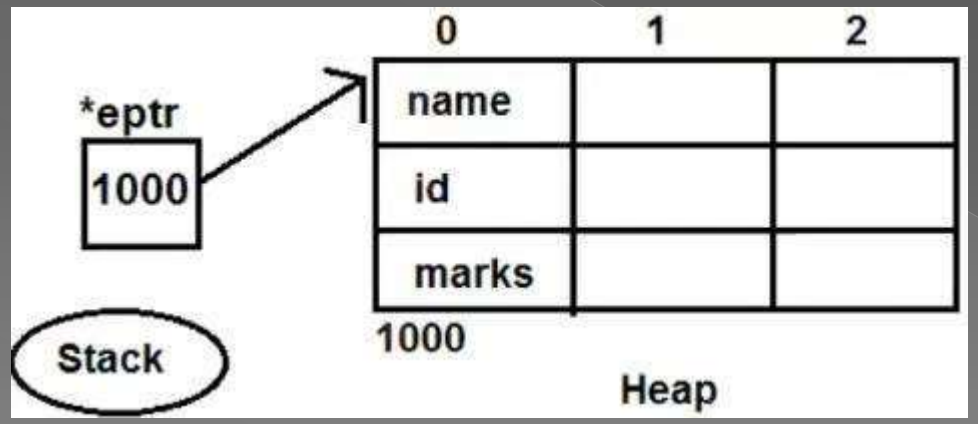
return structure from Function

```
typedef struct Student{
        int rollno;
        char name[20];
        float marks;
}stud;
void display(stud);
stud accept();
int main(){
        stud st1;
        st1=accept();
        display(st1); // pass by value
        return 0;
}
stud accept(){
        stud s1;
        printf("enter rollno,name and marks");
        scanf("%d",&s1.rollno);
        fflush(stdin);
        gets(s1.name);
        scanf("%f",&s1.marks);
        return s1;
}
void display(stud s){ //stud s=st1; // copy st1 struct var. into  s
        printf("\nRollno=%d,name=%s,marks=%.2f",s.rollno,s.name,s.marks);
}
```

DMA for structure Element

Allocate memory for single structure element or for multiple elements is same as allocating memory for built-in type.

*eptr=(*struct Student)malloc(n*sizeof(struct Student));



Nested Structure

```
struct Date{
int day,mon,year;
};
```

```
struct Employee
{
    int eid;
    char ename[20];
    float salary;
    struct Date doj;
};
```

```
struct Student
{
    int id;
    char name[20];
    float marks;
    struct Date dob;
};
```

Nested Structure

```
struct Employee
{
    int eid;
    char name[20];
    float salary;
     struct Date{
        int day,mon,year;
    }doj;
};
```

Access Inner structure Members

Outer_struct_var . Inner_struct_var . Inner_element

Eg:-

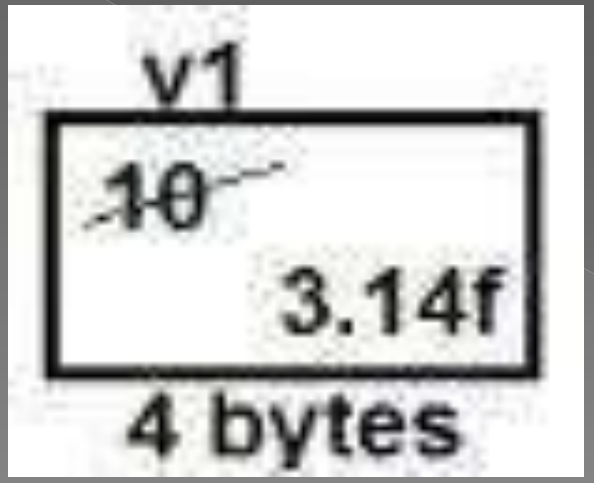
```
Emp e;
e.doj.day;
e.doj.mon;
e.doj.year;
```

Union

- Union is an user defined data-type same as structure.
- It is a collection of variables of different data types in same memory location.
- Union provides an efficient way of using the same memory location for multiple purpose.
- We can define a union with many members, but at a given point of time only one member can contain a value.

