Module-1

What is C++?

→ C++ is an object-oriented programming language. where object is real world entity and helps model programs to real world solutions.

→ C++ is a superset of C. most of C programs are C++ programs, however, there are a few minor differences that will prevent a C program to run under C++ compiler.

→ Most Important facilities that C++ adds on to C are classes, Inheritance, function overloading and operator overloading.

* These features enable creation of abstract data types, inherit properties from existing data types and support polymorphism, thereby making C++ a truly object-oriented language.

→ object-oriented features in C++ allows programmers to build large programs with clarity, extensibility and ease of maintenance, incorporating the spirit and efficiency of C

Applications of C++:

→ It is suitable for virtually any programming task including development of editors, compilers, databases, communication systems and any complex real-life application systems.

* Since C++ allows to create hierarcy-related objects, we can build special object-oriented libraries which can be used later by many programmers.

* C++ is able to map the real-world problem properly the C part of C++ gives the language the ability to get close to the machine-level details.

* C++ programs are easily maintainable and expandable. when new feature needs to be implemented, it is very easy to add to the existing structure of an object.

* It is expected that C++ will replace C as a general-purpose language in the near feature.

A SIMPLE C++ Program

```
#include <iostream>          // include header file
using namespace std;
int main()
{  cout<<" C++ is better than C \n"; // C++ statement
   return 0;
}
```

program features:

* From the example contains only one function, main().
* Execution begins at main(). C++ program must have a main().
* C++ is a free-form language.

Comments:

* C++ introduces a new comment symbol "//" (double slash).
* comment start with a double slash symbol and terminates at the end of the line.
* It is a single line comment (//)
* for multiline comment symbol is /*, */.

Eg: /* This program is of
C++ program */

output Operator:

Cout << "C++ is better than C" << endl;

Cout → Console output

<< → stream insertion operator.

endl → new line

→ Cout is predefined object that represents the standard o/p stream in C++.

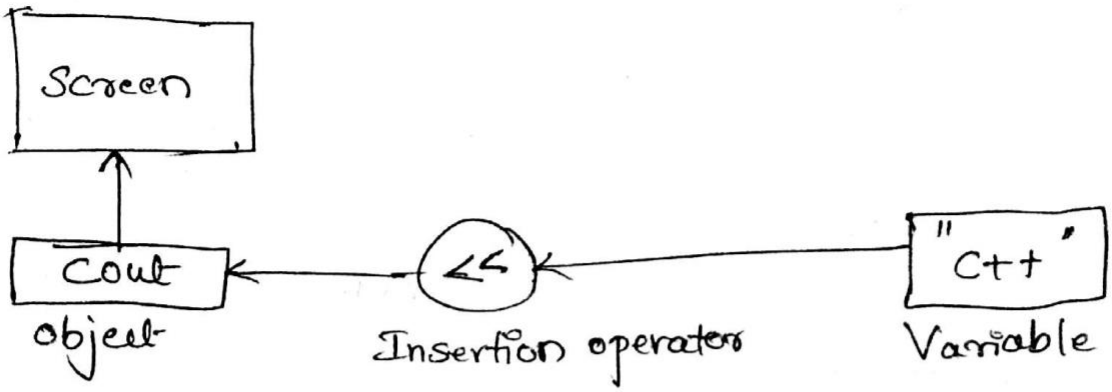


fig: o/p using insertion operator.

→ From above figure the standard o/p stream represents the screen.

→ It is also possible to redirect the o/p to the other output devices.

INPUT OPERATORS

cin >> number1;

where cin → Console input, is a predefined object represents the standard i/p stream.

>> → stream extraction operation

number1 → Variable.

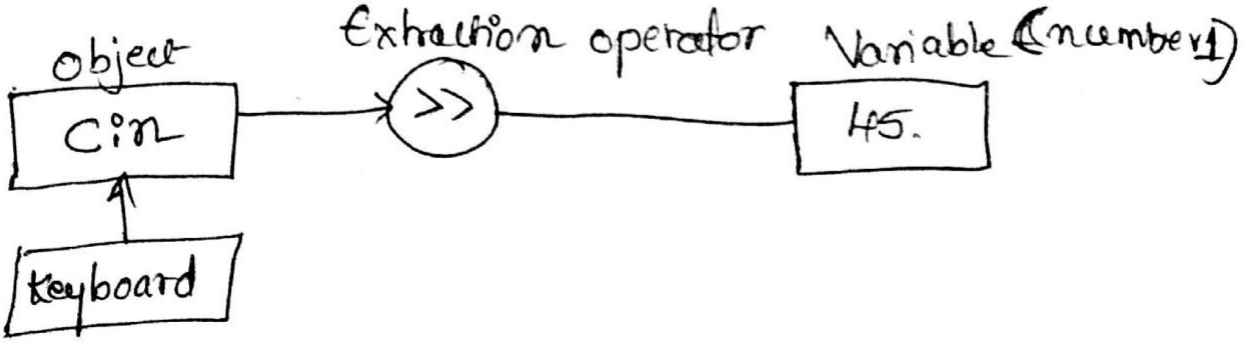


fig: Input using extraction operator

→ The above statement is an input statement and causes the program to wait for the user to type in a number.

→ The number key is placed number 1 from keyboard

→ The Identifier cin is a predefined object in C++. that corresponds to the standard input stream.

The iostream File

→ C++ program typically contains pre-processor directive statements at the beginning.

→ such statements are preceded with a # symbol to indicate the presence of a preprocessor directive to the compiler.

→ All C++ programs starts with "#include" directive that includes the header file contents into the main program

Eg. #include<iostream>

→ It means that "the preprocessor to add the contents of the iostream file" to the program.

→ It contains declarations for the identifier cout and the operator "<<".

→ The header file iostream should be included at the beginning of all programs that use input/output statement

Name Space :

→ This defines a scope for the Identifiers that are used in a program.

→ for using the identifiers defined in the namespace scope we must include the using directive like

```
using namespace std;
```

→ "std" is the namespace where C++ standard class libraries are defined.

→ This will bring all the identifiers defined in std to the current global scope.

→ using and namespace are the new keywords of C++.

Return type of main()

→ In C++ "main()" returns an integer type value to the operating system.

→ ∴ every main() in C++ should end with a return(0) statement.

