2075

Created	@April 2, 2023 12:00 PM
Class	OOPL
Type	Exam Material
Materials	
Reviewed	☑

Group A

1. Explain the concept of user-defined to user-defined data conversion routine located in the destination class.

User-defined to user-defined data conversion routines, also known as type conversion operators, allow the programmer to define how an instance of one user-defined data type can be converted to another user-defined data type. This can be useful when working with complex data types that have a similar structure but different names or representations.

The class to class conversion can be performed either by **defining casting operator function in source class** or **using the constructor in the destination class.**

For the conversion routines located in the destination class, which is the class that the user wants to **convert their object to**. The routine takes an object of the source class (the class that the user wants to **convert from**) as an argument and returns an object of the destination class.

For routine located in destination class

```
#include<iostream>
using namespace std;
class Triangle {
  int base, height;
  float area;
  public:
  Triangle(int b, int h) {
  base = b; height = h;
  area = 0.5 * base * height;
  }
  void print() {
  cout << "Base: " << base << "\nHeight: " << height << "\nArea of Triangle: " << area;
  }
  int getBase() {
  return base;
  }
  int getHeight() {
  return height;
  }
};
class Rectangle {
  int width, length, area;
  public:
  Rectangle(int w, int l) {
  width = w; length = l; area = width * length;
  }
  void output() {
  cout << "\nLength: " << length << "\nWidth: " << width << "Area of rectangle: " << area;
  }
  Rectangle(Triangle t) {
  width = t.getBase();
  length = t.getHeight();
  area = width * length;
  }
};
int main() {
Triangle t(10, 20);
Rectangle r=t; // Triangle to Rectangle [Rectangle(t)]
r.output();
t.print();
return 0;
}
```

For routine located in source class

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

class Rectangle {
  int width, length, area;
  public:
  Rectangle(int w, int l) {
  width = w; length = l; area = width * length;
  }
  void output() {
  cout << "\nLength: " << length << "\nWidth: " << width << "Area of rectangle: " << area;
  }
};
class Triangle {
    int base, height;
    float area;
    public:
    Triangle(int b, int h) {
    base = b; height = h;
    area = 0.5 * base * height;
    }
    void print() {
    cout << "Base: " << base << "\nHeight: " << height << "\nArea of Triangle: " << area;
    }
    operator Rectangle() {
    Rectangle temp(base, height);
    return temp;
    }
};

