2076

Created	@April 2, 2023 8:29 PM
Class	OOPL
Type	Exam Material
Materials	
Reviewed	✓

Group A

1. Write a program according to the specification given below:

- Create a class `Teacher` with data members `tid` & `subject` and member functions for reading and displaying data members.
- Create another class `Staff` with data members `sid` & `position`, and member function for reading and displaying data members.
- Derive a class `Coordinator` from `Teacher` and `Staff` and the class must have its own data member `department` and member functions for reading and displaying data members.
- Create `two` object of `Coordinator` class and read and display their details.

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

class Teacher {
    int tid;
    char subject[20];
    public:
        void getTeacher() {
            cout << "Enter Teacher ID and Subject: ";
            cin >> tid >> subject;
        }

        void displayTeacher() {
            cout << "Teacher ID: " << tid << endl;
            cout << "Subject: " << subject << endl;
        }
};

class Staff {
    int sid;
    char position[20];
    public:
        void getStaff() {
            cout << "Enter Staff ID and Position: ";
            cin >> sid >> position;
        }

        void displayStaff() {
            cout << "Staff ID: " << sid << endl;
            cout << "Position: " << position << endl;
        }
};
```

```
class Coordinator : public Teacher, public Staff {
    char department[20];
    public:
        void getCoordinator() {
            cout << "Enter Department: ";
            cin >> department;
        }

        void displayCoordinator() {
            cout << "Department: " << department << endl;
        }
};

int main() {
    Coordinator c1, c2;

    c1.getTeacher();
    c1.getStaff();
    c1.getCoordinator();
    c1.displayTeacher();
    c1.displayStaff();
    c1.displayCoordinator();

    c2.getTeacher();
    c2.getStaff();
    c2.getCoordinator();
    c2.displayTeacher();
    c2.displayStaff();
    c2.displayCoordinator();

    return 0;
}
```

---

2. Explain the concept of operator overloading? List the operators that cannot be overloaded. Write programs to add two object of distance class with data members feet and inch by using member function and friend function.

Operator overloading is a technique by which operators used in a programming language are implemented in user-defined types with customized logic that is based on the types of arguments passed. To overload an operator, we use a special `operator` function. We define the function inside the class or structure whose objects/variables we want the overloaded operator to work with.

Syntax:

```
class className {
    ... .. ...
    public
       returnType operator symbol (arguments) {
           ... .. ...
       }
    ... .. ...
};
```

Almost all operators can be overloaded except few. Following is the list of operators that cannot be overloaded.

- . (dot)
- :: (scope resolution)
- ?: (ternary conditional operator)
- sizeof

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

class Distance {
    int feet;
    int inch;

    public:
        Distance() {
            feet = 0;
            inch = 0;
        }
        Distance(int f, int i) {
            feet = f;
            inch = i;
        }

        // Member function to add two Distance objects
        void add(Distance d) {
            int f = feet + d.feet;
            int i = inch + d.inch;
            if (i >= 12) {
                f += 1;
                i -= 12;
            }
            cout << "Sum of distances: " << f << " feet " << i << " inches" << endl;
        }

        // Declare the friend function
        friend void add(Distance d1, Distance d2);
};


// Define the friend function
void add(Distance d1, Distance d2) {
    int f = d1.feet + d2.feet;
    int i = d1.inch + d2.inch;
    if (i >= 12) {
        f += 1;
        i -= 12;
    }
    cout << "Sum of distances: " << f << " feet " << i << " inches" << endl;
}


int main() {
    Distance d1(5, 10);
    Distance d2(4, 6);

    d1.add(d2); // calling member function
    add(d1, d2); // calling friend function

    return 0;
}
```

3. Explain types of polymorphism briefly. Write down roles of polymorphism. How can we achieve dynamic polymorphism briefly? Explain with example.

Polymorphism means that have many form. Polymorphism is a concept in object-oriented programming that allows objects of different classes to be treated as if they are objects of the same class. There are two main types of polymorphism in C++:

1. Compile-time polymorphism:

   This is also known as **static polymorphism**, and it involves resolving function calls at **compile-time**.

   This is done using function overloading or operator overloading.

   Function overloading is the practice of defining multiple functions with the same name but different parameter lists, and the appropriate function is chosen at compile-time based on the types of arguments passed to it.

   Operator overloading involves defining new meanings for existing operators, such as the `+` operator for adding two objects of a custom class.

2. Run-time polymorphism:

   This is also known as **dynamic polymorphism**, and it involves resolving function calls at **run-time**.

