End-Sem - 2018

Q1 #define x 5+2

```
main()
{   int i;
    i = x * x * x;
    cout << i;
}
```

#define x 5+2 is macro, wh. means x will be replaced by 5+2 everywhere in program.

So i = x * x * x; becomes.

= 5+2 * 5+2 * 5+2;

= 5 + 10 + 10 + 2

= 27

∴ o/p = 27.

Q2

```
int a = 8
cout << (a >> 3);
```

>> is a rightshift operator by
a >> n means rightshift a, n times
or divide a by 2^n.

a << n is a leftshift operator.
a << n means multiply a by 2^n

∴ a >> 3

= a / 2^3

= 8 / 8

= 1.

Q3 Size of any pointer is 8 Bytes.

char *p;
int *p;
float *p;
} all will store address in 8B.

char *p;

sizeof(*p) = size of data type pointed by p
= Size of char
= 1B.

sizeof(p) = size of ptr
= 8B.

cout << sizeof(*p) << sizeof(p);
↓ 1B ↓ 8B.

Q4 Class Test ... (EXPLANATION)

1) Size of an obj of a class = size of all variables inside that obj.
Any var inside fun r not included

Eg class Test
{
int a; //4B
int b; //4B.
void set(int x, int y, int z) //Not included.
};

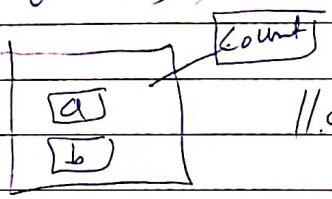
Test obj1;
sizeof(obj1) // sizeof obj1 will be 8B.

2) Size of obj with static var?

- Static var not defined inside obj. They r outside obj, so size of static var is not included in obj.

```
Class Test
{
    static int count; //Not include
    int a; //4
    int b; //4

}
Test Obj1;
Size of (obj1); // = 8B
```



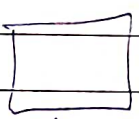
//Static var is //outside class

obj1

3) Size of an empty ~~class~~ class obj?

Size of a ~~class~~ obj with no var. is 1B. It is ~~determined~~ allocated 1B by compiler., so that compiler can distinguish b/w diffent obj

```
Class Test
{
    //empty class
}
Test obj1;
Size of (obj1); // = 1B
                // No var in obj1
```



obj1

Q 4) Class Test
{
static int a;
}
Test obj1;
sizeof(obj1); //=1B

// a is static, so it is not defined inside obj.
// The obj is empty. So size will be 1B

Q 5) ar[3]
can be written as 3[a]

cout<<a[3] << 3[a];
↓ ↓
40 40

b) 5 new operators :-

:: → scope resolution operator
→ 3 uses
1) To access a class member
2) To access a global var
3) To " " namespace

new → for dynamic m/m alloc.

delete → to ~~free~~ release m/m

endl → to print new line

<< → to send o/p to screen.

ii) "Using namespace std"
It means std namespace will be used by default before all the objects.

For eg
```
std::cout<<a;
std::cin>>a;
std::cout>>b;
```
↓
No need to write this compiler will provide this before all cout & cin.

c) i) oop provide several features to overcome the shortcomings ----

ii) Static vs Dy- m/m alloc.

Q2) a)(i) Order of const.

```
class date
{

};
class time
{

};
class train
{
    date ddate; -(3)
    time dtime; -(4)
};
main()
{
    date d1; //-(1)
    time t1; //-(2)
    train tr1; //-(5)
}
```

There 2 const r called first

when this obj. is created

(5)

(ii) Date D2 = D1;
// Here, at D1 is copied to D2 at the time of creation of D2
So ~~SC~~ copy constructor will be called. & ~~obj~~ D1 will be copied to D2.

eg 2: Date D2; // D2 is created
D2 = D1; // D1 is copied to D2.

// Here D1 is copied to D2, but not at the time of creation of D2.
// D2 is created separately & D1 is copied to D2 separately.
// So in this case "Copy Const" will NOT BE CALLED.