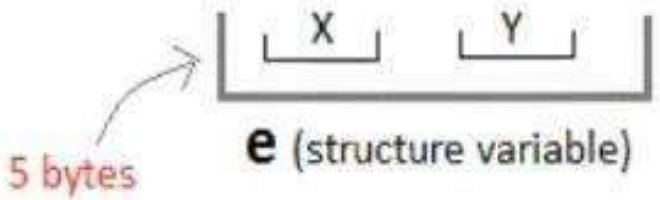
```
union value
{
    int n1;
    float n2;
};
```

```
int main(){
union value v1;
  v1.n1=10;
v1.n2=3.14;
}
```



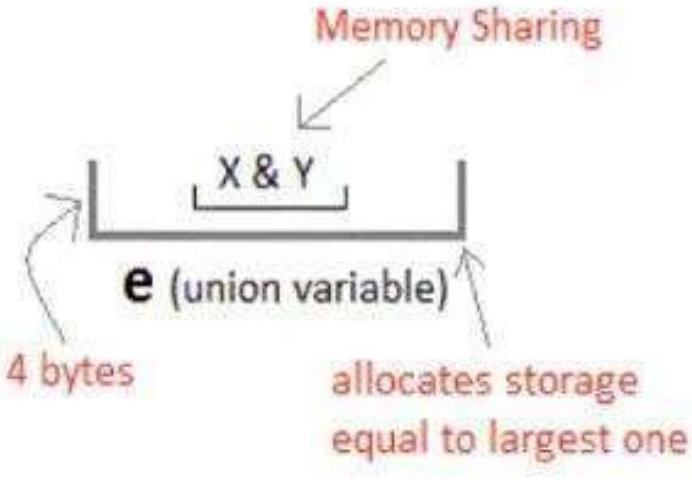
Structure

```
struct Emp
{
 char X ;     // size 1 byte
 float Y ;    // size 4 byte
} e ;
```



