1 C LANGUAGE

1) C-PROGRAMMING

1.1) Basics of c-programming

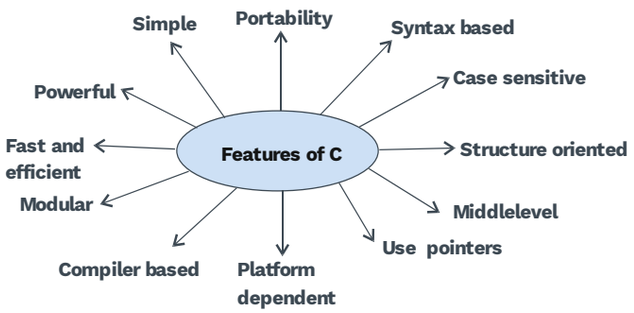


Fig. 1.1 Features of C

Introduction:

- C is a programming language designed and written by Dennis Ritchie.
- It was designed and developed at AT & T Bell Laboratories, USA, in 1972.
- In the late 70's, C began to replace more familiar languages such as ALGOL, PL/I etc. It gradually gained popularity.
- Major portions of operating systems such as Windows, Linux are written in C mainly due to its speed of execution.
- Device drivers and embedded systems are majority written in C.
- Like any other language, C-programming language is composed of alphabets, digits and special symbols which are used to form constants, variables, keywords to write instructions further. These combinations of instructions form a C-program.

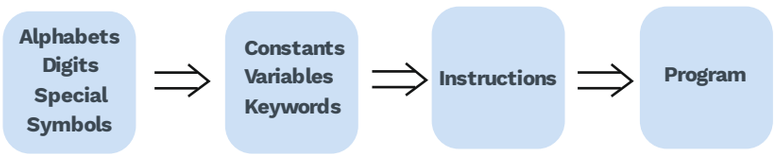


Fig. 1.2

C-character set:

- A character set denotes any alphabets, digits or special symbol used to represent information.
- The following table shows the C-character set:

Alphabets	A – Z, a – z
Digits	0 – 9
Special symbol	~ ' ! @ # % ^ & * () _ - + = | \ { } [ ] : ; " ' < > , . ? /

Token:
Smallest individual unit (Keywords, identifiers, constants, strings, special symbols, operators).

Note

- These are predefined words, each having a specific meaning.
- They are also known as reserved words.
- These words cannot be used as constansts, variables or other identifier names.
- There are total 32 keywords in C.

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

Table 1.1 Keywords in C

Identifiers:

- It is a name used to identify a variable, constant or a function.
- An identifier is a combination of letters, numbers and special symbol (only _ underscore)

Example:

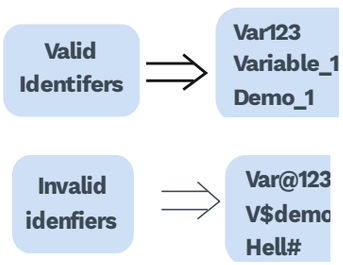


Fig. 1.3

- C does not allow punctuation characters and special symbols such as ?,-, @, #, $, % within identifiers.

Note

C is a case-sensitive language i.e. the identifiers 'Demo' and 'demo' both are two different identifiers in C language.

Whitespace:

- Whitespace in C programming language are blank lines, tabs or new lines, and blank spaces.
- These whitespace helps the compiler in understanding, and identifying when one element in the statement ends, and next element starts.

Example: int age;
A single whitespace or blankspace between int, and age enables the compiler to identify int datatypes, and age as the variable of int datatype.
whereas: int age = age1 + age2;
the space between age, = age1, + and age2 is not necessary, but for readability purposes, it is always preferred to add them.

Comments:

- They are ignored by the compiler.
- Comments are like helping texts in a C-program.
- Mostly added to increase the readability of a program.

Single line comments: //
Multi-line comments: /* _____
_____
_____ */

Instruction:

- It is a combination of keyword, variable, and constants.

Example: int a = 10;

Program

- It is a collection of instructions.

Example:

```
#include <stdio.h>          → header file
void main()                 → function
{  int a = 1; int b = 2;    → variable
   int c = a + b;           → operation
   printf("sum%d", c);
}                           → print statement
```

C-program structure:

- Preprocessor commands
- Functions
- Variables
- Statements and expressions
- Comments

Semicolon:

- A semicolon ";" is a statement terminator in C.
- Every statement must be ended with a semicolon.
- It indicates the end of a logical entry.

Constants and variables:

- Alphabets, numbers and special symbols, when properly combined form constants & variables.
- **Constant** is an entity that does not change during the execution of the program.
- **Variable** is an entity that may change during the execution of the program.

Rules to name variables and constants:

1) The variables or constant name is a combination of 1 to 31 characters, i.e. length can be maximum 30.
2) These characters can be alphabets, digits, underscores.
3) The first character in the variable or constant name must be an alphabet or underscore.
4) No comma, whitespaces are allowed within the variable or constant name.
5) No special symbol other than underscore and no keyword can be used as constant or variable name.

Example:

Valid	Invalid
ab4cd	4abcd
Var_Name	Var@Name
principal_Value1	principal#Val1

Table 1.2

Note

- Variables are the name given to a storage location which can be manipulated by the program written.
- **ivalue**
  An expression which is an Ivalue/ivalue may appear on L.H.S. or R.H.S. of the assignment.
- **rvalue**
  An expression which is a Rvalue/rvalue may appear only on the R.H.S. of the assignment.

Constants:

- Value that cannot be changed during the execution of the program

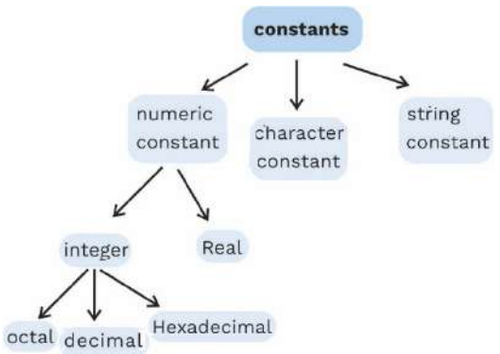


Fig. 1.4 Types of Constants

1) Numeric constant:

- numeric digits (may or may not have decimal point).
- should have at least one digit, no commas or space, either positive or negative sign.
- by default sign is positive.

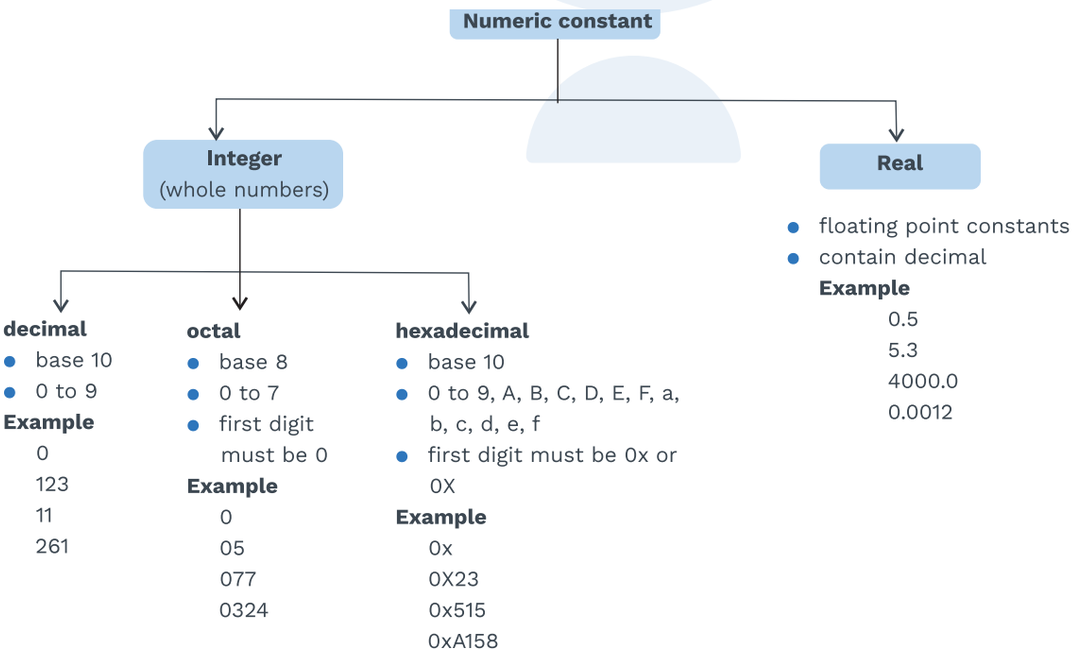


Fig. 1.5 Types of Numeric Constants

2) Character constants:

- Single character enclosed within single quotes.
- A character constant is a single alphabet, a single digit or a single special symbol enclosed within single quotes.

Example: 'a', 'B', '8', '='

Example:

Valid	Invalid
'9'	'four'
'D'	"d"
's'	y
' '	''
'#'	

Table 1.3

Note:

Character	ASCII value
A-Z	65-90
a-z	97-122
0-9	48-57
;	59

Table 1.4

3) String constants:

- It has zero, one or more than one character.
- enclosed within double quotes ""
- at the end of string \0 is automatically placed by compiler.
- \0 refers to as NULL string.

Example:

"Hello"
"8"
"593"
" "
"A"

Note

"A" represents two character: A, \0
'A' represents one character constant with ASCII value 65.

4) Symbolic constants:

- using the keyword: define
- when one constant is to be used several times.
- A symbolic constant is a tag used to replace a number, e.g. pi can be used to replace 3.14.

- These sequences of characters may be numeric constant, character constant or string constant.
- Generally defined at the beginning of the program with header files.

Syntax: #define name value

Example: Defining a constant pi whose value is 3.14159625
#define pi 3.14159625

Some more examples:
#define max 100
#define CH 'a'
#define Name "Somesh"

Note

The names are replaced by their respective values at the time of compilation in the program.

1.2 TYPES OF VARIABLES

Datatypes in C:

- An attribute given to every data used in the program is known as data type for the data. A variable is declared with a data type so that it holds a value of that type.
- The type of variable specifies how much space it requires in storage and how it is stored in memory.

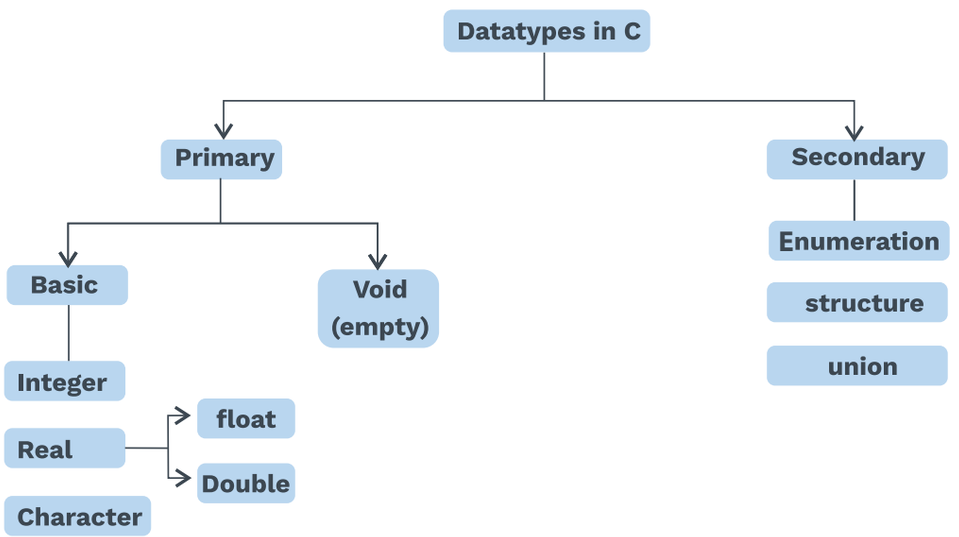


Fig. 1.6 Data-Types in C

1) Primary datatypes:

Integer	Real	Character
Signed • short int • int • long int	float double long double	signed char unsigned char
Unsigned • unsigned short int • unsigned int • unsigned long int		

Table 1.5

Note

- By default, datatypes are signed, we need to specify unsigned to use unsigned datatypes.
- The size of datatypes depends on the machine (compiler and architecture) whether they are 16 bit, 32 bit etc.
- printf("%u", sizeof(int)); //This statement prints the size of datatypes used by the system.

S.No.	Datatype	Size (n)	Range
1)	char	8 bit	– 128 to + 127
2)	unsigned char	8 bit	0 to 255
3)	short int	16 bit	– 32768 to +32767
4)	int	16 bit	–32768 to +32767
5)	unsigned int	16 bit	0 to 65535
6)	long int	32 bit	– (2^{31}) to + $(2^{31} – 1)$
7)	unsigned long int	32 bit	0 to +$(2^{32} – 1)$
8)	float	32 bit	– 3.4e38 to +3.4e38
9)	double	64 bit	– 1.7e308 to +1.7e308
10)	long double	80 bit	– 1.7e4932 to +1.7e4932
11)	unsigned short int	16 bit	0 to 65535

Table 1.6

Note

The sizes and ranges of int, short and long are compiler dependent. Sizes in this figure are for 16-bit compiler.

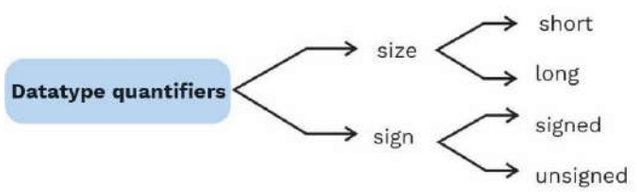


Fig. 1.7

The variable of integer data type in C Programming can be signed or unsigned number. Three different declarations of type integer are short, int, and long. All of them in size, e.g. "short" has a lower range than "int" and "long" has a larger range than "int". Though the actual size in bytes depends on the architecture on which program is being written.

Note

Range for n-bit data:
signed : -2^{n-1} to $2^{n-1} - 1$
unsigned : 0 to $2^{n} - 1$

2) Secondary datatype:

- These data types are divided or defined using primitive datatypes

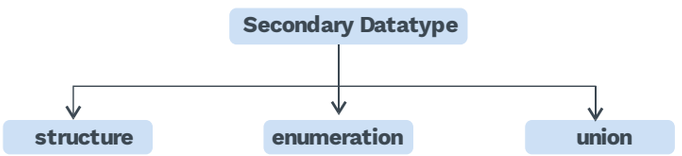


Fig. 1.8 Types of Secondary Datatypes

1.2.2 Scope and lifetime variables:

Storage classes:

- In addition to datatypes, each variable has one or more attribute known as the storage class.

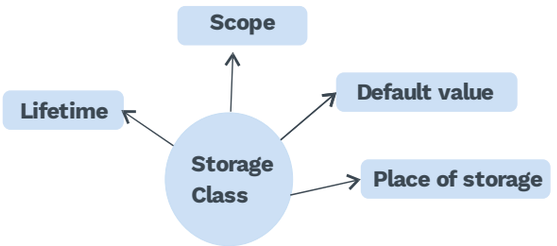


Fig. 1.9 Features of Storage Classes

- The use of storage class makes the programs more efficient and swift.
- The syntax of declaring the storage class of a variable is:

Syntax: Storage_class_name datatype variable_name;

Example: Storage_class_name int a;

- There are four types of storage classes, namely automatic, external, static and register.

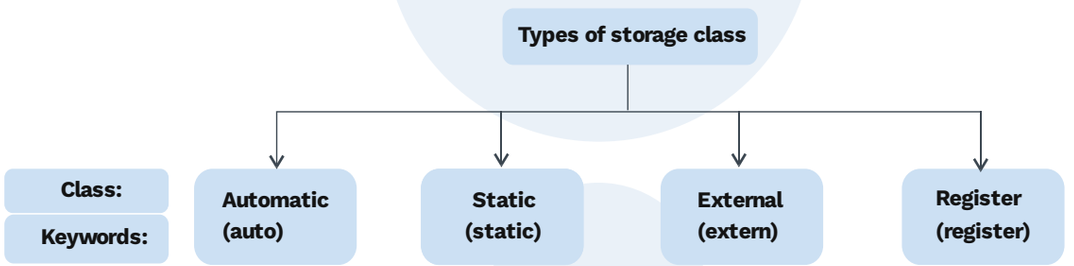


Fig. 1.10

A storage class decides about the following aspects of a variable:

1) **Lifetime:** It is the time between creation & destruction of a variable.
2) **Scope:** The progress range/location where the variable is available for use.
3) **Default Value:** The default value which is taken by uninitialized variable.
4) **Place of storage:** The place in memory which is allocated to the variable.

Note

- When a storage class is not specified, then the compiler assumes a default storage class based on the place of declaration.
- Static scoping is sometimes referred to as lexical scoping.

Types of storage class:

Storage Class Name ⟶	Automatic	External	Static (Global/Local)	Register
Keyword	auto	extern	static	register
Lifetime	function block	whole program	whole program	function block
Scope	Local	Global	Global/local	Local
Initial value	Garbage	Zero	Zero	Garbage
Storage	Stack	Data segment	Data segment	CPU register
Declaration	Inside function block	Outside/Inside function block	Outside/Inside function block	Inside function block

Table 1.7 All about Storage Classes

- To make global static variables, declare the static variable outside of all functions.
- Static variables can only be initialized by constants or constant expressions.
- An static variable is initialized once, and it retains its value during the recursive function calls.

Example 1:

Consider the given C-program statements:

```
int x = 12
static int y = 15;
func()
{    static int x;
     x = x + 2;
     printf("inside func(): x = %d, y = %d\n", x, y);
}
main()
{    int x = 13;
     func();
     func();
     printf("inside main ():x = %d, y = %d\n", x, y);
}
```

Output: Inside func():x = 2, y = 15
Inside func():x = 4, y = 15
Inside main():x = 13, y = 15

In func(), x is initialized to 0 and x = 0 is used, since it is a static variable, hence its value is retained between function calls. Since y has been initialized outside main and func(), it is accessible to both the functions.

Example 2:
Consider the given C-program statements:

```
func1()
{    extern int x;
     x++;
     printf("func1:%d\n", x);
}
     int x = 189;
     func2()
{    x++;
     printf("func2:%d\n", x);
}
main()
{    func1();
     func2();
}
```

Output: func1:190
func2:191

Previous Years' Questions

The value of j at the end of the execution of the following C-program is _____

```
int incr(int i) {
static int count = 0;
count = count + i;
return count;
}
main(){
int i, j;
for (i = 0;i = 4; i++)
j = incr(i);
}
```

a) 10 **b)** 4 **c)** 6 **d)** 7

Sol: Option a) **(GATE-2000)**

Previous Years' Question

Consider the following C-function:

```
int f(int n)
    {  static int i = 1;
       if(n >= 5) return n;
       n = n + i;
       i++;
       return f(n);
     }
```

The value returned by f(1) is:

a) 5 **b)** 6 **c)** 7 **d)** 8

Sol: Option c) **(GATE-2004)**

Previous Years' Question

Early binding refers to a binding performed at compile time, and late binding refers to a binding performed at execution time. Consider the following statements:

i) Static scope facilitates **W1** bindings.
ii) Dynamic scope requires **W2** bindings.
iii) Early bindings **W3** execution efficiency.
iv) Late bindings **W4** execution efficiency.