(iii) Reusability is done thru inheritence

(iv) Access specifies
1) Private - private members can't be accessed by obj

2) Public - pub m/bs can be accessed by obj

3) Protected - inheritence

v). • A copy const is a user defined fun.
• It will copy one obj to another.
• In this fun obj is passed by ref, not by value.

eg. Student :: student (stu & P01)
{
// fun def here
}
// obj passed by ref

// Now if we do this inside main:-
Student obj2 = obj1;

Here, copy const is called & obj1 is the passing obj.

• We know, that if we pass an obj to a fun, then CC will be called.
• If we pass obj by value to CC itself, then CC will be called two times. To avoid this, parameters r passed by ref to a CC.

add_obj ((Student P01, Student P02))
{
// Def
}
// Call.
obj3.add_obj (obj1, obj2);

obj1, obj2 copied to P01, P02 so CC will be called.

Student (student P01)
{
// fn CC
}

obj1 is copied to P01 so CC is called again - (2)

// Call CC
student obj2 = obj1;

CC is called here - (1)

// CC called 2 times here.
// To avoid, use obj by reference

Q2 b) Toll Booth ---

```
Class TB
{
    unsigned int TtCar;
    Double Ttamt;
    Public:
    TB ( );
    Veid payingCar ( );
    Veid NonPayCar ( );
    Veid disp ( );
};

TB::TB ( )
{ //Define Const.
  //It will initialize TtCar, Ttamt to 0;
   TtCar = 0;
   Ttamt = 0;
}
Veid TB:: payingCar ( )
{   //Inc Tt car & Tt amt.
     TtCar ++;
     Tt amt = Tt.amt +50;
}
Veid TB:: Nonpay Car ( )
{   //inc only Tt Car.
     TtCar++;
}
Veid TB:: disp ( )
{     cout << TtCar << Tt amt;
}
```

```
int main()
{   TB obj1;
    Char refeat = 'y';
    Char choice;
    while (refeat == 'y')
    {
        Cout << "enter choice"
             << "P : Paying Car has
                    paned the Toll"
             << "N : Non paying Car paned"
             << "D" : Disfplay values.
             << "E" : exit;

        switch (choice)
        {
            Case 'P': obj1.PayingCar();
                      break;
            Case 'N': obj1.Nonpay Car();
                      brnk;
            Case 'D' : obj1.disp()
                       b;
            Case 'E' : refeat = 'N';
                       brnk;
        }
    }
}
```

Q2 c) • Const vs Dest - Google
• WAP to overload a const <see notes>

Q3 (a)(i) Case-1: Friend fun is friend of two classes.

Q3 (a) (i) A friend fun (FF) can be used in several ways.

1) It can be a member fun (MF) of 1 class & FF of several other classes.
2) It can be a MF of two class & FF of several other classes.
3) However, it can't be a MF of more than 1 class. It is not allowed

Case-1 A FF is a friend of 2 classes.

- In this case, it ~~must~~ is not a MF of any class.
- It must be declared in both classes as a friend fun.
- It must be defined only once after all classes.
- It must be called w/o using any obj.

eg Class 1; // forward declaration
Class 2;

```
Class 1
{
    =
    friend void FF(---); //declare FF
                          in Class-1
};
// Define MF of Class-1
// Do not define FF here.
```

class - 2
{
=
friend void FF(---); // declare FF in class-2
}

// Define MF of class-2

// Finally define FF, at last, only once.

void FF(---) // Do not use any class name b4 FF
{
// Define FF
=
=
}

main()
{
Obj1, obj2 ; // Create obj1, obj2 of different classes.

FF(obj1, obj2); // call FF w/o using any obj.
// Pass obj of different classes.