int main() {
Triangle t(10, 20);
Rectangle r=t; // Triangle to Rectangle
r.output();
t.print();
return 0;
}
```

2. Depict the difference between private and public derivation. Explain derived class constructor with suitable program.

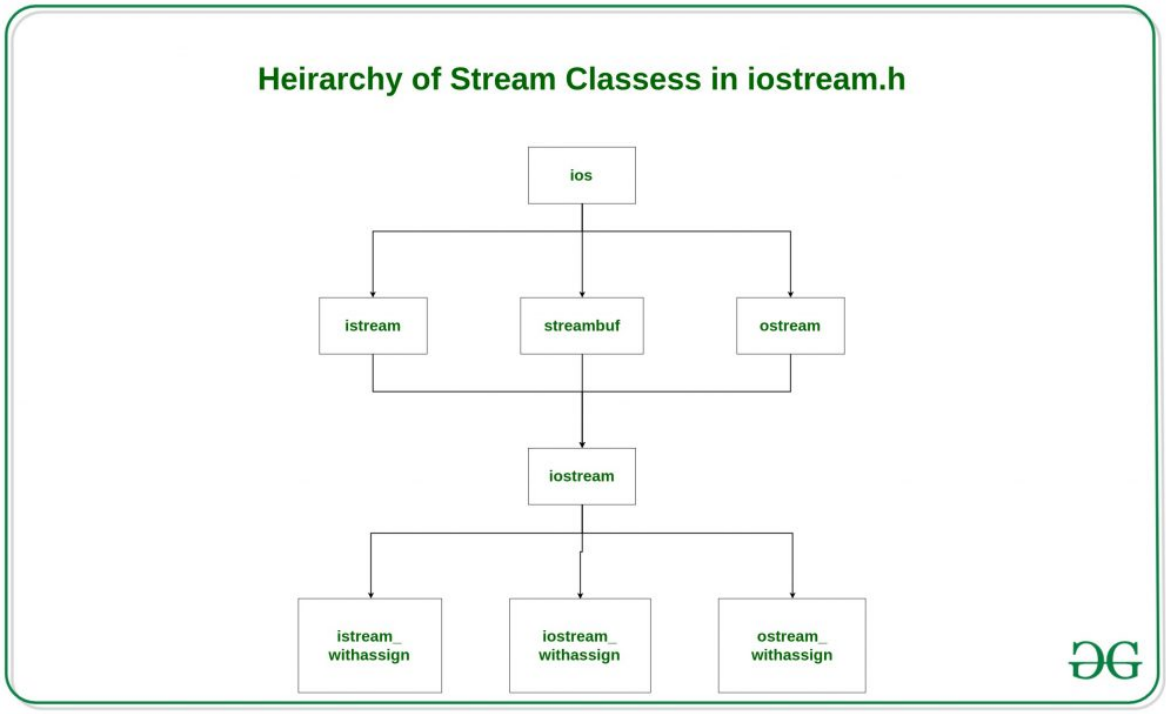
Sr. No.	Publicly derived inheritance	Privately derived inheritance
1	The public derivation of inheritance means that the derived class can access public and protected members of the base class.	The private derivation of inheritance means that the derived class can access public and protected members of the base class privately.
2	The public derivation does not change the access specifiers for inherited members in the derived class. They can be inherited further.	The inherited members become private in the derived class and they can not be inherited further by any other derived class derived from this derived class.
3	The protected member of base class will act as a protected member in derived class.	The protected members of a base class will act as a private member in derived class.
4	The public member of base class will remain public member in derived class.	The public members of a base class will act as a private member in derived class.

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


class Student {
    protected:
    int id, roll;
    public:
        Student(int id, int roll) {
            this->id = id;
            this->roll = roll;
        }
};

class CS : public Student {
    double marks[3];
    public:
        CS(int id, int roll, double m1, double m2, double m3) : Student(id, roll) {
            marks[0] = m1;
            marks[1] = m2;
            marks[2] = m3;
        }

        void display() {
            cout << "CS Student:" << endl;
            cout << "ID: " << id << endl;
            cout << "Roll no: " << roll << endl;
            cout << "Subject 1: " << marks[0] << endl;
            cout << "Subject 2: " << marks[1] << endl;
            cout << "Subject 3: " << marks[2] << endl;
            cout << "---Average---: " << (marks[0] + marks[1] + marks[2])/2 << endl;
        }
};

class Maths : public Student {
    double marks[3];
    public:
        Maths(int id, int roll, double m1, double m2, double m3) : Student(id, roll) {
            marks[0] = m1;
            marks[1] = m2;
            marks[2] = m3;
        }

        void display() {
            cout << "Maths Student:" << endl;
            cout << "ID: " << id << endl;
            cout << "Roll no: " << roll << endl;
            cout << "Subject 1: " << marks[0] << endl;
            cout << "Subject 2: " << marks[1] << endl;
            cout << "Subject 3: " << marks[2] << endl;
            cout << "---Average---: " << (marks[0] + marks[1] + marks[2])/2 << endl;
        }
};

int main() {
    CS c(1, 101, 89, 79, 89);
    Maths m(2, 202, 90, 80, 70);

    c.display();
    m.display();

    return 0;
}
```

---

3. Briefly explain the hierarchy of stream classes. Write a program that overloads extraction and insertion operators.

In C++ there are number of stream classes for defining various streams related with files and for doing input-output operations. All these classes are defined in the file `iostream.h`.



1. `ios` **class** is topmost class in the stream classes hierarchy. It is the base class for `istream`, `ostream`, and `streambuf` class.
2. `istream` and `ostream` serves the base classes for `iostream` class. The class `istream` is used for **input** and `ostream` for the **output**.
3. Class `ios` is indirectly inherited to `iostream` class using `istream` and `ostream`. To avoid the duplicity of data and member functions of `ios` class, it is declared as virtual base class when inheriting in `istream` and `ostream`.