   This is achieved through virtual functions and inheritance.

   Virtual functions are functions declared in a base class that can be overridden in derived classes. When a virtual function is called on an object, the appropriate implementation is chosen based on the actual type of the object at run-time.

   Inheritance allows objects of derived classes to be treated as objects of the base class, which enables run-time polymorphism.

Dynamic polymorphism in C++ can be achieved through the use of virtual functions and inheritance. A virtual function is a function declared in a base class that can be overridden in derived classes. When a virtual function is called on an object, the appropriate implementation is chosen based on the actual type of the object at run-time.

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

class Shape {
    public:
        virtual void draw() {
            cout << "Drawing a shape." << endl;
        }
};

class Circle : public Shape {
    public:
        void draw() {
            cout << "Drawing a circle." << endl;
        }
};
```

```
class Rectangle : public Shape {
    public:
        void draw() {
            cout << "Drawing a rectangle." << endl;
        }
};

int main() {
    Shape* s;

    Circle c;
    Rectangle r;

    s = &c;
    s->draw(); // Calls Circle's draw()

    s = &r;
    s->draw(); // Calls Rectangle's draw()

    return 0;
}
```

Group B

4. How object oriented programming differs from object based programming language? Discuss benefits of OOP.

Object-oriented programming (OOP) and object-based programming (OBP) are two different approaches to programming that involve the use of objects. While they share some similarities, there are also some key differences between the two:

Object-Oriented Programming (OOP)	Object-based Programming (OBP)
Classes and objects are organized into a hierarchy with inheritance and polymorphism allowing for objects of different types to be treated as objects of the same type.	Objects are not organized into a hierarchy and do not have inheritance or polymorphism.
Object-oriented languages do not have the inbuilt objects.	Object-based languages have the inbuilt objects, for example, JavaScript has window object.
Example: C++, Java, and Python	Example: JavaScript, VBScript
It emphasizes **encapsulation,** which is the concept of hiding the internal details of an object and exposing only the public interface. **This helps to prevent unintended access and modification of an object's state.**	It may or may not provide encapsulation.
It also emphasizes **abstraction**, which is the concept of modeling real-world entities as objects with attributes and behaviors. **Abstraction allows for complex systems to be modeled in a simpler way, making it easier to reason about them.**	It may or may not provide abstraction.

Object-Oriented Programming (OOP)	Object-based Programming (OBP)
It provides **polymorphism**, which allows objects of different classes to be treated as if they are objects of the same class. **This allows for more flexibility and reusability in code.**	It may or may not provide polymorphism.

Benefits of OOP are as follows:

1. **Modularity:** OOP promotes modularity, which means breaking down a program into smaller, more manageable parts. This makes it easier to understand, maintain, and update code.
2. **Reusability:** OOP emphasizes code reuse, which means using existing code to create new software. This saves time and effort, as well as reduces the likelihood of introducing errors.
3. **Encapsulation**: OOP encourages encapsulation, which means hiding the internal details of an object and exposing only the public interface. This improves code security and reduces the likelihood of unintended access and modification of an object's state.
4. **Abstraction**: OOP promotes abstraction, which means modeling real-world entities as objects with attributes and behaviors. Abstraction makes it easier to reason about complex systems and makes code more maintainable.
5. **Polymorphism**: OOP provides polymorphism, which allows objects of different classes to be treated as if they are objects of the same class. This enables more flexibility and reusability in code.
6. **Inheritance**: OOP provides inheritance, which allows a subclass to inherit properties and behaviors from its superclass. This enables code reuse and makes it easier to maintain and update code.
7. **Flexibility**: OOP is highly flexible and adaptable, making it suitable for a wide range of applications and programming tasks.

---

5. What is the use of new and delete operators? Illustrate with example. What are advantages of new malloc.

In C++, `new` and `delete` operators are used for **dynamic memory allocation and deallocation**.

new: The new operator is used to allocate memory for a new object at runtime. It returns a pointer to the memory location allocated.

Syntax:

```
datatype *pointer_name = new datatype;
```

delete: The delete operator is used to deallocate the memory that was allocated by the new operator.

Syntax:

```
delete pointer_name;
```

Example:

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

int main() {
    // Allocate memory for an integer using the new operator
    int *ptr = new int;

    // Assign a value to the integer
    *ptr = 42;

    // Print the value of the integer
    cout << "The value of the integer is: " << *ptr << endl;

    // Deallocate the memory using the delete operator
    delete ptr;

    return 0;
}
```

In C++, new and malloc are both used for dynamic memory allocation, but they have some differences in terms of usage and functionality.