The right choices of W1, W2, W3 and W4 (in that order) are:

a) Early, late, decrease, increase **b)** Late, early, increase, decrease
c) Late, early, decrease, increase **d)** Early, late, increase, decrease

Sol: Option d) **(GATE-2007)**

Previous Years' Question

Consider the program given below, in a block-structured pseudo-language with lexical scoping, and nesting of procedures permitted.

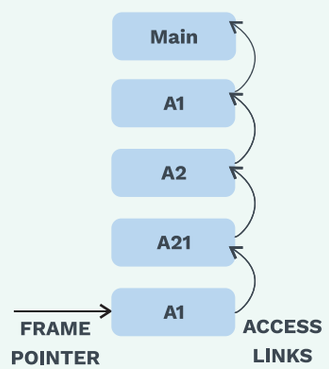
```
Program main;
    Var ...
        Procedure A1;
            Var ……..
            Call A2;
            End A1
        Procedure A2;
            Var ...
            Procedure A21;
                Var ...
                Call A1;
                End A21
        Call A21;
    End A21
        Call A1 ;
    End main.
```

Consider the calling chain:

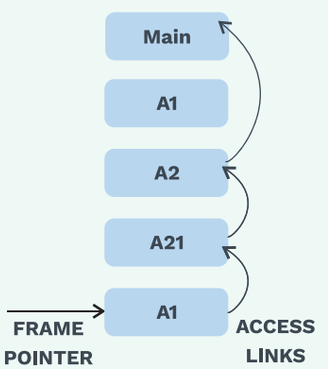
Main → A1 → A2 → A21 → A1

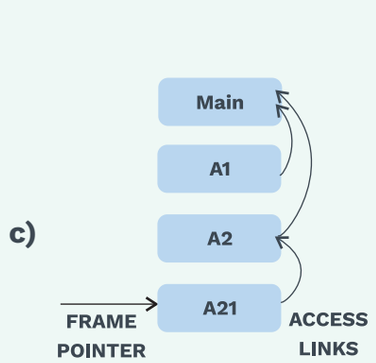
The correct set of activation records along with their access links is given by:

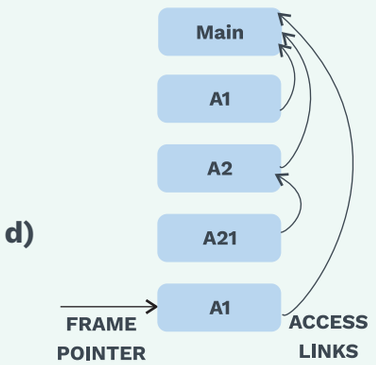
a)



b)







Sol: Option d) **(GATE-2012)**

1.3 OPERATORS IN C

- An operator specifies an operation to be performed that yields a value.
- An operand is a data item on which an operator acts upon.

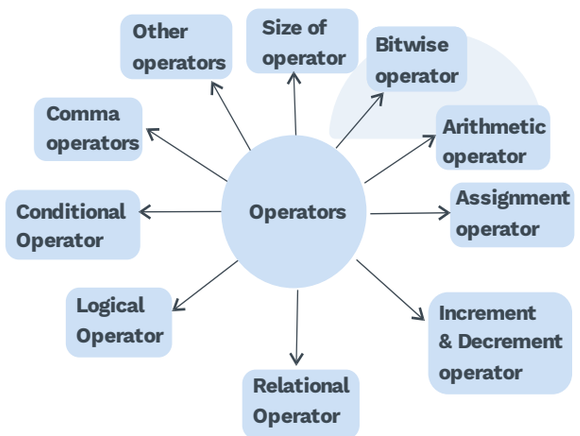


Fig. 1.11 Types of Operators in C

- Operators can be of two types primarily:
  1) unary operator **2)** binary operators

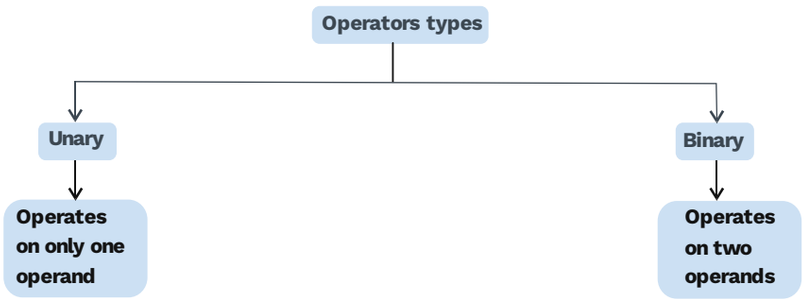


Fig. 1.12

Bitwise operator:

- Manipulation at bit level can be performed using bitwise operators.
- These operators perform operation on individual bits.

Operator	Meaning
&	bitwise AND
|	bitwise inclusive OR
^	bitwise XOR
~	One's compliment (unary)
<<	Bitwise left shift
>>	Bitwise right shift

Table 1.8 Bitwise Operators

Note

- Bitwise operators operates on integral operands only.
- All bitwise operators are binary except compliment operator, which is unary.
- All bitwise operators except compliment can be obtained with the assignment operators.

&=,|=, <<=, >>=, ^ =
Compound Assignment Operators

Bitwise AND (&):

- Binary operator represented as &

Bit of operand 1	Bit of operand 2	Resulting bit
0	0	0
0	1	0
1	0	0
1	1	1

Table 1.9

Syntax: operand 1 & operand 2

Bitwise inclusive OR (|):

- Binary operator represented by |.

Bit of operand 1	Bit of operand 2	Resulting bit
0	0	0
0	1	1
1	0	1
1	1	1

Table 1.10

Syntax: operand 1 | operand 2

Bitwise XOR (^):

- produces output 1 for only those input combinations that have odd number of 1's.
- Binary operator represented by ^.

Bit of operand 1	Bit of operand 2	Resulting bit
0	0	0
0	1	1
1	0	1
1	1	0

Table 1.11

Syntax: operand 1 ^ operand 2

Compliment (~):

- One's compliment operator represented by ~.
- unary operator

Bit of operand	Resulting bit
0	1
1	0

Table 1.12

Syntax: ~ operand.

Bitwise left shift (<<):

- Binary operator represented by <<.

Syntax:

Operand 1 ≪ Operand 2

↓ ↓

Operand whose bits are to be shifted left — No. of bits to be shifted

- Shifting bits results in equal no. of bits being vacated on the other side, these vacated spaces are filled with 0 bits.

Example:

Let x = 0001 0011 0000 0100, then x<<4 means left shift x by 4 bits, then:

0001 0011 0000 0100 0001

Lost bits **Filled bits**

∴ **After left shifting:** 0011 0000 0100 0000

Bitwise right shift (>>):

- Binary operator represented by >>.
- Similar to left shift except it shifts bit in the opposite direction as of left shift i.e. shift bit to the right side.
- Similar to left shift, here the bits are shifted to right & bits are vacated in the left and right shifting on unsigned quantity always fill with zero in the vacated positions. In right shifting on signed quantity, then will fill with sign bits ("arithmetic shift") on the vacated positions.

Example:

Let unsigned int x= 0001 0011 0000 0100, then x>>4 means right shift x by 4 bits, then:

0000 0001 0011 0000 0100

Filled bits **Lost bits**

After right shifting: 0000 0001 0011 0000

Note

In right shift if the first operand is signed integer, then the result is compiler dependent.

Arithmetic operator:

- On the basis of no. of operands:

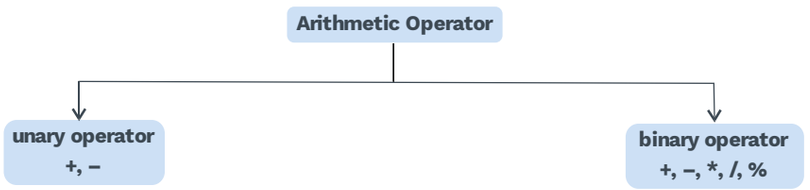


Fig. 1.13

- On the basis of value of operands:

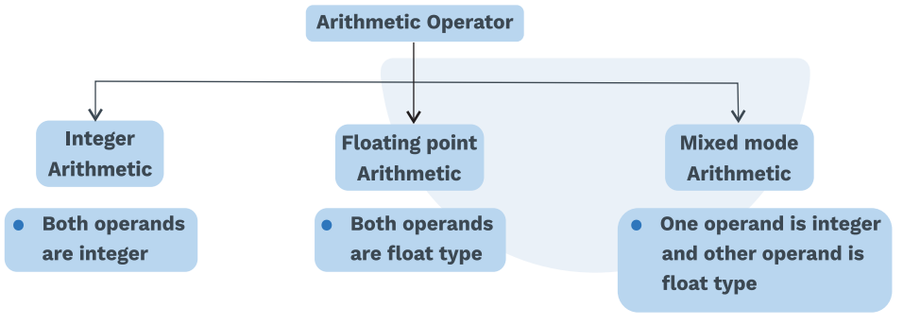


Fig. 1.14

Binary arithmetic operator:

Operator	Meaning
+	Addition
–	Subtraction
*	Multiplication
/	Division
%	Modulus (gives remainder in integer division)

Table 1.13

Note

% (Modulus operator cannot be applied to floating point operands). There are no exponent operator in C, whereas the library function pow () is used as exponentiation operation.

Relational operator:

- These operators are used to compare values of two expressions depending on their relations.
- Relational expression is an expression with relational operators.

Operator	Meaning
<	less than
<=	less than or equal to
==	equal to
!=	not equal to
>	greater than
>=	greater than or equal to

Table 1.14

Note

Value of relational expression = 1, if relation is True.
Value of relational expression = 0; if relation is False.

'=' and "= =" have entirely different meaning.

↓ → checks

assignment equality

operator

Logical operators:

- They are also known as Boolean operators.
- Used for combining expressions.
- They return 0 for false and 1 for true.

Operator	Meaning
&&	AND
||	OR
!	NOT

Table 1.15

Note

Logical NOT is a unary operator whereas && and || are binary operators.
In C, any non-zero value is regarded as true and 0 is regarded as false.

Non – zero value → True
Zero value → False

AND operator (&&):

- Binary operator represented as &&.

Condition 1	Condition 2	Result
False	False	False
False	True	False
True	False	False
True	True	True

Table 1.16 Truth table for AND

Syntax: (condition 1) && (condition 2)

OR operator (||):

- Binary operator represented as ||.

Condition 1	Condition 2	Result
False	False	False
False	True	True
True	False	True
True	True	True

Table 1.17 Truth table for OR

Syntax: (condition 1) || (condition 2)

NOT operator (!):

- unary operator represented by !.
- negates the value of the condition.

Condition	Result
False	True
True	False

Table 1.18 Truth table for NOT

Syntax: ! (Condition)

Assignment operator:

- used to assign values to variables, denoted by symbol '='.

Syntax:

Operand 1 = **Operand 2**

↓ Should be a variable

↓ Can be a variable, expression or a constant

Compound assignment operator:

- Combination of arithmetic operator with assignment operator.

Compound assignment operator	Alternate
x + = 5	x = x + 5
x – = 5	x = x – 5
x/=5	x = x/5
x% = 5	x = x%5
x*=5	x = x*5

Table 1.19

Increment and decrement operator:

- Unary operator
- Increment operator represented by ++.
- Decrement operator represented by ––.

Operator	Meaning	Equivalent
++x	increments the value of variable by 1	x = x + 1
––x	decrement the value by variable by 1	x = x – 1

Table 1.20

- ++ increment by 1
- -- decrement by 1.
- The increment and decrement operator can be either: (1) prefix; (2) postfix;

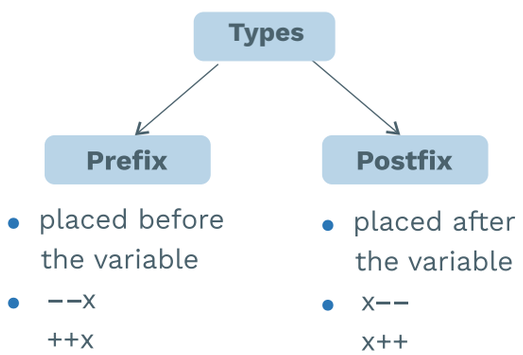


Fig. 1.15

Prefix increment/decrement:

- The value of the variable is incremented or decremented, then the value is used in the operation.

Syntax: ++x or Operator variable;
––x

Example:
Let x =5, then y=++x means increment the value of x by 1 and then use the value to be assigned to y. ∴ y = 6

Similarly, y=––x means decrement the value of y by 1 and then use the value to be assigned to y. ∴ y = 1

Postfix increment/decrement:

- First the value of variable is used, then it is incremented or decremented.

Syntax: x++ or Variable operator;
x––

Example:
Let x =5, then y = x++ means assign y = 5, then increment the value of x by 1 i.e. after execution y = 5 & x = 6.
Similarly, y = x–– means assign y = 5, then decrement the value of x by 1 i.e. after execution y = 5 & x = 4.

Conditional operator:

- It is a ternary operator which requires three expressions as operands.
- Represented by ? and : space
- **Syntax:** Text expression ? expression 1: expression 2

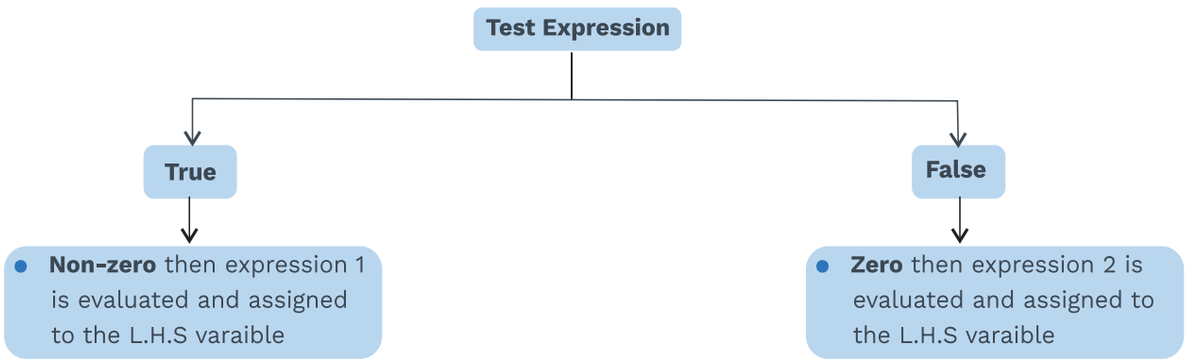


Fig. 1.16

Example:
max = a > b ? a: b

This translates to that if (a > b) ⟶ True, then max = a
⟶ False, then max = b

- printf statements can also be used.

Example: a > b ? printf (“a is larger”): printf (“b is larger”);

Rack Your Brain

Which of the following is not a unary operator?
1) ++ **2)** – **3)** sizeof **4)** ?:

Comma operator:
- Represented by ‘,’
- It is used to allow different expressions to appear in places where exactly one expression would be used.
- Also commonly used as a separator.

Example:

Without comma operator	With comma operator
a = 8	a = 8, b = 7, c = a + b;
b = 7;	
c = a + b;	

Table 1.21

Example:
Sum = (a = 10, b = 7, c = 3, a + b + c);
Here sum would be a + b + c i.e. sum = 20.

Sizeof operator:
- unary operator
- returns the size of operand in bytes.
- The parameter passed can be a variable, constant or any datatype (int, float, character).

Syntax: sizeof(parameter);
↓
Variable\ constant\ Datatype

Example:
sizeof(int); would return the bytes occupied by integer datatype.

Note

Another operator called the typecast operator is used for carrying out type conversion.

Rack Your Brain

Consider the given C-program:

```
#include <stdio.h>
int main()
{printf("%u%u%u%u",   sizeof(schar),   sizeof(int),   sizeof(float)
sizeof(double));
return 0;
}
```

1) 1 4 4 8 **2)** 1 2 4 8 **3)** 1 4 8 16 **4)** Machine dependent

Previous Years' Question

Consider the following C program:

```
#include<stdio.h>
int main()
{
        int m = 10;
        int n,n1;
        n =++m;
        n1 = m++;
        n--;
        --n1 ;
        n-=n1;
        printf ("%d", n);
        return 0;
}
```

The output of the program is _____.

Sol: 0) **(GATE-2017 (Set-2))**

Previous Years' Question

Consider the following C-program:

```
int a, b, c = 0;
Void prtFun(void);
main()
{    static int a = 1; /*Line 1*/
     prtFun();
     a+=1;
     prtFun();
     printf ("\n%d %d", a, b);}
void prtFun(void)
{    static int a = 2; /*Line 2*/
     int b = 1;
     a+ = ++b;
     printf ("\n%d%d",a,b);
}
```

What output will be generated by the given code segment?

a)
3 1
4 1
4 2

b)
4 2
6 1
6 1

c)
4 2
6 2
2 0

d)
3 1
5 2
5 2

Sol: c)

What output will be generated by the given code segment if:
Line 1 is replaced by auto int a=1;
Line 2 is replaced by register int a=2;

a)
3 1
4 1
4 2

b)
4 2
6 1
6 1

c)
4 2
6 2
2 0

d)
4 2
4 2
2 0

Sol: d)

(GATE-2012 (Common Data Question))

1.4 TYPE CONVERSION

- It includes converting data type of one operand into data type of another operand to perform operations.

