// Board Questions
2017

Group 'A'

1a) Explain features of OOP.

=) The features of OOP are :-

i) Class
ii) Objects
iii) Encapsulation
iv) Abstraction
v) Polymorphism
vi) Inheritance
vii) Dynamic Binding
viii) Message passing

i) (lass -) The building block of C++ that leads to Object-Oriented programming is a class. It is a user defined ~~data~~ data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is an blueprint for an object.

ii) Object :- An object is an identifiable entity with some characteristics and behaviour. An Object is an instance of a class. When a class is defined, no memory is allocated but when it is instantiated (ie. an object is created) memory is allocated

Object take up space in memory and have an associated address like a record in pascal or structure or union in C.
When a program is executed the objects interact by sending messages to one another. Each objects contain data and code to manipulate the data.

iii) Encapsulation

Encapsulation is defined as wrapping up of data and information under a single unit. In Object-Oriented programming, Encapsulation is defined as binding together the data and the functions that manipulate them. By default data is not accessible to outside world and they are only accessible through the functions which are wrapped in a class. Prevention of data from direct access by the program is called data hiding or information hiding

iv) Data abstraction

Data abstraction is one of the most essential and important features of OOP in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

v) Polymorphism

↳ Polymorphism comes from the Greeks words "poly" and "morphism". "poly" means many and "morphism" means form i.e, many forms. Polymorphism means the ability to take more than one form. For example, an operation have different behaviour in different instances. The behaviour depends upon the type of the data used in the operator.

Different ways to achieving polymorphism in c++ program:

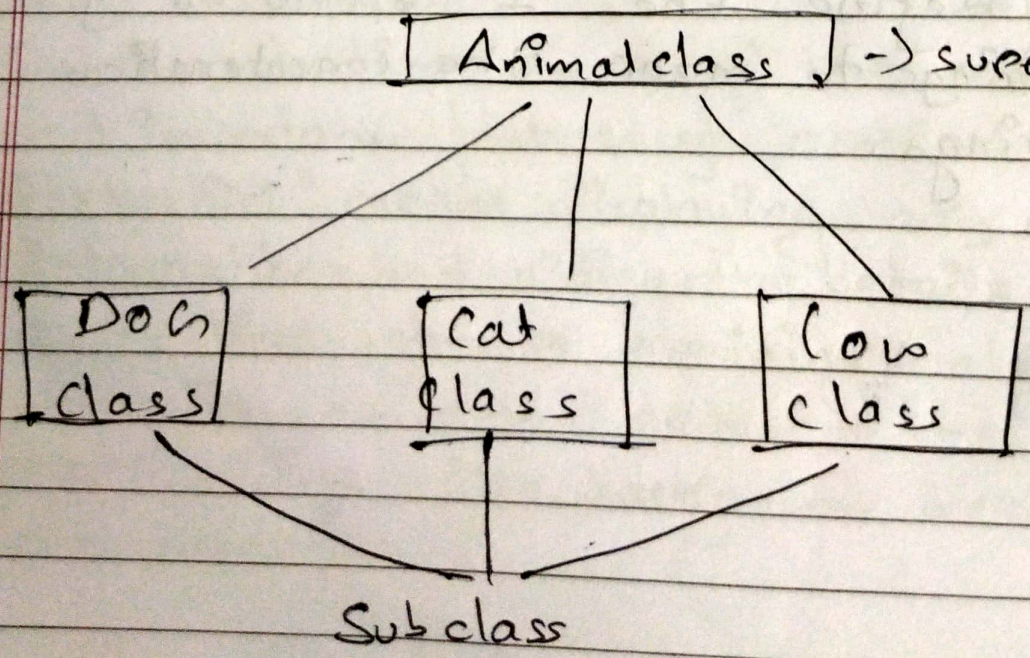
1) Function Overloading
2) Operator Overloading

C++ is able to express the operation of addition by a single operator say '+'. when this is possible you use the expression x+y to denote the sum of x and y, for many different types of x and y; integers, float and complex no. You can even define the '+' operation for two strings to mean the concatenation of the strings.

vi) Inheritance

↳ The capability of a class to derive properties and characteristics from another class is called inheritance. Inheritance is one of the most important features of object - Oriented programming.