→ Return type for main() is integer specified as int

```
int main()
{ Statement;

return(0);
}
```

Structure of C++ Program

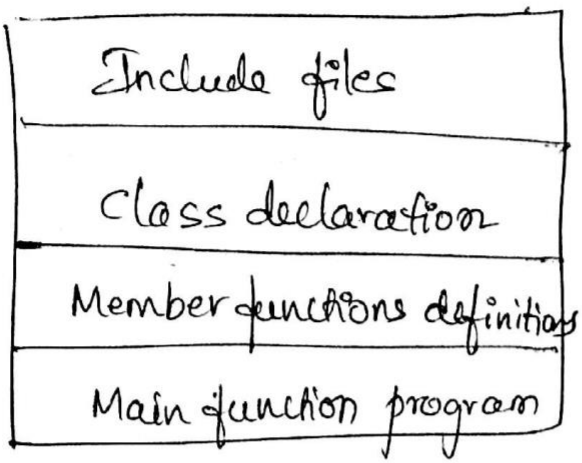
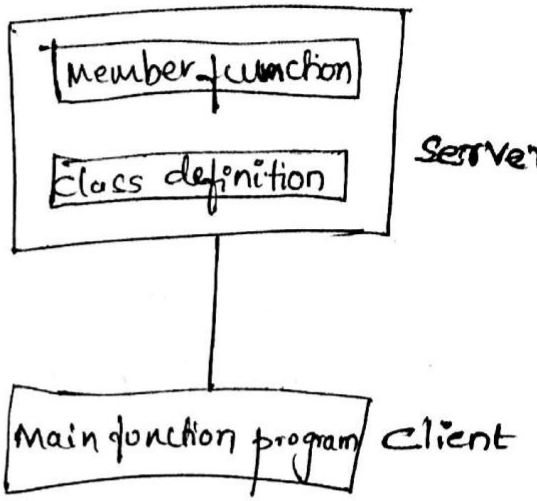


fig: (a) structure of C++ program



fig(b): The client-server model.

→ C++ program contains four sections as shown in fig (a).

→ It is common practice to organize a program into three seperate files

→ The class declarations are placed in a header file and definitions of member functions go into another file.

→ This approach enables the programmer to separate the abstract specification of the interface (class defination) from the implementation details (member function details).

→ finally the main program that uses the class is placed in a third file which "includes" the previous two file as well as any other files required.

→ This approach is based on the concept of client server model as shown in fig(b);

→ The class defination including the member functions constitute the server that provides services to the main program known as client

→ The client uses the server through the public interface of the class.

Keywords;

→ Keywords are reserved identifiers and are explained to the compiler and cannot be used as names for the program variable (or other user defined program elements.

C++ keyword ar as follows: [48 keywords]

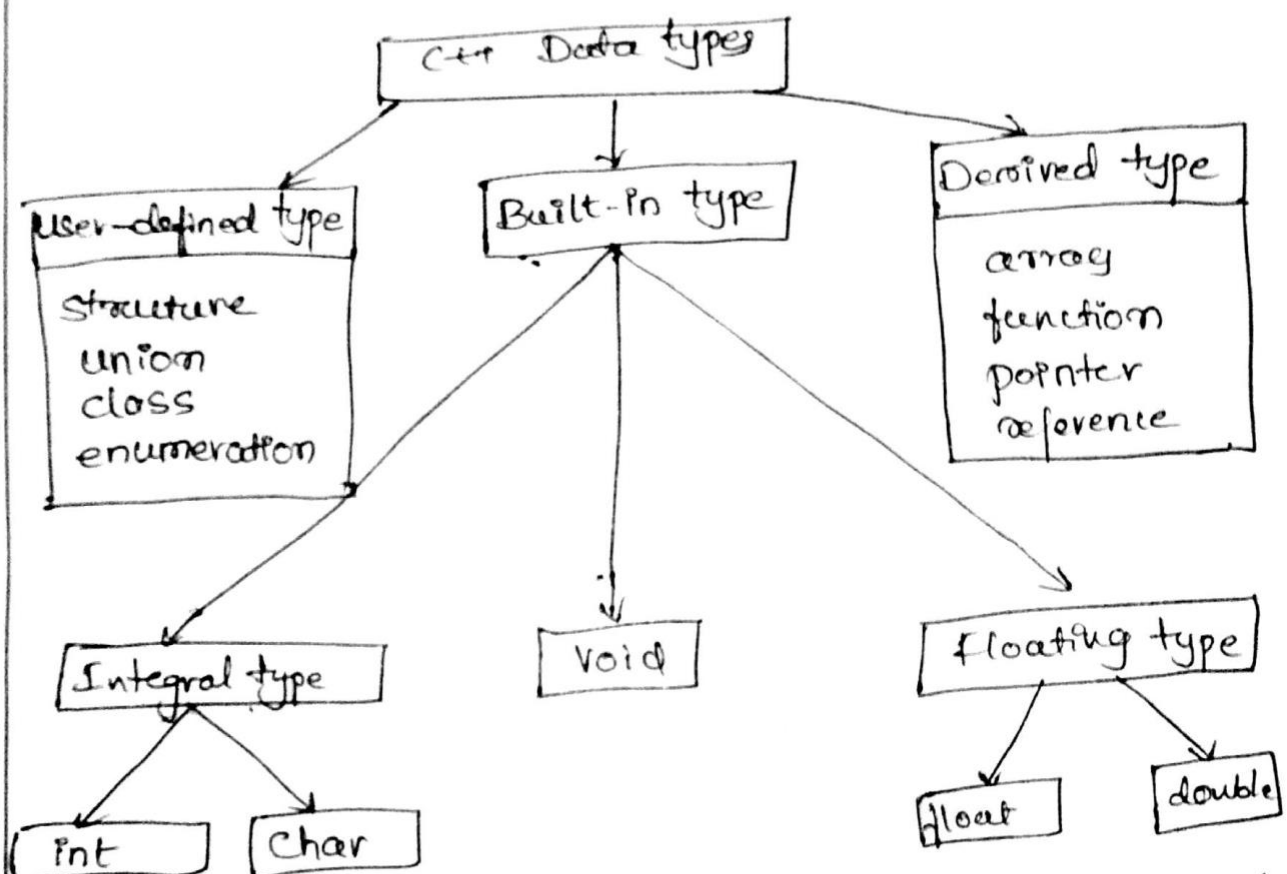
common keywords b/w C and C++;

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

In C++; ~~& otherthan abstract and inherit~~ asm, catch, class, delete, friend, inline, new, operator, private, protected, public template, this, throw, try virtual, bool, export.