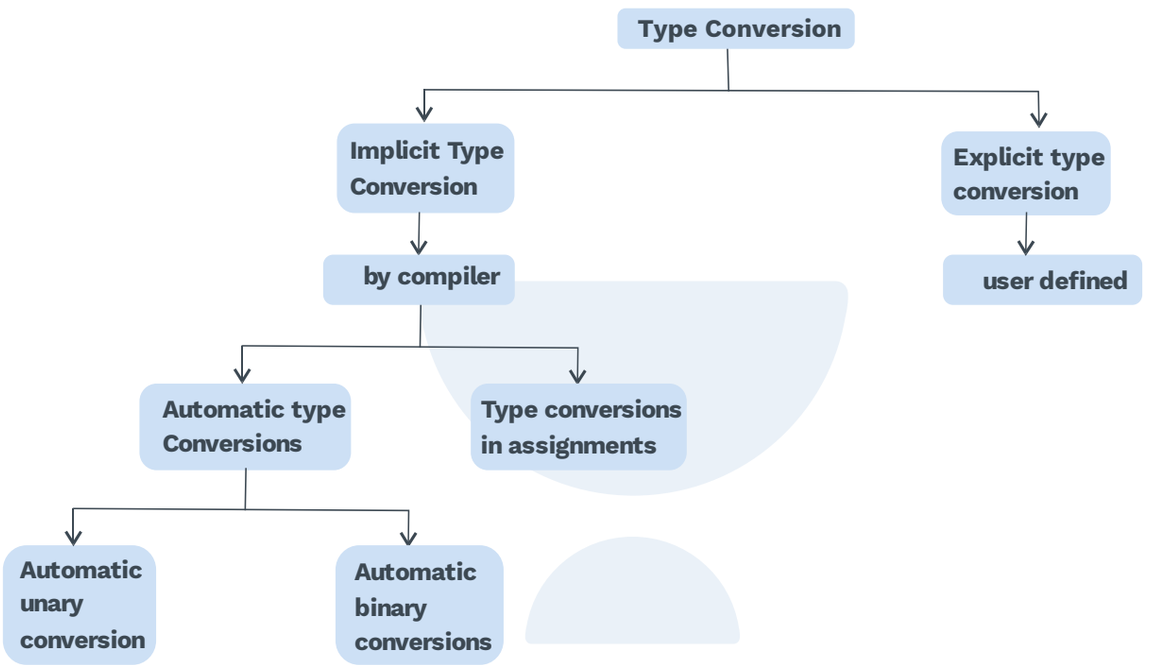


Fig. 1.17

Implicit type conversion:

- Done by the C compiler based on some predefined C-language rules.

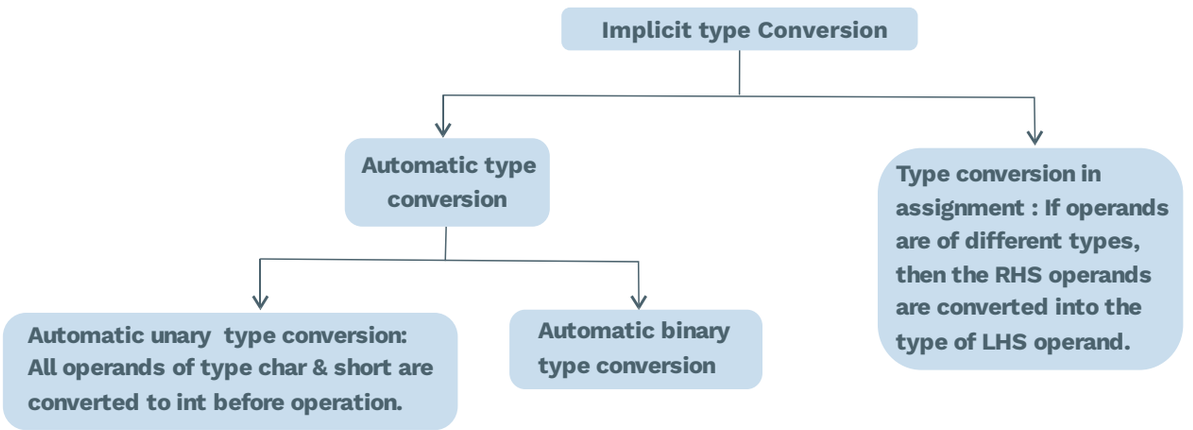


Fig. 1.18

Note

- For automatic unary type conversion, some compilers convert all float operands to double before operation.
- There are some rules and rank hierarchy followed for automatic binary type conversion.
- In type conversion during assignment, the RHS operand are demoted or promoted based on the rank of LHS operand.

Rules for automatic binary type conversion:

1) With a binary operator, if both the operands have dissimilar data types then the operand with lower rank gets converted into the data type of higher rank operand. It is known as data type promotion.

2) **In case of unsigned datatypes:**

i) One operand is unsigned long int, others are converted to unsigned long int as well as the result is stored as unsigned long int.

ii) One operand is unsigned int and other is long int, then:

Case I If long int can represent all values of unsigned int, then unsigned is converted to long int and the result is also stored in long int etc.

Case II Both are converted to unsigned long int along with the result.

iii) One operand is unsigned int, then others along with the result are converted and stored as unsigned int.

Consequences of these promotions and demotions

1) Few higher order bits get dropped when higher order rank data type is converted to a lower order rank data type. E.g. when int is converted to short.

2) During the conversion of the float data type into int, the fractional part gets removed.

3) Digits are rounded off while conversion of double type to float type.

4) The sign may be dropped in case of conversion of signed type to unsigned types.

5) There is no increased accuracy or precision when an int is converted to float or float to double.

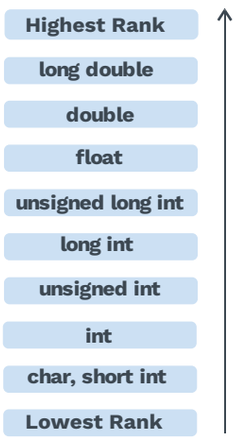


Fig. 1.19 Rank of Datatypes in C

Explicit type conversion (Type casting):

Some cases such as:

```
float z;
int x = 20, y = 3;        ...(i)
z = x / y;
```

The value of z would be 6.0 and not 6.666666

- These cases require the implementation of explicit type conversion.
- This allows user to specify our own/user defined conversion.
- Also known as Type casting or Coercions.
- Type casting is implemented using the cast operator.
- The cast operator is a unary operator which converts an expression to a particular datatype temporarily.

Syntax : (Datatype) expression;
Cast Operator

- Using the cast operator (float) in (i).

z = (float) x/y;

then the value of z would be 6.666667

Fig. 1.20

Precedence and associativity of operators:

- For an expression having more than one operator, there exists certain precedence and associativity rules for its evaluation.

Example:

Given an expression: 2 + 3 * 5

It contain 2 operations: + and *, if + performed before *, then result would be 25 and if * performed before +, then result would be 17.

∴ To remove this ambiguity of which operation to perform first the C-languages specifies the order of precedence & associativity of operators.

- **Operator precedence:** Determines which operator is performed first in an expression.
- **Operator associativity:** It is used when two operators of same precedence appear in an expression.

Associativity can be: Left to right

or

Right to left

Operator	Description	Precedence	Associativity
( )	Parentheses or function call	1	Left to right
[ ]	Brackets or array subscript		
●	Dot or member selection operator		
→	Arrow operator		
+	Unary plus	2	Right to left
–	Unary minus		
++	Increment		
––	Decrement		
!	Logical NOT		
~	One's complement		
*	Indirection or dereference operator		
&	Address		
(Datatype)	Type cast		
sizeof	Size in bytes		
*	Multiplication	3	Left to right
/	Division		
%	Modulus		

+	Addition	4	Left to right
–	Subtraction		
<<	Left shift	5	Left to right
>>	Right Shift		
<	Less than	6	Left to right
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	equal to	7	Left to right
!=	not equal to		
&	Bitwise AND	8	Left to right
^	Bitwise XOR	9	Left to right
!	Bitwise OR	10	Left to right
&&	Logical AND	11	Left to right
‖	Logical OR	12	Left to right
?:	Conditional operator	13	Right to left
=	Assignment Operator	14	Right to left
*=,/=, %=			
+=, –=			
&=,^=, /=			
<<=, >>=			
,	Comma Operator	15	Left to right

Table 1.22 Operator Precedence and Associativity Table

Example: 5 + 16 / 2 * 4

- Since / and * have higher precedence than +

∴ they are evaluated first.

- / and * have same precedence, so which would be evaluated first amongst the two is determined by the associativity rules. Since, /and * are left to right associative.

∴ / is performed before *.

The given expression can be considered as:

5 + (16/2) * 4

Solving: 5 + (8 * 4)

⇒ 5 + 32

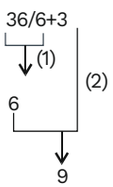
⇒ 37

Role of parentheses:

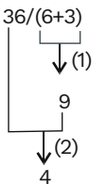
- Parenthesis are used to change the order of precedence of any operation.
- All the operations enclosed in parenthesis are performed first.

Example:

I. Without parenthesis



II. With parenthesis



- Sometimes to increase the readability of the code and expression parenthesis are used.

Example: x = a!=b & c * d >= m%n
Better: x = (a!= b) & ((c*d)>=(m%n))

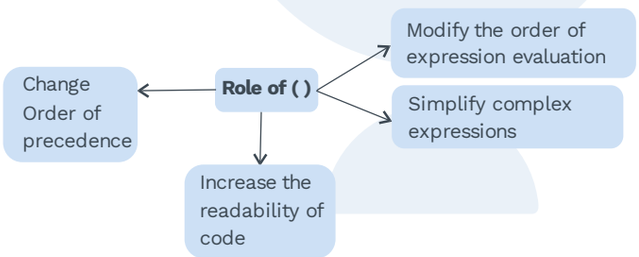


Fig. 1.21

1.5 FLOW CONTROL IN C

- Control statements enable us to specify the order in which the various instructions in the program are to be executed.
- It determines the flow of control in C.

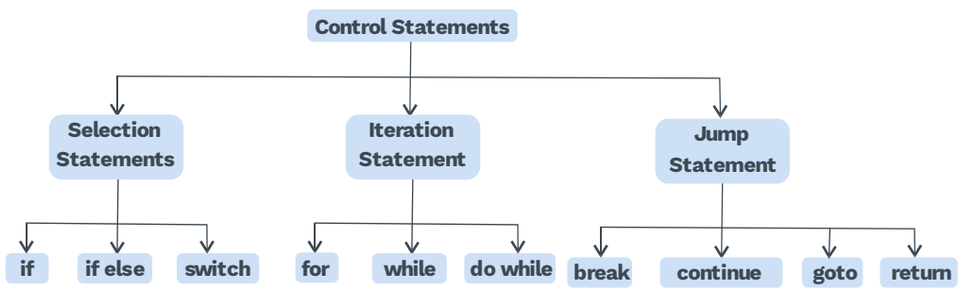


Fig. 1.22 Types of Control Statements

- Compound statements or a block are a group of statements which are enclosed within a pair of curly braces { }.

A compound statement is syntactically equivalent to a single statement.

If else statements:

- Bi-directional conditional control statement.
- The statement tests one or more conditions and executes the block based on the outcome of the test.
- Any non-zero value is regarded as true, whereas 0 is regarded as false.

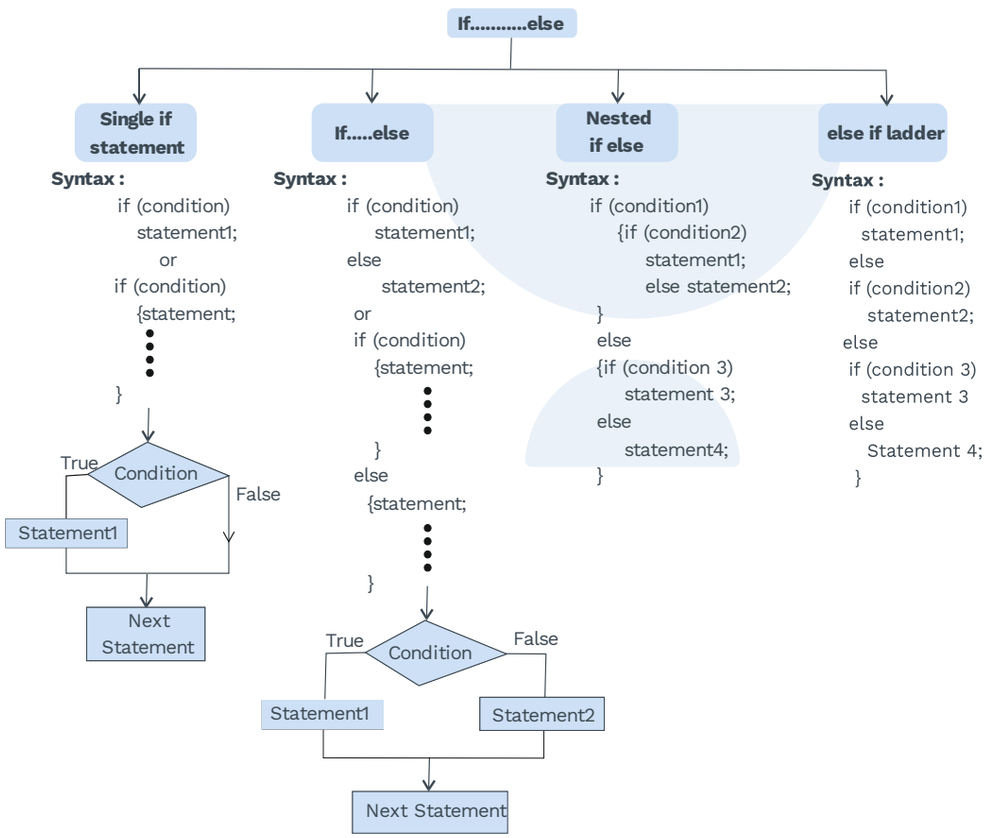


Fig. 1.23 Types of If-else Statements

Loops:

- Used when one wants to execute a part of a program or a block of statements several times.
- With the help of loops, one can execute a part of program repeatedly till some condition is true.

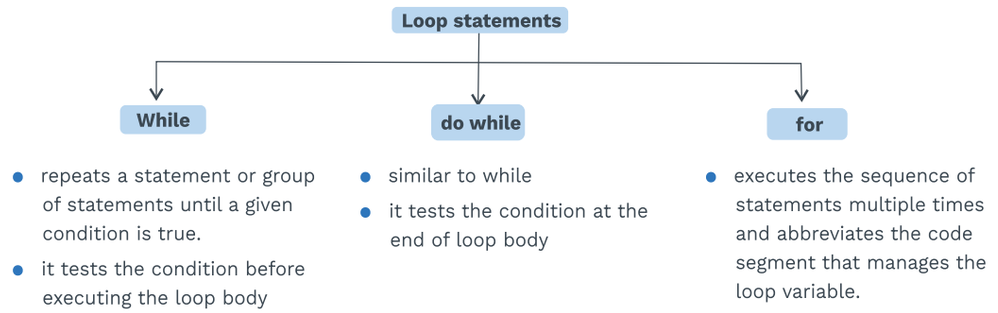


Fig. 1.24 Types of Loop Statements

While loop:

- First the condition is evaluated, if it is true, then the body of loop is executed.

Syntax: While (condition)
Statement;

OR

While (condition)
{ Statement;
Statement;
⋮
⋮
}

Note: While loops are used when the number of iterations are not known in advance.

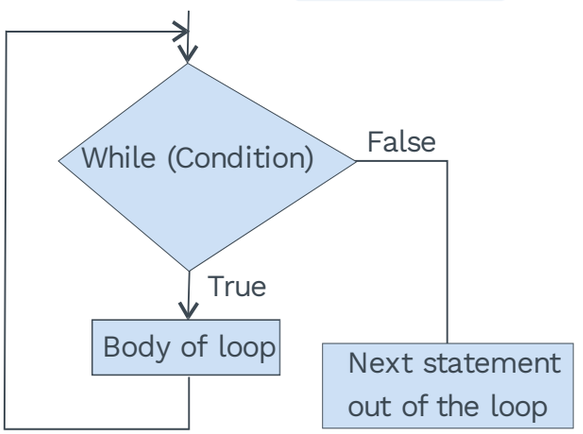


Fig. 1.25 Flow Chart for While Loop

- Each execution of loop body is called an iteration.

Example: #include <stdio.h>
main()

```
{ int i = 1;
        while(i <=5)
        { printf("%d\t", i);
        i++;
        }
}
```

Output: 1 2 3 4 5

This C-program prints the numbers from 1 to 5. The variable i is initialized to 1 and then the condition is checked whether (i < = 5) if true, then the printf statement is executed. The value of i is incremented by 1 each time the loop is executed.

Previous Years' Question

Consider the following C-program:

```
#include <stdio.h>
int main()
{float sum=0.0, j=1.0, i=2.0;
While(i/j>0.0625)
{
j=j+j;
sum=sum+i/j;
printf("%f\n", sum);
}
 return 0;
}
```

The number of times the variable sum will be printed, when the above program is executed is _____.

Sol: 5 to 5 **(GATE-2019)**

Do-while loop:

- Similar to while loop just that in do while loop first the statements are executed, then the condition is checked.
- This results in the execution of the statements at least once even if the condition is false for the first iteration.

Syntax:

```
do
   Statement;
while (condition);
```

or

```
do
{ Statement;
      Statement;
      ⋮
} while (condition);
```

- In do while loop, a semicolon is placed after the while (condition).

Example:

```
#include <stdio.h>
main()
{ int i = 1;
     do
     { printf ("% d\t", i);
       i++ ;
     } while(i <=5);
}
```

output: 1 2 3 4 5

For a program to count the no. of digits, it is better to use do while loop over while loop. This program if written using while loop returns the number of digits as 0 for n = 0. Therefore, here using do while loop is preferred.

```
#include <stdio.h>
main()
{    int n, count = 0;
     printf("enter a no.");
     scanf("%d",&n);
     do {    n/=10;
             count ++;
         } while(n > 0);
     printf("no. of digits = %d", count);
}
```

For loop

- Combined of three expression separated by semicolons.

Syntax:

```
for(expression 1; expression 2; expression 3)
      Statement;
            Or
for(expression 1; expression 2; expression 3)
      {  Statement;
         Statement;
            ⋮
      }
```

- The body of for loop can have single statement or a block of statements.

- In for loop:

Expression 1: Initialization expression, executed only once, and is generally an assignment expression.

Expression 2: Test expression or condition, tested before each iteration, generally uses relational or logical operators.

Expression 3: Update expression or updation, executed each time for body of loop is executed.

- Firstly, the initialization expression is executed, and the loop variables are initialized, then the condition is checked.
- If the condition expression is true, then the body of the loop is executed, and the updation expression is executed.
- This continues till the condition expression is true, and once the condition expression becomes false, the loop terminates and control is transferred to the statement following the for loop.

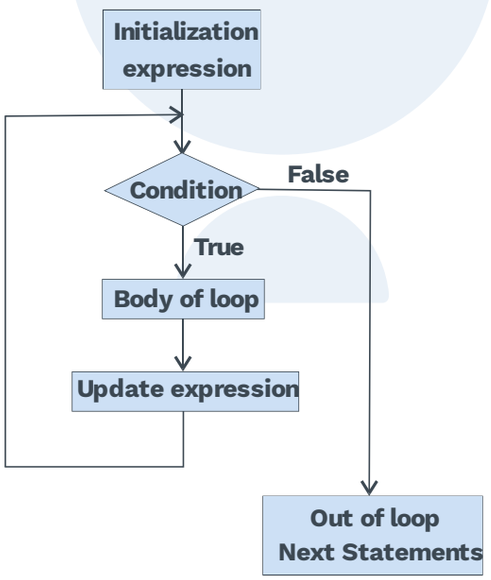


Fig. 1.26 Flow Chart of "for" Loop

- For loops are used when the number of iterations are known in advance.