- Sub class :- The class that inherits properties from another class is called Sub class or Derived class.
- Super class :- The class whose properties are inherited by sub class is called Base class or Super class
- Reusability :- Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing



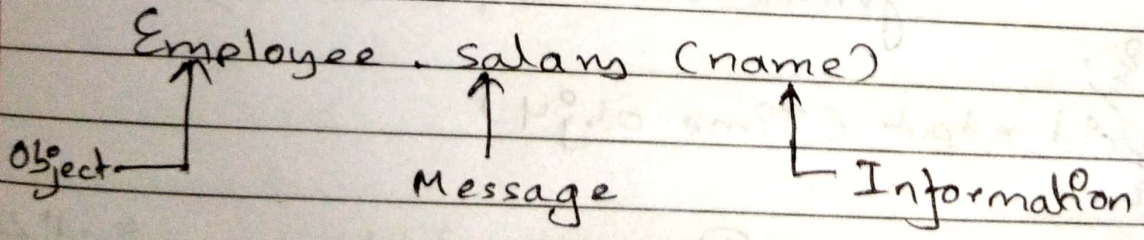
vi) Dynamic Binding

i) Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means the code associated with a given procedure call is not known until the time of the call at run time. It is associated with a polymorphic reference depends upon the dynamic type of that reference.

vii) Message passing

i) An object oriented program consists of a set of object that communicate with each other.

A message for an object is a request for execution of a procedure and therefore will invoke a function (procedure) in the receiving object that generates the desired result. Message pasing involves specifying the name of the object, the name of the function (message) and information to be sent.



15) Write a program to add two objects of a class using friend function. The class consists of hour and minute as its data member, input(), sum() and output() as its member functions.

```
#include <iostream>
using namespace std;
class Time {
        int hrs, min;
        public:
          void input (int x, int y)
          {
              hrs = x
              min = y
          }
          void sum (Time p, Time r)
          {
              hrs = p.hrs + r.hrs
              min = p.min + r.min
          }
          friend void output (Time);
};
void output (Time obj4)
{
    cout << "\nThe sum of the hour is: " << obj4.hrs;
    cout << "\n The sum of min is: " << obj4.min;
}
```

```
int main()
{
    Time obj1, obj2, obj3;
    cout<<"\n For the first object "<<endl;
    int a,b;
    cout<<"\nEnter the hrs:";
    cin>>a;
    cout<<"\nEnter the min:";
    cin>>b;
    obj1.input(a,b);
    cout<<"\n For the second object "<<endl;
    int c,d;
    cout<<"\n Enter the hrs:";
    cin>>c
    cout<<"\nEnter the min:";
    cin>>d;
    obj2.input(c,d);
    obj3.sum(obj1,obj2)
    output(obj3);
    return 0;
}
```

2a) What do you mean by type conversion? Explain its types.

=) Type conversion in c++ is to allow the data conversion of one type to that of another type.

There are two types of type conversion

i) Imple Implicit type conversion
ii) Explicit type conversion

i) Implicit type conversion

-) These are the type conversion done automatically by the compiler. It is also called Automatic conversion

```
#include<iostream>
using namespace std;
int main()
{
        int num_int = 9
        double num_double;

        num_double = num_int;
        cout<< "integer value" << num_int;
        cout<< "double value" << num_double;

        return 0;
}
```

ii) Explicit conversion
- C-style type casting
- Function notation
- Type conversion operators

Explicit conversion

i) When user manually changes data from one type to another, this is also known as Explicit conversion

→ C-style type casting
This type casting is favoured by the C-programming language.

```
#include<iostream>
using namespace std;
int main()
{
    int num_int = 112;
    double num_double;

    num_double = (double) num_int;

    cout<<"Integer value:"<< num_int;
    cout<<"Double value :"<< num_double;

    return 0;
}
```

-) Function - style casting
-) It is a c++ old style type casting where the variable is used as function

```
#include<iostream>
using namespace std;
int main()
{
        int num_int = 36;
        double num_double;

        numdouble = double (num_int);

        cout<<"Integer value:" << num_int;
        cout<<"Double value :" << num_double;

        return 0;
}
```

-) Type casting
Combination of both C-style type casting and Function style type casting

```
#include<iostream>
using namespace std;
int main()
{
        double num_double = 3.56
        cout<<" num_double =" << num_double << endl;
```

```
//c-style conversion from double to int
int num_int1 = (int)num_double;
cout << "num_int1 = " << num_int1 << endl;

//function-style conversion from double
  to int
int num_int2 = int(num_double);
cout << "num_int2 = " << num_int2 << endl;

return 0;

}
```

5) What is inline function? using inline function, write a program to add two integers and return sum.

