2078

Created	@April 4, 2023 7:03 PM
Class	OOPL
Type	Exam Material
Materials	
Reviewed	☑

Group A

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

- Create a class Account with data members `accno`, `balance`, and `min_balance(static)`
- Include `methods` for reading and displaying values of objects
- Define static member function to display `min_balance`.
- Create `array` of objects to store data of `5` accounts and `read` and `display` values of each object.

```
#include <iostream>
using namespace std;
#define SIZE 5

class Account{
    int acc_no;
    int balance;
    public:
        //static data member
        static int min_balance;

        //read the data
        void readData(){
            cout << "\nEnter Account No: ";
            cin >> acc_no;

            cout << "Enter balance: ";
            cin >> balance;
        }

        //print data
        void displayData(){
```

```
            cout << "Account No: " << acc_no << endl;
            cout << "Balance: " << balance << endl;
        }

        //read minimum balance
        static void readMinBalance(){
            cout << "\nEnter Minimum Balance: ";
            cin >> min_balance;
        }

        //static function to print static member
        static void dispMinBalance(){
            cout << "Minimum Balance: " << min_balance << endl << endl;
        }
};

//Initialize static data member
int Account::min_balance = 0;

int main(){

    Account acc[SIZE];
    int i = 0;

    //Read the account data
    Account::readMinBalance();

    cout << "\n=======Enter Account Data=======" << endl;
    for(i = 0; i < SIZE; i++){
        acc[i].readData();
    }

    //print the account data
    cout << "\n=======Account Details=======\n" << endl;
    for(i = 0; i < SIZE; i++){
        acc[i].displayData();
        Account::dispMinBalance();
    }

    return 0;
}
```

2. What is meant by type conversion? Define two way of converting one user defined data type (object) to another user defined object? Write a program that converts object of another distance class with data members feet and inch.(Assume 1m = 3.3 feet and 1cm = 0.4 inch)

Type conversion in C++ is the process of **converting a variable or value of one data type to another data type**. There are two types of type conversion in C++:

`implicit` and `explicit`.

1. **Implicit Type Conversion:**

   Implicit type conversion is **done automatically by the compiler**. This happens when you assign a value of one data type to a variable of another data type. For example, if you assign an integer value to a float variable, the compiler will automatically convert the integer value to a float value.

```
int num1 = 10;
float num2 = num1; // Implicit conversion from int to float
```

2. Explicit Type Conversion:

   Explicit type conversion, **also known as** `type casting`, is **done manually by the programmer**. In this method, the programmer tells the compiler to convert a value of one data type to another data type. Explicit type conversion can be done using casting operators.

```
int num_int = 26; // initializing int variable
double num_double; // declaring double variable

// converting from int to double
num_double = (double)num_int; // c-style type casting
num_double = double(num_int); // function-style type casting
```

When we assign an object of a class into the object of another class then it is called as class to class conversion. In C++, there are **two ways** to convert one user-defined data type (object) to another user-defined object. 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**.

NOTE: 1m = 100cm

1. Using casting operator in source class

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

// destination class
class FeetI {
```

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

        void display() {
            cout << feet << "'" << inch << "''" << endl;
        }
};

// source class
class MeterC {
    double meter, centi;
    public:
        MeterC(double m=0, double c=0) {
            meter=m; centi=c;
        }

        void display() {
            cout << meter << " meters and " << centi << " centimeter" << endl;
        }

        // MeterC = FeetI garda yo function trigger huncha
        operator FeetI() {
            double totalCenti = (100*meter) + centi;

            // formulas
            // didn't use double inorder to not get 4.9' 11.0551' :)
            int inch = totalCenti / 2.54;
            int feet = inch / 12;

            inch = inch - (12*4);

            FeetI tfeetI(feet, inch);

            return tfeetI;
        }
};

int main() {
    MeterC mc(1, 50); // 1m 50cm == 150cm
    FeetI fi;

    fi = mc;

    fi.display();

    return 0;
}

/*
OUTPUT:
4'11''
*/
```

2. Using constructor in destination class

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

// source class
class MeterC {
    double meter, centi;
    public:
        MeterC(double m=0, double c=0) {
            meter=m; centi=c;
        }

        void display() {
            cout << meter << " meters and " << centi << " centimeter" << endl;
        }

        double getMeter() { return meter; }
        double getCenti() { return centi; }
};