Basic Data Types

→ Data types in C++ can be classified under various categories as follows



→ The basic data types may have several modifiers preceding them to serve the needs of various situations.

→ The modifiers **signed**, **unsigned**, **long**, and **short** may be applied to character and integer basic data types.

→ the modifier **long** also be applied to **double**.

→ Lists all combinations of the basic data types and modifiers along with their size and range for a 16-bit word machine.

Type	Bytes
char	1
unsigned char	1
signed char	1
int	2
unsigned int	2
signed int	2
short int	2
unsigned short int	2
signed short int	2
long int	4
signed long int	4
unsigned long int	4
float	4
double	8
long double	10

~~User defined data type~~

Void:

i) to specify the return type of a function when it is not returning any value.

ii) To indicate an empty argument list to a function

Ex: Void function(Void);

It is also used in the declaration of generic pointers

```
void *gp;   // gp becomes generic pointer
```

Ex:
```
int x=10;
char y='c';
void *ptr
ptr = &x;
printf("%d", *(int*) ptr);
ptr = &y;
printf("%c", *(char*)ptr);
```

User defined data types:

Structures:

→ Structures are collection of (or) group of dissimilar data types.

general format of a structure defination is as follows;

```
struct name
{
    data type member1;
    data type member2;
    :
};
```

Ex: book: It has several attributes such as

→ number of pages
→ title
→ price etc:

```
struct book
{
    char title[25];
    char author[25];
    int pages;
    float price;
};
```

struct book book1, book2, book3

where:→ book1, book2 and book3 are structure variable of type book;

→ we can access the member elements of a structure by using dot (.) operator. as shown below,

```
book1.pages = 50
book2.price = 225.75
```

Union

→ These are similar to structures as they allow us to group dissimilar type elements inside a single unit.

→ main difference b/w structure and union is implementation is concerned.

→ The size of structure is sum of the sizes of individual member type.

→ The size of Union is equal to the size of its largest member element.

declaration of Union as follows:

```
Union result
{
    int marks;
    char grade;
    float percentage;
};
```

→ Above union example occupy 4-bytes of memory, because largest element of union is float (of size 4-bytes).

→ Unions are memory-efficient alterations of structures particularly in situations where it is not required to access the different member elements simultaneously.

Difference btw structures and Unions

Structure	Union
1) defined as 'struct' keyword	1) defind as 'union' keyword
2) All members of a structure can be manipulated simultaneously	2) members of a union can be manipulated one at a time
3) Size is equal to the sum of the individual sizes of member objects.	3) Size of union object is equal to the size of largest member object.
4) members are allocated distinct memory locations	4) Common memory space for their exclusive usage.
5) not memory efficient in comparison to unions	5) memory efficient in the situations when the members are not required to be not accessed simultaneously.

Classes

→ Classes are user defined data type used to create ~~the~~ object. ~~for~~.

→ objects are the central focus of object-oriented programming.

Enumerated Data Type

→ It is another user-defined data type, which provides a way for attaching names to numbers, thereby increasing comprehensibility of the code.

→ 'enum' keyword automatically enumerates a list of words by assigning them values 0, 1, 2, and so on.

→ Syntax of an 'enum' statement is similar to that of the 'struct' statement.

→ ~~Syntax~~ Example

```
enum shape { circle, square, triangle };
enum colour { red, blue, green, yellow };
```

* where shape and colour are tag names become new type names.

→ By using tag name we can declare new variable

Eg: `shape ellipse;`
`colour background;`

where 'ellipse' & 'background' are enumerator Variable of type 'shape' and 'colour' respectively.

→ C++ does not permit an 'int' value to be automatically converted to an **enum** value.

[in 'C' defines the types of 'enums' to be 'int']

→ In C++ enumerated data type retain its own separate type.

Ex: Colour background = blue; // allowed
colour background = 7; // error in C++
colour background = (colour) 7; // allowed

→ Enumerated value can be used in place of an 'int' value.

Ex: int c = red // valid, 'colour' type promoted to 'int'

→ By default the enumerators are assigned integer values starting with '0' for the first enumerator, '1' for the second and so on.

Ex: enum colour {red, blue=4, green=8};
where → red is assigned with '0'

Ex: enum colour {red=5, blue, green};
where → blue → 6;
green → 7;

→ C++ also permits the creation of anonymous 'enum' (i.e enum without tag names)

Ex: enum { off, on };

Where → off → 0
on → 1

⇒ Enumeration is used to define symbolic constant for a switch statement.

Ex:
```
enum shape
 { circle;
   rectangle,
   triangle };
int main()
{ cout << "Enter shape code" << endl;
  int code;
  cin >> code;
while ( code >= circle && code <= triangle)
{ switch (code)
  { case circle:
      cout << "Area of circle πr²" << endl
      break;
    case rectangle:
      cout << "Area of rectangle = lbh" << endl;
      break;
    case triangle:
      cout << "Area of triangle = lb" << endl;
      break;
```

```
    }
    cout << "Enter shape code" << endl;
    cin >> code;
  }
  cout << " BYE " << endl;
  return 0;
}
```

Derived data types

Arrays:

→ It is similar to 'C'.

→ The only exception is the way character arrays are initialized. When initializing a character array, the compiler will allow us to declare the array size as the exact length of the string constant

Ex: char string[3] = "xyz"; → is valid in 'C'

→ It assumes that the programmer intends to leave out the null character '\0' in the definition.

In C++: size should be one larger than the number of characters in the string.

char string[4] = "xyz";

Functions: → It is same as 'C' with, function declaration, function definition and function call.

Pointers;

pointers are declared and initialized as in C

Eg: int *ptr;
int x;
* ptr = &x;
*ptr = 10

C++ adds the concept of constant pointer and pointer constant.

char x = 'c';
char * const ptr1 = "xxxxx"; // constant pointer.

→ and we cannot modify the address of 'ptr1' is initialised too.

Eg: int const *ptr2 = &m; // pointer to a constant

ptr2 → is declared as pointer to a constant.

SYMBOLIC CONSTANTS:

→ There are two ways of creating symbolic constant
* Using the qualifier 'const'
* Defining a set of integer constants using 'enum' keyword.

→ Value declared as constant cannot be modified by the program in any way.