Advantages of new over malloc:

- `new` does not need the `sizeof()` operator where as `malloc()` needs to know the size before memory allocation.
- Operator `new` can make a call to a constructor where as `malloc()` cannot.
- `new` can be overloaded, `malloc()` can never be overloaded.
- `new` could initialize object while allocating memory to it where as `malloc()` cannot.
- If the `new` operator fails to allocate memory, it throws an exception, which can be handled using the `try-catch` mechanism.

Advantages of malloc over new:

- `malloc()` is a C standard library function and can be used in C++ as well, making it more compatible with older C code.
- `malloc()` allows for allocation of memory blocks of any size, while `new` only allocates memory for objects of a specific type.
- `malloc()` is generally faster than `new` due to its simplicity.

6. What is destructor? Write a program to show the destructor call such that it prints the message “memory is released”.

Destructors are special members functions in a class that delete an object. They are called when the class object goes out of scope such as when the function ends, the program ends, etc.

It is used to clean up the resources that were allocated to the object during its lifetime.

The destructor has the same name as the class, but with a tilde (~) symbol in front of it.

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

class MyClass {
    public:
        MyClass() {
            cout << "Memory Occupied!" << endl;
        }

        ~MyClass() {
            cout << "Memory Released!" << endl;
        }
};

int main() {
    MyClass obj;

    return 0;
}
```

---

7. What is this pointer? How can we use it for name conflict resolution? Illustrate with example.

In C++, the `this` pointer is a special pointer that points to the object for which the member function is called. It is used to refer to the current object.

- used to pass current object as a parameter to another method.
- can be used to refer current class instance variable.
- used to declare indexers.

In C++, the `this` pointer can be used for name conflict resolution when a local variable or a function parameter has the same name as a member variable of a class. This can be resolved by using the `this` pointer to explicitly refer to the member variable.

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

class MyClass {
    int x;
    public:
        void setX(int x) {
            this->x = x; // using this pointer to refer to the member variable
        }
        void printX(int x) {
            cout << "Local x = " << x << endl; // printing the local variable
            cout << "Member x = " << this->x << endl; // using this pointer to refer to the member variable
        }
};

int main() {
    MyClass obj;
    obj.setX(42);
    obj.printX(99);
    return 0;
}
```

OR:

```
// this is easier than of above
#include <iostream>
using namespace std;

class Distance{
    float inch;

    public:
        void setValue(float inch){
            this->inch = inch;
        }

        void display(){
            cout << "Distance: " << inch;
        }
};

int main(){
    Distance d1;
    d1.setValue(10);
    d1.display();
    return 0;
}
```

In second example, in `setValue()` function we have passed inch as parameter and we are setting value also on member variable `inch`. In this case, compiler get confused because passed parameter and set variable name are same. So to fix this issue, we will use `this` pointer.

---

8. How can you define catch statement that can catch any type of exception? Illustrate the use of multiple catch statement with example.

Exceptions are run-time anomalies or abnormal conditions that a program encounters during its execution.

To catch any type of exception in C++, we can use an ellipsis (`...`) in the catch block, which is known as the catch-all block. The catch-all block must be placed at the end of the list of catch blocks, as it can catch any type of exception, including system-level exceptions.

We can use catch multiple times to catch multiple types of exception.

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

```
int main() {
    try  {
       throw 10;
    }
    //catching character exception
    catch (char c)  {
        cout << "Caught " << c;
    }
    //catching all exception
    catch (...){
        cout << "Exception Occured\n";
    }
    return 0;
}
```

---

9. Which functions can be used for reading and writing object? Describe briefly. Write a program that read values of two objects of student class(assume data members are sid , sname, and level) and display the data in monitor.

Reading and writing data to and from files requires another standard library of C++ `<fstream>`. The three main data types of `fstream` are:

- **ifstream** − represents input file stream and reads information from files.
- **ofstream** − represents output file stream and writes information to files.
- **fstream** − represents general file stream and has capabilities of both.

```
file1.write((char*)&emp1, sizeof(Emp) ); // to write object to the file
file2.read((char*)&emp2, sizeof(Emp) ); // to read object from the file
```