// destination class
class FeetI {
    int feet, inch;
    public:
        FeetI(int f=0, int i=0) {
            feet=f; inch=i;
        }

        void display() {
            cout << feet << "'" << inch << "''" << endl;
        }

        // MeterC = FeetI triggers this function
        FeetI(MeterC a) {
            double totalCenti = (100* (a.getMeter())) + a.getCenti();

            // formulas
            // didn't use double inorder to not get 4.9' 11.0551' :)
            int tinch = totalCenti / 2.54;
            feet = tinch / 12;

            inch = tinch - (12*4);
        }
};

int main() {
    MeterC mc(1, 50); // 1m 50cm == 150cm
    FeetI fi = mc;

    fi.display();

    return 0;
}

/*
```

```
OUTPUT:
4'11''
*/
```

3. Using casting operator in source class (number 1) **BUT** with the given values(formula) in the question

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

// destination class
class FeetI {
    int feet, inch;
    public:
        FeetI(int f=0, int i=0) {
            feet=f; inch=i;
        }

        void display() {
            cout << feet << "'" << inch << "''" << endl;
        }
};

// source class
class MeterC {
    double meter, centi;
    public:
        MeterC(double m=0, double c=0) {
            meter=m; centi=c;
        }

        void display() {
            cout << meter << " meters and " << centi << " centimeter" << endl;
        }

        // MeterC = FeetI triggers this function
        operator FeetI() {
            double totalCenti = (100*meter) + centi;

            // formulas
            int feet = 3.3 * meter;
            int inch = 0.4 * centi;

            if (inch >= 12) {
                feet += 1;
                inch -= 12;
            }

            FeetI tfeetI(feet, inch);

            return tfeetI;
        }
};
```

```
int main() {
    MeterC mc(1, 50); // 1m 50cm == 150cm
    FeetI fi;

    fi = mc;

    fi.display();

    return 0;
}

/*
OUTPUT:
4'8''
*/
```

---

3. How ambiguity arises in multipath inheritance? How can you remove this type of ambiguity? Explain with suitable example.

In multipath inheritance, a **derived class inherits from two or more base classes**, which can lead to ambiguity **if the base classes have members with the same name**.

Example:

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

class A {
public:
    void show() {
        cout << "Hello from A \n";
    }
};

class B : public A {};

class C : public A {};

class D : public B, public C {};

int main() {
    D d;
    d.show();

    return 0;
}
```

Since both `B` and `C` classes inherit from class `A`, and have their own versions of the `show()` function, we need to explicitly specify which version of the `show()` function to use or use **virtual inheritance, meaning that only one instance of the class should be present in the inheritance hierarchy, even if the class is inherited multiple times**. If we do not do so, it will lead to ambiguity.

1. Using Virtual Inheritance

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

class A {
public:
    void show() {
        cout << "Hello from A \n";
    }
};

class B : public virtual A {};

class C : public virtual A {};

class D : public B, public C {};

int main() {
    D d;
    d.show();

    return 0;
}
```

2. Using Scope Resolution Operator

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

class A {
    public:
        void func() {
            cout << "I am in class A" << endl;
        }
};

class B {
    public:
        void func() {
            cout << "I am in class B" << endl;
        }
};

class C: public A, public B {};
```

```
int main() {
    C c;

    // Calling function func() in class A
    c.A::func();

    // Calling function func() in class B
    c.B::func();

    return 0;
}
```

Group B

4. What is structured programming? Discuss characteristics and problems associated with structured programming.

Structure Programming is a programming language that contain well-structured steps and procedures that uses different functions for different tasks in a program. In structured programming, a program is broken down into smaller, more manageable modules that can be easily understood, modified, and tested.

Characteristics of Structured Programming Language

- Structured programming emphasizes **top-down design**, where a program is designed from the main module to sub-modules, and sub-modules are designed from general to specific.
- Structured programming emphasizes **modular programming**, where a **program is divided into smaller modules, and each module performs a specific task.**
- Structured programming emphasizes **clear control flow**, where a program follows a sequential flow.
- Structured programming **uses structured control statements** such as `if-else`, `do-while`, and `switch-case` statements to control the flow of the program.

Problems Associated with Structured Programming Language:

- **Difficulty in Handling Complex Programs**: Structured programming can be challenging to apply to complex programs that require more extensive use of

control structures.

- **Code Repetition**: Structured programming can sometimes lead to code repetition because of the need to define functions or modules for each specific task.
- **Maintenance Issues**: Although structured programming can help improve the maintainability of a program, it can lead to some maintenance issues that can make it challenging to maintain and modify a program.

---

5. What is the use of `get` and `getline` functions? Explain with suitable example.

- `get()`:
  - `cin.get()` is used for accessing character array.
  - It includes white space characters.
  - Generally, `cin` with an extraction operator (`>>`) terminates when whitespace is found.
  - However, `cin.get()` reads a string with the whitespace.

  Syntax:

  ```
  cin.get(string_name, size);
  ```

  Example:

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

  int main() {
      char name[25];
      cin.get(name, 25);
      cout << name;

      return 0;
    }
  ```

- `getline()`:

- The `cin` is an object which is used to take input from the user but does not allow to take the input in multiple lines.
- To accept the multiple lines, we use the `getline()` function.
- It is a pre-defined function defined in a `<string.h>` header file used to accept a line or a string from the input stream until the delimiting character is encountered.

Syntax:

```
cin.getline(string_name, size); // without delimiting character
cin.getline(string_name, size, delimiting_character); // with delimiting character
```

Example:

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

int main() {
    char name[40];

    cout << "Enter your name: ";
    cin.getline(name, 40); // without delimiting character
    cout << "Hello " << name << endl;

    cout << "Enter your name: ";
    cin.getline(name, 40, ' '); // with delimiting character
    cout << "Hello " << name << endl;

    return 0;
}
```

---

6. What is meant by pass by reference? How can we pass arguments by reference by using reference variable? Illustrate with example.

Pass by Reference:

Pass by reference is something that C++ developers use to allow a function to modify a variable without having to create a copy of it.

To pass a variable by reference, we have to declare function parameters as references and not normal variables.

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

void duplicate(int& b){
    b*=2;
}

int main() {
    int x = 25;

    cout << "The original value of x is " << x << endl;
    duplicate(x);
    cout << "The new value of x is " << x;

    return 0;
}
```

---

7. What is constructor? Explain the concept of default and default copy with suitable example.

- Constructor:

  A constructor in C++ is a special **member function or method** having the same name as that of its class which is used to initialize some valid values to the data members of an object. It is executed automatically whenever an object of the class is created.

- Default Constructor:

  A Default Constructor is a constructor which has no arguments.

- Default Copy Constructor:

  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.

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

class Exam {
    double marks;
    public:
        // dafault constructor
        Exam() {
            marks = 40.5;
        }
```

```
        void getVal() {
            cout << "Enter marks: ";
            cin >> marks;
        }

        void display() {
            cout << "Marks: " << marks << endl;
        }
};

int main() {
    // default constructor example
    Exam e1;
    e1.display();

    // default copy constructor example
    Exam e2;
    e2.getVal();

    Exam e3 = e2; // or Exam e3(e2);
    e3.display();

    return 0;
}
```

---

8. What is the concept of friend function? How it violates the data hiding principle? Justify with example.

In C++, a friend function is a function that is not a member of a class but has access to the `private` and `protected` members of that class. **Friend functions are declared inside the class, but their definition is outside the class**.

Friend functions violate the data hiding principle **because they can access private and protected members of a class**, **which are intended to be hidden from outside the class**. **This can lead to security issues and make the code harder to maintain.**

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

class MyClass {
    int x;

    public:
        MyClass(int val) {
            x = val;
        }
```

```
        friend void printX(MyClass obj); // Declaration of a friend function
};

void printX(MyClass obj) { // defination of a friend function
    cout << obj.x << endl; // Accessing private member x of MyClass
}

int main() {
    MyClass obj(10);
    printX(obj); // Output: 10

    return 0;
}
```

---

9. What is exception? Why exception handling is better to use? Explain exception handling with try….. catch by using suitable example.

Exceptions are run-time anomalies or abnormal conditions that a program encounters during its execution. An exception is an error or an unexpected event that occurs during the execution of a program. Exceptions can occur due to various reasons such as incorrect input, insufficient memory, or unexpected external events.