Example:

```
# include <stdio.h>
main()
{ int i;
    for(int i = 1; i < = 5; i++)
    printf("%d\t",i);
}
```

Output: 1 2 3 4 5

Note

- The variable/loop variable i can be declared before or at the time of initialization in expression 1.
- All three expressions of for loop are optional. One can avail any one or all three expressions.
- The two separating semi-colons are mandatory.
- Expression 1 can be omitted if initialization is done before the loop.
- Expression 2 which is the condition if omitted is always considered true. Hence, loop will never stop executing resulting in an infinite loop.
- Expression 3 which is the updation can be omitted if it is present inside the loop body.

Some loops that are valid:

- for(; n > 0; n/=10)//initialization of n should be mentioned before loop.
- for(i = 0, j = 10; i < =j; i++, j--)//multiple expression separated by comma & semi-colon.
- for(; ;)//infinite loop.

Example:

```
#include<stdio.h>

main()
{   int n, sum = 0;
    printf("Enter the no.:");
    scanf("%d", & n);
    for(; n > 0; n/=10)
    {   int t = n% 10;
        sum + = t;
    }
printf(" sum = %d", t);

}
Example:
        #include<stdio.h>
        main()
        {   int i, j;
            for (i = 0, j = 10; i<=j; i++, j–=2)
                {printf ("i =%d\t    j = %d\n", i, j);
                }
        }

Output:                    i = 0                    j = 10
              i = 1              j = 8
              i = 2              j = 6
              i = 3              j = 4
```

Previous Years' Question

Consider the following C code. Assume that unsigned long int type length is 64 bits.

```
unsigned long int fun(unsigned long int n)
    {unsigned long int i, j=0, sum=0;
    for(i=n;i>1;i=i/2) j++;
    for(;j>1;j=j/2) sum++;
    return(sum);
    }
```

The value returned when we call fun with the input 240 is:

a) 4 **b)** 5 **c)** 6 **d)** 40

Sol: b)

Hint : when n=240, j=log(240) **(GATE-2018)**

Nesting of loops:

- Loop within a loop
- Any type of loop can be nested inside any other type of loop.

Infinite loops:

- Loops that go on executing infinitely.
- The loops that do not terminate are called infinite loops.

Example:

while(1)
{.....
.....
}
 	
Infinite while loop
	for (; ;)
	{.....
	.....
	}
	
	Infinite for loop
		do
		{.....
		.....
		} while (1);
		
		Infinite do while loop

- To come out of such loops break or goto statements are used.

1.6 BREAK, CONTINUE AND GOTO STATEMENTS

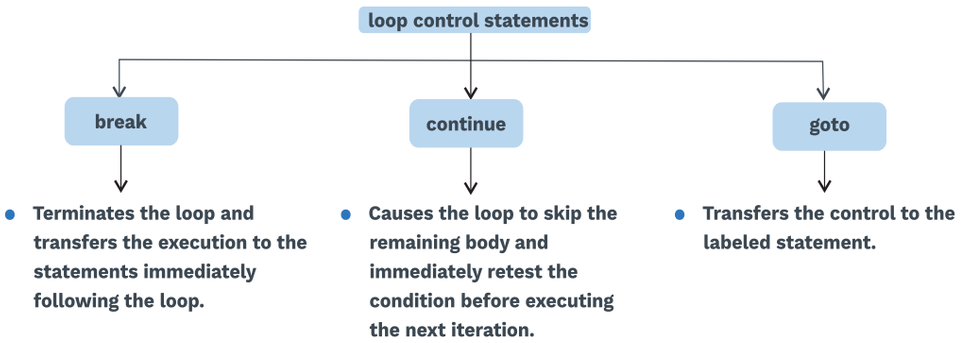


Fig. 1.27 Types of Control Statements

- These loop control statements enable the control of programs/loop to be transferred.

Break statement:

- Used when it becomes necessary to come out of the loop even before the loop condition becomes false.
- Break statement causes the immediate exit from the loop.

Syntax: break;

- Popularly used in switch statements.

While loop:

```
while (condition)
    {  Statement;
       if (condition to break){
          break;
       }
       Statement;
    }
Control
```

Note

As soon as the break statement is encountered, the control is transferred to the next statements after the loop.

Do-While Loop

```
do{
        Statement;
        if (condition to break){
        break;
        }
        Statement;
    } while (condition);
Control
```

For Loop

```
      Initialization   Condition      Updation
           ↑              ↑             ↑
for (expression 1; expression 2; expression 3)
   {  Statement;
      if (condition to break) {
      break ; }
      Statement;
   }
Control
```

Continue statement:

- Used when one wants to skip some statements and execute the next iteration.

Syntax: continue;

- Generally used with condition statements, when a condition statement is encountered and is true, then the statements after the keyword continue are skipped & the loops condition is checked for next iteration.

Note

In break, the loop terminates and control is transferred to outside. In continue the current iteration terminates and the control is transferred to the beginning of the loop.

Example:

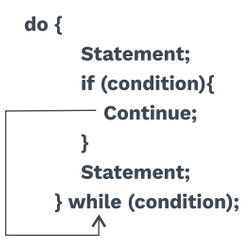
While Loop

```
while (condition) {
        Statement;
        if (condition)
        }
        Statement;
```

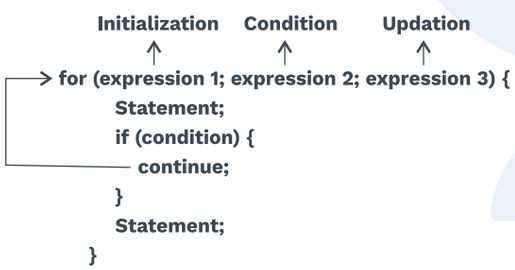
Note

As soon as the condition of if is true, the continue statement is executed. This results in control being given to the while condition and next statements are skipped.

Do-while Loop:



For Loop



Goto statement:

- Unconditional control statement
- Transfers the flow of control to another part of the program.

Syntax: goto **label;**

Statement;
Statement;
.....
.....
.....

label:
Statement;
Statement;
.....
.....

Note

The control is transferred immediately after the label.

- The label can be placed anywhere:

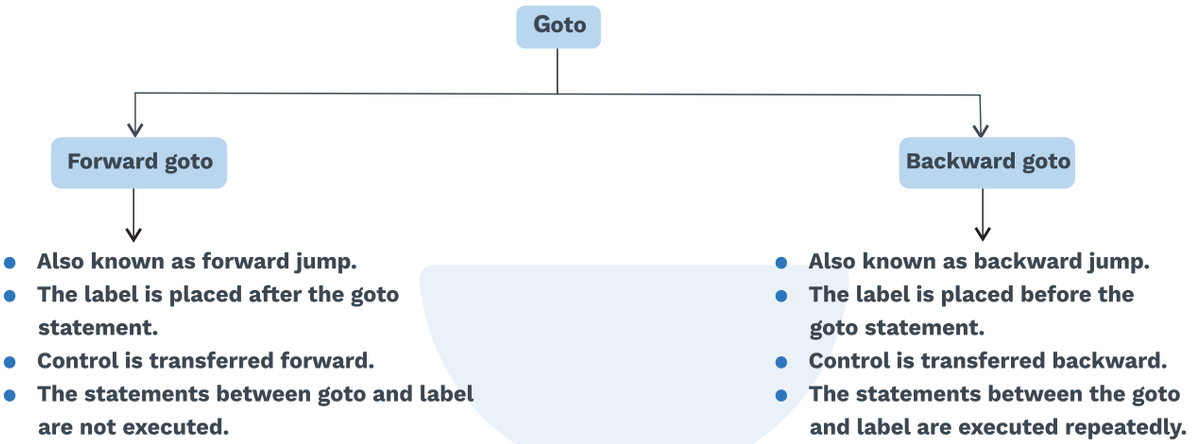


Fig. 1.28 Types of Goto Statements

Note

The control in goto can be transferred within a function.

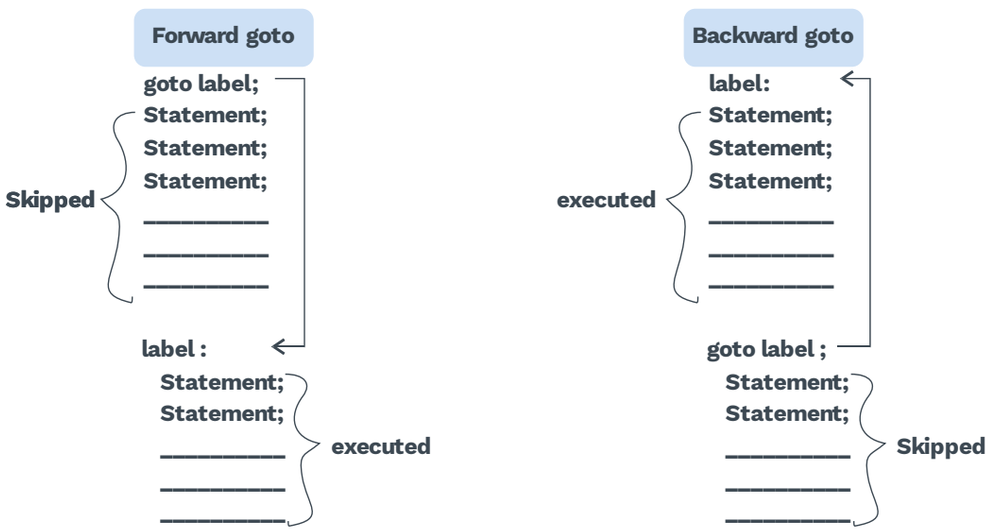


Fig. 1.29

Note

Use of goto statements is not preferred as it makes it difficult to understand where the control is being transferred.

Often leads to spaghetti code (Ambiguous & not understandable)

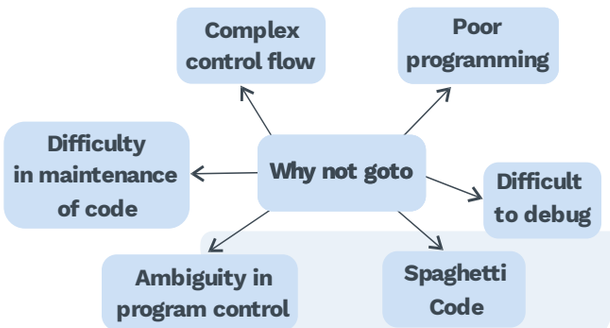


Fig. 1.30

Although there might be situation such as complex nested loops or deeply nested loops where goto or jump statements can improve the readability of code.

Example:

```
for (initialization; Condition; updation)
  {
     while (condition)
     {   for (initialization; condition; updation)
           { do
              { _____
                _____
                _____
              }while(condition);
           while (condition)
           { if  (condition)
              {goto stop;
              }
           }
        }
     }
  }
  Stop :
  _____                  * exit from the deeply
  _____                    nested loops.
  _____
```

Return statement:

- ends the execution of function and returns the control of execution to the calling function.
- does not mandatorily need any conditional statements.
- Depending on the type of function it may or may not return values.

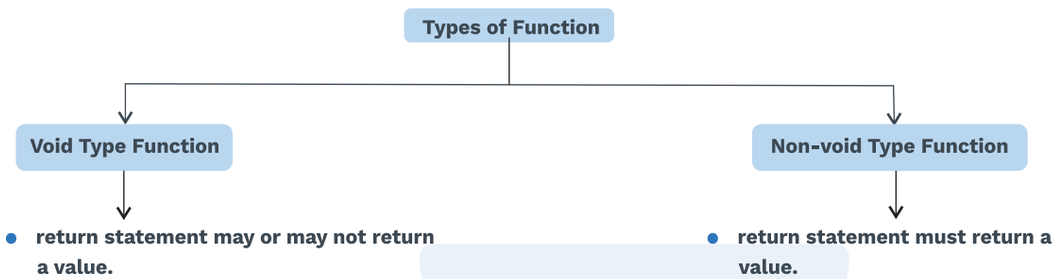


Fig. 1.31 Types of Functions

Syntax: return expression;

Example:

Void Function

```
void func()
   {
   }
```

Non-void Function

```
return_datatype func()
        { return value;
        }
```

1.7 SWITCH CASE

- Multi-directional conditional control statement.
- Popularly used for menu driven programs.
- **Uses three keywords:** switch, case, break, default.

Syntax:

```
switch(expression)
{ case constant 1: Statement;
                   break;
case constant 2: statement;
                   break;
_____
                _____
                _____
default: Statements;
}
```

- **Expression:** can be anything:
  1) An expression yielding an integer value.
  2) Value of any integer
  3) Character variable
  4) Function call returning a value.

- **Constants:**
  1) Should be integer or character type.
  2) can be constants or constant expressions.

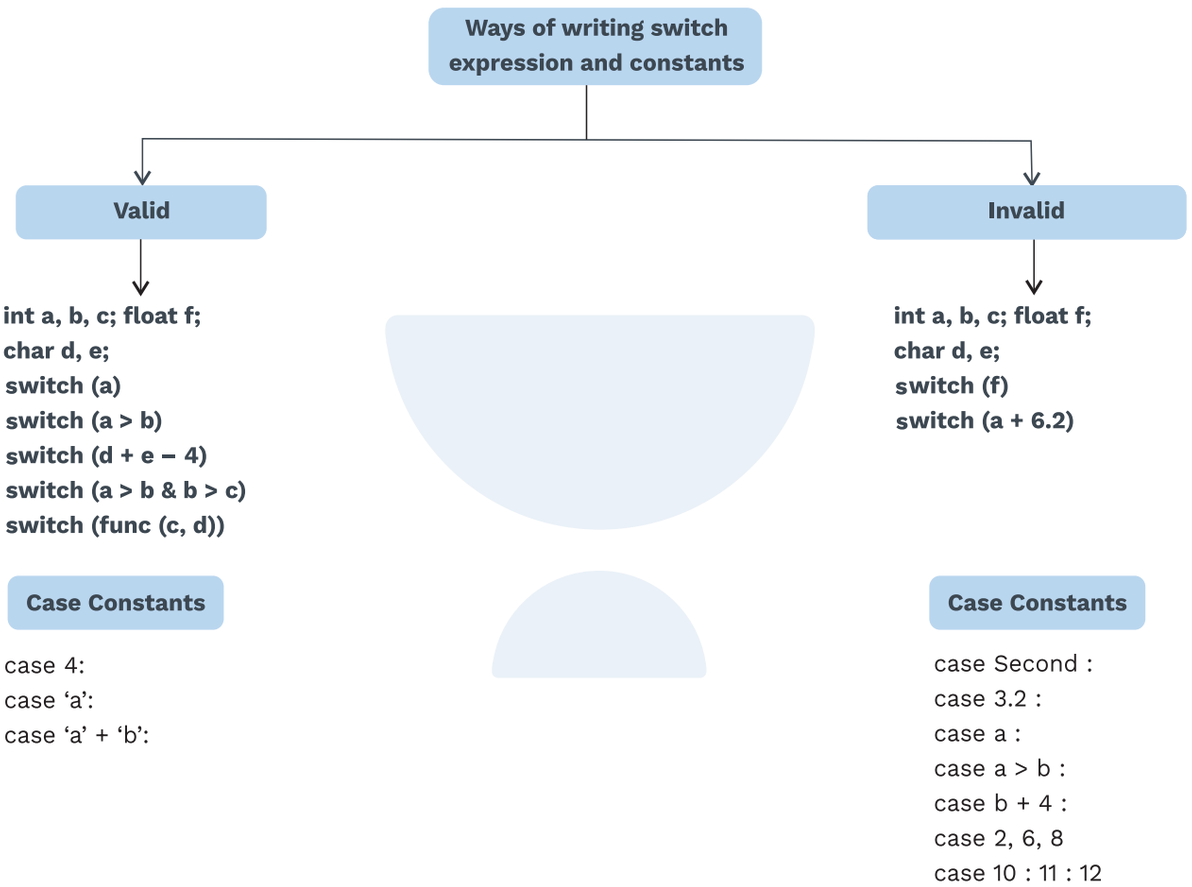


Fig. 1.32

Note

- If break statements are not used, then the control falls through the case in switch. As soon as one case matches, the cases following it are also executed in the absence of break statement.
- This is popularly known as fall-through cases.
  Default can be executed if no other case constant matches the switch expression.

Previous Years' Question

What will be the output of the following C program segment?

```
char inchar='A';
switch(inchar)
{
        case 'A': printf("choice A\n");
        case 'B':
        case 'C': printf("choice B");
        case 'D':
        case 'E':
        default: printf("No choice");
}
```

No choice

a) choice A
b) choice A
c) Choice B No choice
d) program gives no output as it is erroneous.

Sol: c) **(GATE-2012)**

1.8 FUNCTIONS AND PROGRAM STRUCTURES

1.8.1 Basics of functions:

- Self-contained sub-program with a well-defined task.
- Although function calls are overhead but are still used.