Eg: const int size = 10;
char name[size];

→ This would be illegal in C;

→ As with long and short, if we use the const Modifier alone, it default to int.

Ex: const size=10;

means ⇒
const int size=10;

→ The named constants are just like variables except that their values cannot be change.

→ 'const' in c++ defaults to the internal linkage and therefore it is local to the file where it is declared.

→ In 'C' 'const' values are global in nature. they are visible outside the file in which they are declared. It can be made local by declaring them as "static"

→ To give a 'const' value an external linkage so that it can be referenced from other file, we must explicitly define it as an 'extern' in c++

ex: extern const total=100;

other method ⇒ of naming integer constants is by enumerator

ex: enum {X, Y, Z};

⇒ where X, Y and Z are as integer constant with values 0, 1 and 2 respectively. This is equivalent to
const x=0; const z=2;
const Y=1;

We can also assign values to X Y and Z explicitly

Eg: enum { X=100, Y=50, Z=200 };

Declaration of Variables

→ Variable declaration is as 'C'.

→ major difference b/w C and C++ in declaring the variable is,

* In 'C' all variable to be declare (~~de~~) defined at the begining of a scope. where we read C program.

* In 'C++' allows the declaration of a variable anywhere in the scope. This means that a variable can be declared right at the place of its first use. This makes the program much easier to write and reduces the errors.

Eg:

```
int main()
{
    float x; // declaration of variable
    float Sum=0;
    for (int i=1; i<5; i++) // declaration of variable
    {
        cin>>x;
        Sum= sum+x;
    }
    float avg;          // declaration of variable
    avg = sum/(i-1)
    cout<< avg;
    return 0; }
```

DYNAMIC INITIALIZATION OF VARIABLES

[In C Dynamic initialization of Variables are done at the time of compilation]

→ In C++ Dynamic initialization of variable is done at the time of run time, This is called as dynamic initialization

→ A variable can be initialized at run time using expressions at the place of declaration

Example:

```
int n = strlen (string);
float area = 3.14159 * rad * rad
float avg; // declare where it is necessary
avg = sum/i;
```

can be combined into a single statement

```
float avg = sum/i; // initialize dynamically at run time
```

→ Dynamic intialization is extensively used in object-oriented programming.

REFERENCE VARIABLES

→ A reference variable provides an alias (alternative name) for a previously defined variable.

Ex: if we make the variable 'sum' a reference to the variable 'total', the 'sum' and 'total' can be used interchangeably to represent the variable.

Syntax:

```
data-type &reference-name = variable_name
```

Ex:
```
float total = 100;
float &sum = total;
```

⇒ 'total' is of 'float' is already declared: 'sum' is the alternative name declared to represent the variable total.

⇒ both variables refers to the same data object in the memory.

⇒ the statements;

```
cout << total;
```

and

```
cout << sum;
```

both statements will print value of 100.

Ex: total = total + 10;

both 'total' and 'sum' value will be '10'.

Exercise:

if sum = 0

again 'sum' and 'total' value will be '0'.

** A reference variable must be initialized at the time of declaration

** C++ assigns additional meaning to the symbol '&'

⇒ Here '&' is not an address operator. The notation float & means reference to 'float'

Ex:

```
int n[10];
int &x = n[10] // x is alias for n[10]
char &a = 'h' // initalize reference to a literal.
```

Ex:

```
int x;
int *p = &x;
int & m = *p;
```

⇒ where 'm' is to refer to 'x' which is pointed to by the pointer 'p'.

→ Major application of reference variables is in passing arguments to function:

```
Void disp(int &x)   // uses reference variable
{
    x = x+10;       // x is incremented
}
int main()
{
    int m=10;
    disp(m);   // function call.
}
```

when `&x = m`

Hence it is called "call by reference" [where 'x' is alias for 'm']

→ when the function increments 'x', 'm' is also incremented

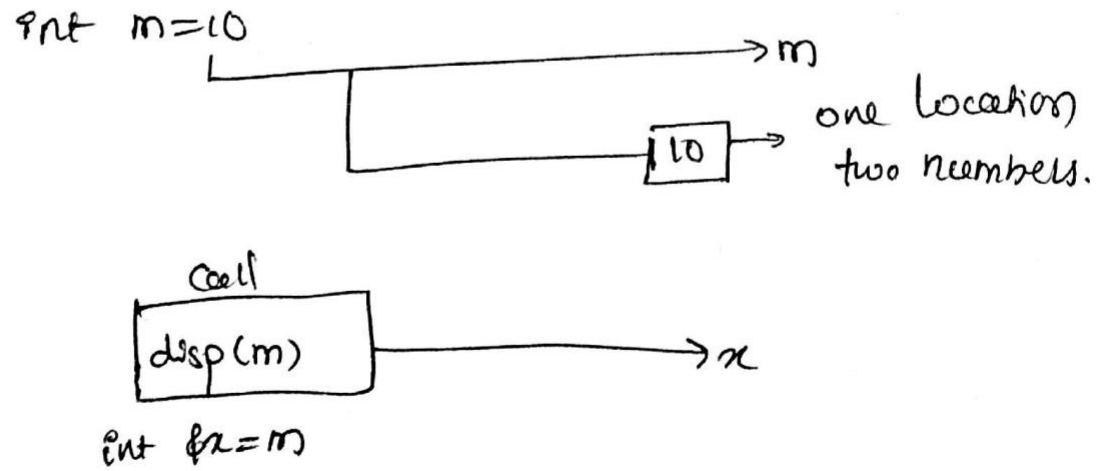


fig. Call by reference.

Operators in C++

→ All 'C' operators are valid in C++.