Case-2 A FF is a friend of 1 class and MF of another class

- Inside class-1 declare it as MF
- " class-2 " " " FF but also specify the class-1 *** // see eg. below in the declaration
- Define it after all the classes, at last, only once
- Call it using obj of class-1.

```
Class-1 // forward declaration
Class-2 //  "       "
Class-1
{
    =
    =
    void M & FF( - - );// declare as a
                          MF of class-1.
}
// Define fun of class-1 except M & FF.
  =
  =

Class-2
{
    =
    =
    ~~void~~                    ***
    friend void class-1 :: M & FF( - - - )
    // when declaring inside class-2 as
    FF, the name of class-1 is also
    specified.
}
// Define fun of class-2.
```

// finally at last define the MFF

```
Void class-1:: MFF(---)
{
    // define fun
    =
}

int main()
{
    obj1; // create obj of 2 diff
    obj2; // classes

    obj1.MFF(obj2); //
    // called using obj of class-1
    // bcoz it is a MF of class-1.
```

Case-3 Friend class.

1) One class can be a friend of another class.
2) This class will have access to all private/public members of other class.
3) It doesn't inherit the other class. It only has access to private var of other class.

eg

```
class-1
{
    =
    friend class class-2; // class-2 is
                          a friend of
                          class-1
}
```

class - 2
{
=
void fun(class-1 obj);
}
void fun(class-1 obj)
{
// This fun of class-2 can
access members of class-1
}

Q3) (a) (ii) WAP swap data members of 2 diff classes.

A. Friend fun should be used.

```
class B;
class A
{
    int a;
    public:
    friend void swap(class A &P01, B &P02);
    //                obj of class A   obj of class B
// swap is done by call by reference, so
   use & before obj name.
};

class B
{
    int b;
    public:
    friend void swap(A &P01, B &P02);
};
```

```
//Define the swap fun.

void swap (A &P01, B &P02
{
    int temp;
    temp = ~~A.a~~ P01.a;
    P01.a = P02.b;
    P02.b = temp;

}

int main ()
{
    A obj1;
    B obj2;
    swap (obj1, obj2);
    //Call swap FF w/o any obj
    //Pass obj of 2 diff classes.

}
```

~~Q3 b) Question~~

Q3 b) Correction in this question
=> and << can't be overloaded by a MF. So overload >> and << using FF
and overload ++, --, - using MF

```
class loc
{
    int long;
    int lat;
    public
    void set (int x, int y);
    loc
    // declare ++, --, - as MF
    loc operator++ ( );
    loc operator-- ( );
    loc operator- (loc P01);

    // declare >> & << as FF
    friend ostream & operator<<(ostream& P01,
                                loc& P02 );
    friend istream& operator>>(istream & P01,
                               loc& P02);
// Why P01 is of type ostream&
- if you r overloading <<, then you r
  using it like this
      cout << obj1;
- cout will be P01
  and obj1 " " P02

- cout is of type ostream
& obj1 " " " loc.

- Both r passed by ref
- So P01 is of type ostream&
& P02 " of type loc&.
```

```
// Why return type is of ostream&
  - operator<< fun will return POI
    wch is of type ostream&, so
    return type is ostream&.

// Similarly for >> operator,
    ostream& replaced by istream&
    bcz cin is of type istream.

};

//Define these fun.
void loc::set(int x, int y)
{
        long = x;
        lat = y;
}

loc
void loc::operator++()
{
        ++long;
        ++lat;
        return *this;
}

loc   loc::operator--()
{
        --long; --lat;
        return *this;
}

loc   loc::operator-(loc POI)
{     loc temp;
      temp.long = long - POI.long;
      temp.lat = lat - POI.lat;
      return temp;
}
```

```
ostream& operator<<(ostream& PO1,
                    loc & PO2)
{   //PO1 is in fact cout
    PO1<< PO2.long << PO2.lat;
    return PO1;
}   //This fun will always return PO1

istream& operator>>(istream& PO1,
                    loc & PO2)
{   //PO1 is in fact cin
    cin>> PO2.long >> PO2.lat;
    return PO1;
}   // This fun will always return PO1,

int main()
{
    loc obj1, obj2;
    obj1.set(11, 11);
    obj2.set(22, 22);

    loc obj3;
    //Call ++ operator.
    obj3 = ++obj1;
    //Print obj3 using << operator.
    cout << obj3;

    //Call -- operator
    obj3 = --obj1;
    //Print obj3 using << operator.
    cout << obj3;

    //Cal - operator.
    obj3 = obj2 - obj1;
    cout << obj3;
```

```
// input obj3 using >> operator.
cin >> obj3;
cout << obj3;

}
```

Note** cout << obj3;
~~cout~~ cin >> obj3;
These statement are not valid under normal circumstances.
In this program >> & << r overloaded so cout << obj3 & cin >> obj3 r valid statements.

