Mid-Sem-2018

Q1) int a = 10 + 20 / 30 % 4 - 5 && 7 != 2;

2(a) 1(a) 1(b) 2(b) 4 3

Here /, % has highest precedance i.e 1
+, - " next " " i.e 2
!= " " " " 3
&& " " " # 4.

Associativity of /, % is left to right
So 1a will be executed first then 1b.
Similarly 2a " " " " " 2b.

10 + 20/30 % 4 - 5 && 7 != 2;

10 + 0 % 4 - 5 && 7 != 2;

10 + 0 - 5 && 7 != 2;

10 - 5 && 7 != 2;

5 && 7 != 2;

This is True, so o/p = 1

5 && 1

Both r non zero, so o/p = 1

1

Q2. a = 10, b = 4;
bool res = (a == b) && cout << "Hello";
cout << res;

1) What does cout return?
Depends on state of output stream
If stream is in bad state, then
cout returns - NULL - ** NULL is considered as 0
Otherwise cout returns - void * pointer

The value is not specified, but
You can consider it as non
zero.

So cout can return 2 values :-
1) NULL - wh. is 0
2) void * - " " non zero.

2) In a logical expression, if 1st part
is 0, then second part is not
evaluated.

(a == b) && cout << "Hello"

This part is 0 — So this part will not be evaluated
So, Hello is not
printed.

cout << "Hello" && (a == b)

"Hello" will be
printed — This part is
0
This part is not 0
so second part will
be evaluated

3) How does && work?

ans) ( ) && ( )

If any part is 0 then o/p will be 0.

Now: a = 10, b = 4

bool res = (a == 6) && cout << "Hello"

This part is 0 — not evaluated

So o/p will be 0. or false

Cout << res;

// o/p = 0

// "Hello" is not printed

Q3 int a[] = {10, 15, 20, 25, 30};

int *p1, *p2;

p1 = a

p2 = &a[3];

cout << p2 - p1;

A • Difference of two pointer is the No. of el. b/w them + 1.

a or &a[0] &a[1] &a[2] &a[3]

| 10 | 15 | 20 | 25 | 30 |

↑ p1 ↑ p2

So p2 - p1 = 3.

Q4
```
void show(int a, int b=10)
{
    cout<<a+b;
}

int main()
{
    show(20,30); // 50
    show(20); // 30
}
```

This is called a default argument.

→ def. argument is ignored

→ def. argument is considered as 2nd param.

Answer If you do not provide second parameter then default argument will be taken by compiler.

o/p of show(20); will be 30 bcoz value of a=20, & b=10.

o/p of show(20,30) will be 50 bcoz value of a=20, b=30

So if you provide second argument/parameter then default argument will be ignored by compiler.

Q5 int n = 20; // Global n, written
int main()
{
n = 40;
cout << n // o/p = 40
cout << ::n; // o/p = 20
{ int n = 60;
cout << n; // o/p = 60
cout << ::n; // o/p = 20
}
}

A ::n will always print global n.
n will print the most local n.

o/p = 40
20
60
20.

Q6) Class vs struct

Q7) const keyword.

Q8 Manipulators
eg endl, setw, setfill(*)

Q9 insertion & ext^n.

Q10 Type casting

Q11 Imp of destructor?
A ~ can be used to perform any task that need to be done whenever obj is destroyed.

• Most common use of d is to perform file handling related tasks.
Sometimes User may forget to store same a file opened by an obj. In that case, when obj is destroyed, d will be automatically called & it will save that file.

Sec-B

Q3 a) OOP vs POP

b) Inline fun
• Fun with has inline keyword in the fun declaration.
• ~~A normal fun~~
• During a normal fun call, the program jumps from one m/m loc to another. So there is some o/head which decreases efficiency.
• So inline fun are needed to reduce this o/h. Inline fun are expanded not called, with means, the code of inline fun is copied in place of fun call.

eg
```
inline add (int a, int b)
{
    int sum;
    sum = a+b;
    cout<<sum;
}

int main()
{
    int a=10, b=20;
    add(a,b); // This fun call will be expanded
}
```

c) class flight
{ private:
int flight#;
String dest;
float distance;
float fuel;
void calfuel ( );

public:
void feedinfo ( )
void showinfo ( )