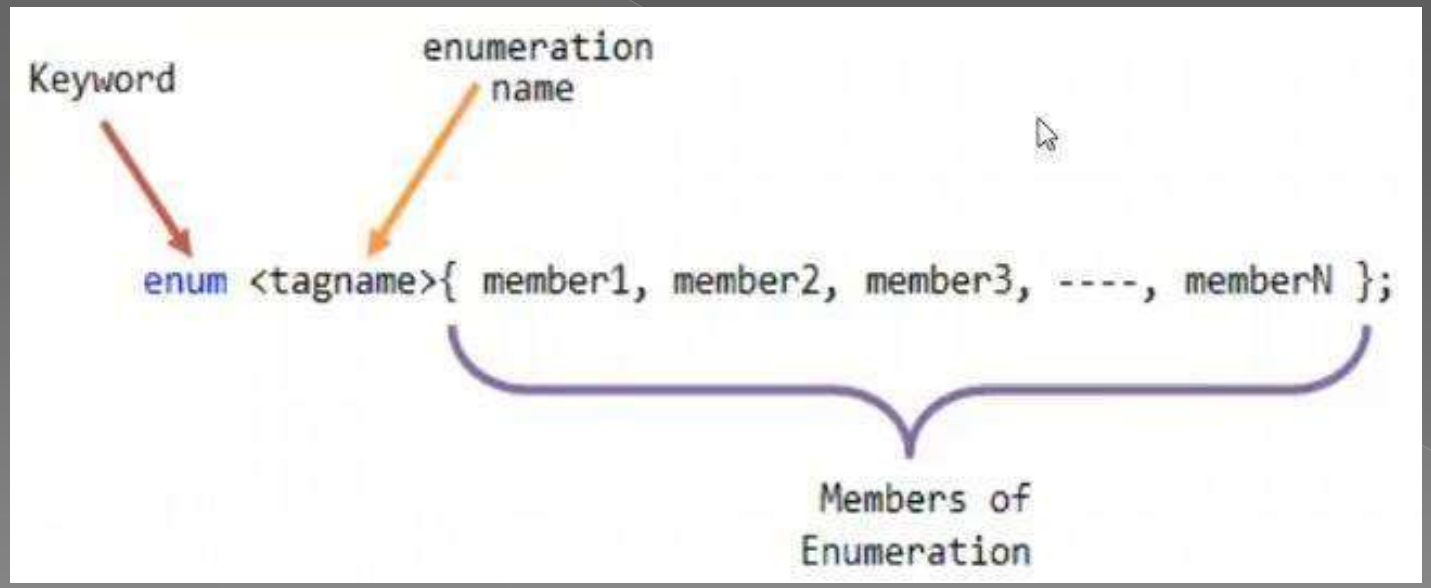
Unions

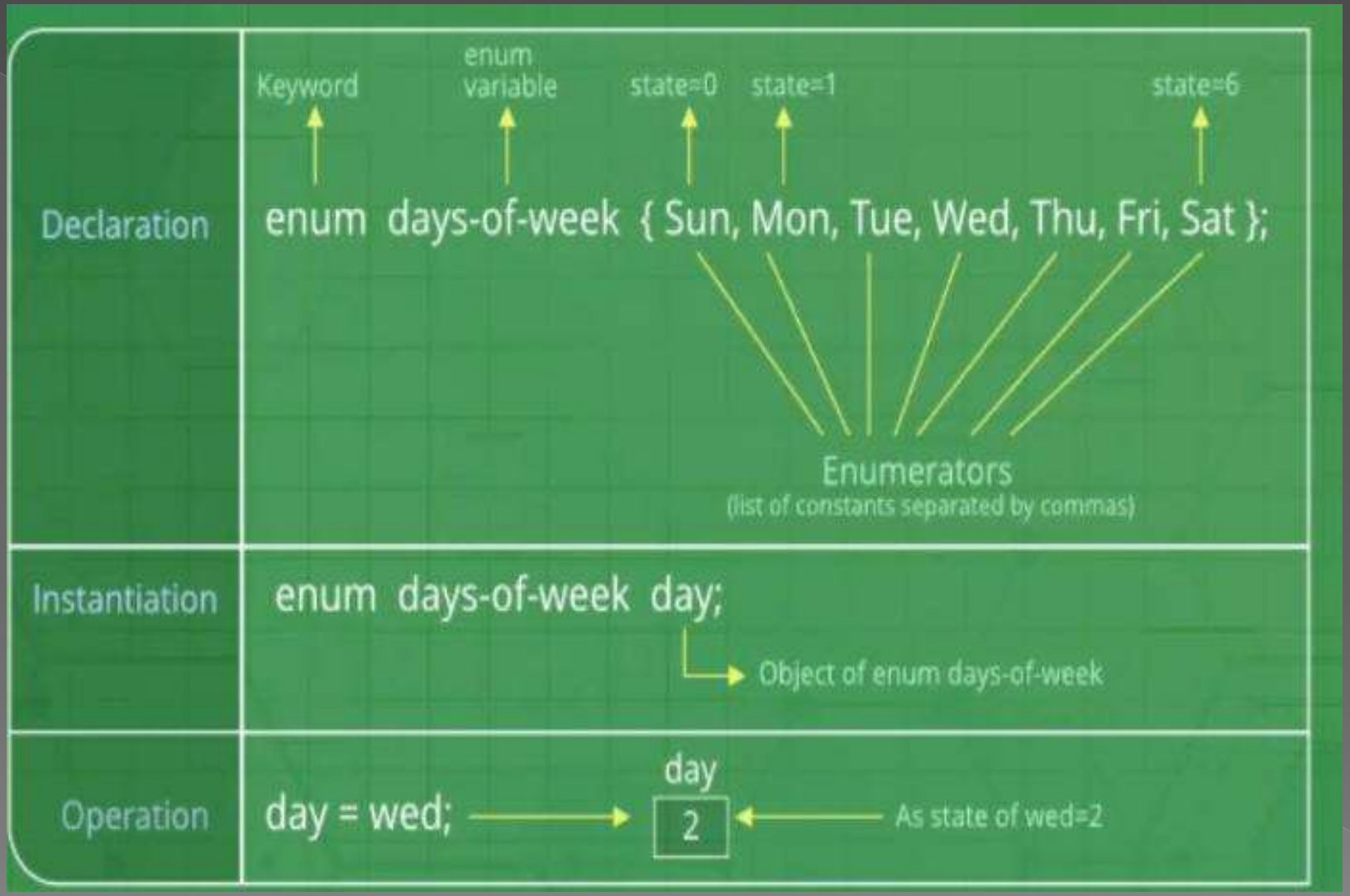
```
union Emp
{
 char X ;
 float Y ;
} e ;
```