=) Inline function are those function whose definitions are small and be substituted at the place where its function call is happened. It shortens the compilation time and there work burden in compiler will be less.

```
// Program
#include<iostream>
using namespace std;
inline sum (int n, int y)
{
        return (n+y);
}
int main()
{
        int a,b;
        cout<<"Enter the first integer:";
        cin>>a;
        cout<<"Enter the second integer:";
        cin>>b;
        cout<<"The sum of integers are:"
            <<sum(a,b);
        return 0;
}
```

Sa) What is access specifier ? Discuss its types.

=) Access specifiers in a class are used to assign the accessibility to the class member. It gives access to the object of class to use the data member and member function

Types of access specifier:-

i) public access specifier
ii) private access specifier
iii) protected access specifier

i) Public access specifier

=) The public access specifier can be used from to make the data member and member function public which means the public class member and functions can be used from outside of a class by any function or other classes. You can access public data member or function directly by using dot operator (.) or (arrow operator -> with pointer).

Syntax

```
class X{
    public:
        int n=10;
        void output()
        {
            cout<<n;
        }
};
int main()
{
```

// Syntax/code

```
class x {
        public: // access specifier
        int n=10;
        void output()
        {
            cout << n;
        }
};
int main()
{
        x obj;
        obj.output();
        return 0;
}
```

(ii) private access specifier

-) By the use of private access specifier the class members and functions can be used only inside of class and by friend function and classes. There is no direct access to the private data member and member function. Indirectly by the use of public member function the private data member can be used.

Syntax

```
class x {
        private: // private access specifier
        int y;
};
int main()
```

{

Xobj;

Obj.y = 50; // Not allowed (private)

return 0;

}

iii) Protected access specifier

=) By the use of protected access specifier the class members and member function can be cannot be accessed from outside the class, however they can be accessed by inherited class or friend function

Syntax

class Z {

protected: // protected access specifier

int height; // protected data member

int weight; // protected data member.

};

iv) file pointer

Group B

4. Write a program with an overloaded function calc_area() that calculates and return area of circle and rectangle. Assume appropriate number and type of arguments and return type. 8

```
#include <iostream>
using namespace std;
void calc_area (int p)
{
    float pie = 3.14, a=0;
    a = pie * p*p;
    cout<<"\n The area of circle is :"<<a;
}
void calc_area (int x, int y)
{
    int ar=0
    ar = x*y;
    cout<<"\n The area of rectangle
        is :"<<ar;
}
int main()
{
    int l, b;
    float r;
    cout<<"\n For the calculation of
        area of rectangle ---"<<endl;
    cout<<"\n Enter the length of the
        rectangle :";
    cin>>l;
```

```
cout<<" Enter the breadth of rectangle :";
cin>>b;
calc_area(l,b);
cout<<"\n\n For the calculation of area of
circle ---";<<endl;
cout<<"\n Enter the radius of the circle:";
cin>>r;
calc_area(r);
return 0;
}
```

5. What are the roles of constructor and destructor? Write a program to demonstrate ~~can parod~~ parameterized and copy constructor.

=) The roles of constructor and destructor are:

i) Constructor is used to initialize member variable to pre-defined values as soon as an object of a class is declared.

ii) constructor function gets invoked when an object of a class is constructed (declared)

iii) Constructor function is used to initialize the objects (at the time of creation), and they are automatically invoked.

iv) Destructor function is used to release dynamic allocated memory. Destructors are used to free memory, realese resources and to perform other clean up.

v) Destructors are automatically named when an object is destroyed.

```
// code for parameterized constructor
#include<iostream>
using namespace std;
class x {
    int a, b;
    public:
    x(int c, int d)
    {
        a = c;
        b = d;
    }
    void output ()
    {
        cout<<"\n The value of a is: "<<a;
        cout<<"\n The value of b is: "<<b;
    }
};
int main()
{
    x obj1(10, 12);
    obj1.output();
    return 0;
}
```

```
// code for copy constructor
#include<iostream>
using namespace std;
class Demo{
    private:
    int num1, num2;
    public:
    Demo(int n1, int n2)
    {
        num1 = n1;
        num2 = n2;
    }
    Demo (const Demo &n)
    {
        num1 = n.num1;
        num2 = n.num2;
    }
    void display()
    {
        cout<<"num1 = "<<num1<<endl;
        cout<<"num2 = "<<num2<<endl;
    }
};
int main()
{
Demo obj1(10,20);
Demo obj2 = obj1;
        obj1.display();
        obj2.display();
        return 0;
}
```

6. Mention rules for overloading an operator. Write a program to convert class type of data into basic type.

=) The rules for overloading an operator are:-

i) Only built-in operators can be overloaded. New operators can not be created.

ii) Arity of the operator cannot be changed.

iii) Precedence and associativity of the operator cannot be changed.

iv) Overloaded operators cannot have default arguments except the function call operator () which can have default arguments.

v) Operators cannot be overloaded for built in types only. At least one operand must be used defined type.