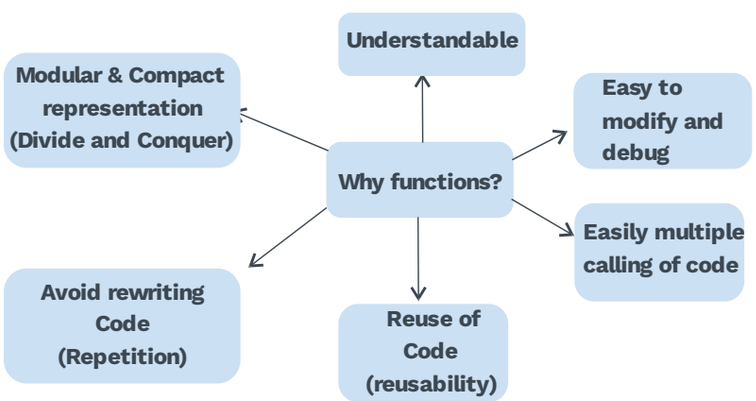


Fig. 1.33

Types of functions:

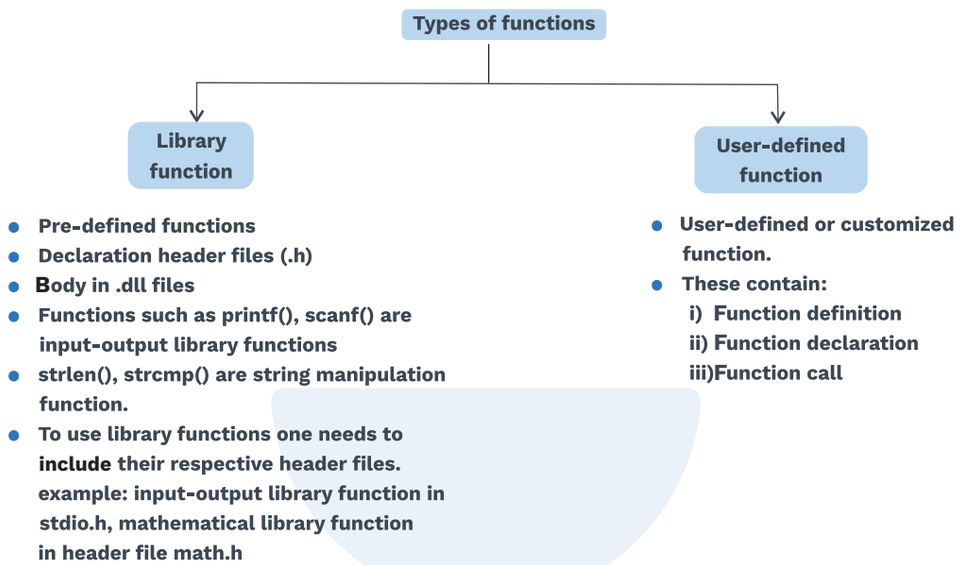


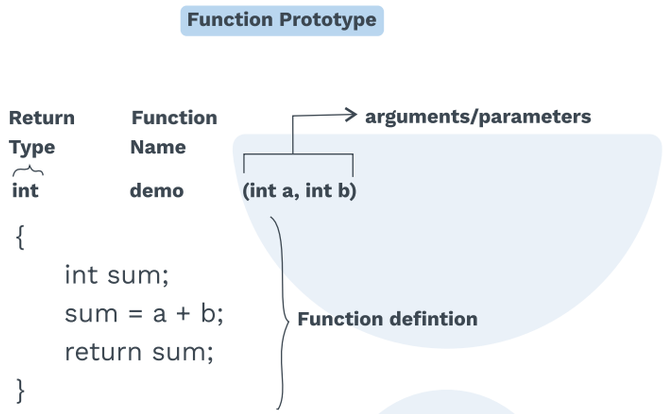
Fig. 1.33 Types of Functions

S.No.	Function Aspects	Syntax	Meaning
1)	Function declaration	return_type function_name (argument)	• Must be declared globally for the compiler to know about the function parameters and return type.
2)	Function call	function_name (argument)	• Can be called from anywhere in a program. • Parameter should be same for function declaration & functions call.
3)	Function definition	return_type function_name (argument) {function body; }	• Contains actual statements which are to be executed.

Table 1.22

Syntax:

```
return_type function_name (arguments)
{
        Statement;
        _____
        _____
        _____
}
```



Example: Program to find sum of two numbers:

```
#include <stdio.h>
int sum (int x, int y); //function declaration
main ()                                          Calling function ?
{       int a, b, s;                             → main()
        printf("enter the value of a & b:");     Called function?
        scanf("%d%d", &a, &b);                          → sum()
        s = sum(a, b); //function
        printf("sum=%d",b);
}
int sum (int x, int y) //function definition
{
        int s;
        s = x + y;
        return s;
}
```

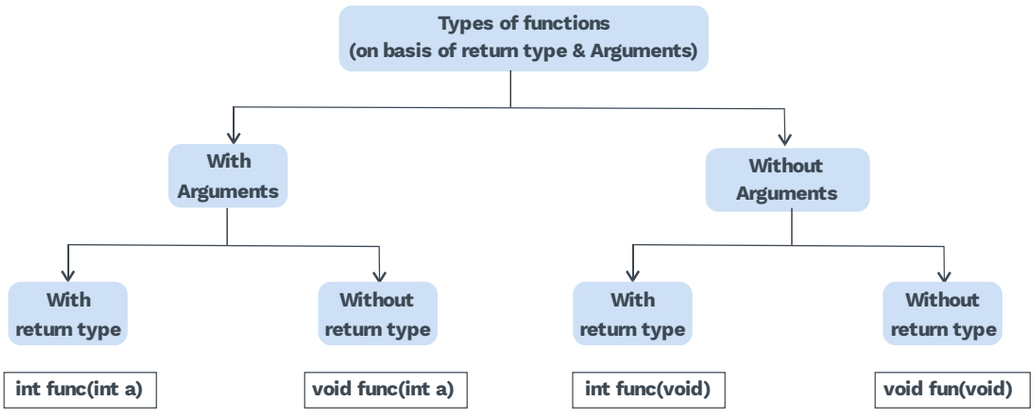


Fig. 1.34 Types of Functions Basis of Return Type and Arguments

Function arguments:

- The calling function sends some values to the called function, these values are known as arguments or parameters.

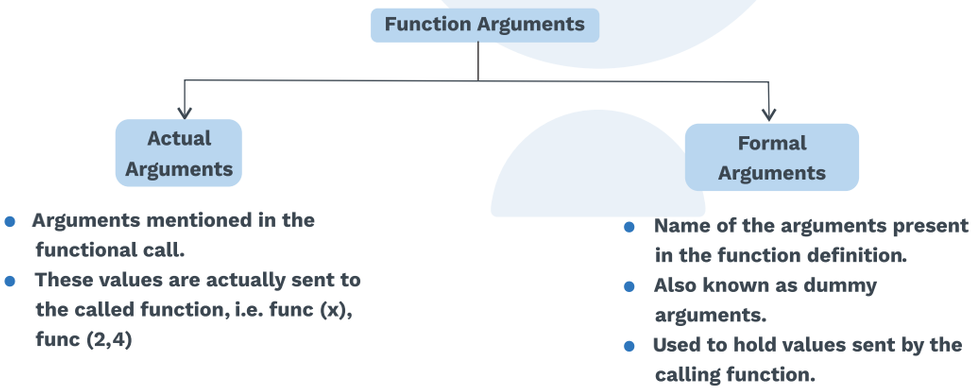


Fig. 1.35 Types of Function Arguments

Note

The order, number and type of actual arguments in the function call should match with the order, number and type of formal arguments in the function definition.

Example: Program showing formal and actual arguments:

```
#include <stdio.h>
mul(int p, int q)//formal arguments
{       int prod;
        prod = p*q;
        return prod;
}
sum (int p, int q)//formal arguments
```

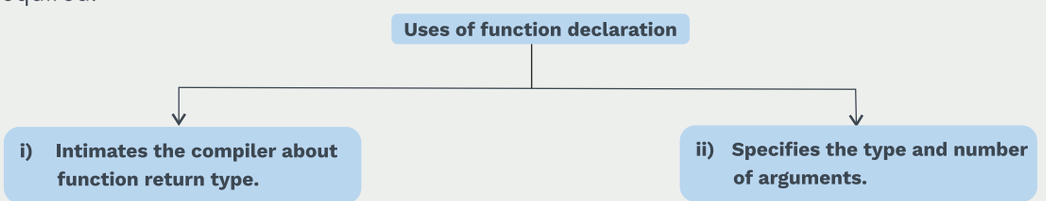
```
{       int s;
        s = p + q;
        return s;
}
main()
{int a = 2, b = 3;
        printf("%d\t", mul(a, b));//actual arguments
        printf("%d\t", mul(5, 4));//actual arguments
        printf("%d\t", mul(a+b, b–a));//actual arguments
        printf("%d\t", sum(a, b));
        printf("%d\t", sum(mul(a, b), b));
}
```

Output: 6 20 5 5 9

Grey Matter Alert!

1) If function definition occurs before the function call, then function declaration is not required.

2)



3) Function declaration is optional, but is a GOOD PRACTICE.

4) Declaration is absent.

Case I: Actual Arguments more than formal arguments.

- extra actual arguments are ignored.

Case II: Actual Arguments are less than formal arguments.

- extra formal arguments receive garbage value.

Example : function (int x, int y, int z)

```
{ _____
  _____
  _____
}
```

Case I:

```
function (1, 2, 3, 4, 5);
                  └──→Actual Arguments ignored.
```

Case II:

```
function (1, 2):
          └──→Missing actual argument:  The formal
              arguments takes garbage value.
```

Case II: function (1, 2):
→ Missing actual argument: The formal arguments takes garbage value.

5) Order of evaluation of function arguments:

- unspecified and compiler dependent

Example: int a = 4, m ;
m = multiply(a, a++);.....(1)

Let function multiply(a,b) be performing a * b, and returning the product, then if (1) is evaluated from left to right, answer would be 16.

But if (1) is evaluated from right to left, then answer would be 20.

Result is unpredictable and varies from compiler to compiler.

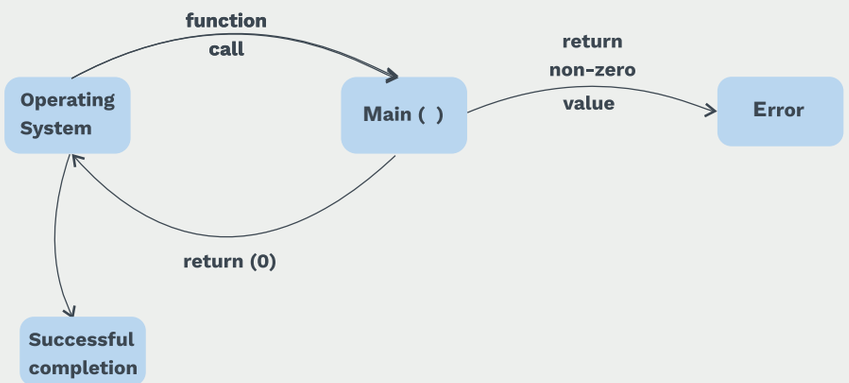
"Avoid such argument expression."

6) main() function

- Only one main() function.
- Each function is directly or indirectly called main().
- Each function after execution returns the control back to the main().

S.No.	Task	Performed by
1)	Function Declaration	C-compiler
2)	Function Definition	Programmer
3)	Function call	Operating System

- main () can also take arguments.



- Calling an exit() function with an integer return type is equivalent to returning values from main().
- If no return and if type specified for main () then any garbage value is returned automatically.

automatically.

7) L brary functions are not formally a part of C-language but, are supplied with every C-compiler.

8) One can define their own C-function with the same name and arguments as that of a pre-defined function. Here the user-defined function, then takes precedence over the library function.

String library functions:

- There are several library functions to manipulate strings, all present in the header file strings.

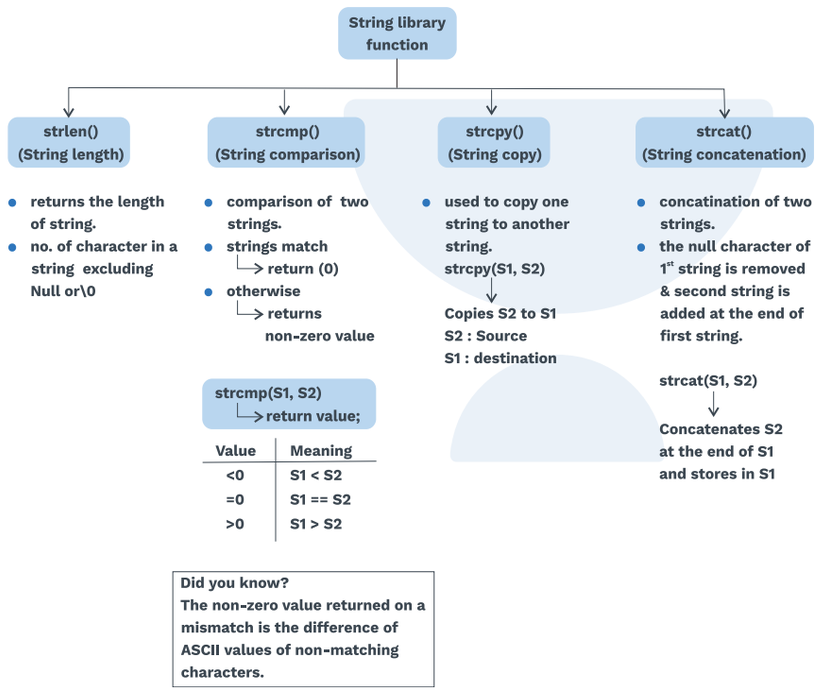


Fig. 1.36 String Library Functions

Example: Consider the given C-program

```
#include < stdio.h>
#include < string.h>
main ()
{       char S1 = “Banglore”;
        char str S3[10];
        char S2 = “Manglore”;

        int length1=strlen (S1);
        int length2=strlen (S2);
        printf (“length S1: %d\n”,length1);
        printf(“length S2:%d\n”,length2);
        if ((strcmp(S1, S2)) == 0)
```

```
            printf("strings are same\n");
        else
            printf("strings are different\n");
        strcpy (S3, S1);
        printf ("S3 string: %s\n", S3);

        strcpy (S1, S2);
        printf ("new string: %s\n", S1);
}
```

Output:

```
length S1: 8
length S2: 8
Strings different
S3 string: Banglore
New string: Banglore, Manglore
```

Local, global and static variables:

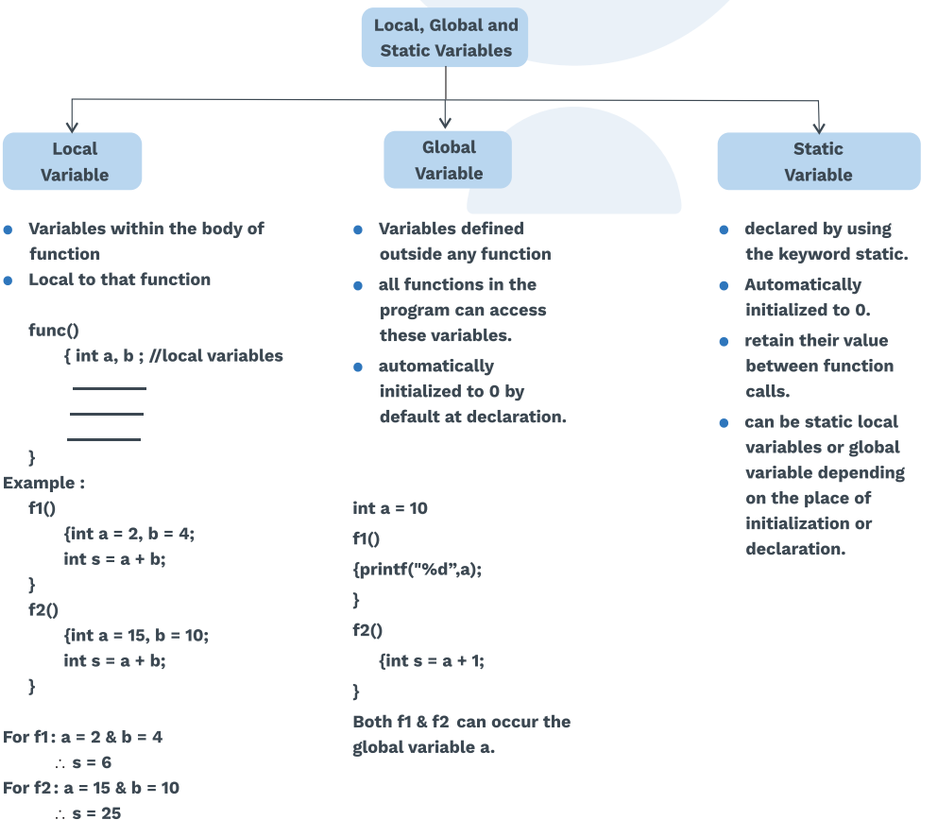


Fig. 1.37 Types of Variables

```
# include <stdio.h>
   double x;  ------>global variable
   static int y ;  ------>Static global variable

   void func1 ( int a, int b )  ------>  parameter variables
       {static int z;  ------>  Static Local variable
        double sum;  ------>  Local variable
          _____
          _____
          _____
          _____
       }

   void func2 ( double a, double b )  ------>parameter variables
       { float P;  ------>  local variable
          _____
          _____
          _____
       }
```

Recursion:

- It is the process when a function calls itself.
- Powerful technique whose complicated algorithms are solved by the divide and conquer approach.
- The sub-problems are defined in terms of the problem itself.

A function should have the capability of calling itself.

> The function that calls itself, again and again, is called a recursive function.

Example:

```
main ( )
{  _____
   _____
   rec ( ) ;
   _____
}
rec ( )
{  _____
   _____
   rec ( );  ------>recursive call
}                    |
                     v
```

As rec ( ) is called within the body of function rec ( ).

Point to remember:

- One should be able to define the solution of the problem in terms of a similar type of smaller problem.
- There should be a termination condition; otherwise, the recursive call would never terminate.

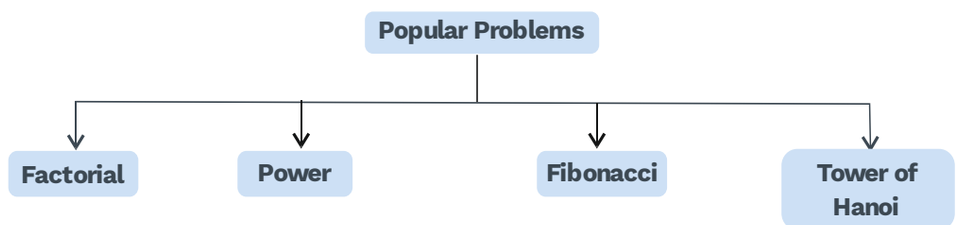


Fig. 1.38

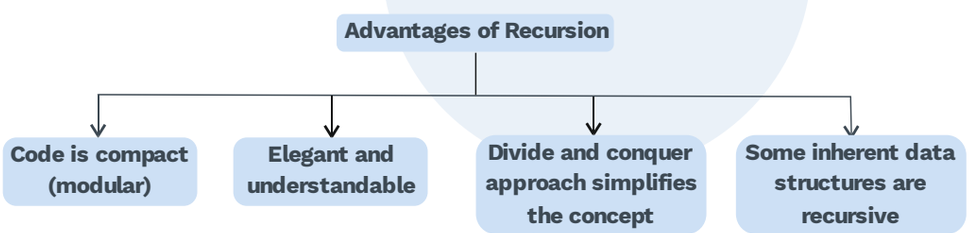


Fig. 1.39

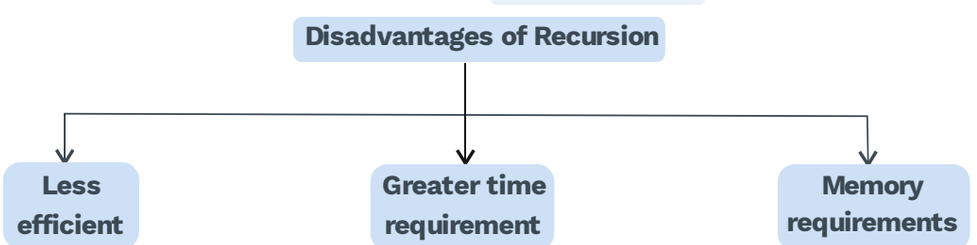


Fig. 1.40

Factorial:

Factorial of a positive integer n can be represented as a product of all integers from 1 to n.

n! = 1 * 2 * 3 * ..... *n

It can be written as:

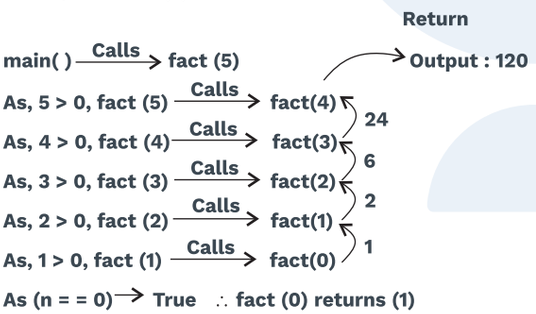
n! = n * (n – 1)!