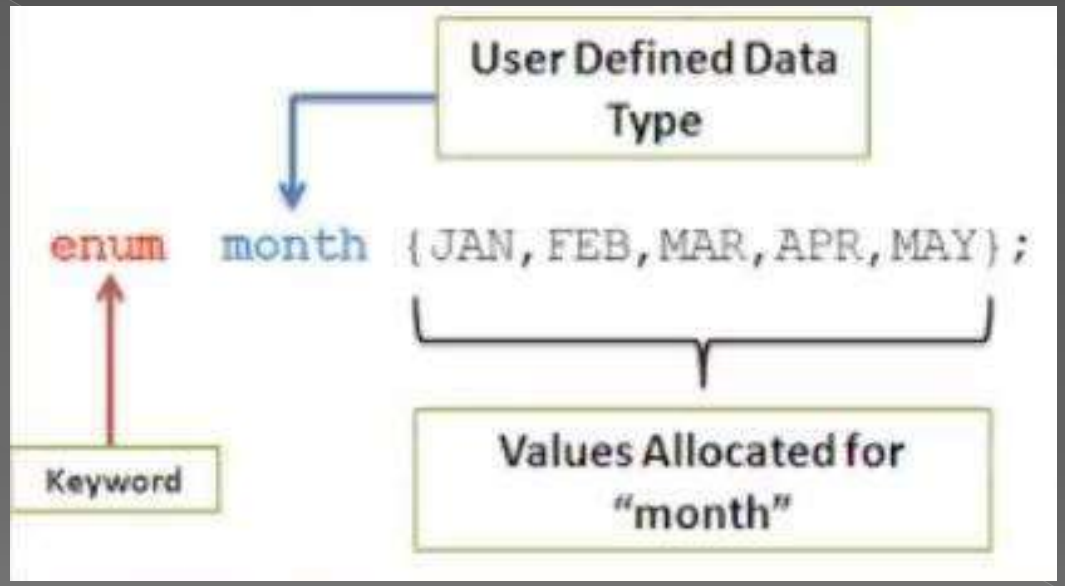


Enum

- A set of named values called elements, members, enumeral or enumerators of the type.
- Enumerator names are identifiers that behaves as constants in the language and enables for a variable to be a set of predefined constants.



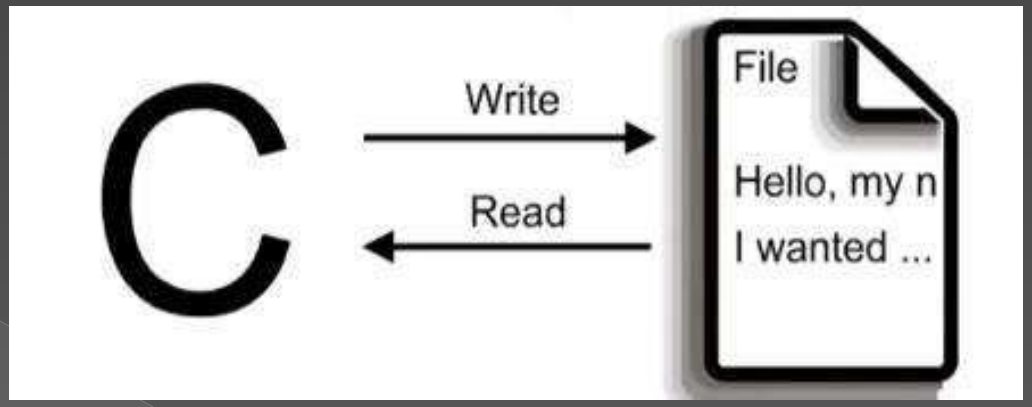

Declaration
Keyword
enum variable
state=0
state=1
state=6
enum days-of-week { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
Enumerators
(list of constants separated by commas)
Instantiation
enum days-of-week day;
Object of enum days-of-week
Operation
day = wed;
day
2
As state of wed=2

