|  |  |  |  |
| --- | --- | --- | --- |
| Date: | **27-06-2020** | Name: | **Varun G Shetty** |
| Course: | **C ++ programming** | USN: | **4AL17EC093** |
| Topic: | * Operating with files * Virtual function | Semester & Section: | **6th & ‘B’** |
| GitHub Repository: | **Varunshetty4** |  |  |

**Type of Inheritance:** Access specifiers are also used to specify the **type of inheritance**.

**Public**Inheritance: public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

**Protected**Inheritance: public and protected members of the base class become protected members of the derived class.

**Private**Inheritance: public and protected members of the base class become private members of the derived class.

Public inheritance is the most commonly used inheritance type. If no access specifier is used when inheriting classes, the type becomes **private**by default. When inheriting classes, the base class' constructor and destructor are not inherited. However, they are being called when an object of the derived class is created or deleted. let's create a sample class that includes a constructor and a destructor. Next, let's create a **Daughter**class, with its own constructor and destructor, and make it a derived class of the **Mother**.

|  |  |  |
| --- | --- | --- |
| class Mother {  public:  Mother()  {  cout <<"Mother ctor"<<endl;  }  ~Mother()  {  cout <<"Mother dtor"<<endl;  }  }; | class Daughter: public Mother {  public:  Daughter()  {  cout <<"Daughter ctor"<<endl;  }  ~Daughter()  {  cout <<"Daughter dtor"<<endl;  }  }; | int main() {  Daughter m;  }  /\*Outputs  Mother ctor  Daughter ctor  Daughter dtor  Mother dtor  \*/ |

Note that the base class' constructor is called first, and the derived class' constructor is called next. When the object is destroyed, the derived class's destructor is called, and then the base class' destructor is called.

**Polymorphism:** The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by **inheritance**. C++ polymorphism means that a call to a member function will cause a **different**implementation to be executed depending on the **type**of object that invokes the function. Simply, polymorphism means that a single function can have a number of different implementations.

Suppose you want to make a simple game, which includes different enemies: monsters, ninjas, etc. All enemies have one function in common: an **attack**function. However, they each attack in a different way. In this situation, polymorphism allows for calling the same **attack**function on different objects, but resulting in different behaviors. The first step is to create the **Enemy**class.

Our second step is to create classes for two different types of enemies, **Ninjas**and **Monsters**. Both of these new classes inherit from the **Enemy**class, so each has an attack power. At the same time, each has a specific **attack**function. As you can see, their individual **attack**functions differ.  
Now we can create our **Ninja**and **Monster**objects in main. **Ninja**and **Monster**inherit from **Enemy**, so all **Ninja**and **Monster**objects are **Enemy**objects.

|  |  |  |
| --- | --- | --- |
| class Enemy {  protected:  int attackPower;  public:  void setAttackPower(int a){  attackPower = a;  }  }; | class Ninja: public Enemy {  public:  void attack() {  cout << "Ninja! - "<<attackPower<<endl;  }  };  class Monster: public Enemy {  public:  void attack() {  cout << "Monster! - "<<attackPower<<endl;  }  }; | int main() {  Ninja n;  Monster m;  Enemy \*e1 = &n;  Enemy \*e2 = &m;  e1->setAttackPower(20);  e2->setAttackPower(80);  n.attack();  m.attack();  }  /\* Output:  Ninja! - 20  Monster! - 80  \*/ |

**Virtual Function**: Continuing to above example, To be able to call the corresponding attack() function for each of the derived classes using Enemy pointers, we need to declare the base class function as **virtual**. Defining a virtual function in the base class, with a corresponding version in a derived class, allows polymorphism to use Enemy pointers to call the derived classes' functions. Every derived class will override the attack() function and have a separate implementation. A virtual function is a base class function that is declared using the keyword **virtual**.

|  |  |
| --- | --- |
| class Enemy {  public:  virtual void attack() {  }  };  class Ninja: public Enemy {  public:  void attack() {  cout << "Ninja!"<<endl;  }  };  class Monster: public Enemy {  public:  void attack() {  cout << "Monster!"<<endl;  }  }; | int main() {  Ninja n;  Monster m;  Enemy \*e1 = &n;  Enemy \*e2 = &m;  e1->attack();  e2->attack();  }  /\* Output:  Ninja!  Monster!  \*/ |

As the attack() function is declared virtual, it works like a template, telling that the derived class might have an **attack()** function of its own.  We are using **Enemy**pointers to call the same **attack()** function, and generating different results. If a function in the base class is **virtual**, the function's implementation in the derived class is called according to the actual type of the object referred to, regardless of the declared type of the pointer. A class that declares or inherits a virtual function is called a **polymorphic**class.