// program to convert class type of data into basic type

```
#include<iostream>
using namespace std;
class X{
    int hrs, min;
    public:
      X(int , int);
      operator int();
     ~X()
     {
        cout<<"\n Destructor called.....";
     }
};
X::X (int a, int b)
{
    cout<<"\n constructor called with two
      parameter.....";
    hrs = a;
    min = b;
}
X:: operator int()
{
    cout<<"\n class type to basic type
          conversion ...";
    return (hrs* 60+min);
}
```

```
int main()
{
    int h, m, duration;
    cout << "\n Enter the hour : ";
    cin >> h;
    cout << "\n Enter the minutes : ";
    cin >> m;
    t t(h, m);
    duration = t;
    cout << "\n Total minutes are :" << duration;
    cout << "\n\n In second method operator
            overloading ----- ";
    duration = t.operator int();
    cout << "\n Total minutes are :" << duration;
    getch();
}
```

7. Assume that a bank maintains two kinds of accounts for customers, one called as saving account and the other as current amount. The saving account provides compound interest at the annual rate of 10%. Current account holders get simple interest of 5% per year. Create a class ~~name~~ account that stores customer name and account ~~name~~ number. From this, derive the classes : current_account and saving_account. Include necessary member functions and calculate the total amount of money in an account for both types of customer.

// code

```
#include <iostream>
#include <math.h>
using namespace std;

class Account {
        protected:
        string customer_name;
        int account_num;
        int balance;
        int t;
        public:
        void get_details ()
        {
            cout<<"Name of the customer:";
            getline (cin, customer_name);
            fflush(stdin);
            cout<<"\n Enter the account number of
                    the customer :";
            cin>> account_num;
            fflush(stdin);
            cout<<"\n Enter the balance amount :";
           cin>> balance;
           fflush(stdin);
           cout<<"\n Enter the time period:";
            cin>> t;
            fflush (stdin);
}
```

```
void display ()
{
    cout<<" ********** Details
        of person ********** "<<endl;
    cout<<"\n Customer name : "<<customer_name
      <<endl;
    cout<<"\n Customer acc number :"
        << account_num<<endl;
}
};
class current_acc : public Account
{
    public :
    void simpleInterest ()
    {
        int SI;
        int r1=5;
        SI = (balance *t * r1)/100;
        display();
        cout<<"\n The total balance
        amount with the simple
        interest is : "<<SI <<endl;
    }
};
class Saving_acc : public Account
{
    public :
        void compound Interest ()
        {
            int CI;
            int r2 = 10;
```

```
        CI = balance * pow(1.1, t);
        display();
        cout << "\n The total balance amount
             with the compound interest is:"
             << CI << endl;
    }
};
int main()
{
    int acc_type;
    Current_acc obj2;
    Saving_acc obj3;
    label1:
    cout<< " Enter the account type :" <<endl;
    cout<<" 1. Current Account " <<endl;
    cout<<" 2. Saving Account " <<endl;
    cin >> acc_type;
    if (acc_type == 1)
    {
        obj2.get_details();
        obj2.simpleInterest();
    }
    else if (acc_type == 2)
    {
        obj3.get_details();
        obj3.compoundInterest();
    }
    else
    {
        cout<< "Invalid choice";
        goto label1;
    }
    return 0;
}
```

8. When do we use virtual function? Differentiate early and late binding.

=) Virtual function is a member function which is declared within a base class and is re-defined (overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

Virtual functions play an important role in making programming experience effective and efficient. They are the direct supporter of object-oriented programming. A virtual function is used to perform late binding as well as dynamic linkage operations by telling the compiler. Therefore, it is used to point to the base class.

Difference between early and late binding

Early Binding	late Binding
i) The process of using the class information to resolve method calling that occurs at compile time is called Early Binding.	The process of using the object to resolve method calling that occurs at run time is called the late Binding
ii) Early Binding happens at compile time.	Late Binding happens at run time.
iii) Early Binding uses the class information to resolve method calling	Late Binding uses the object to resolve method calling.
iv) Early Binding is also known as static binding	Late Binding is also known as dynamic binding.
v) Overloading methods are bonded using early binding	Overridden methods are bonded using late binding
vi) Execution speed is faster in early binding	Execution speed is lower in late binding.