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

class Complex {
    private:
        int real, imag;

    public:
        Complex(int r = 0, int i = 0) {
            real = r;
            imag = i;
        }

    friend ostream &operator<<(ostream &out, Complex &c){ // output
        out << c.real;
        out << "+i" << c.imag << endl;
        return out; // return
    }

    friend istream &operator>>(istream &in, Complex &c){ // input
        cout << "Enter Real Part ";
        in >> c.real; //
        cout << "Enter Imaginary Part ";
        in >> c.imag;
        return in; // return
    }
};

int main() {
    Complex c1;
    cin >> c1;
    cout << "The complex object is ";
    cout << c1;
    return 0;
}
```

Group B

4. Write a member function called `reverseit()` that reverses a string (an array of character). Use a for loop that swaps the first and last characters, then the second and next-to last characters and so on. The string should be passed to `reverseit()` as an argument.

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

class Stri {
    public:
        void reverseit(char *str, int len) { // <- formal arg as a pointer
            int start = 0;
            int end = len - 1; // index starts from 0
            int i = 0;

            while (start < end) {
                char temp = str[start];
                str[start] = str[end];
                str[end] = temp;

                start++;
                end--;
            }
        }
};

int main() {
    Stri s;

    char str[10] = "Mansij";

    s.reverseit(str, 6); // total length counted from 1

    cout << str;

    return 0;
}
```

---

5. What is the principle reason for using default arguments in the function? Explain how missing arguments and default arguments are handled by the function simultaneously?

A function can be called without specifying all its arguments, but it **does not work on any general function**. The function declaration **must provide default values for those arguments that are not specified**. When the arguments are missing from function call, default value will be used for calculation. Default values should be assigned from right to left.

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

int sum(int a, int b=5, int c=10) { // <- right to left
    return a + b + c;
}

int main() {
    cout << sum(11) << endl;

    cout << sum(11, 12) << endl;

    cout << sum(11, 12, 13) << endl;

    return 0;
}
```

---

6. “An overloaded function appears to perform different activities depending the kind of data send to it” Justify the statement with appropriate example.

In programming, overloading functions allow multiple functions with the **same name to be defined with different input parameters**. This enables the same function name to perform different actions depending on the type or number of arguments passed to it.

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

int calc(int i, int j) {
    return i+j;
}

float calc(int i, float j) {
    return i*j;
}

int main() {
    int a=5, b=1;
    float x=8.1, y=6.8;

    cout << calc(a, b) << endl;
    cout << calc(a, x) << endl;

    return 0;
}
```

For example, consider a function `calc` that takes two arguments. The function performs different calculations depending on the data types of the arguments passed to it. If the first argument is an integer and the second argument is also an integer, the function calculates the **sum** of the two arguments. If the first argument is an integer and the second argument is a float, the function calculates the **product** of the two arguments passed.

---

7. Explain the default action of the copy constructor. Write a suitable program that demonstrates the technique of overloading the copy constructor.

In C++, a copy constructor is a special member function that creates a new object by initializing it with an existing object of the same class. When a copy constructor is called, it typically performs a member wise copy of the source object's data members into the new object.

When a copy constructor is not defined, the C++ compiler automatically supplies with its self-generated constructor that **copies the values of the object to the new object.** This is called **Default Copy Constructor.**

Assuming: Overloaded copy constructor means copy constructor that is overloaded with different data types passed.

```
/*
Write a suitable program that demonstrates the technique of overloading the copy constructor.
*/
#include<iostream>
using namespace std;
class Area{
    float area;
    public:
        Area() {} // 1
        Area(double r) { // 2
            area=3.14*r*r;
        }
        Area(Area &i) { // 3
            area = i.area;
        }
        Area(Area &i, bool add) { //4
            if (add)
                area = i.area + 5;
        }

        void display() {
            cout << area << endl;
        }
};

int main(){
    Area obj1(1.2), obj2; // second and first construtor respectively
    obj1.display();

    obj2=obj1; // third construtor
```

```
    obj2.display();

    Area obj3(obj1, true); // fourth construtor
    obj3.display();

    return 0;
}
```

In the example above, in statement `obj2=obj1` the third constructor is called and the value of `area` of `obj1` is stored to in the value of area of `obj2` . And in the statement `Area obj3(obj1, true)` , two arguments are passed: `obj1` and `true` . Where if `true` is passed to the `add` variable of boolean data type then as per the code, it adds `5` to the value or `area` of `obj1` and stores the result in the variable `area` of `obj3` .