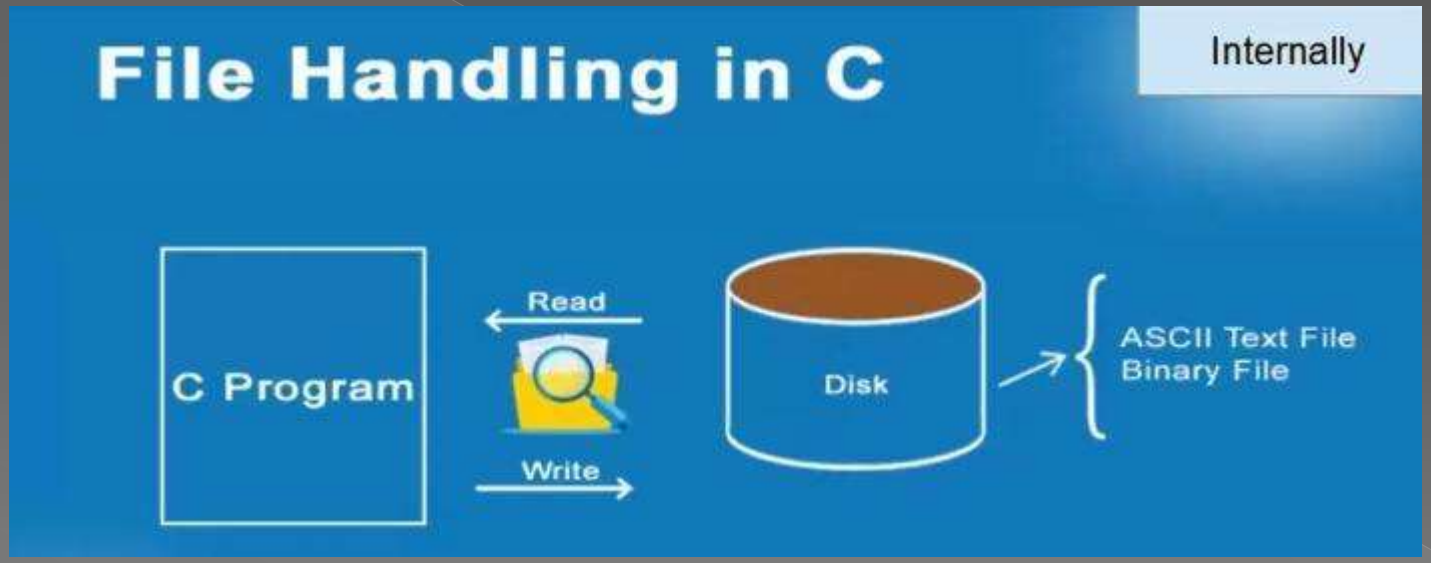

User Defined Data Type
enum month {JAN,FEB,MAR,APR,MAY};
Keyword
Values Allocated for "month"

File Handling

Need

- Used to store data to use or access in future.
- Using C program when we derive output that is stored on temporary memory.
- Soft copies are easy to handle, maintain and share.


C
Write
Read
File
Hello, my n
I wanted ...


File Handling in C
Internally
C Program
Read
Write
Disk
ASCII Text File
Binary File

Types of Files

- Text Files
  - Normal .txt files.
- Binary Files
  - .bin files. Instead of storing data in plain text, they store it in the binary form (0's and 1's).