→ In addition to 'C', C++ introduces some new operators such as

1) :: scope resolution operator
2) ::* pointer to-member declaration
3) ->* pointer to member operator
4) .* pointer to member operator
5) delete memory release operator
6) endl line feed operator
7) new memory allocation operator
8) setw Field width operator

SCOPE RESOLUTION OPERATOR

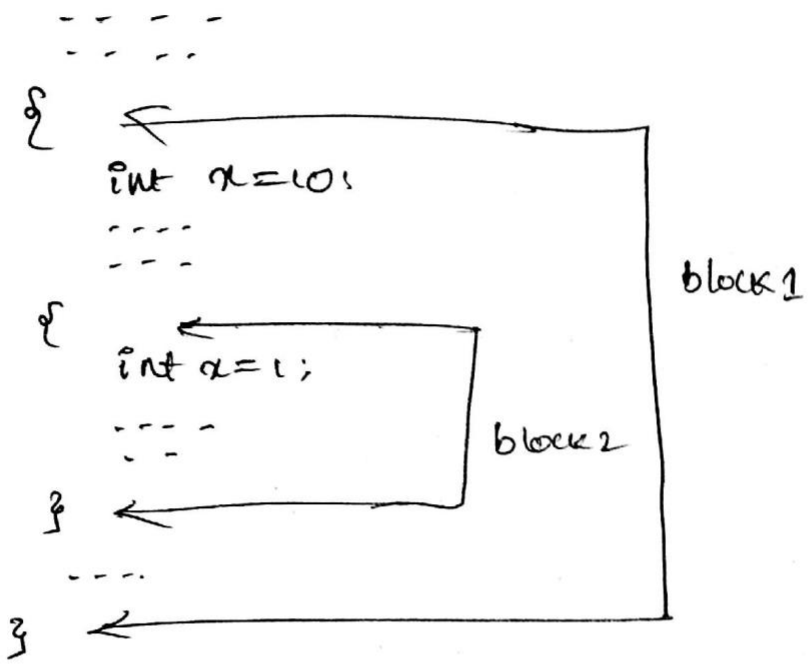
→ C++ is block structure language. Blocks and scops can be used in constructing programs.

→ Same variable name can be used to have different meanings in different blocks.

→ Variable declared inside a block is said to be local to that block.

Ex:
```
{ int x = 10;
  ...
  {
  }
{ int x = 1;
  }
```

→ The two declarations of 'x' refere to two different memory locations containing different values.



→ Block2 contains Block1

→ declaration in an inner blocks hides a declaration of the same variable in an outer blocks.

∴ each declaration of 'x' causes it to refer to a different data object.

→ Global version of a variable cannot be accessed from within the inner blocks.

→ C++ resolves this problem by a new operator "::" called scope resolution operator

Syntax: :: variable name;

* this operator allows access to the global version of a variable

Example:

```
#include <iostream>
using namespace std;
int m=10;  // global m
int main()
{
    int m=20;  // m is redeclared as local to main
    {
        int k=m
        int m=30; // m declared again local to
                  // inner block
        cout<< "we are in inner block m"<<endl
        cout<< " k ="<< k << endl;
        cout << "m = "<< m << endl;
        cout << "::m ="<< ::m <<endl;
    }
    cout << "we are in outer block"<<endl;
    cout << "m ="<< m<< endl;
    cout << "::m="<< ::m <<endl;
return 0;
}
```

MEMORY DEREFERENCING OPERATOR

→ C++ permits us to define a class containing various types of data and function as members.

→ C++ also permits us to access the class members through pointers.

→ To achive this C++ provides a set of three pointer to pointer member operators.

1) ::* To declare a pointer to a member of a class
2) .* To access a member using object name and a pointer to the member.
3) ->* To access a member using a pointer to the object and pointer to that member.

MEMORY MANAGEMENT OPERATORS

→ C++ support memory allocation and deallocation using unary operator **new** and **delete** respectively.

→ object can be created by using **'new'** and destroyed by using **'delete'**, as and when required.

→ a data object created inside a block with new, will remain in existance until it is explicitly destroyed by using **"delete"**

Syntax:

pointer-variable = new data-type;

Here, pointer-variable is a pointer of type data-type.

→ The new operator allocates sufficient memory to hold a data ~~datatype~~ data object of type data-type and returns the address of the object.

→ pointer_variable holds the address of the memory space allocated.

Example

```
p = new int;
q = new float;
```

p → is a pointer of type int
q → is a pointer of type float.

Ex:
```
int *p = new int;
float *q = new float;

*p = 25;
*q = 7.5;
```

→ 25 - is assigned to newly created int.
→ 7.5 to the float object

~~Here value specifies the initial value~~

~~→ We can also Initialise the~~

Initialise memory using new:

Syntax:

pointer_variable = new data-type(value);

Ex:

int *p = new int(25);

float *q = new float(7.5);

→ 'new' can be used to create a memory space for any data type including user defined types such as arrays, structures and classes.

for arrays:

Syntax:

pointer_variable = new data-type[size];

where → size specifies number of array elements

Ex:

int *p = new int[10];

→ create a memory space for an array of 10 integers

P[0] ⟶ first element

P[1] ⟶ second element so, on....

We can create multi-dimensional arrays with new.

Ex: ~~array_ptr = new~~

int *ptr = new int[3][5]; // legal

int *ptr = new int[][5]; // illegal

int *ptr = new int[m][5]; // legal

for classes:

→ using 'new' operator can be used for dynamically creating class object.

→ The size of the class object is automatically determined and the memory space is allocated accordingly.

general form is:

```
class sample
{   private:
        data_type d1;
        data_type d2;
        :
    public:
        display(sample *s);
        :
};
int main()
{
    :
    sample *ptr = new sample;
    cout << display(ptr);
    delete ptr;
}
```

⇒ when data object is no longer needed, it is destroyed to release the memory space for reuse.

Syntax:

```
delete pointer_variable;
```

→ for deleting allocated memory for array

Syntax: delete [size] pointer_variable;

```
int *ptr = new int[10];
delete [10]ptr;
```

Example: : using array :

```
#include <iostream>
using namespace std;
int main()
{
    int *arr;
    int size;
    cout<<" Enter the size of the integer array: "<<endl;
    cin>>size;
    cout<<" create an array of size "<<size<<endl;
    arr = new int[size];
    cout<<" memory allocation is done"<<endl;

    delete [size] arr;
return 0;
}.
```

Using classes: Example:

```
#include<iostream>
using namespace std;
class sample
{  private:
       int data1;
       char data2;
   public:
      void set (int i, char c)
      {  data1 = i;
         data2 = c
      }
      void display (void)
      {  cout<< data1;
         cout<< data2;
      }
};
int main()
{  sample *ptr = new sample
   ptr -> set(25, 'A');
   ptr -> set(26, 'B');
   ptr -> display( );
  delete ptr;
return 0;
}
```

MANIPULATORS

→ Manipulators are operators that are used to formate the data display.

→ Commonly used manipulators are 'endl' and 'setw'.

endl → are used in an o/p statement, causes a linefeed to be inserted. It has same effect as using new line character "\n".