Here: $n! = \begin{cases} 1 & ;n = 0 \\ n * (n-1)! & ;n > 0 \end{cases}$

Program to find the factorial:

```
#include <stdio.h>
long fact (int n);
main()
{       int num;
        printf("enter the number");
        scanf("%d" & num);
        printf("factorial: %ld", fact(num));
}
long fact(int n)
{       if (n = = 0)
          return(1);
        else
          return (n*fact(n–1));
}
```

Let num = 5, then fact(5) :



Fibonacci

```
int fib (int n)
    {if (n = = 0 || n = = 1)
        return (1);
    else
        return (fib (n – 1) + fib (n –2));
    }
```

$$fib(n) = \begin{cases} 1 & ; (n = 0) \text{ or } (n = 1) \\ fib(n-1) + fib(n-2) & ; (n > 1) \end{cases}$$

Tower of hanoi

```
TOH (source, temporary, destination, n)
{   if (n > 0)
        TOH (source, destination, temporary, n – 1);
        printf("move disk from %d%c →%c\n", n, source destination);
    TOH (temporary, source, destination, n – 1);
    }
```

Rack Your Brain

1) GCD of two numbers.
2) Finding second maximum of three distinct numbers.
Hint. **1)** GCD of two numbers:

```
f(a, b)
{ if(a == 0)
    return b;
if(b = = 0)
    return a;
if(a == b)
    return a;
else
    if(a > b)
    return f(a – b, b);
else
    return f(a, b – a)
}
```

3) Second maximum of three distinct numbers:

```
int func(int a, int b, int c)
            {   if((a >= b) && (c < b))
                  return b;
                else
                if(a > = b)
                  return func(a, c, b);
                else
                return func(b, a, c); }
```

Value passing in functions:

Type of Value passing	Actual Arguments changed	Formal Arguments changed	Analogous value passing
1) Call by value	x	√	—
2) Call by reference	√	√	—
3) call by constant	x	x	—
4) Call by need	x	√	Call by value
5) Call by result	√	√	Call by reference
6) Call by restore	√	√	Call by reference
7) Call by text	√	—	Call by reference
8) Call by name	√	—	Call by reference

Table 1.23

Grey Matter Alert!

- Call by reference and call by name parameter passing produces different results when address of an array element is passed as an argument (parameters).
- If actual arguments are:
  - **Scalar variable : Call by name is equivalent to call by reference.**
  - **Constant expression : Call by name is equivalent to call by value.**
- Value passing is often referred to as parameter passing or argument passing.
- Here is a parameter to the name for the data that goes into a function.
  Pre-defined function. Here the user-defined function then takes precedence over the library function.

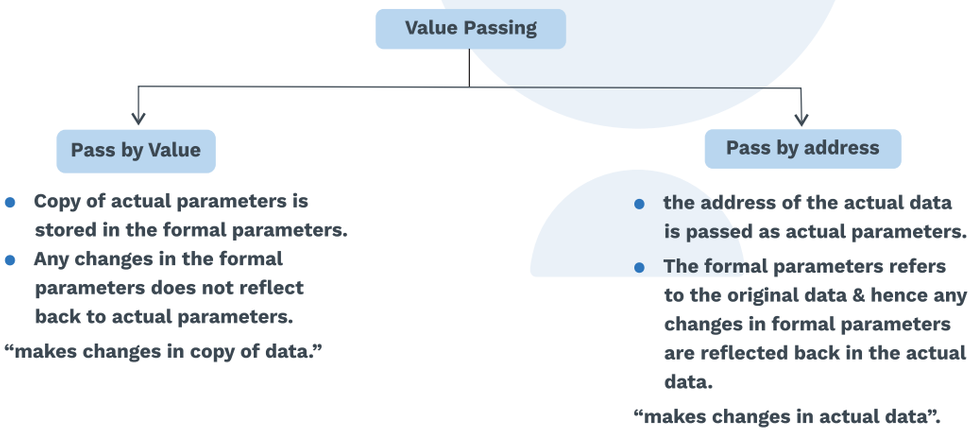
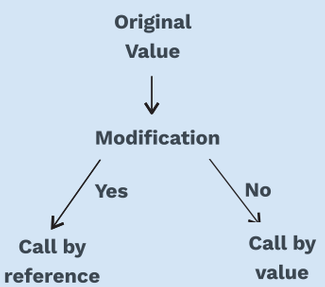


Fig. 1.41 Types of Value Passing Techniques

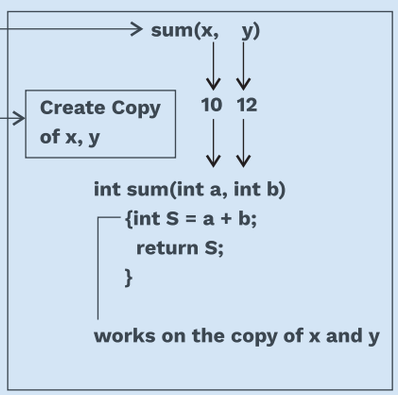
Note

- Words such as argument & parameter are used interchangeably. Similarly, call and pass are used. Call by value is the same as pass by value.
  Arrays are always passed by reference.
- **Address variable:** stores the address of another data variable. They are usually pointer variable.
  Example: int i = 10; //Data variable;
  int *ptr = &i;//pointer variable ptr storing address of variable i.
- **Reference variable:** alias to another variable.
  Example: int i = 10; //a data variable i
  int &r = i;//reference r to the data variable i.
- C only supports call by value and call by address. Call by reference is a feature of C++.

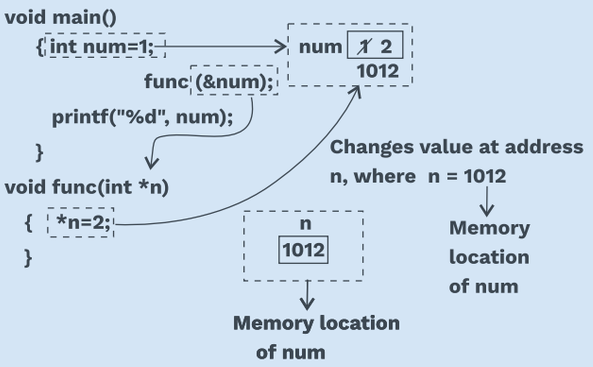


Example: Call by Value

```
(All by value)
# include <stdio.h>
int sum(int a, int b);
    main()
    { int x = 10, y = 12;
      int S = sum(x, y);
      printf("Sum=%d", S);
    }
int sum(int a, int b)
    { int S = a + b;
      return S;
    }
Output : Sum = 22
```



Example: Call by address



	Call by value	Call by reference	Call by address
1) **Definition**	A way of calling function where the actual arguments are copied to formal arguments.	A way of passing the arguments by copying the reference of an arguments into the parameters.	A way of passing values where the address of actual arguments are passed and is copied to formal arguments.
2) **Memory**	Memory is allocated for both actual and formal arguments.	Memory is allocated only for actual arguments. The formal arguments share the memory.	Memory is allocated to both actual arguments and formal arguments.
3) **Actual parameters**	Variables or constants	Variable	Address of variable
4) **Formal parameters**	Variables	Reference variable	Pointer variable (Address variable)

5) **Example**	void main()
	{int i=10;
	func(i);
	printf(”%d”,i);
	}
	func(int x)
	{x=x+1;
	printf(”%d”,x);
	}
		void main()
		{int i=10;
		func(i);
		printf(”%d”,i);
		}
		func(int &p)
		{p=20
		printf(”%d”,p);
		}
			void main()
			{int i=10;
			func(&i)
			printf(”%d”,i);
			}
			func(int *n)
			{ *n=20;
			print(”%d”,n);
			}

6) **Output**	10 11	20 20	20 20

Table 1.24 Call by Value vs Call by Reference vs Call by Address

Previous Years' Question

Consider the following C program:

```
#include < stdio.h>
    int r()
    {   static int num = 7;
        return num--;
    }
    int main()
    {   for(r() ; r(); r())
            printf("%d", r());
        return 0;
    }
```

Which one of the following values will be displayed during the execution of the programs?

a) 41 **b)** 630 **c)** 63 **d)** 52

Sol: d) **(GATE-2019)**

Hint:

for(expression 1 ; expression 2 ; expression 3)

- expression 1: executed only once
- expression 2: executed every iteration before code block execution
- expression 3: executed every iteration after code block execution

Previous Years' Question

Consider the following C function:

```
void convert(int n ) {
    if(n < 0)
    printf("%d", n);
else {
        convert(n/2);
        printf("%d",n%2);
    }
}
```

Which one of the following will happen when the function convert is called with any positive integer n as an argument?

a) It will print the binary representation of n in the reverse order and terminate.
b) It will print the binary representation of n but will not terminate.
c) It will not print anything and will not terminate.
d) It will print the binary representation of n and terminate.

Sol: d) **(GATE-2019)**

Previous Years' Question

Consider the following recursive C function:

```
void get(int n) {
    if(n < 1) return;
    get(n – 1);
    get(n – 3);
    printf("%d", n);
}
```

If get(6) function is being called in main(), then how many times will the get() function be invoked before returning to the main()?

a) 15 **b)** 25 **c)** 35 **d)** 45

Sol: b) **(GATE-2015 (Set-3))**

Previous Years' Question

Consider the following C program:

```
#include <stdio.h>
int f1(void);
int f2(void);
int f3(void);
int x = 10;
int main()
{   int x = 1;
    x+ = f1() + f2() + f3() + f2();
    printf("%d", x);
    return 0;
}
int f1(){int x = 25; x++; return x;}
int f2(){static int x = 50; x++; return x;}
int f3(){x * = 10; return x;}
```

The output of the program is _____.

Sol: 230 to 230 **(GATE-2015 (Set-3))**

1.9 POINTERS AND ARRAYS

1.9.1 Pointers and addresses:

- A pointer is a variable which stores the memory address of another variable in the memory.
- Real power of C lies in pointers.
- The memory of the computer is made up of bytes arranged in a sequential.
- Each byte has an index number called the address of that byte.
- If there are n bytes, the address would vary from 0 to (n – 1) bytes.

Example: 64 MB RAM

then, 64 MB

$\Rightarrow$ 64×2^{20} bytes

$\Rightarrow$ 67108864 bytes

Then: the address of these bytes would be: 0 to 67108863.

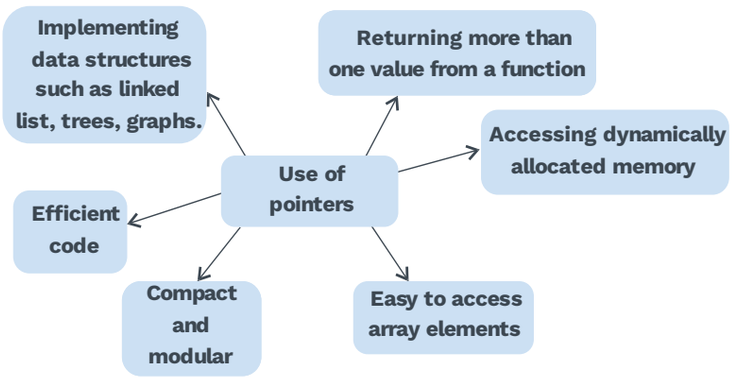


Fig. 1.42

Address operator:

- denoted by '&' (Ampersand) and read as 'address of'.
- returns the address of a variable when placed before it.

Example: &a: address of a

- Popularly used in scanf() function.

Program to print address of variable.

```
#include <stdio.h>
main ()
        {       int var = 10;
                float demo = 11.1;
                printf("value of var = %d, address of var = %u\n", var, &var);
                printf("value of demo= %f, address of demo = %u\n",
        demo, & demo);
        }
```

Output: Value of var = 10, address of var = 65524
Value of demo = 11.100000, address of demo = 65520

Note

- %u control sequence to print address; it doesn't mean that addresses are unsigned. Since there are no specific control sequences for addresses and addresses are also just whole numbers.
- The addresses printed may be different each time one runs the program. It depends on the part of the memory allocated by the operating system.
- Address operator cannot be used with constant or expression:
  valid : &j; &arr [1];
  Invalid : &12; &(j+k);

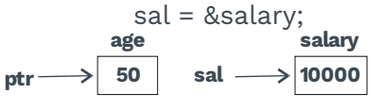
Pointer variables:

- Stores the memory address.
- Like all variables, it has a name, is declared and occupies space in memory.
- **Pointer:** Because it points to a memory location.

Syntax: data_type *pointer_name;
int *ptr

- * (asterik) often read as "value at".

Example: int *ptr, age = 50;
int *sal, salary = 10000;
ptr = &age;
sal = &salary;



- *ptr means value at variable ptr, since ptr stores the address of age.

∴ *ptr = * (&age) which is value at address of age.

This returns the value of variable age = 50

- Similarly: as sal = &salary

∴ *sal= * (&salary), which means the value at the address of salary.

- One can access the variable 'age' by using (*ptr) since ptr is storing the location of age, hence (*ptr), can be used in place of the variable name. This is called **dereferencing pointer variables.**

Example: int a = 10;
int b = 21;
int *ptr1=&a;
int *ptr2=&b;
Now:

Statement	Equivalent to
*ptr1 = 9	a = 9;
(*ptr1)++;	a++;
x=*ptr2+1;	x = b + 1;
printf("%d%d",*ptr1, *ptr2);	printf("%d%d", a, b);
scanf("%d%d", ptr1, ptr2);	scanf("%d%d", & a, & b);

Table 1.25

- Often referred to as indirection operation translating to "value at the address."

Note

- Value at and address of operator cancel each other out. (*, & operators cancel out each other.)
  Example: int age = 10;
  int *ptr = &age;
  then: *(&age) translates to value at address of age which is the value of variable age.
  hence: * (&age) = age

Example:

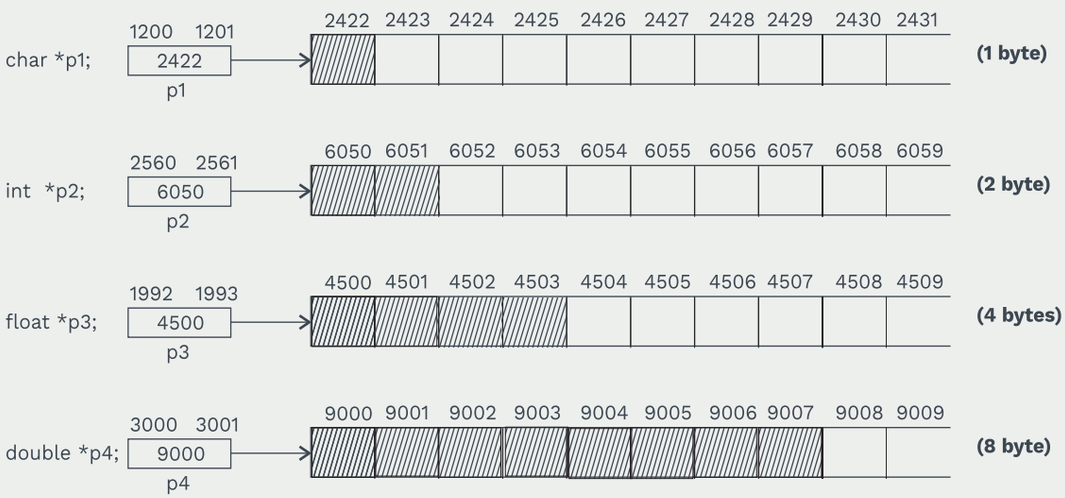
```
#include <stdio.h>
main()
{       int a = 67;
    float b = 45.6;
    int*ptr1=&a;
    float *ptr2 = &b;
    printf("Value of ptr1 = Address of a = %u\n", ptr1);
    printf("Value of ptr2 = Address of b = %u\n", ptr2);
    printf("Address of ptr1 = %u\n", &ptr1);
    printf("Address of ptr2 = %u\n", &ptr2);
    printf("Value of a = %d%d%d\n", a, *ptr1, *(&a));
    printf("Value of b = %f%f%f\n", b, *ptr2, *(&b));
}
```

Output:

```
Value of ptr1 = Address of a = 65524
Value of ptr2 = Address of b = 65520
Address of ptr1 = 65518
Address of ptr2 = 65516
Value of a = 67 67 67
Value of b = 45.600000 45.600000 45.600000
```

Grey Matter Alert!

- The size of the pointer variable is the same for all types of pointers but the memory that will be accessed while dereferencing is different.



size of the pointer variable	size of the value the pointer points to
sizeof(p1)=2	sizeof(*p1)=1
sizeof(p2)=2	sizeof(*p2)=2
sizeof(p3)=2	sizeof(*p3)=4
sizeof(p4)=2	sizeof(*p4)=8

- The size of int, float, and double are compiler, and architecture-dependent. This example is of a 16-bit compiler.

Pointer arithmetic:

- All types of operations are not possible with pointers.

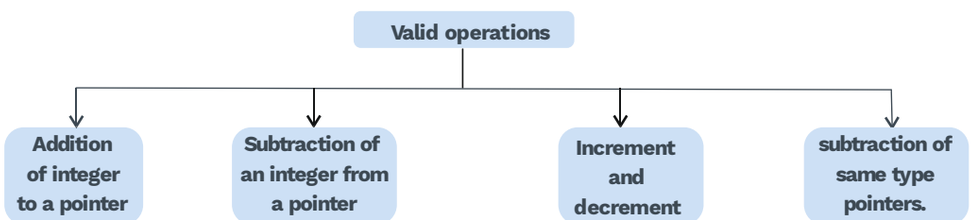
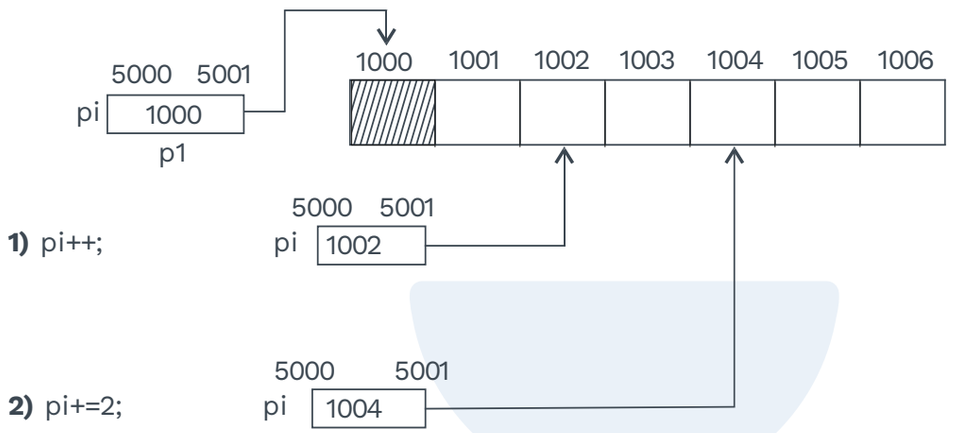


Fig. 1.43 Types of Value Passing Techniques

- Pointer arithmetic is different since it is performed relative to the size of the base type of pointer.