Virtual functions can also have their implementation in the base class. Now, when you create an **Enemy**pointer, and call the **attack()** function, the compiler will call the function, which corresponds to the object's type, to which the pointer points(the object address saved in the ponter it may be base class object).

**Pure Virtual function**: In some situations, you'd want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class. The virtual member functions without definition are known as **pure virtual functions**. They basically specify that the derived classes define that function on their own. The syntax is to replace their definition by =0 (an equal sign and a zero):

virtual void attack() = 0; -> The = 0 tells the compiler that the function has no body.

Every derived class inheriting from a class with a pure virtual function **must** override that function. If the pure virtual function is not overridden in the derived class, the code fails to compile and results in an error when you try to instantiate an object of the derived class.

**Abstract Class:** You **cannot**create objects of the base class with a pure virtual function. These classes are called **abstract**. They are classes that can only be used as base classes, and thus are allowed to have pure virtual functions.  It can be used to create pointers and take advantage of all its polymorphic abilities.

**Function Templets**: With function templates, the basic idea is to avoid the necessity of specifying an exact type for each variable. Instead, C++ provides us with the capability of defining functions using placeholder types, called **template type parameters**. To define a function template, use the keyword **template**, followed by the template type definition:

**template** <class T>

We named our template type **T**, which is a generic data type.

|  |  |
| --- | --- |
| template <class T>  T sum(T a, T b) {  return a+b;  }  int main () {  int x=7, y=15;  cout << sum(x, y) << endl;  }  // Outputs 22 | template <class T>  T sum(T a, T b) {  return a+b;  }  int main () {  double x=7.15, y=15.54;  cout << sum(x, y) << endl;  }  // Outputs 22.69 |

The function returns a value of the generic type T, taking two parameters, also of type T. The compiler automatically calls the function for the corresponding type. When creating a template type parameter, the keyword **typename**may be used as an alternative to the keyword **class**: **template <typename T>**. In this context, the keywords are identical, but throughout this course.

Function templates also make it possible to work with **multiple**generic data types. Define the data types using a comma-separated list. Let's create a function that compares arguments of varying data types (an **int**and a **double**), and prints the smaller one.

template <**class T, class U**>

|  |  |
| --- | --- |
| template <class T, class U>  T smaller(T a, U b) {  return (a < b ? a : b);  } | int main () {  int x=72;  double y=15.34;  cout << smaller(x, y) << endl;  }  // Outputs 15 |

It's not necessary to use **T**, however; you can declare your type parameters using any identifiers that work for you. The only terms you need to avoid are C++ keywords.

**Class Template:** Just as we can define function templates, we can also define **class templates**, allowing classes to have members that use template parameters as types.

|  |
| --- |
| template <class T>  class MyClass {  }; |

Just as with function templates, you can define more than one generic data type by using a comma-separated list. As an example, let's create a class **Pair**, that will be holding a pair of values of a generic type. A specific syntax is required in case you define your member functions outside of your class - for example in a separate source file. You need to specify the **generic type** in angle brackets after the class name. For example, to have a member function **bigger()**defined outside of the class, the following syntax is used. The **bigger**function returns the greater value of the two member variables.

|  |  |
| --- | --- |
| template <class T>  class Pair {  private:  T first, second;  public:  Pair (T a, T b):  first(a), second(b) {  }  T bigger();  };  template <class T>  T Pair<T>::bigger() {  return (first>second ? first : second);  }} | Pair <int> obj(11, 22);  cout << obj.bigger();  // Outputs 22  Pair <double> obj(23.43, 5.68);  cout << obj.bigger();  // Outputs 23.43 |

To create objects of the template class for different types, specify the data type in angle brackets, as we did when defining the function outside of the class.

**Template Specialization**: **Template specialization** allows for the definition of a different implementation of a template when a specific type is passed as a template argument. For example, we might need to handle the character data type in a different manner than we do numeric data types.

|  |  |
| --- | --- |
| template <class T>  class MyClass {  public:  MyClass (T x) {  cout <<x<<" - not a char"<<endl;  }  };  template < >  class MyClass<char> {  public:  MyClass (char x) {  cout <<x<<" is a char!"<<endl;  }  }; | int main () {  MyClass<int> ob1(42);  MyClass<double> ob2(5.47);  MyClass<char> ob3('s');  }  /\* Output:  42 - not a char  5.47 - not a char  s is a char!  \*/ |

First of all, notice that we precede the class name with **template<>**, including an empty parameter list. This is because all types are known and no template arguments are required for this specialization, but still, it is the specialization of a class template, and thus it requires to be noted as such. But more important than this prefix, is the <**char**> specialization parameter after the class template name. This specialization parameter itself identifies the type for which the template class is being specialized (**char**).