};

void student :: calfuel ( )
{
if (distance <= 1000)
fuel = 500;
if (distance > 1000 && distance <= 2000)
fuel = 1100;
if (distance > 2000)
fuel = 2000;
}

void student :: feedinfo ( )
{
~~cin~~ ~~int a, flight#, d~~ ; int x;
~~String~~ ; String y;
~~float distance~~ ; float z;
~~cin >> a >> s >> d;~~
~~flight #~~
cin >> x >> y >> z;
flight # = x;
dest = s;
distance = z;
calfuel ( );
}

```
Void student :: showinfo( )
{
    cout << flight# << dest << distance << fuel;
}
```

Q4 a) time class.

```
class Time
{
    int h, m, s;
    public:
     void set ( ); // read values by
                   // input - so no param.
    Void get ( ); // print
    void add_obj (Time P01,
                  Time P02 );
};

Void Time :: set ( )
{   int x, y, z;
    cin >> x >> y >> z;
     h = x.
     m = y;
     s = z;
}
Void Time :: get ( )
{   Void<<
    if ( h < 10 )
    {   cout << 0 << h << ":";
    }
    else  cout << h << ":";
    if ( m < 10)
          cout << 0 << m << ":";
    else  cout << m << ":";
```

```
if (s<10)
    cout<<0<<s;
else cout<<s;
}

void Time :: add_obj (Time po1,
                      Time po2)
{   s = po1.s + po2.s;
    m = s/60;
    s = s%60;
    m = m + po1.m + po2.m;
    h = m/60;
    m = m%60;
    h = h + po1.h + po2.h;
}

int main()
{
    Time obj1, obj2, obj3;
    obj1.set();  // No param, values i/p
                 // by user
    obj2.set();
    obj3.add_obj(obj1, obj2); // add to obj3
    obj3.get(); // Print obj3.
```

Q5 CBV vs CBA vs CBR - See notes.

Q6) Constructor - A fun wh. is called auto when an obj is created.
It can perform any task at the time of obj creation. It has same name as class.

Properties - 1) Same names as class
2) No return type
3) Can be parameterized or non-P
4) Called automatically.
5) Called only at time of obj creation
6) Called only once.

Types

1) Parameterized Constructor

```
Student :: Student (int x, int y, int z)
{
    rn = x;
    m1 = y;
    m2 = z;
}
```

Used to create obj like this
Student obj1 (1, 11, 11);

2) Non parameterized const.

```
Stud :: St ( )
{
    // Nothing here.
}
```

Used to create obj like this
Student obj1;

Q5. (a) static data member - see notes.
static member fun - used to access static members in a class.
Static fun can't access a non static member
Declared like this,
static void set ( );

WAP to count No. of obj - see notes.

...

b) double power(double m, int n = 2)
// def. argument is 2
{
    double result = 1;
    for(i=0; i<n; i++)
    {
        result = result * a;
    }
    return result;
}

int readvalues(int &x, int &y)
{
    int choice;
    cout<<"How many values you want to enter";
    cin>>choice;
    if(choice == 1) cin>>x;
    if(choice == 2) cin>>x>>y;
    return choice
}

int main()
{
    int choice, m, n = 0;
    ~~cout<<"enter values~~
    choice = readvalues(m, n);
    if(choice == 1)
        cout<<power(m); // 2nd argument is not provided, So def argument will be taken as 2nd arg.
    if(choice == 2)
        cout<<power(m, n);

Here both arguments r provided so def. argument is ignored.

}

• Const means, it can't be changed

1) int const *p;

↓

Here *p is int as well as const

↓

Here *p is const, but p is not const

↓

You can't change *p, but you can change p.

Eg
```
int a=10, b=20;
int const *p = &a;
*p = 50; // Not allowed
p = &b; // Allowed.
```

2) int * const p;

↓

Here p is const, but *p is not const.

↓

You can't change p, but *p can be changed.

Eg
```
int a=10; b=20;
int const *p = &a;
*p = 50 // Allowed
p = &b // Not allowed.
```

③ int const * const p;

Here p & *p both are const.
You can't change p & *p;

eg int a = 10, b = 20;
int const * const p = &a;
*p = 50; // Not allowed
p = &b; // Not allowed.

- Other ways to write these

①	②	③
int const *p; const int *p; const int * const *p; int const const *p; ↓ *p is const or *p is read only.	int * const p; ↓ p is const or p is read only	int const * const p; const int * const p; ↓ Both *p & p are const or both r read only.