Operations

- Create and Open File
- Read
- Write
- Append
- Close

Modes

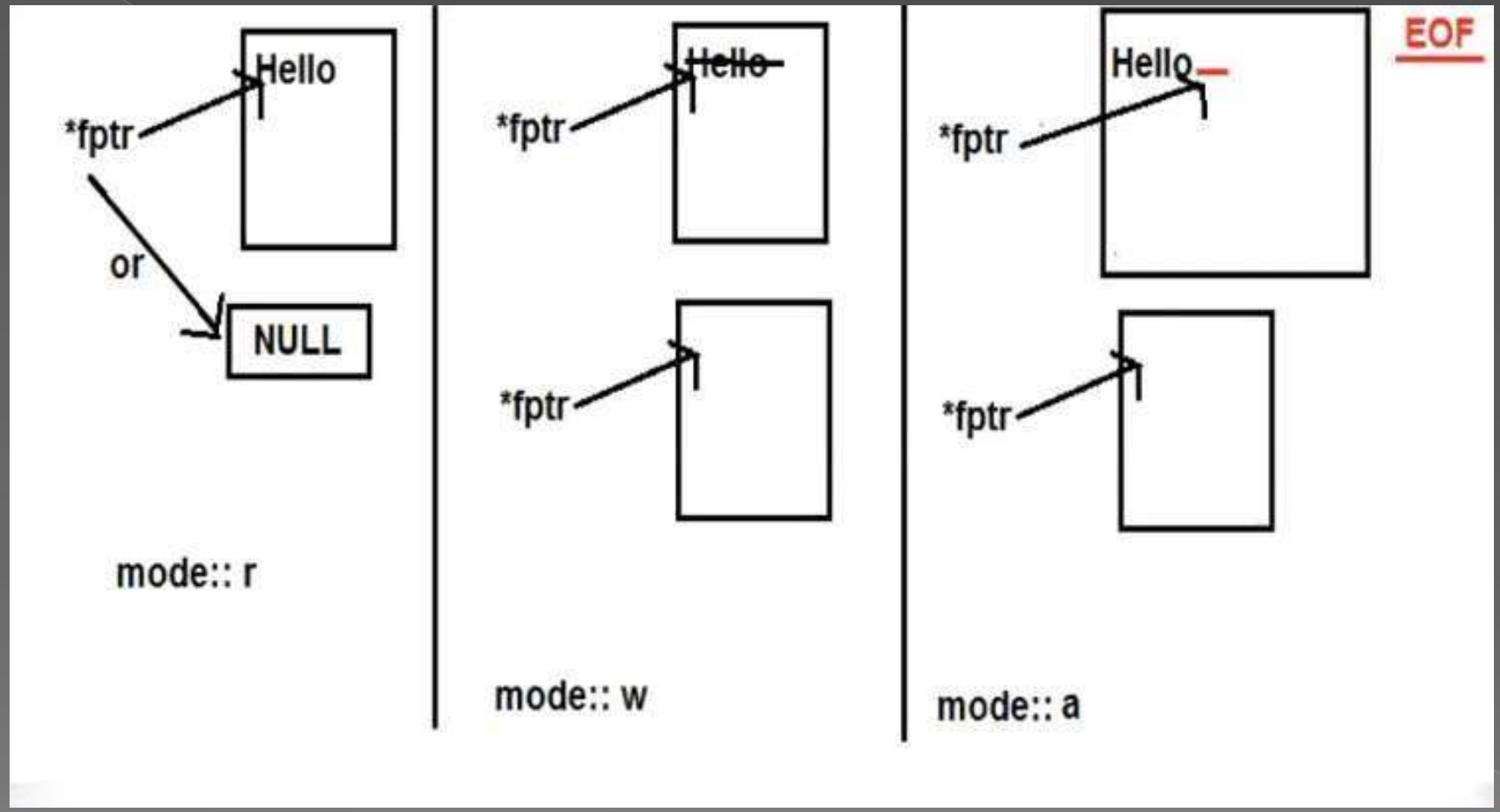
Syntax:- (FILE *) fopen("filepath","mode");

- Modes-

r - Open for reading. If file does not exits. fopen() returns NULL.

w - Open for writing. If file exists, its contents overwritten and if not exists it will be created.

a – Open for append. Data is added to the end of the file. If not exits it will be created.


Hello
*fptr
or
NULL
mode:: r
Hello
*fptr
*fptr
mode:: w
Hello
EOF
*fptr
*fptr
mode:: a

r+ : Opens a text file for both reading and writing.

