Object Orientation Part 1 Overview, Java

|  |  |
| --- | --- |
| **OO in C++**   * For reusability, a **class header** (.h) file is used to specify the class prototype information. That file can be used by consumers of the class to have C++ compiler type checking. * The **actual class .cpp file** also includes the .h file and then provides detailed function definitions. Any static variables also must be defined. * Objects can be created as **automatics** (allocated on the runtime memory stack) or as **dynamics** (heap memory) * If created on the heap, you must free via delete *obj* or delete[] *arrayObj*. * Allows multiple inheritance classes; therefore, interfaces are not needed. * C++ provides controls on inheritance via   **private** not known outside the class  **public** known inside, in subclasses, and outside  **protected** known inside, known in subclasses, not known outside   * **Templates** are used to provide generic typing of classes. With templates, we do not separate the class definition into two files; instead, we only provide a header file. * **Reflection** in C++ is still maturing. Unlike Java, C++ doesn't require runtime metadata; therefore, it is more difficult to obtain it. There are preprocessor tools that can help provide it. | // Assuming the Employee class from below  #include "Employee.h"  #include <string>  #include <stdio.h>  int main (int argc, char\*argv[])  {  // emp and empM are automatic //pointing to automatics//don’t need to be freed  Employee emp = Employee("Dr", "Rob Benz", 125000);  Employee empM[3] = { Employee("Dr", "Rob Benz", 125000)  , Employee("", "Gibb Sun", 70000)  , Employee("Dr", "Rob N Sun", 70000) };  // print them  printf("%9.2lf %s\n"  , emp.salary  , emp.getFormattedName().c\_str());  for (int i = 0; i < 3; i++)  printf("%9.2lf %s\n"  , empM[i].salary  , empM[i].getFormattedName().c\_str());  // pEmp and otherEmpM are dynamic on the heap  Employee \*pEmp = new Employee("Ms", "Mae Nard", 75000);  Employee \*otherEmpM = new Employee[3];    // introduce a set function to initialize an  // employee after allocation  otherEmpM[0].set("", "Harry Fingers", 50000);  otherEmpM[1].set("Gen", "Dee Livery", 11000);  otherEmpM[2].set("Col", "Pop Corn", 85000);  // print them  printf("%9.2lf %s\n"  , pEmp->salary  , pEmp->getFormattedName().c\_str());  for (int i = 0; i < 3; i++)  printf("%9.2lf %s\n"  , otherEmpM[i].salary  , otherEmpM[i].getFormattedName().c\_str());  // free the dynamically allocated  delete pEmp;  delete[] otherEmpM;  } |
| **Defining a Class - Header File (.h file)**  In C++, classes are more reusable if class declarations are separated into two files. In the header file, you must specify the class and instance variables. Class variables are specified using the **static** keyword.  It is best practice to start the .h file with preprocessor controls. This helps avoid double declarations causing compilation to error. This need is more apparent when a class is subclassed and a consumer needs both.  The class definition begins with a class statement:  class *className*  {  ...  };  If it is a subclass, use:  class *className* : public *parentClass*  {  ...  };  In the example main function above, where did we use the Employee() constructor that doesn't have parameters?  ?? | // Employee .h file  #ifndef EMPLOYEE\_H  #define EMPLOYEE\_H  #include <string>  class Employee  {  private:  // class variables  static long lastId; // last generated id  // instance variables  std::string name;  std::string namePrefix;  int exemptions;  long id;    // private class method  static long generateId();  public:  double salary;  // Constructors  Employee(std::string prefix, std::string nm, double pay);  Employee();  void set(std::string prefix, std::string nm, double pay);  std::string getFormattedName();  double calculatePay();  };  #endif |
| **Defining a Class - C++ file**   * The .cpp file must reference the .h file. * Any class variables must be defined. Note that we leave off the **static** keyword in the .cpp file unlike the .h file. * Without specifying a namespace, you must specify the class name in front of the method name. * Destructor automatic unless explicitly created | // Employee.cpp  #include "Employee.h"  #include <string>  long Employee::lastId = 0; // last generated id  // Constructors  Employee::Employee(std::string prefix, std::string nm, double pay)  {  namePrefix = prefix;  name = nm;  salary = pay;  exemptions = 1;  id = generateId();  }  Employee::Employee()  {  namePrefix = "";  name = "";  salary = 0;  exemptions = 1;  id = generateId();  }  void Employee::set(std::string prefix, std::string nm, double pay)  {  namePrefix = prefix;  name = nm;  salary = pay;  exemptions = 1;  }  std::string Employee::getFormattedName()  {  if (namePrefix.compare("")==0)  return name;  else  return namePrefix + " " + name;  }  double Employee::calculatePay()  {  double pay = salary / 26;  if (exemptions == 1)  return pay;  else  return pay \* 0.9;  }  long Employee::generateId()  {  if (Employee::lastId == 0)  Employee::lastId = 111; // db.getLastId();  Employee::lastId++;  return Employee::lastId;  } |