Example: int *pi;
if: pi = 1000, let int size be 2 bytes, then (pi)++; is 1002 and not 1001.
This is because the data type int occupies 2 bytes considering 16-bit computer.



pi = (pi + 2) = 1000 + 2*(size of int)
pi = (pi + 2) = 1000 + 2*2
pi = (pi + 3) = 1004

Note

Precedence and Associativity of Dereferencing operator (*), increment (++) and decrement (––) operator is same and from right to left respectively.

S.No.	Pointer Expression	Meaning

1)	x = *ptr++;	x = *(ptr++);
		Since associativity is from right to left.
		∴ (++) associativity first to ptr.
		x = *(++ptr);

2)	x = *++ptr;	x = *(++ptr);
3)	x=++*ptr;	x = ++(*ptr); Here (++) is applied to (*ptr) not ptr.
4)	x = (*ptr)++;	First assign value of (*ptr) to x and then increment (*ptr); since it is postfix increment.

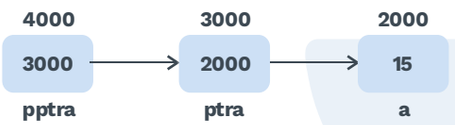
Table 1.26

Pointer to pointer:

- When the address of a pointer variable is stored in another variable, it is called pointer to a pointer variable.
- Similarly, one can have a pointer to pointer to a pointer variable, and this can be extended to any limit.

Syntax: data_type **ptr;
here: ptr is a pointer to pointer

Example:
```
int a = 15;
int *ptra = &a;
int **pptra = &ptra;
```



	Representation			Meaning/value
Value of a	a	*ptra	**pptra	5
Address of a	&a	ptra	*ptra	2000
Value of ptra	&a	ptra	*ptra	2000
Address of ptra		&ptra	pptra	3000
Value of pptra		&ptra	pptra	3000
Address of pptra			pptra	4000

Table 1.27

Previous Years' Question

Consider the following C-program:

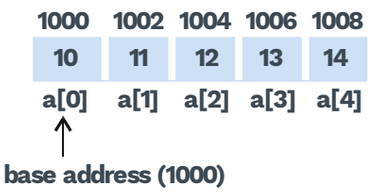
```
#include <stdio.h>
int main() {
static int a[] = {10, 20, 30, 40, 50};
static int *p[] = {a, a+3, a+4, a+1, a+2};
int **ptr = p; ptr++;
printf(" %d %d", ptr-p, **ptr);
}
```

The output of the program is _____.

Sol: 140 to 140 **(GATE-2015 (Set-3))**

Pointers and arrays:

- Elements of an array are stored in contiguous memory location.
  int a[5] = {10, 11, 12, 13, 14};
  Let int occupy 2 bytes.
  Then:



- There is a close-knit relationship between a pointer and an array. Array name is a pointer that only refers to the initial address of the array known as the base address of the array.
- Compiler accesses the array elements by converting subscript notation to pointer notation.
- Since the name of the array is a constant pointer that points to the first element of the array, therefore by pointer arithmetic when pointer variable is incremented, it points to fix the location of its base type, thereby making it possible to access all the elements of the array.

Example:
int a[5] = {10, 11, 12, 13, 14};

Representation	Equivalent to	Value
a	&a[0]	1000
a+1	&a[1]	1002
a+2	&a[2]	1004
a+3	&a[3]	1006
a+4	&a[4]	1008
*a	a[0]	10
*(a+1)	a[1]	11
*(a+2)	a[2]	12
*(a+3)	a[3]	13
*(a+4)	a[4]	14

Table 1.28

Note

- Accessing arrays by pointer notation is faster than accessing them by subscript notation since the compiler also ultimately changes the subscript notation to pointer notation.
- Array sub-scripting is commutative i.e. a[i] = i[a]

Pointer to an array:

- Useful in multidimensional Arrays

Syntax:

data_type pointer[array size];
int *ptr[10];//ptr is a pointer to an array of size 10.

Note

Here ptr is a pointer to an whole array of 10 elements and not to the first element of array.

Example:

Program to differentiate between pointer to integer and pointer to an array of integer.

```
#include <stdio.h>
main() {
    int *p;
    int(*ptr)[5];
    int a[5] = {1, 2, 3, 4,};
    p = a; // points to the first element of array a
    ptr = a; // points to the whole array a.
    printf(" p = %u , ptr = %u \n", p, ptr);
    p++; ptr++;
    printf(" p = %u, ptr = %u \n", p, ptr);
}
```

Output:

```
//(let int occupy 2 byte)
p = 1000, ptr = 1000
p = 1002, ptr = 1010
```

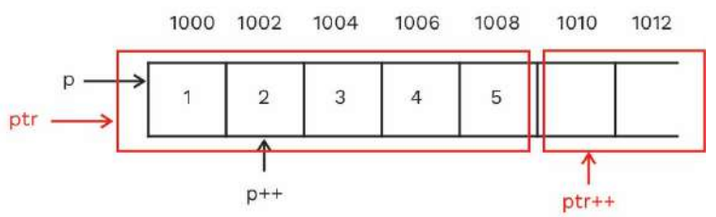


Fig. 1.44

Also, sizeof(p) = 2 sizeof(*p) = 2
sizeof(ptr) = 2 sizeof(*ptr) = 10

Pointers and 2D arrays:

- In 2-D arrays, the first subscript represents rows and the second subscript represents column.

Example: int a[3][4] = {{10, 11, 12, 13}, {20, 21, 22, 23}, {30, 31, 32, 33}};

	Column 0	Column 1	Column 2	Column 3
Row 0	10	11	12	13
Row 1	20	21	22	23
Row 2	30	31	32	33

- 2D arrays are stored in Row major order, i.e. rows are placed next to each other.

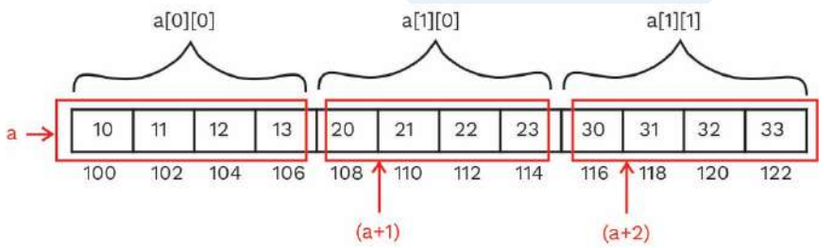


Fig. 1.45

a: points to 0th 1-D array →address(100)
(a+1): points to 1st 1-D array →address(108)
(a+2): points to 2nd 1-D array →address(106)

Hence: (a+i) points to ith element of a or points to ith 1-D array of a

Now: *(a+i) gives the base address of ith 1-D array.

Both(a+i) and *(a+i) are pointers but their base types are different, here:
(a+i) is an array of 4 integers
*(a+i) is equivalent to a[i] which is integer value here.

Representation	Meaning
a	Points to 0^{th} 1-D array
*a	Points to 0^{th} element of 0^{th} 1-D array
(a+i)	Points to i^{th} 1-D array
*(a+i)	Points to 0^{th} element of i^{th} 1-D array
*(a+i)+j	Points to j^{th} element of i^{th} 1-D array
((a+i)+j)	value of j^{th} element of i^{th} 1-D array

Table 1.29

Considering the above Example:

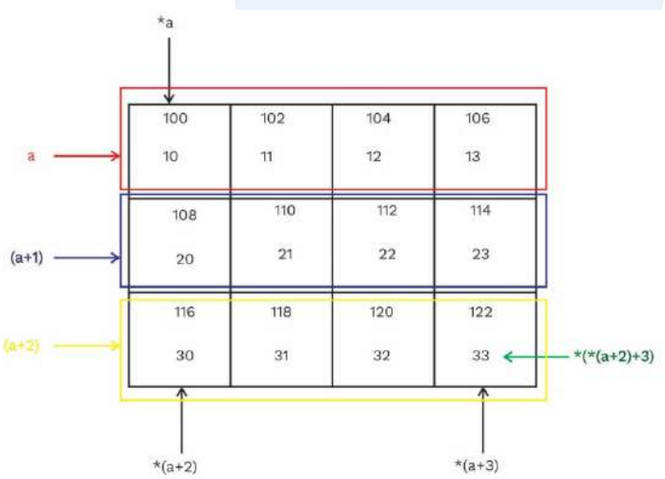


Fig. 1.46

Pointers and 3-D arrays:

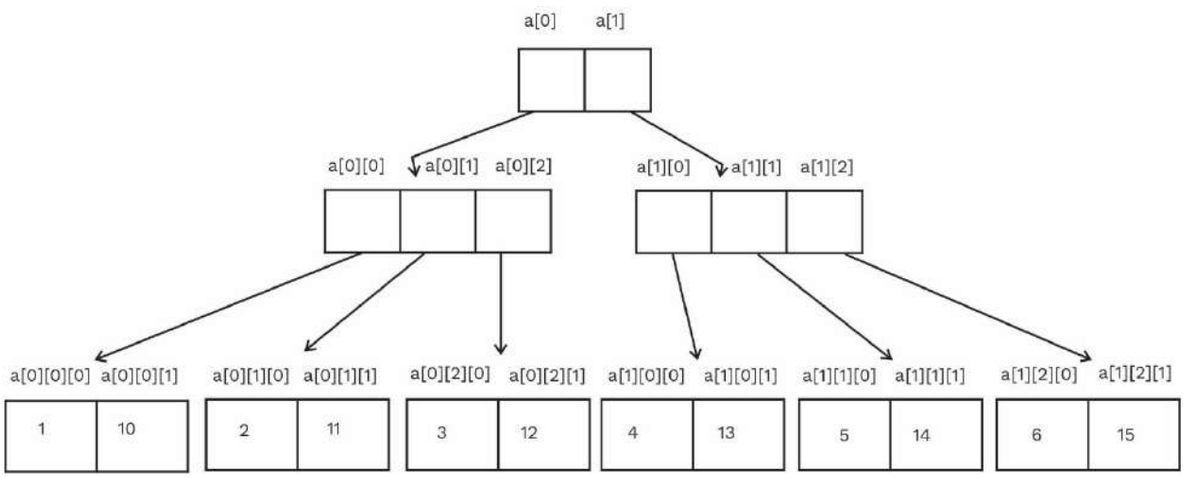
- Elements are accessed using 3 sub-scripts

int a[2][3][2] = {{{1, 10}, {2, 11}, {3, 12}},
{{4, 13}, {5, 14}, {6, 15}}};

- A 3-D array is considered to be an array of 2-D arrays, where each elements is a 2-D array.

Representation	Meaning
a	points to 0^{th} 2-D array
(a+i)	points to i^{th} 2-D array
*(a+i)	gives base address of i^{th} 2-D array (points to 0^{th} element of i^{th} 2-D array). Since each 2-D array is a 1-D array ∴ it points to the 0^{th} 1-D array of i^{th} 2-D array. (equivalent to a[i])
*(a+i)+i	points of j^{th} 1-D array of i^{th} 2-D array
((a+i)+j)	gives base address of j^{th} 1-D array of i^{th} 2-D array so points to 0^{th} element of j^{th} 1-D array of i^{th} 2-D array. (equivalent to points to a[i][j])
((a+i)+j)+k	points to k^{th} element of j^{th} 1-D array of i^{th} 2-D array.
((a+i)+j)+k	gives value of k^{th} element of j^{th} 1D array of i^{th} 2-D array. (equivalent to a[i][j][k]).

Table 1.30



Grey Matter Alert!

a→ address of a[0] ≈ &a[0]

a[0] → address of a[0][0]≈&a[0][0]

a[0][0]→address of a[0][0][0]≈&a[0][0][0]

Rack Your Brain

Let int a[3][2][2] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
(Assume int of 2 bytes and array start from 100), then fill the table:

Representation	Meaning
a	
a+1	
*a	
*a+1	
&a+1	
**a	
***a	
**a+1	
***a+1	

Table 1.31

Previous Years' Question

Consider the following C-program:

```
#include <stdio.h>
    int main()
    {int a[] = {2, 4, 6, 8, 10};
    int i, sum = 0, *b = a+4;
    for(i=0; i <5; i++)
                    sum = sum +(*b-i) – *(b-i);
    printf("%d \n", sum);
    return 0;
    }
```

The output of the above C program is _____.

Sol: 10 to 10 **(GATE-2019)**

Previous Years' Question

Consider the following C-program:

```
#include <stdio.h>
        int main() {
        int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 5},
        *ip = arr +4;
        printf("%d\n", ip[1]);
        return 0;
}
```

The number that will be displayed on execution of program is _____ .

Sol: 6 to 6 **(GATE-2019)**

Pointers and functions:

- Arguments to the functions can be passed in two ways:
  1) Call by value.
  2) Call by reference.
- function can return pointers too.

Syntax:

```
type *func(type1, type2 ....);
float *func(int, char); // function returns pointer to float.
```

Example:

```
int *fun()
{int x = 5;
    int *p = &x;
       return p;
}
main()
{int *ptr;
    ptr = fun();
       _____
       _____
       _____
    }
```

Let x be stored at 1000 address, then p = 1000.

Character pointer and functions:

- Character pointers are used to initialize string constant.

Example: char *ptr = “Demo”;

Here ptr is a character pointer which points to the first character of the string constant “Demo”, i.e, the base address of this string constant.

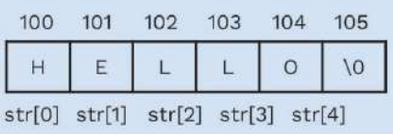
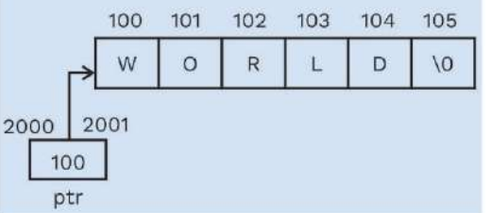
String as arrays	String as pointers
char str[] = “Hello” ; char str[] = { ‘ H’, ‘E’, ‘L’, ‘L’, ‘O’, ‘\0’};  	char *ptr = “World” ; 

Table 1.32 String as Arrays vs String as Pointers

Rack Your Brain

1) Consider the given C-program segments and assume header files are included.

```
main()
    {int i , arr[5] = {2, 3, 5, 4, 6}, *p;
    p = &arr[4];
    for(i = 0; i <5; i ++)
    printf("%d \t%d \t", *(p-i), p[-i]);
}
```

2)

```
main()
    {int i, j;
        int arr[10] = {3, 2, 4, 1, 5, 9, 8, 10, 7, 6};
        for(i=0; i <10; i++)
        for(j=0; j <10-i-1; j++)
        if(*(arr+j) > *(arr + j+1))
        swap(arr+j, arr+j + 1);
        for(i=0; i <10; i++)
        printf("%d \t", arr[i]);
        printf("\n");
    }
        swap(int *b, int *c)
    {int temp;
    temp = *b, *b= *c, *c = temp;
    }
```

Hint : Sorting using bubble sort.

3) If a is declared in C programming language as a 1-D array then which of these statements is/are correct?

1) *(a+i) is same as *(&a[i])
2) *(a+i) is same as *a+i
3) &a[i] is same as a+i-1
4) *(a+i) is same as a[i]

Previous Years' Question

Consider the following C-program:

```
#include <stdio.h>
    void fun1(char *s1, char *s2) {
    char *tmp;
    tmp = s1;
    s1 = s2;
    s2 = tmp;
}
    void fun2(char **s1, char **s2) {
    char *tmp;
    tmp = *s1;
    *s1 = *s2;
    *s2 = tmp;
}
    int main() {
    char *str1 = "Hi", *str2= "Bye";
    fun1(str1, str2); printf("%s %s", str1, str2);
    fun2(&str1, &str2); printf("%s %s", str1, str2);
    retrun 0;
}
```

The output of the program is:

a) Hi Bye Bye Hi
b) Hi Bye Hi Bye
c) Bye Hi Hi Bye
d) Bye Hi Bye Hi

Sol: Option a) **(GATE-2018)**

Previous Years' Question

Consider the following C program

```
#include <stdio.h>
#include <string.h>
int main()
    {char *c = "GATE CSIT 2017";
    char *p = c;
printf("%d",(int) strlen(c+2[p] – 6[p]-1));
return 0;
}
```

The output of the program is _____.

Sol: 2 to 2 **(GATE-2017 (Set-2))**

Previous Years' Question

Consider the following C program

```
#include <stdio.h>
#include <string.h>
        void printlength(char *s, char *t)
        {unsigned int c = 0;
        int len =((strlen(s) – strlen(t)) > c) ? strlen(s): strlen(t);
        printf("%d\n", len);
}
        void main()
        {char *x = "abc";
        char *y = "defgh";
        printlength(x,y);
}
```

Recall that strlen is defined in string.h as returning a value of type size_t, which is an unsigned int. The output of the program is _____.

Sol: 3 to 3 **(GATE-2017 (Set-1))**

1.10 Structures:

Basics of structures:

- A structure is a group of items in which each item is identified by its own identifier, each of which is known as a member of the structure.
- Used to store related fields of different data types.
- Capable of storing heterogeneous data.