Eg:
```
cout << "m = " << m << endl;
cout << "n = " << n << endl;
cout << "p = " << p << endl;
```

Assume m = 2597, n = 14 and p = 175

m = | 2 | 5 | 9 | 7 | n = | 1 | 4 | p = | 1 | 7 | 5 |

→ Setw: specifies a common field width for all the members.

Eg: `cout << setw(5) << sum << endl;`

setw(5) → Is a manipulator specifies a field width 5 for printing the value of variable sum.

Example:
```
int sum = 345;
```

op : | | 3 | 4 | 5 |

TYPE CAST OPERATOR

→ C++ permits type conversion of variable (or) expressions using the type cast operator.

→ ~~@e cast~~

Syntax:

(type_name) expression // C notation

type_name (expression) // C++ notation.

Example:

average = sum / (float) i ; // C notation

average = sum / float(i); // C++ notation

→ ~~It can be used only if the type is a identifier.~~

Ex: p = int * (P); // is illegal.

p = (int*) P;

→ alternatively we can use 'typedef' to create an identifier of the required type and use it in the functional notation.

Example:

typedef int * ptr;

p = ptr(9);

Expressions and Their types

→ It is a combination of operators, constants and Variables arranged as per the rules of the language.

→ It may include functions calls which return values.

→ An expression may consist of one or more operands, and zero (or) more operators to produce a value.

Expressions may be of the following seven types:

* Constant expressions
* Integral expressions
* float expressions
* pointer expressions
* Relational expressions
* Logical expressions
* Bitwise expressions.

Constant expressions:

→ It consists of only constant values

Ex: 1) 15
2) 20+5/2.0
3) 'x'

Integral expressions:

→ Integral expressions are those which produces integer results after implementing all automatic and explicit type conversions.

Example:

```
int m, n;
m;
m*n - 5
m * 'x'
5 + int(2.0);
```

Float Expressions:

→ Float expressions are those which, after all conversions, produce floating-point results.

Ex:

```
float x, y;
x * y;
x * y / 10;
5 + float(10)
10.75
```

Pointer Expressions:

→ pointer expressions produces address values.

Ex:

```
int *ptr, x = 10;
ptr = &x;
ptr + 1;
```

Relational Expressions

→ Relational expressions yield results of type bool which takes a values 'true' or 'false'

```
int x, y, a, b;
bool val;
if (x <= Y)
    value = TRUE;
```

Logical expression:

→ Logical expressions combine two (or) more relational expressions and produces **bool** type result.

Example:

```
( a>b) && (x==10)
(x==10) || (y==5)
```

Bitwise Expressions:

→ These expressions are used to manipulate data at bit level. They are basically used for **testing** (or) **shifting** bits.

Example:

```
x<<3 ; // shift 3 bit position to left
y>>1 ; // shift 1 bit position to right.
```

→ Shift operator are used for multiplication and Division by powers of two. (Binary)

SPECIAL ASSIGNMENT EXPRESSIONS:

Chained Assignment:

Ex: `x = (y = 10);`

(or)

`x = y = 10`

first y is assigned to 10 and then to x

Note: Chained statements cannot be used to initialize

Ex: float a=b=12.34 // illegal

this can be written as

float a=12.34, b=12.34; // Correct

Embedded Assignment

Ex: x=(y=50)+10;

where (y=50) is an assignment expression known as embedded assignment, Here the value 50 is assigned to 'y' and then the result 50+10=60 is assigned to 'x'. This statement is identical to

```
y=50;
x=y+10;
```

Compound Assignment:

~~→ C++ supports a compound expression known as embedded assignment.~~

→ C++ supports a compound assignment operator which is a combination of the assignment operator with a binary arithmetic operator.

Ex: x=x+10;

may be written as.

x+=10;

OPERATOR OVERLOADING

→ Overloading means assigning different meanings to an operation, depending on the context.

→ C++ permits overloading of operators, thus allows us to design multiple meaning to operators.

ex:① * the operator * when applied to a pointer variable gives the value pointed to by the pointer.

→* But it is also commonly used for multiplying two numbers.

* The number and type of operands decides the nature of operation to follow.

ex② * I/p & O/p operator << and >> are examples for operator overloading.

* where << can also used for shifting of bits to left and also used for displaying the values of various data types.

* where the member of overloading of << operator is

```
cout << 75.86 ; // for invoking for displaying a double type value
cout << "well done"; // invokes for displaying a char value
```

→ Hence C++ operators can be overloaded with a few exceptions such as the member-access operators (. and .*), conditional operator (?:), scope resolution operator (::) and size of operator (sizeof)

operator precedence

Note: Precedence is from top to bottom
⇒ top → highest, bottom → lowest priority.

Operators	Associativity
::	Left to right
-> . () [] postfix ++, --	Left to right
prefix ++, prefix --, ~, ! ~~*~~, &, sizeof, new, delete	right to left
->* *	left to Right
* / %	L to R
+ -	L to R
<< >>	L to R
<, <=, >>=,	L to R
==, !=	L to R
&	L to R
^	L to R
|	L to R
&&	L to R
||	L to R
?:	L to R
=, *=, /=, %=, +=, =	Right to left
<<=, >>=, &=, ^=, |=	~~L to Right~~ left to Right
, (comma)	

Control structure:

→ C++ a large number of functions are used that pass message, and process the data contained in objects

→ Experience has also shown that the number of bugs that occur is related to the format of the program.

→ The format should be such that it is easy to trace the flow of execution of statements.

→ This will help not only in debugging but also in the review and maintainance of the program later.