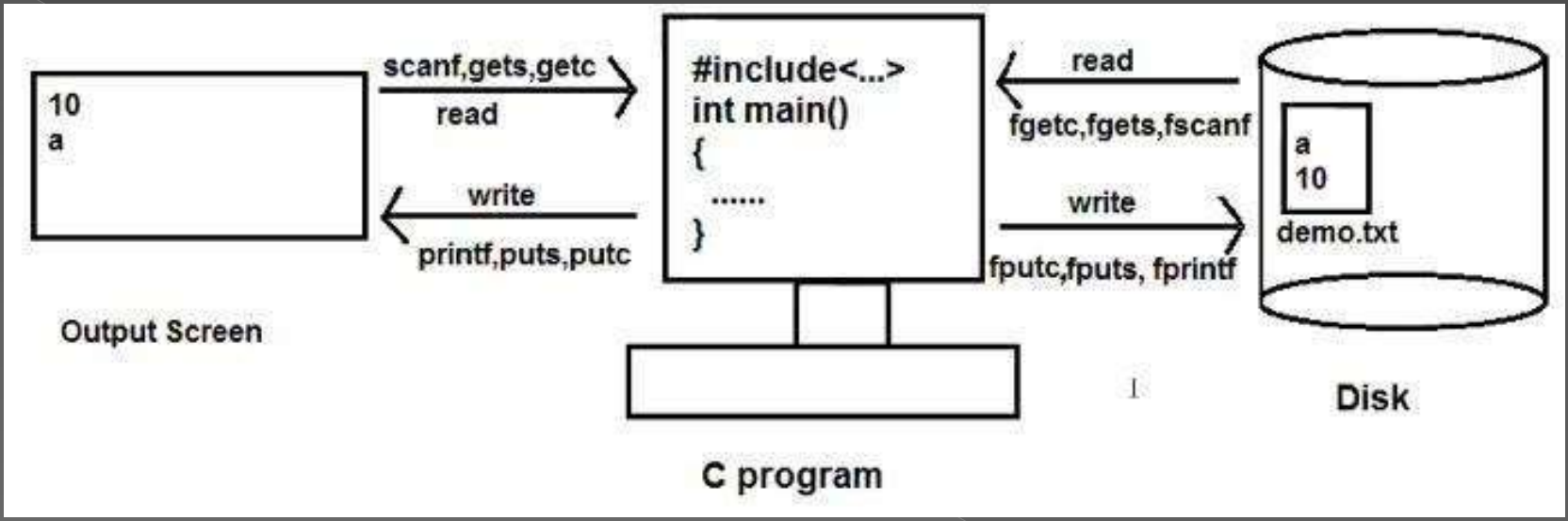
w+ : Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist.

a+ : Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended

Binary mode –

rb	rb+
wb	wb+
ab	ab+

Reading and Writing Functions



- Reading from file – fgetc, fgets or fscanf
- Writing to file – fputc, fputs or fprintf
- Writing and reading block of data respectively – fwrite and fread
- Random access or moving to specific location in file – fseek, rewind
- Getting current position of file pointer - ftell

Command Line Arguments

Command Line Arguments

- The arguments passed from command line are called command line arguments.
- Command line arguments are simply arguments that are specified after the name of the program in the system's command line, and these argument values are passed on to your program during program execution.
- Arguments are handled by main() function.

Components of Command Line Arguments

- There are 2 components of Command Line Argument in C/C++:

1. argc: It refers to “argument count”. It is the first parameter that we use to store the number of command line arguments. It is important to note that the value of argc should be greater than or equal to 0.

2. agrv: It refers to “argument vector”. It is basically an array of character pointer which we use to list all the command line arguments.

Properties of Command Line Arguments

1. They are used to control program from outside instead of hard coding those values inside the code.
2. argv[argc] is a NULL pointer.
3. argv[0] holds the name of the program.
4. argv[1] points to the first command line argument and argv[n] points last argument.

Syntax: int main(int argc, char *argv[])

```c
#include <stdio.h>
int main(int argc, char *argv[] ){
    int i,a,b,sum=0;
    printf("Example of Command Line Arguments in C!\n");
    printf("Program name is: %s\n",argv[0]);
    printf("Arguments passes: %d\n", argc);
    if(argc!=3)
        {
            printf("please use \"prg_name value1 value2 \"\n");
            return -1;
        }
        a = atoi(argv[1]);
        b = atoi(argv[2]);
        sum = a+b;
        printf("Sum of %d, %d is: %d\n",a,b,sum);
    return 0;
}
```

```
E:\PGDAC_MAR22>gcc first.c

E:\PGDAC_MAR22>a.exe 1 2
Example of Command Line Arguments in C!
Program name is: a.exe
Arguments passes: 3
Sum of 1, 2 is: 3

E:\PGDAC_MAR22>a.exe a b
Example of Command Line Arguments in C!
Program name is: a.exe
Arguments passes: 3
Sum of 0, 0 is: 0
```

END

Thank you !!