---

8. Briefly explain types of inheritance used in object oriented Programming.

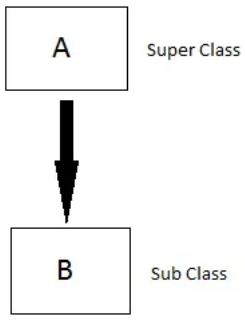
Inheritance is a process in which **one object acquires all the properties and behaviors of its parent object automatically**. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class. For example, if we have a base class called `Animal`, we might create a derived class called `Cat` that inherits from `Animal`. We would say that a `Cat` "is-a" kind of `Animal`.

Inheritance is a powerful technique that allows us to create complex class hierarchies and **reuse** code across multiple classes, making our code **more modular, maintainable, and extensible**.

Types of inheritance:

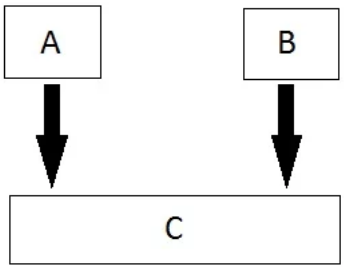
1. Single Inheritance

   Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.



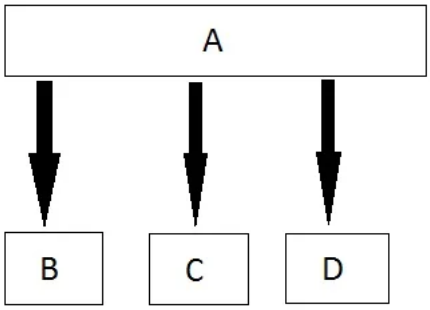
2. Multiple Inheritance

   Multiple inheritance is a type of inheritance in which a class derives from more than one classes. As shown in the above diagram, class C is a subclass that has class A and class B as its parent.



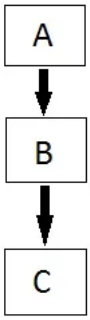
3. Hierarchical Inheritance

   In hierarchical inheritance, more than one class inherits from a single base class.



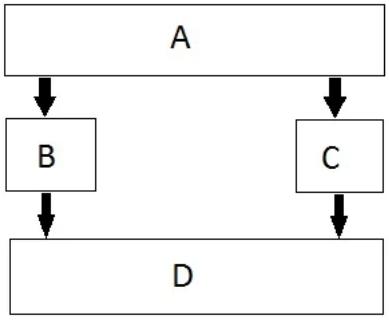
4. Multi-level Inheritance

   In multilevel inheritance, a class is derived from another derived class. This inheritance can have as many levels as long as our implementation doesn't go wayward.



5. Hybrid Inheritance

   Hybrid inheritance is usually a combination of more than one type of inheritance. In the below representation, we have multiple inheritance (B, C, and D) and multilevel inheritance (A, B and D) to get a hybrid inheritance.



---

9. Create a real scenario where static data members are useful. Explain with suitable example.

Static data members are class members that are declared using the static keyword. There is only one copy of the static data member in the class, even if there are many class objects. This is because all the objects share the static data member. The static data member is always initialized to zero when the first class object is created.

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

class Student{
    int rollNo;
    char name[10];
    int marks;
    public:
```

```
        static int objectCount;
        Student() {
            objectCount++;
        }

        void getdata() {
            cout << "Enter roll number: " << endl;
            cin >> rollNo;
            cout << "Enter name: " << endl;
            cin >> name;
            cout << "Enter marks: " << endl;
            cin >> marks;
        }

        void putdata() {
            cout << "Roll Number = " << rollNo << endl;
            cout << "Name = " << name << endl;
            cout << "Marks = " << marks << endl;
            cout << endl;
        }
};

int Student::objectCount = 0; // initializing static var to 0

int main() {
    Student s1;
    s1.getdata();
    s1.putdata();
    Student s2;

    s2.getdata();
    s2.putdata();
    Student s3;

    s3.getdata();
    s3.putdata();
    cout << "Total objects created = " << Student::objectCount << endl; // <- getting static var with scope resolution, can't use

    return 0;
}
```

---

10. Create a function called `swaps()` that interchanges the values of the two arguments sent to it (pass these arguments by reference). Make the function into a template, so it can be used with all numerical data types (`char`, `int`, `float`, and so on). Write a main() program to exercise the function with several types.