In the example above, the first class is the generic template, while the second is the specialization. If necessary, your specialization can indicate a completely different behavior from the behavior of the generic template.

**Exceptions:** C++ exception handling is built upon three keywords: **try**, **catch**, and **throw**. **throw** is used to throw an exception when a problem shows up.

|  |
| --- |
| int motherAge = 29;  int sonAge = 36;  if (sonAge > motherAge) {  throw "Wrong age values";  } |

In the **throw**statement, the operand determines a type for the exception. This can be any expression. The type of the expression's result will determine the type of the exception thrown. A **try**block identifies a block of code that will activate specific exceptions. It's followed by one or more **catch**blocks. The **catch**keyword represents a block of code that executes when a particular exception is thrown. Code that could generate an exception is surrounded with the **try/catch**block. You can specify what type of exception you want to catch by the exception declaration that appears in parentheses following the keyword **catch**.

|  |
| --- |
| try {  int motherAge = 29;  int sonAge = 36;  if (sonAge > motherAge) {  throw 99;  }  }  catch (int x) {  cout<<"Wrong age values - Error "<<x;  }  //Outputs "Wrong age values - Error 99" |

Multiple **catch**statements may be listed to handle various exceptions in case multiple exceptions are thrown by the try block. In our case, we catch exceptions of type **integer**only. It's possible to specify that your catch block handles any type of exception thrown in a try block. To accomplish this, add an **ellipsis (...)**between the parentheses of catch:

|  |
| --- |
| try {  // code  } catch(...) {  // code to handle exceptions  } |

Working with Files: Another useful C++ feature is the ability to read and write to files. That requires the standard C++ library called **fstream**. Three new data types are defined in fstream:  
**ofstream**: Output file stream that creates and writes information to files.

**ifstream**: Input file stream that reads information from files.

**fstream**: General file stream, with both ofstream and ifstream capabilities that allow it to create, read, and write information to files.

To perform file processing in C++, header files <**iostream**> and <**fstream**> must be included in the C++ source file. A file must be opened before you can read from it or write to it.  
Either the **ofstream**or **fstream**object may be used to open a file for writing.  
Let's open a file called "**test.txt"** and write some content to it:

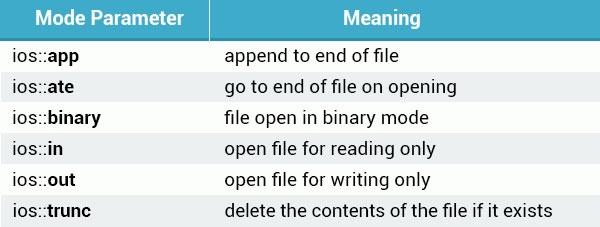
|  |  |
| --- | --- |
| #include <iostream>  #include <fstream>  using namespace std;  int main() {  ofstream MyFile;  MyFile.open("test.txt");  MyFile << "Some text. \n";  MyFile.close();  } | #include <fstream>  using namespace std;  int main() {  ofstream MyFile("test.txt");  MyFile << "This is awesome! \n";  MyFile.close();  } |

The above code creates an **ofstream**object called MyFile, and uses the **open()**function to open the "test.txt" file on the file system. As you can see, the same stream output operator is used to write into the file. If the specified file does not exist, the **open**function will create it automatically. When you've finished working with a file, close it using the member function **close()**. You can also provide the path to your file using the **ofstream**objects constructor, instead of calling the **open** function as in second code.

Under certain circumstances, such as when you don't have file permissions, the **open**function can fail. The **is\_open()** member function checks whether the file is open and ready to be accessed.

|  |
| --- |
| int main() {  ofstream MyFile("test.txt");  if (MyFile.is\_open()) {  MyFile << "This is awesome! \n";  }  else {  cout << "Something went wrong";  }  MyFile.close();  } |

**File Opening Modes**: An optional second parameter of the **open**function defines the **mode**in which the file is opened. This list shows the supported modes.



All these flags can be combined using the bitwise operator OR (|). For example, to open a file

in write mode and truncate it, in case it already exists, use the following syntax:

ofstream outfile;

outfile.open("file.dat", ios::out | ios::trunc );

**Reading From a File**: You can read information from a file using an **ifstream**or **fstream** object.

|  |
| --- |
| #include <iostream>  #include <fstream>  using namespace std;  int main () {  string line;  ifstream MyFile("test.txt");  while ( getline (MyFile, line) ) {  cout << line << '\n';  }  MyFile.close();  } |