It is better to use because of following reasons:

- Exception handling can control run tine errors that occur in the program.
- It can avoid abnormal termination of the program and also shows the behavior of program to users.
- It can provide a facility to handle exceptions, throws message regarding exception and completes the execution of program by catching the exception.
- It can separate the error handling code and normal code by using try-catch block.
- It can produce the normal execution flow for a program.
- It can implement a clean way to propagate error. When an invoking method cannot manage a particular situations, then it throws an exception and asks the invoking method to deal with such situation.
- It develops a powerful coding which ensures that the exceptions can be prevented.

- It also allows to handle related exceptions by single exception handler. All the related errors are grouped together by using exceptions. And then they are handled by using single exception handler.
- Exceptions provide a way to transfer control from one part of a program to another.

C++ exception handling is built upon three keywords: `try`, `catch`, and `throw`.

- **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.
- **throw**: A program throws an exception when a problem shows up. This is done using a `throw` keyword.
- **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.

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

int main () {
    int i = 24;
    int j = 2;
    float k;
    try {
        if (j==0) {
            throw "Can't divide by 0 idiot.";
        }
        else {
            k = i/j;
            cout << k << endl;
        }
    } catch (const char* e) {
        cerr << "Error: " << e << endl;
    }
    return 0;
}
```

---

10. When class templates are useful? How can you define a class that can implement stack with integer as well as sack of strings? Illustrate with example.

Class templates are useful when we want to create a class that can work with multiple types of data. They allow us to define a generic(not specific) class that can work with different types of data without having to write separate code for each type.

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

template<class T>
class show {
    T a, b; // attributes with datatype T, T supports any datatype
    public:
        show(T p, T q) {
            a=p; b=q;
        }

        void get() {
            cout << "a: " << a << "\tb: " << b << endl;
        }
};

int main() {
    show<char> objC('a', 'e'); // class_name<data_type> class_object
    show<int> objI(1, 2);

    objC.get();
    objI.get();

    return 0;
}
```

---

11. What is meant by stream? Write a program that reads content of file data.txt and displays the content in monitor.

Streams in C++ are an important abstraction that provide a uniform way of working with input and output data from various sources and destinations.

- `istream`**:** It is a general purpose **input stream**. `cin` is an example of an `istream`.
- `ostream`**:** It is a general purpose **output stream**. `cout` is an examples of `ostreams`.
- `ifstream`**:** It is an **input file stream**. It is a special kind of an `istream` that reads in data from a data file.
- `ofstream`**:** It is an **output file stream**. It is a special kind of `ostream` that writes data out to a data file.

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

int main() {
    fstream file;
    file.open("data.txt");

    char ch[50];
    while(!file.eof()) {
        file.getline(ch, 50);
        cout << ch << endl;
    }

    file.close();

    return 0;
}
```

---

12. Write short notes on:

- **Manipulators**:

  Manipulators are operators that are used to format the data display. Manipulators are helping functions that can modify the input/output stream. It does not mean that we change the value of a variable, it only modifies the I/O stream using insertion (`<<`) and extraction (`>>`) operators.

  - `setw`: The setw manipulator helps to shift the output statements to **right.** It is used for shifting output rightwards according to need. While using `setw()` we should include `<iomanip>` i.e. `#include<iomanip>` preprocessor file.

    ```
    cout << setw(5) << sum << endl;
    ```

    The manipulator `setw(5)` in above example will right shift the output value of sum in the screen.

  - `endl`: The endl manipulator works same as the `"\n"` (newline). The endl manipulator when used in an output statement, causes a linefeed to be inserted.

```
cout << "m=1" << endl;
cout << "n=2" << endl;
```

This `endl` in above example will cause to print `m=1` in one line and `n=2` in next line.

- **Protected Access Specifier**:

  In C++, the protected access specifier is one of the three access specifiers that can be used to control the visibility of class members from outside the class. Members declared as `protected` are accessible from within the class itself and its derived classes, but not from outside the class or its derived classes. When a class member is declared as `protected`, it means that it can be accessed by any member function of the class and any member function of its derived classes. This is useful when you want to restrict access to a member variable or member function to the class itself and its derived classes, but not to outside code.

```
class MyClass {
  protected:
    int myProtectedVariable;
  public:
    void setProtectedVariable(int value) {
      myProtectedVariable = value;
    }
};

class MyDerivedClass : public MyClass {
  public:
    void doSomething() {
      myProtectedVariable = 42; // Accessible from derived class
    }
};

int main() {
   MyClass myObj;
   myObj.myProtectedVariable = 42; // Error: not accessible from outside class
}
```