C++ insertion(<<) and extraction(>>) Operators

The insertion and extraction operators are used to write information to or read information to or read information from, respectively, ostream and istream objects.

By default, white space is skipped when the insertion and extraction operators are used.

The insertion operator << (Common Output - cout) :

The insertion operator (<<) points to the ostream object wherein the information is inserted. The normal associativity of the << -operator remains unaltered, so when a statement like :

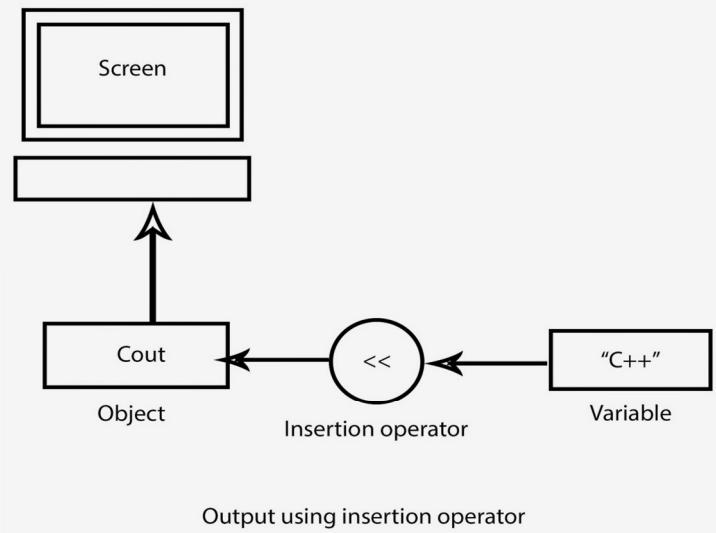
cout << "hello" << "world";

is encountered, the leftmost two operands are evaluated first (cout << hello).

cout is an ostream object, into which information can be inserted. This stream is normally connected to the screen.

cout << is used for displaying the message or a result of an expression.

It is same as of printf() function used in c language.



Output using insertion operator

Syntax :

cout <<"Message to print" or result to be show

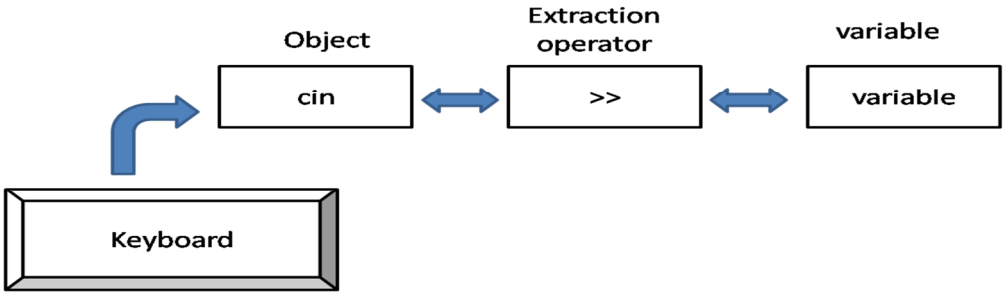
Eg : cout << "Welcome to c++ ";

The extraction operator >> (Common Input- cin) :

cin is an istream object from which information can be extracted. This stream is normally connected to the keyboard.

cin is used to insert the information or the value of a variable.

This works same as scanf() works in C language.



Syntax :

cin >> variable value;

Eg : cin >> x;

where x is a variable.

iostream.h header file :

To use cin >> and cout << we have to include the "iostream.h" header file in our program. This header file contains the definition (prototype) of both cout and cin.

Features	Structure	Class
Definition	A structure is a grouping of variables of various data types referenced by the same name.	In C++, a class is defined as a collection of related variables and functions contained within a single structure.
Basic	If no access specifier is specified, all members are set to 'public'.	If no access specifier is defined, all members are set to 'private'.

Declaration	`struct structure_name{`
	`type struct_member 1;`
	`type struct_member 2;`
	`type struct_member 3;`
	`.`
	`type struct_memberN;`
	`};`
		`class class_name{`
		`data member;`
		`member function;`
		`};`

Instance	Structure instance is called the 'structure variable'.	A class instance is called 'object'.
Inheritance	It does not support inheritance.	It supports inheritance.