Q3 c) Rules to o/l operator thru MF

1) These operator fun should be defined as MF of the class.
2) obj on left side of operator is calling obj & " right " " " passing obj
3) For unary operator, there is only calling obj & no passing obj
4) Calling obj is returned using *this.

Rules to o/l more FF

1) These opr should be defined as FF
2) There will be no calling obj.
   Both obj on left & right side of operator r passing obj.
3) For unary opr, there will be only 1 passing obj & no calling obj
4)

Common rules for both FF & MF for opr overloading

1) At least one of the operand must be a class obj.

   eg `obj1 + obj2;` } allowed
   `obj1 + 10;` }
   `10 + 10.5;` } not allowed. to overload for these operands.

2) Only existing opr can be o/loaded.

3) ?:, sizeof, ::, ., .*, (cast)
   these opr can't be overloaded.

4) Can't rename existing opr

5) Can't change No. of operands.

6) Can't change precedence & associativity.

- Operators which can't be o/l these MF are >> and <<
- o/l for sub. can't be o/l these FF are =

Q4 a) Inheritence
b) "

c) Inline fun — <see notes>
This ptr — < " " >.

Q5 a)
b) } Not covered.
c)

5.	**How does OOP overcome the shortcomings of traditional programming approaches?**
Ans.	OOP provides the following advantages to overcome the shortcomings of traditional programming approaches: ✓ OOPs is closer to real world model. ✓ Hierarchical relationship among objects can be well-represented through inheritance. ✓ Data can be made hidden or public as per the need. Only the necessary data is exposed enhancing the data security. ✓ Increased modularity adds ease to program development. ✓ Private data is accessible only through designed interface in a way suited to the program.

3.13 Operators in C++

C++ has a rich set of operators. All C operators are valid in C++ also. In addition, C++ introduces some new operators. We have already seen two such operators, namely, the insertion operator <<, and the extraction operator >>. Other new operators are:

```
::          Scope resolution operator
::*         Pointer-to-member declarator
->*         Pointer-to-member operator
.*          Pointer-to-member operator
delete      Memory release operator
endl        Line feed operator
new         Memory allocation operator
setw        Field width operator
```

S. No	Static Memory Allocation	Dynamic Memory Allocation
1	In the static memory allocation, variables get allocated permanently, till the program executes or function call finishes.	In the Dynamic memory allocation, variables get allocated only if your program unit gets active.
2	Static Memory Allocation is done before program execution.	Dynamic Memory Allocation is done during program execution.
3	It uses stack for managing the static allocation of memory	It uses heap for managing the dynamic allocation of memory

4	It is less efficient	It is more efficient
5	In Static Memory Allocation, there is no memory re-usability	In Dynamic Memory Allocation, there is memory re-usability and memory can be freed when not required
6	In static memory allocation, once the memory is allocated, the memory size can not change.	In dynamic memory allocation, when memory is allocated the memory size can be changed.
7	In this memory allocation scheme, we cannot reuse the unused memory.	This allows reusing the memory. The user can allocate more memory when required. Also, the user can release the memory when the user needs it.

8	In this memory allocation scheme, execution is faster than dynamic memory allocation.	In this memory allocation scheme, execution is slower than static memory allocation.
9	In this memory is allocated at compile time.	In this memory is allocated at run time.
10	In this allocated memory remains from start to end of the program.	In this allocated memory can be released at any time during the program.
11	**Example:** This static memory allocation is generally used for array.	**Example:** This dynamic memory allocation is generally used for linked list.