→ one method of achieving the object of an accurate, error-resistant and maintainable code is to use one or any combination of the following three control structures:

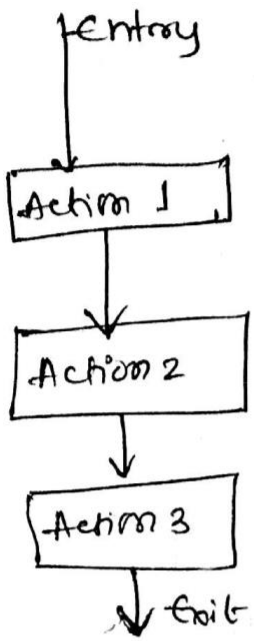
i) Sequence structure (straight line)
ii) Selection structure (branching)
iii) Loop structure (iteration or repetition)

→ ~~It is~~

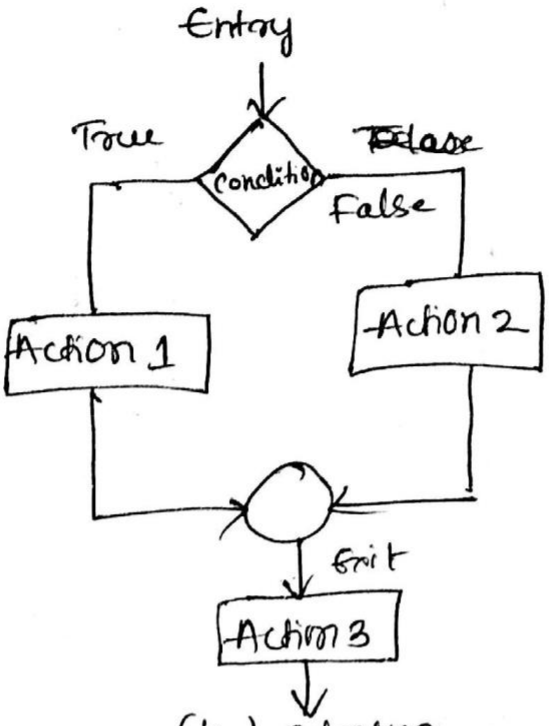
→ It is important to understant that all program processing can be coded by using only these three logic structure.

→ The approach of using one (or) more of these basic control constructs in programming is known as structure programming.

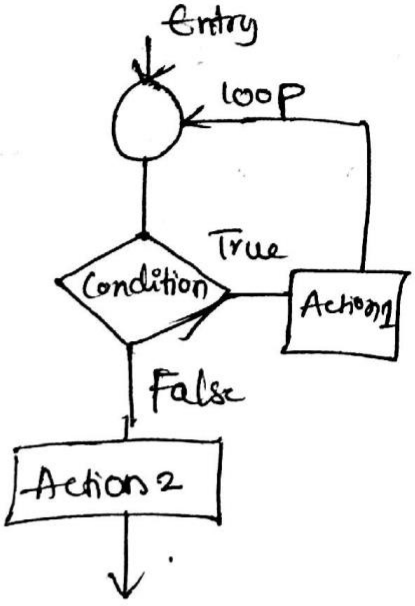
⇒ figure below fig(a) shows how these structures are implemented using one-entry, one exit concept, a popular approach used in modular programming.



(a) Sequence



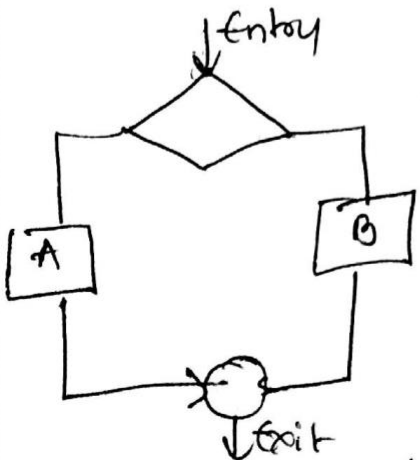
(b) Selection



(c) loop

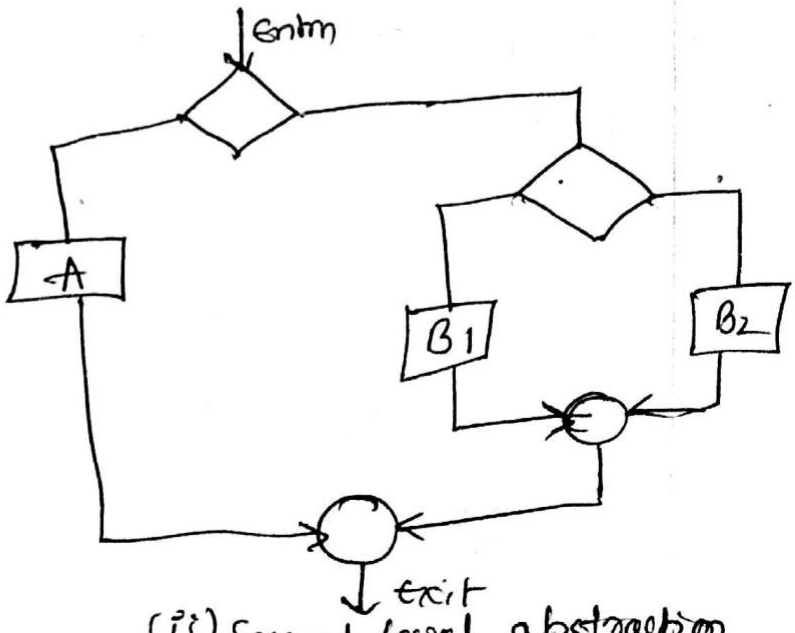
fig(a): Basic control structures

→ From fig(b) shows a function structure either in detail (or) in summary. using these three basic structures.