```
// NOTE: In pass by reference, address operator (&) in paramemter list and no address operator(&) in arguments
#include<iostream>
using namespace std;

template<class T>
void swaps(T &a, T &b) {
    T temp = a;
    a = b;
    b = temp;
}

int main() {
    int a=1, b=2;
    char i='m', j='a';
    float k=8.9, l=6.9;

    cout << "Before Swap" << endl;
    cout << "a: " << a << "\nb: " << b << endl;
    cout << "i: " << i << "\nj: " << j << endl;
    cout << "k: " << k << "\nl: " << l << endl;

    swaps(a, b);
    swaps(i, j);
    swaps(k, l);

    cout << "After Swap" << endl;
    cout << "a: " << a << "\nb: " << b << endl;
    cout << "i: " << i << "\nj: " << j << endl;
    cout << "k: " << k << "\nl: " << l << endl;

    return 0;
}
```

Explain how exceptions are used for handling C++ error in a systematic and OOP-oriented way with the design that includes multiple exceptions.

Exceptions are run-time **anomalies or abnormal conditions** that a program encounters during its execution. Exception Handling in C++ is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors. In C++, we use 3 keywords to perform exception handling:

- try
- catch
- throw

try: The `try` statement allows us to define a block of code to be tested for errors while it is being execute**d.**

throw: The `throw` keyword throws an exception when a problem is detected, which lets us create a custom error.

catch: The `catch` statement allows us to define a block of code to be executed, if an error occurs in the try block.

Syntax:

```
try {
    //your code
} catch(exceptionName e1) {
    // catch block
} catch(exceptionName e2) {
    // catch block
} catch(exceptionName en) {
    // catch block
}
```

Example for multiple exception handling:

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

void test(int x) {
    try {
        if (x > 0)
            throw x;
        else
            throw 'x';
    }

    catch (int x) {
        cout << "\nCaught a integer and that integer is:" << x;
    } catch (char x) {
        cout << "\nCaught a character and that character is:" << x;
    }
}

int main() {
    test(10);
    test(0);

    return 0;
}
```

12. How is character I/O different from Binary I/O? Explain with examples.

The key difference between them is how the data is represented in the file.

Character I/O	Binary I/O
Text files are used to store data more user friendly.	Binary files are used to store data more compactly.
In the text file, a special character whose ASCII value is 26 inserted after the last character to mark the end of file.	In the binary file no such character is present. Files keep track of the end of the file from the number of characters present.
Content written in text files is human readable.	Content written in binary files is not human readable and looks like encrypted content.

Character I/O	Binary I/O
In the text file, the newline character is converted to carriage-return/linefeed before being written to the disk.	In binary file, conversion of newline to carriage-return and linefeed does not take place.
In text file, text, character, numbers are stored one character per byte i.e. 32667 occupies 5 bytes even though it occupies 2 bytes in memory.	In binary file data is stored in binary format and each data would occupy the same number of bytes on disks as it occupies in memory.

Character I/O example:

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

int main() {
    ofstream file("example.txt");
    file << "Hello, world!";
    file.close();

    ifstream input("example.txt");
    char str[30];
    while(!input.eof()) {
        input.getline(str, 30); // variable, size
        cout << str << endl;
    }
    input.close();

    return 0;
}
```

Binary I/O example:

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

class Person {
    public:
        int id, age;

        Person() {}
        Person(int i, int j) {
            id = i;
            age = j;
        }
};

int main() {
    int id, age;
    cout << "Enter id and age to insert: ";
    cin >> id >> age;

    Person p1(id, age);

    ofstream output("person.dat", ios::binary);
    output.write((char*)&p1, sizeof(Person)); // pointer to data of type char -> (char*)&obj
    output.close();

    Person p2;
    ifstream input("person.dat", ios::binary);
    if (input) {
        input.read((char*)&p2, sizeof(Person));
        cout << "ID: " << p2.id << endl;
        cout << "Age: " << p2.age << endl;
    }
    input.close();

    return 0;
}
```