Memory Allocated	Memory is allocated on the stack.	Memory is allocated on the heap.
Nature	Value Type	Reference Type
Purpose	Grouping of data	Data abstraction and further inheritance.
Usage	It is used for smaller amounts of data.	It is used for a huge amount of data.
Null values	Not possible	It may have null values.
Requires constructor and destructor	It may have only parameterized constructor.	It may have all the types of constructors and destructors.

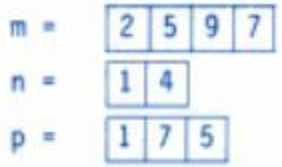
3.17 Manipulators

Manipulators are operators that are used to format the data display. The most commonly used manipulators are **endl** and **setw**.

The **endl** manipulator, when used in an output statement, causes a linefeed to be inserted. It has the same effect as using the newline character "\n". For example, the statement

```
.....
.....
cout << "m = " << m << endl
     << "n = " << n << endl
     << "p = " << p << endl;
.....
.....
```

would cause three lines of output, one for each variable. If we assume the values of the variables as 2597, 14, and 175 respectively, the output will appear as follows:



It is important to note that this form is not the ideal output. It should rather appear as under:

```
m =  2597
n =    14
p =   175
```

Here, the numbers are *right-justified*. This form of output is possible only if we can specify a common field width for all the numbers and force them to be printed right-justified. The **setw** manipulator does this job. It is used as follows:

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

The manipulator **setw(5)** specifies a field width 5 for printing the value of the variable sum. This value is right-justified within the field as shown below:

Program 3.2 illustrates the use of **endl** and **setw**.

USE OF MANIPULATORS

Program 3.2 illustrates the use of **endl** and **setw**.

USE OF MANIPULATORS

```
#include <iostream>
#include <iomanip>   // for setw

using namespace std;

int main()
{
    int Basic = 950, Allowance = 95, Total = 1045;

    cout  << setw(10) << "Basic" << setw(10) << Basic << endl
          << setw(10) << "Allowance" << setw(10) << Allowance << endl
          << setw(10) << "Total" << setw(10) << Total << endl;

    return 0;
}
```

PROGRAM 3.2

Output of this program is given below:

```
     Basic    950
Allowance      95
     Total   1045
```

note

Character strings are also printed right-justified.

We can also write our own manipulators as follows:

```
#include <iostream>
ostream & symbol(ostream & output)
{
   return output << "\tRs";
}
```

The **symbol** is the new manipulator which represents **Rs**.The identifier **symbol** can be used whenever we need to display the string **Rs**.

Procedural Oriented Programming	Object Oriented Programming
In procedural programming, program is divided into small parts called ***functions***.	In object oriented programming, program is divided into small parts called ***objects***.
Procedural programming follows ***top down approach***.	Object oriented programming follows ***bottom up approach***.
There is no access specifier in procedural programming.	Object oriented programming have access specifiers like private,

Adding new data and function is not easy.	Adding new data and function is easy.
Procedural programming does not have any proper way for hiding data so it is ***less secure***.	Object oriented programming provides data hiding so it is ***more secure***.
In procedural programming, overloading is not possible.	Overloading is possible in object oriented programming.
In procedural programming, function is more important than data.	In object oriented programming, data is more important than function.
Procedural programming is based on ***unreal world***.	Object oriented programming is based on

Procedural programming is based on ***unreal world***.	Object oriented programming is based on ***real world***.
Examples: C, FORTRAN, Pascal, Basic etc.	Examples: C++, Java, Python, C# etc.

3.18 Type Cast Operator

C++ permits explicit type conversion of variables or expressions using the type cast operator.

Traditional C casts are augmented in C++ by a function-call notation as a syntactic alternative. The following two versions are equivalent:

```
(type-name) expression  // C notation
type-name (expression)  // C++ notation
```

Examples:

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

A type-name behaves as if it is a function for converting values to a designated type. The function-call notation usually leads to simplest expressions. However, it can be used only if the type is an identifier. For example,

```
p = int * (q);
```

is illegal. In such cases, we must use C type notation.

```
p = (int *) q;
```

Alternatively, we can use **typedef** to create an identifier of the required type and use it in the functional notation.

```
typedef int * int_pt;
p = int_pt(q);
```

ANSI C++ adds the following new cast operators:

- const_cast
- static_cast
- dynamic_cast
- reinterpret_cast

Application of these operators is discussed in Chapter 16.