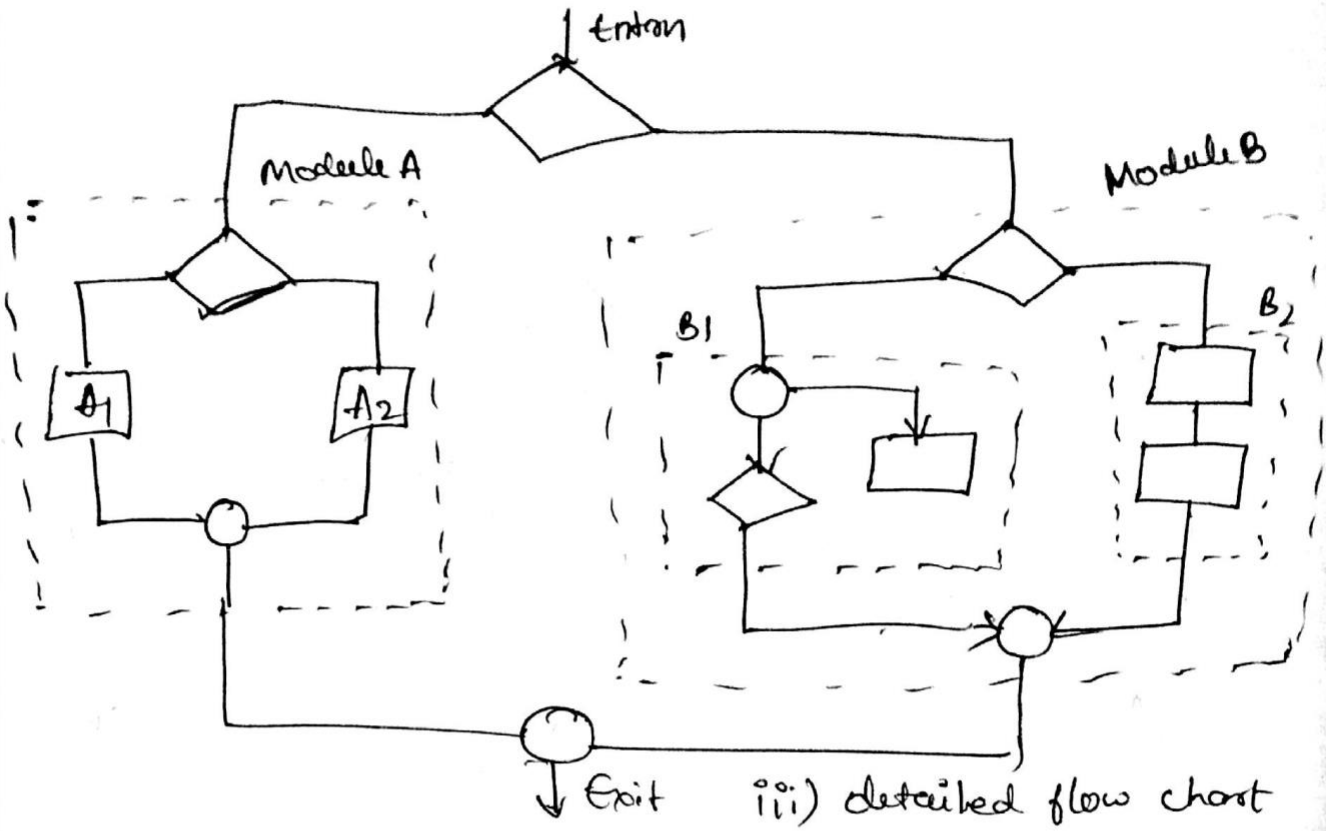


i) first level abstraction

fig(b)



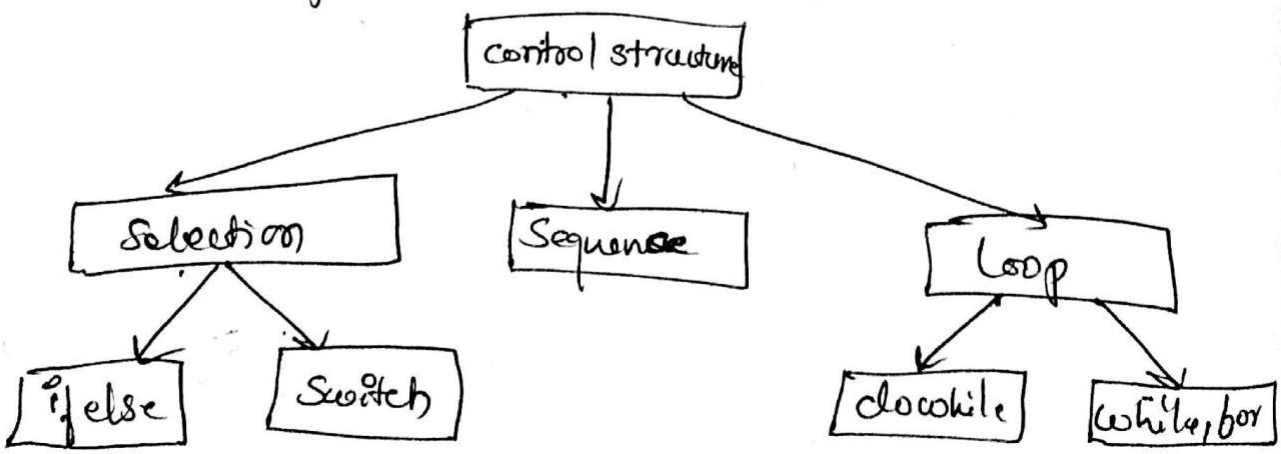
(ii) second level abstraction



fig(b) Different levels of abstraction.

→ like in C & C++ also supports all three basic control structures and implemented them using various control statement as shown in fig (c).

→ Hence c++ combines the power of structure programming with the object-oriented paradigm.



The if Statement

→ if statement Statement is implemented in two forms:

* simple if statement
* if...else statement

Eg: for if statement

```
if (expression is true)
{
  action1;
}
action2;
action3;
```

Eg: for if-else

```
if (expression is true)
{
  action1;
}
else
{
  action2;
}
action3;
```

The Switch Statement

→ This is multiple-branching statement, based on a condition, the control is transfered to one of the many possible points.

→ It is implemented as

```
Switch (expression)
{
   Case 1:
     { action1;
     } break;

   Case 2:
     {
        action2:
        break;
     }
   Case3:
      {
         action3;
         break;
      }
   default:
     {
         action4;

      }
}
action5;
```

The while statement:

This is a loop structure, but is an entry-controlled one, The Syntax is as follows:

```
initialization;
  while(condition is true)
{
     action1;

  increment:
}
action2;
```

The do-while statement

→ do-while is an exit-controlled loop. Based on a condition, the control is transferred back to a particular point in the program.

The syntax is as follows:

```
initialization;
do
{ action1;
} while (condition is true);
action2;
```

For statement:

The 'for' is an entry-controlled loop and is used when an action is to be repeated for a number of times.

Syntax is:

```
for (initialvalue; test; increment)
{ action1;
}
action2;
```

Programming Exercises

① Write a function using reference variables as arguments to swap the values of a pair of integers.

program:

```
#include<iostream>
Void swap_fun(int &a, int &b)
{ cout<< "Before swapping"<<endl;
  cout<< "a= "<<a<<endl<<"b="<<b<<endl;
  int temp;
  temp=a;
  a= b ;
  b= temp;
 cout<< "After swapping"<<endl;
 cout<<"a="<<a<<endl<< "b="<<b<<endl;
}

int main()
{
   int x, y;
   cout<<" Enter the two integer values : "<<endl;
   Cin>>x>>y;
   swap_fun(x, y);
   return 0;
}
```