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

class Student {
    int sid;
    char sname[30], level[30];
    public:
        void setData() {
            cout << "Enter id, name and level: ";
            cin >> sid >> sname >> level;
        }

        void getData() {
            cout << "\nID: " << sid;
            cout << "\nName: " << sname;
            cout << "\nLevel: " << level;
        }
};

class StudFile {
    public:
        void addStud(Student s) {
            ofstream outfile;
            /* opening in app mode becz we want to store 2 obj,
            if not opened in app mode the first obj will be replaced by second */
            outfile.open("students.txt", ios::app);
            outfile.write((char*)&s, sizeof(Student));
            outfile.close();
        }

        void getStud() {
            ifstream infile;
            infile.open("students.txt");
            Student sd;

            while (!infile.eof()) {
                infile.read((char*)&sd, sizeof(Student));
                if( infile.eof() ) break;
                sd.getData();
            }

            infile.close();
        }
};

int main() {
    StudFile sf;
    Student s1, s2;

    s1.setData();
    sf.addStud(s1);
    s2.setData();
    sf.addStud(s2);

    sf.getStud();

    return 0;
}
```

```
OUTPUT:
Enter id, name and level: 1 mansij batch
Enter id, name and level: 2 ankita batch

ID: 1
Name: mansij
Level: batch
ID: 2
Name: ankita
Level: batch
```

---

(10) 6. What is meant by return by reference? How can we return values by reference by using reference variable? Illustrate with examples. (was not there in hamrocsit.com)

Return by reference means returning the memory address of a variable instead of its value. When a function returns a reference, the caller can use this reference to access the original data directly without creating a copy of it.

Syntax:

```
int& multiply(int& x, int& y) {
    int result = x * y;
    return result;
}
```

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

int& get_larger(int& a, int& b) {
    // cout << &a << " " << &b << endl;
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

int main() {
    int x = 5;
    int y = 10;

    int& larger = get_larger(x, y);

    // cout << larger << endl; // 10
    // cout << &larger;

    larger = 20; // setting larger to 20

    cout << "x: " << x << endl; // Output: x: 5
    cout << "y: " << y << endl; // Output: y: 20

    return 0;
}
```

---

11. Write short notes on:

- **Cascading of IO operators**

  When an object calls an operator function by passing an argument and the returned value of the operator function calls the next operator function in the same expression, it is called as **cascading of operators.**

```
// Cascading IO operators

#include <iostream>
using namespace std;

class Height {
  int feet, inches;

    public:
        // default constructor
        Height() {
            feet = 0;
            inches = 0;
        }

        // parameterized constructor
        void setData(int x, int y) {
            feet = x;
            inches = y;
        }

        void showData() {
            cout << feet << "'" << inches;
        }

        // Function for overloading of operator +
        Height operator+(Height H) {
            Height temp;

            temp.feet = feet + H.feet;
            temp.inches = inches + H.inches;

            return temp;
        }

        // Function to normalize the height
        // into proper terms of 1 feet
        // per 12 inches
        void normalize() {
            // Update the feets
            if (inches >= 12) {
                feet += 1;
                inches -= 12;

                // OR: feet = feet + inches / 12;
            }
        }
};

int main() {
  Height h1, h2, h3, h4;

  // Initialize the three heights objects
  h1.setData(5, 9);
  h2.setData(5, 2);
  h3.setData(6, 2);

  // Add all the heights using cascading of operators
  h4 = h1 + h2 + h3;

  // Normalize the heights
  h4.normalize();

  // Print the height h4
  h4.showData();

  return 0;
}
```

Here at first `h1` object called `(+)` operator and passes `h2` as an argument in the operator function call and the `returned value` of this operator function calls again `(+)` operator and passes `h3` as an argument in the same expression, at last, the `returned value` of this second operator function is assigned in `h4`.

- **Pure Virtual Function**

  Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation. Such a class is called **abstract class**. For example, let `Shape` be a base class. We cannot provide implementation of function `draw()` in `Shape`, but we know every derived class must have implementation of `draw()`. Similarly an `Animal` class doesn't have implementation of `move()` (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.

A pure virtual function (or abstract function) in C++ is a virtual function for which we can have implementation, But we must override that function in the derived class, otherwise the derived class will also become abstract class. A class is abstract if it has at least one pure virtual function.

```
// An abstract class
class Test {
  public:
  // Pure Virtual Function
  virtual void show() = 0;
};
```

A pure virtual function is implemented by classes which are derived from a Abstract class. Following is a simple example to demonstrate the same.

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

class Base {
    int x;
    public:
        virtual void fun() = 0;
        int getX() {
            return x;
        }
};

// This class inherits from Base and implements fun()
class Derived: public Base {
  int y;
    public:
        void fun() {
            cout << "fun() called";
        }
};

int main(void) {
  Derived d;
  d.fun(); // OUTPUT: fun() called

  return 0;
}
```