Name - Tanmay R. Mathkar YEAR: SE
PRN: F22111018 ; DIV: COMP I

DATE | PAGE NO.

# OOP Assignment

Q1. what is meant by polymorphism? Explain the types of polymorphism with an example of each.

Ans. • Polymorphism simply means more than one form.

• That is, the same entity (function or operator) behaves differently in different scenarios.

→ Types of polymorphism:-

i) Compile Time polymorphism:-

• You invoke the overloaded functions by matching the number and type of arguments.

• The information is present during compile-time.

1. Function overloading:-

• Function overloading occurs when we have many functions with similar names but different arguments.

• The arguments may differ in terms of number or type.

eg:

```
#include<iostream>
using namespace std;
int sum(int num1, int num2)
{
      return num1+num2;
}
double sum(double num1, double num2)
{
        return num1+num2;
}
int sum(int num1, int num2, int num3)
{
        return num1 + num2 + num3;
}
```

```
int main()
{
    cout <<"sum1="<<sum (5,6) <<endl;
    cout<< "sum2=" <<sum (5.6, 5.7) <<endl;
    cout<<" sum3=" << sum(1,2,3)<<endl;
    return 0;
}
// O/P:
sum1 = 11
sum2=11.3
sum3= 6
```

2. Operator overloading

- Operator overloading is basically function overloading, where different operator functions have the same symbol but different operands.
- eg:

```
class Count
{   public:
    int a;
    int b;
    Count (int x, int y)
    {
            a=x;
            b=y;
    }
    void show()
    {
            cout<< "A="<<a<<" " <<"B="<<b<<endl;
    }
    void operator -()
```

```
        {
            a=-a;
            b=-b;
        }
};
int main()
{       Count obj(20,30);
        obj.show();
        -obj;
        obj.show();
        return 0;
}
```

//O/P:

```
A = 20   B=30
A = -20   B=-30
```

iv) Runtime Polymorphism:-

- Runtime polymorphism is achieved through function overriding.
- The function to be called/invoked is established during runtime.

eg: Virtual function:

- A virtual function is a member function that is declared within a base class and redefined by a derived class.
- eg:

```
class b
{ public:
        virtual void show()
        {
                cout<<"\n showing base class :....";
        }
```

```
          void display ()
          {
                cout <<"\n Displaying base class...";
          }
};
class d : public b
{ public:
          void display()
          {
               cout <<"\n Displaying Derived
                          class.....";
          }
          void show()
          {
               cout<< "\n Showing derived class...";
          }
};
int main()
{
        b B;
        b *ptr;
        cout <<"\n\t P points to base: \n";
              ptr=&B;
              ptr→display ();
        ptr→show();

        cout <<"\n\n\t P points to derive! \n";
              d D;
              ptr=&D;
              ptr→display ();
      ptr→ show();
}
```

```
//O/P:
P points to base:
Displaying base class.....
Showing base class.......

        P points to derive :

Displaying base class....
Showing derived class.....
```

Q2. Explain the use of keywords. Explain mutable and explicit keywords with examples of each.

Ans. • The two unusual keywords : explicit and ~~implicit~~ mutable have quite different effects, but both ~~se~~ are grouped together here because they both modify class members.

- The explicit keyword relates to data conversion, but mutable has a more subtle purpose.

1. Mutable Keyword

- mutable keywords come in handy when in a const declared object, you want to update a few const data members without updating other data members.
- eg:

```
class B
{ public:
        mutable string name;
        int roll
        void nam (string n, int no)
```

```
            {
                roll=no;
                name=n;
            }
            void changename (string p) const
            {
                name=p;
            }
            void display()
            {
                cout<<"Name is: "<<name<<endl;
                cout<<"Roll no is: "<<roll<<endl;
            }
};
int main()
{
    string p1,p2;
    int rollno;
    B b;
    cout<<"Enter name: ";
    cin >>p1;
    Cout <<"Enter roll no: ";
    cin >>rollno;
    b.nam (p1,rollno);
    b.display();
    cout<<"Enter name again: ";
    cin >>p2;
    b.changename(p2);
    b.display ()
    return 0;
}
```

```
//O/P:
Enter name: Tanmay
Enter roll no: 18
Name is: Tanmay
Roll no is: 18
Enter name again: Rajesh
~~Enter~~
Name is: Rajesh
Roll no is: 18
```

2. Explicit keyword:-

- ~~eg.~~ As discussed in C++, if a class has a constructor which can be called with a single argument, then this constructor becomes conversion constructor because such a constructor allows conversion of the single argument to the class being constructed.
- Prefixing the explicit keyword to the constructor prevents the compiler from using that constructor for implicit conversions.
- eg:

```
Class Blah
{ public:
        explicit Blah (int blah)
        {
            m_blah = blah;
        }
        int getBlah ()
        {
            return m_blah;
        }
```

```
    private:
            int m_blah;
};
void Ext_Blah (Blah blah)
{
      int x = blah. getBlah ();
}
int main ()
      // Your code goes here
      Ext_Blah (3);
```

- The above code will give the following error:-

```
could not convert '3' from 'int' to 'Blah'
            Ext_Blah (3);
                      ^
```

- It is now necessary to call for a conversion explicitly with Ext_Blah(Blah(3)) as shown below:

```
int main()
{
      Ext_Blah (Blah (3));
}
```

Q3. Explain typecasting concept with example. Also include its types.

Ans.
- Type ~~c~~ casting is converting one data type into another one.
- It is also called as data conversion or type conversion.
- There are two types of type casting operations;

1. Implicit Type Conversion:-
- The type conversion that is done automatically by the compiler is known as implicit type conversion.

- It does not require any effort from the programmer. The C++ compiler has a set of predefined rules.
- Based on these rules, the compiler automatically converts one data type to another.

eg:

```
int main()
{
    int num;
    double num1 = 6.28;
    num=num1;
    cout<<"The value of the int variable is: "
          <<num <<endl;
    cout <<"The value of the double variable is: "
          << num1 <<endl;
    return 0;
}
```

```
//O/P:
The value of the int variable is: 6
The value of the double variable is: 6.28
```

2. Explicit Type conversion:-

- Explicit type conversions are those conversions that are done by the programmer manually.
- In other words, explicit conversion allows the programmer to typecast (change) the data type of a variable to another type.

i) ~~Explicit type~~ C-style type casting:

- This type casting is usually used in the C programming language. It is also known as cast notation.

- Syntax:

```
(datatype) expression;
```

- eg:

```
int main()
{
      char char_var = 'a';
      int  int_var;
      int_var = (int) char_var;
      cout << "char_var: " << char_var;endl;
      cout << "int_var: " << int_var << endl;
      return 0;
}
//O/P:
char_var: a
int_var: 97
```

2. Function style casting:-

- As the name suggests, we can perform explicit typecasting using function style notations.
- Syntax: datatype (expression);
- eg:

```
int main()
{
      int int_var =17
      float float_var ;
      float_var = float (int_var)/2;
      cout << "float_var: " << float_var << endl;
      return 0;
}
//O/P:
float_var: 8.5
```