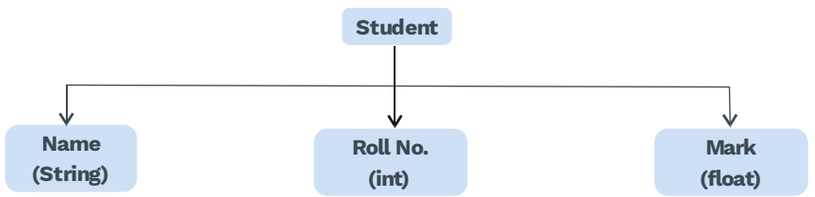


Fig. 1.47

Syntax:

```
struct student{
    char name[20];
    int roll no;
      float marks;
};
```

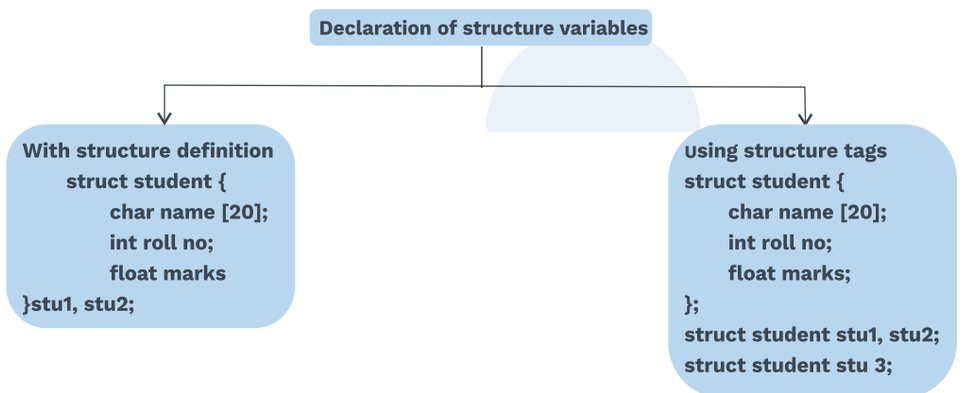


Fig. 1.48

Both declaration creates three variable which enables access individual structure members as:

stu1.name, stu1.rollno, stu1.marks;

Let char occupy 1 byte, int 2 bytes and float 4 bytes then the entire structure student would reserve (1×20+2+4) = 26 bytes.

Structure initialization:

- The number, order, and type of values must be same as the structure template/ definition.

- The initializing values can only be constant expressions.

```
struct student {
char name[20];
int roll no;
float marks;
} stu1 = {“Ravi”, 13, 98};
struct student stu2 = {“Ravi”, 24, 86};
```

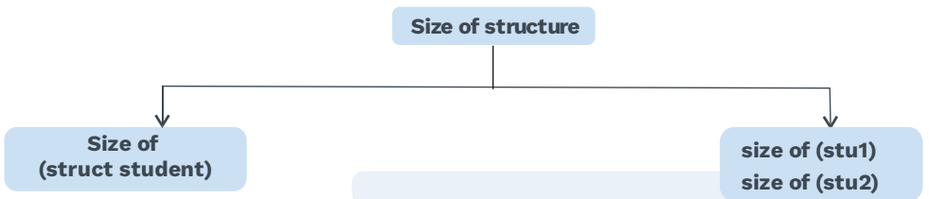


Fig. 1.49

1.10.2 Array of structures:

- each element of an array is a structure type.

Example: struct student stu[10];
Here stu is an array of 10 elements where each element is a structure having three members name, roll no &marks.

stu[0].name	stu[0].roll no	stu[0].marks
stu[1].name	stu[1].roll no	stu[1].marks
.		
.		
.		
.		
.		
.		
.		
stu[9].name	stu[9].name	stu[9].name

Example:

Program to sort by roll no.(bubble sort)

```
#include <stdio.h>
    struct stud
    {int roll;
    char code[25];
    float marks;
    };
```

```
main()
{struct stud class[100], t;
int j, k, n; scanf ("Enter the number of students", & n);
for (k=0; k<n; k++)
scanf(\"Enter roll no., dept code and marks"
%d %s %f", &class[k].roll, &class[k].code, &class[k].marks);
for(j=0; j<n-1; j++)
for(k=j+1; k<n; k++)
{if(class[j].roll > class[k].roll}
{t=class[j];
class[j]=class[k];
class[k]=t;
}
}
for(k=0; k<n; k++){
printf("roll no %d code %s marks%f\n", class[k].roll, class[k].code,
class[k].marks);
}
```

Arrays within structure:

- C allows the use of arrays as structure members.

Example:

```
struct student {
char name[20];
int rollno;
int submarks[4];
};
```

The array submarks represent subject marks. Each student has Four subject and each subject mark is denoted by submarks[0], submarks[1], submarks[2], submarks[3].

Note

one can have nested structures also, i.e. structure within structure.

Rack Your Brain

```
#include <stdio.h>
   struct class
     {int students[7];
     };
int main()
{    struct class c={20, 21, 30, 31, 40, 41};
     int *ptr;
     ptr=(int*) &c;
     printf("%d", *(ptr+2));
     return 0;
}
```

What is the output of the c –program_____ ?

Pointers to structures:

- A pointer to a structure stores the start address of a structure variable.

Example:

```
struct student {
char name[20];
int roll no;
int marks;
};
struct student stu, *ptr;
```

- ptr is a pointer that can point to the variable of type struct student.

```
ptr = &stu;
```

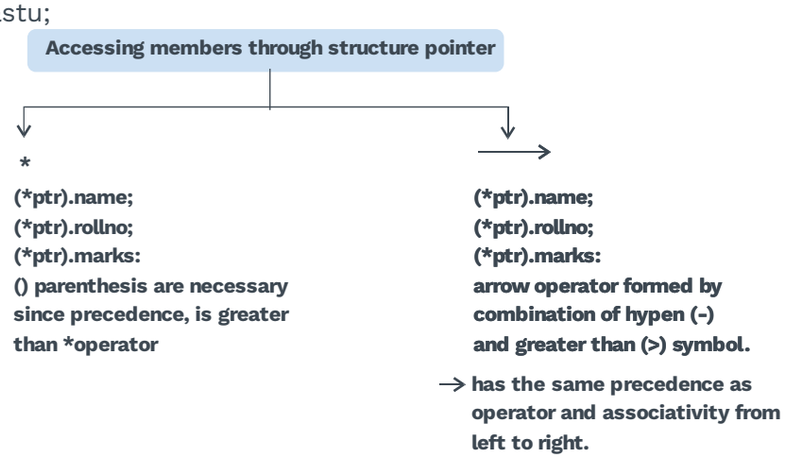


Fig. 1.50

Rack Your Brain

1) i) ptr → roll
   ii) (*ptr).roll
   iii) *ptr.roll
   Does I, II and III mean the same thing?

2) What does the follow represent incrementing ptr or roll no.?
   i) ++ptr → roll
   ii) (++ptr) → roll

Structures and functions:

- structures can be passed as arguments to functions. A function can have a return value as a structure.

Pass as argument to function:	Function can return:
1) pass individual members	1) a structure member
2) whole structure variable	2) whole structure variable
3) structure pointers	3) pointer to a structure

Table 1.33

Example:

Program to add two complex numbers.

```
#include <stdio.h>
struct complex {float re; //real part
                float in; //imaginary part
        }
struct complex x, y;
struct complex add(x,y)
{struct complex t;
t.re=x.re+y.re;
t.im=x.im+y.im;
return(t);}
main()
        {struct complex a, b, c; // variables a, b, c
printf("enter two complex numbers");
scanf("%f %f\n", &a.re, &a.im);
```

```
scanf("%f %f\n", &b.re, &b.im);
c=add(a,b);
printf("The addition of real no. a and b are %f %f \n", c.re, c.im);
}
```

Here the complex no. a has two parts real represented as a.re and imaginary represented as a.im. Similarly, for complex no. b and the result of the addition of complex no. c.

Example:

Program to add complex no. using pointers.

```
#include <stdio.h>
        struct complex{
        float re;
        float im;
} struct complex *x, *y, *t;
void add(x, y, t)
{       t → re = x → re + y → re;
        t → im = x → im + y → im;
}
main()
{       struct complex a, b, c;
printf("enter two complex no.");
scanf("%f %f\n" &a.re, &a.im);
scanf("%f %f", &b.re, &b.im);
add(&a, &b, &c);
printf("the addition of a and b are %f %f", c.re, c.im);
}
```

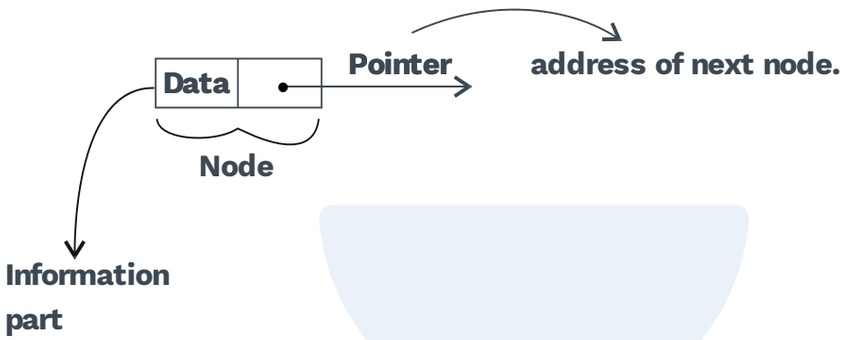
Self-referential structures:

- Structure that contains pointers to structures of its own type

Syntax:

```
struct tag {
        datatype member1;
        datatype member2;
        _________
        _________
        _________
        struct tag *ptr1;
        struct tag *ptr2;
        }
```

Here struct tag is a self-referential structure as it contains Two pointers ptr1 and ptr2 of type struct tag.
Linked lists are the most popular application of self-referential data structures.
A linked list has two parts, a data and a pointer, which together form a node.



Syntax: struct node {int info;
struct node *link;
}

Union:

- It is also a collection of heterogeneous data types.
- The only difference between struct and union is the way memory is allocated to its members.
- In structure: each member has its own memory location
- In union: members share the same memory location.
- When a or the union is declared, the compiler allocates sufficient memory to hold the largest member in the union.

Note: Union is used for saving memory

Syntax: union name

```
{
   datatype member1;
   datatype member2;
   ____
   ____
};
```

Similar to structures, unions can have union variables to access the members

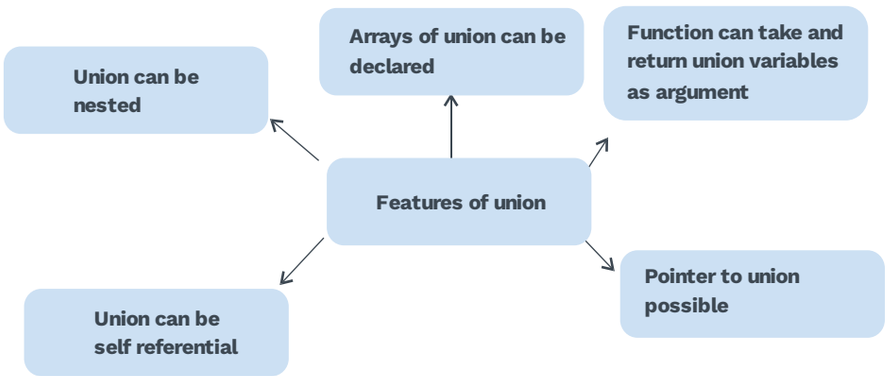


Fig. 1.51

Previous Years' Question

Consider the following C-programs ?

```
#include <stdio.h>
struct ournode
{char x, y, 2;
};

int main() {
struct ournode p = {'1', '0', 'a'+2};
struct ournode *q=&p;
printf("%c, %c",((char*)q+1), *((char*)q+2));
return 0;
}
```

The output of the program is:

a) 0, c **b)** 0, a+2 **c)** '0', 'a+2' **d)** '0', 'c'

Sol: Option a) **(GATE-2018)**

Previous Years' Question

The following c declaration

```
struct node {
int I;
float j;
};
struct node *s[10]:
```

Define s to be.

a) An array, each element of which is a pointer to a structure of type node.
b) A structure of 2 fields, each field being a pointer to an array of 10 elements.
c) A structure of 3 fields: an integer, a float, and an array of 10 elements.
d) An array, each element of which is a structure of type node.

Sol: Option a) **(GATE-2000)**

Typedef:

- allows you to define a new name for an existing datatype.

Syntax: typedef datatype_name new_name;

Example: typedef int marks;

- now marks is a synonym for integer i.e marks sub1, sub2;

	Syntax
1) Pointers	typedef datatype *pointer_name; **Example :** typedef float *fptr; fptr is synonym for float pointer **Valid :** fptr p, q, *r;
2) Arrays	typedef datatype arrayname[size]; **Example :** typedef int arr[10]; arr is a synonym for integer array of 10 elements. **Valid :** arr a, b, c [10];
3) Functions	typedef datatype func_name(arguments) **Example :** typedef floatfunc(float, int); func is synonym for any function taking two argument one float & other integer. **Valid :** func add, sub, null;

4) Structures	typedef struct structure_namevariable_name; **Example :** typedef struct student std; now structures student has a synonym std. **Valid :** std stu 1, stu 2;

Rack Your Brain

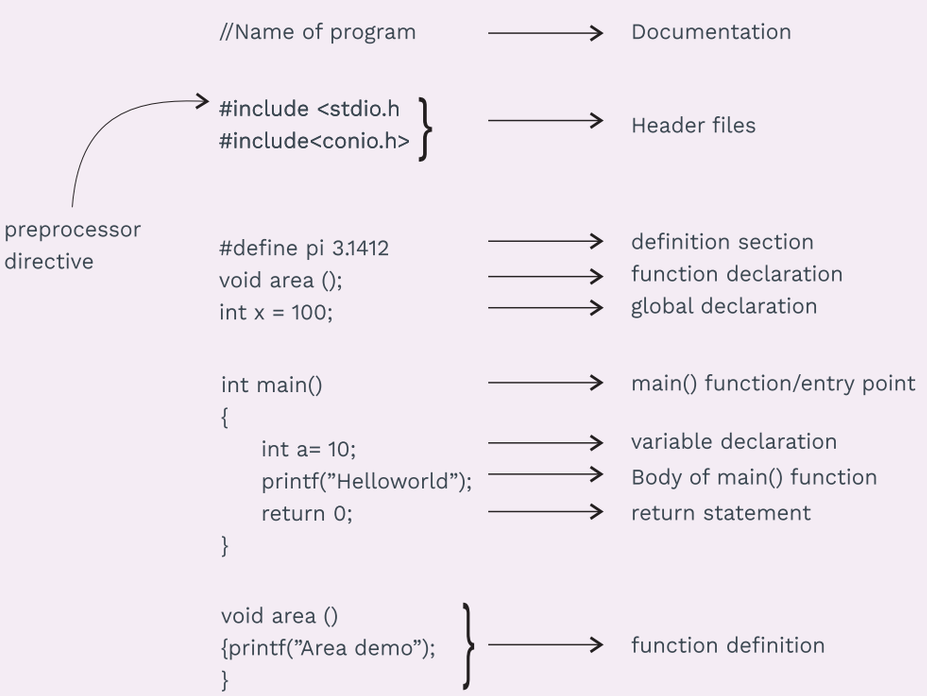
1) Let:
```
struct student
{char name [20];
 int marks;
}
```
then: typedef struct student std;
Which of the following variable declaration is true?
1) student stu1, stu2;
2) struct student stu1, stu2;

2) typedef int a arr; then what would arr A[15]; mean?

3) typedef float *fptr;
Then what would fptr *r; mean?

Chapter Summary

- C-programming given by Dennis Ritchie in 1972.
- Some of its features are portability, syntax based, case-sensitive, middle level, simple yet powerful, fast and efficient.
- The character set includes alphabets, digits and symbols. C has a total of 32 keywords.
- Components of basic C-program;

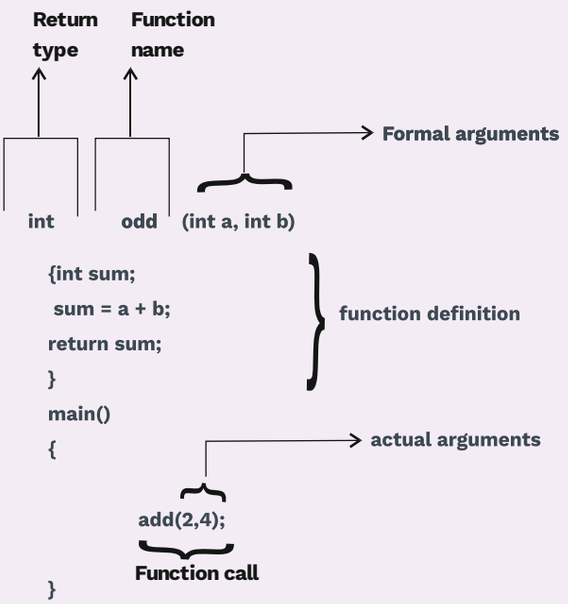


- C has integer, float, double and character as primary data types and structure, union, and enumeration as secondary or user-defined data types.
  int will normally be the natural size for a particular machine. short is often 16 bits long, and int is of 16 or 32 bits. Each compiler allocates storage to each data type based on the hardware, subject only to the the restriction that shorts and ints are at least 16 bits, longs are at least 32 bits, and short is no longer than int which is no longer than long.
- Datatype size is machine-dependent.
- C has four storage classes, namely automatic, static, external and register, that specify the lifetime, scope, initial value & place of storage of a variable.

- C has Arithmetic, Assignment, increment decrement, relational, logical, conditional, comma, sizeof, and bitwise operators to perform various operations.
- Operators are either unary(one operand) or binary(two operands)
- The conditional operator is a ternary operator(?:)
- C supports both implicit and explicit type conversion.
- Precedence and associativity play a vital role in expression evaluation.

- C has three control statement categories namely: 1. selection statements: if, if-else, switch, 2. iteration statements: for, while, do while, 3. jump statements: break, continue, goto, return.
- C has two type of function: user-defined and library functions.

- The library functions are in the header files; hence to use these functions in a program one, needs to include the header files.
- Functions that are defined by the programmer are called user-defined functions.
- Functions aim at making the code modular, compact, understandable, easy to modify, reusability and altogether avoid repetition of code.
- Function can be called with some argument; these are actual and formal arguments/ parameters.
- Actual arguments / parameters are the ones in the function calls.
- Formal arguments / parameters are the ones in the function definition.



- Recursion is a major application of function but is not preferred as a good programming practice due to the memory and time requirement overhead.
- Pass by value/call be value and pass by reference/ Call by reference are two major value passing techniques in C.
- In pass by value, a copy of actual arguments is stored in formal arguments. The changes in formal argument not reflected back to the actual arguments.
- In pass by reference, the address of actual arguments is passed, and changes in formal arguments are reflected in actual arguments.
- A pointer is a variable which stores the memory address of another variable. Pointers are what make C-programming language really powerful.
- Pointers help in implementing various data structures such a linked lists, trees, and graphs, helps in returning one or more values from functions, provide easy access to array elements, and play major role in dynamic memory allocations. Also makes the code compact, modular and efficient.
- Some of the valid operations on pointer include the addition of integer to a pointer, subtraction of an integer from a pointer, increment and decrement of pointers, subtraction of same type pointers.
- Pointers are used in function in the call by reference function calling.
- Structure and unions are user-defined structures which are heterogeneous. Both can store multiple data types.
- The only difference is that structures reserve memory locations for each datatype, whereas unions reserve memory location for only the largest number of union(members share the memory location).
- Self-referential structures are used to implement data structures such as linked lists.