9. Write a program to sort N numbers in ascending order using a function template.

```
// code
#include<iostream>
using namespace std;
template <typename R>
void sortt (R b[ ], R y)
{
    R i=0, j=0, temp;
    for (i=0; i<y; i++)
    {
        for (j=i+1; j<y; j++)
        {
            if (b[i] > b[j])
            {
                temp = b[j];
                b[j] = b[i];
                b[i] = temp;
            }
        }
    }
    cout << "Ascending order \n";
    for (i=0; i<y; i++)
    {
        cout << b[i] << endl;
    }
}
```

```
int main()
{
    int x[10];
    int n, i=0, j=0
    cout << "\n Enter the number to be
            sorted :";
    cin>>n
    cout << "Enter the numbers :\n";
    for (i=0; i<n; i++)
    {
        cin>>n[i];
    }
    Sortt <int>(x,n);
    return 0;
}
```

11. Why do we need exception handling? Explain with an example.

=) Exception handling is necessary for the separation of Error handling code from Normal code. Exception handling handles the errors (Exception) and makes the code less readable and maintainable. The Exception handling is built upon three keywords: try, catch and throw.

i) throw - A program throws an exception when a problem shows up. This is done using a throw keyword

ii) catch - A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

iii) try - A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

// code

```
#include<iostream>
using namespace std;

double division (int a, int b){
        if (b==0)
        {
            throw "Division by zero condition!";
        }
        return (a/b);
}
int main(){
        int x=50;
        int y=0;
        double z=0;

        try{
            z= division (x,y);
            cout << z<<endl;
            }
        catch (const char* msg)
        {
            cerr << msg <<endl;
        }
        return 0;
}
```

12. Write short notes:

a) static data member and static member function

=) Static data member is a variable which is declared with the static keyword, it is also known as class member, thus one single copy of the variable creates & for all objects. Any changes in the static data member through one member function will reflect in all other objects member function.

Declaration

static data_type member_name;

Defining the static data member
It should be defined outside of the class following this syntax:

data_type class_name :: member_name = value;

- If you are calling a static data member within a member function, member function should be declared as static (i.e, a static member function can access the static data member)

static member function is a special function in a programming language, which is to access only static data members and other static member functions. There is only one copy of the static member no matter how many objects of a class are created.

A static member is shared by all objects of the class made. There is also one more highlighting feature of static data that it is initialized to zero when the first object is created. The static member function are accessed using the only class name and the scope resolution operation ::

```
// code
#include <iostream>
using namespace std;
class notebook {
        static int page_number;
        public:
        static int value()
        {
            return page_number;
        }
};
int notebook::page_number =1;
int main()
{
    cout << "Number= " << notebook::value() << endl;
    return 0;
}
```

5) This operator

→ Every object in c++ has access to its own address through an important pointer called this pointer. The this pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Friend functions do not have a this pointer, because friends are not members of a class. Only member function have a this pointer

```
// code
#include<iostream>
using namespace std;
class student {
        public:
        int roll_no;
        string name;
        float marks;
        student (int x, string y, float z)
        {
            this -> roll_no = x;
            this -> name = y;
            this -> marks = z;
        }
```

```
    void show()
    {
    cout << " student name: " << name << endl;
    cout << " student roll: " << roll << endl;
    cout << " student Marks: " << marks << endl;
    }
};
int main(void)
{
    student stu = student (102, "shishir", 90);
    stu.show();
    return 0;
}
```

c. Namespaces

=) Namespaces allow us to group named entities that otherwise would have global scope into narrower scopes, giving them namespace scope. This allows organizing the elements of programs into different logical scopes referred to by names.

- Namespace ~~allows~~ is a feature added in C++ and not present in C.
- A namespace is a declarative region that provides a scope to the identifiers (names of the types, function, variables etc) inside it.
- Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.
- Namespace declarations appear only at global scope
- Namespace declarations can be nested within another namespace.
- Namespace declarations don't have access specifiers (public or private)
- No need to give semicolon after the closing brace of definition of namespace
- We can split the definition of namespace over several units.

```
//code
#include <iostream>
using namespace std;

namespace ns1
{
        int value()
        {
            return 5;
        }
}
namespace ns2
{
    const double x = 100;
    double value()
    {
        return 2*n;
    }
}
int main()
{
    cout<< ns1::value()<< "\n";

    cout<< ns2::value()<< '\n';

    cout<< ns2::x << '\n';

    return 0;
}
```