|  |  |  |  |
| --- | --- | --- | --- |
| Date: | **26-06-2020** | Name: | **Poojary Sushmita** |
| Course: | **C ++ programming** | USN: | **4AL16EC046** |
| Topic: | * Composition * Friend function etc | Semester & Section: | **6th & ‘B’** |
| GitHub Repository: | **Sushmita-s** |  |  |

**Composition:** In C++, object composition involves using classes as member variables in other classes.  
This sample program demonstrates composition in action. It contains **Person**and **Birthday**classes, and each **Person**will have a **Birthday**object as its member. Let's also add a **printDate()**function to our Birthday class: Next, we can create the **Person**class, which includes the **Birthday**class. The Person class has a **name**and a **Birthday**member, and a constructor to initialize them.  
Ensure that the corresponding header files are included. Let's add a **printInfo()**function to our Person class, that prints the data of the object.

|  |  |
| --- | --- |
| **Birthday.h**  class Birthday {  public:  Birthday(int m, int d, int y)  : month(m), day(d), year(y)  {  }  void printDate()  {  cout<<month<<"/"<<day  <<"/"<<year<<endl;  }  private:  int month;  int day;  int year;  }; | #include <string>  #include "Birthday.h"  class Person {  public:  Person(string n, Birthday b)  : name(n),  bd(b)  {  }  void printInfo()  {  cout << name << endl;  bd.printDate();  }  private:  string name;  Birthday bd;  }; |

Composition is used for objects that share a **has-a** relationship, as in "A **Person**has a **Birthday**". Notice that we can call the **bd**member's **printDate()** function, since it's of type **Birthday**, which has that function defined. Now that we've defined our **Birthday**and **Person**classes, we can go to our main, create a **Birthday**object, and then pass it to a **Person**object.

|  |
| --- |
| int main() {  Birthday bd(2, 21, 1985);  Person p("David", bd);  p.printInfo();  }  /\*Outputs  David  2/21/1985  \*/ |

In general, **composition**serves to keep each individual class relatively simple, straightforward, and focused on performing one task. It also enables each sub-object to be self-contained, allowing for reusability (we can use the **Birthday**class within various other classes).

**Friend Function:** Normally, private members of a class cannot be accessed from outside of that class.  
However, declaring a **non-member**function as a **friend**of a class allows it to access the class' private members. This is accomplished by including a declaration of this external function within the class, and preceding it with the keyword **friend**. In the example below, **someFunc()**, which is not a member function of the class, is a friend of **MyClass**and can access its private members.

|  |  |
| --- | --- |
| class MyClass {  public:  MyClass() {  regVar = 0;  }  private:  int regVar;    friend void someFunc(MyClass &obj);  };  void someFunc(MyClass &obj) {  obj.regVar = 42;  cout << obj.regVar;  } | int main() {  MyClass obj;  someFunc(obj);  }  //Outputs 42 |

Note that when passing an object to the function, we need to pass it **by reference**, using the & operator. The function **someFunc()** is defined as a regular function outside the class. It takes an object of type **MyClass**as its parameter, and is able to access the private data members of that object. We can create an object in main and call the **someFunc()** function. Similar to friend functions, you can define a **friend class**, which has access to the private members of another class.

**This Pointers:** Every object in C++ has access to its own address through an important pointer called the **this**pointer. 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. The **printInfo()** method offers three alternatives for printing the member variable of the class.

|  |
| --- |
| class MyClass {  public:  MyClass(int a) : var(a)  { }  void printInfo() {  cout << var<<endl;  cout << this->var<<endl;  cout << (\*this).var<<endl;  }  private:  int var;  }; |

All three alternatives will produce the same result. **this**is a **pointer**to the object, so the arrow selection operator is used to select the member variable.

**Operator Overloading**: Most of the C++ built-in operators can be redefined or **overloaded**. Thus, operators can be used with user-defined types as well (for example, allowing you to **add**two objects together). This chart shows the operators that can be overloaded.

  
 (Operators that can't be overloaded include :: | .\* | . | ?: )Overloaded operators are functions, defined by the keyword **operator**followed by the symbol for the operator being defined. An overloaded operator is similar to other functions in that it has a **return type** and a**parameter list**. In our example we will be overloading the **+ operator**. It will **return**an object of our class and take an object of our class as its **parameter**. We need our + operator to return a new **MyClass**object with a member variable equal to the sum of the two objects' member variables.

|  |  |
| --- | --- |
| class MyClass {  public:  int var;  MyClass() {}  MyClass(int a)  : var(a) { }  MyClass operator+(MyClass &obj) {  MyClass res;  res.var= this->var+obj.var;  return res;  }  }; | int main() {  MyClass obj1(12), obj2(55);  MyClass res = obj1+obj2;  cout << res.var;  }  //Outputs 67 |

Here, we declared a new **res**object. We then assigned the sum of the member variables of the current object (**this**) and the parameter object (**obj**) to the **res**object's var member variable. The **res**object is returned as the result. This gives us the ability to create objects in main and use the overloaded + operator to add them together.

**Inheritance:** The class whose properties are inherited by another class is called the **Base** class. The class which inherits the properties is called the **Derived** class. The derived class inherits all feature from the base class, and can have its own additional features. The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS-A mammal, hence dog IS-A animal as well. This syntax derives the **Daughter**class from the **Mother**class.

|  |
| --- |
| **class Daughter : public Mother**  {  public:  Daughter() {};  }; |

The Base class is specified using a **colon**and an **access specifier**:**public**means, that all public members of the base class are public in the derived class. As all public members of the Mother class become public members for the Daughter class, we can create an object of type Daughter and call the **sayHi()** function of the Mother class for that object:

|  |
| --- |
| #include <iostream>  using namespace std;  class Mother  {  public:  Mother() {};  void sayHi() {  cout << "Hi";  }  };  class Daughter: public Mother  {  public:  Daughter() {};  };  int main() {  Daughter d;  d.sayHi();  }  //Outputs "Hi" |

A class can be derived from multiple classes by specifying the base classes in a **comma-separated** list. A derived class inherits all base class methods with the following exceptions:

- Constructors, destructors

- Overloaded operators

- The friend functions