| **Important Concepts - Inheritance in .h file**  **Inheritance**  a class can be defined as a subclass of another class allowing it to inherit variables and methods from the superclass. The subclass has a "is-a" relationship to the superclass.  In the example, Manager is an Employee. It inherits the instance variables: name, namePrefix, exemptions, salary, and id. Manager has its own additional attribute bonus.  The class statement has the following syntax to define a subclass:  class *className* : public *parentClass*  {  ...  }; | // Manager.h  #ifndef MANAGER\_H  #define MANAGER\_H  #include "Employee.h"  #include <string>  class Manager : public Employee  {  private:  double bonus;  public:  Manager(std::string prefix, std::string nm, double pay, double mgrBonus);  double calculatePay();  };  #endif |
| **Inheritance in .cpp file**  For C++ constructors needing to invoke the superclass constructor, C++ is syntactically more difficult than Java.  *className*::*constructor*(*subclassParameterList*)  : *superConstructor*(*superclassParameterList*)  {  *statements*;  } | // Manager.cpp  #include "Manager.h"  #include <string>  // Manager constructor invokes Employee constructor  Manager::Manager(std::string prefix, std::string nm, double pay  , double mgrBonus) :Employee(prefix, nm, pay)  {  bonus = mgrBonus;  }  // Manager overrides Employee calculatePay  double Manager::calculatePay()  {  double basePay = Employee::calculatePay(); //invoke the superclass calculatePay(), not a function in Employee  return basePay + bonus/12;  };  With the current definition of Employee, can Manager::calculatePay() access exemptions? No, it’s a private var to Employee so it cant be known outside the class  How can it access exemptions without making it available to consumers which are not subclasses? Make a get method. Can make it protected but would allow the variable to be changed. |
| **Important Concepts - Polymorphism**  **Polymorphism**  With polymorphism, subclasses can have their own implementation of a commonly named method. This allows a caller to use the common method name. Subclasses can override the implementation of a superclass' method.  In the example, Employee is used to declare both pEmp and pEmp2; however, pEmp2 is referencing a Manager object. | Employee and Manager use the same getFormattedName method. Additionally, both have a calculatePay, but Manager overrode it  #include "Employee.h"  #include "Manager.h"  #include <string>  #include <stdio.h>  int main (int argc, char\*argv[])  {  Employee \*pEmp = new Employee("Ms", "Mae Nard", 75000);  printf("%9.2lf %s\n"  , pEmp->salary  , pEmp->getFormattedName().c\_str());  Employee \*pEmp2 = new Manager("Dr.", "R. Pepper", 50000.00, 500.00);  printf("%9.2lf %s\n"  , pEmp2->salary  , pEmp2->getFormattedName().c\_str());  delete pEmp;  delete pEmp2;  return 0;  } |
| **Important Concepts - Encapsulation**  **Encapsulation** (aka, Information Hiding)  the internals of a class design can be hidden from the public perspective  In this example, StackOne uses an array for its implementation.  Notice that StackOne throws two exceptions StackOverflow and StackUnderflow. | // StackOne.h  #ifndef STACKONE\_H  #define STACKONE\_H  #define STACK\_MAX 10  #include "StackOverflow.h"  #include "StackUnderflow.h"  class StackOne  {  private:  int dataM[STACK\_MAX];  int count = 0;  public:  // constructor  StackOne();  void push(int item) throw (StackOverflow);  int pop() throw (StackUnderflow);  bool isEmpty();  int size();  };  #endif |
|  | // StackOne.cpp uses an array to implement the stack  #include "StackOne.h"  #include "StackOverflow.h"  #include "StackUnderflow.h"  // Constructor  StackOne::StackOne()  {  count = 0;  }  void StackOne::push(int item) throw (StackOverflow)  {  if (count == STACK\_MAX)  {  throw StackOverflow(STACK\_MAX);  }  dataM[count] = item;  count++;  }  int StackOne::pop() throw (StackUnderflow)  {  if (count == 0)  throw StackUnderflow();  count--;  return dataM[count];  }  bool StackOne::isEmpty()  {  return count == 0;  }  int StackOne::size()  {  return count;  } |
| In this implementation of StackTwo, a linked list is used for the Stack. The actual implementation is hidden from consumers of StackOne and StackTwo.  The private nested Node class is only known to StackTwo.  Since StackTwo creates other objects and holds on to them, it is necessary for it to have a destructor. That was not shown, but is shown later in the notes. (See **Destructor** below.) | // StackTwo.h using a linked list  #ifndef STACKTWO\_H  #define STACKTWO\_H  #include "StackUnderflow.h"  class StackTwo  {  private:  // private nested class for the linked list nodes  class Node //subclass  {  public:  int data;  Node \*next;  Node(int data, Node \*next)  {  this->data = data;  this->next = next;  }  };  Node \*pTop;  int count;  public:  // constructor  StackTwo();  void push(int item);  int pop() throw (StackUnderflow);  bool isEmpty();  int size();  };  #endif |
| Notice that pop must delete the node it pops. | // StackTwo.cpp using a linked list  #include "StackTwo.h"  #include "StackUnderflow.h"  #include <stdio.h>  //#define nullptr \_\_null // g++ didn't have it  // constructor  StackTwo::StackTwo()  {  pTop = nullptr;  count = 0;  }  void StackTwo::push(int item)  {  pTop = new Node(item, pTop);  count++;  }  int StackTwo::pop() throw (StackUnderflow)  {  if (pTop == nullptr)  throw StackUnderflow();  int item = pTop->data;  Node \*pDelete = pTop;  pTop = pTop->next;  count--;  printf("Deleting Node %d\n", pDelete->data);  delete pDelete; // free it as we pop  return item;  }  bool StackTwo::isEmpty()  {  return pTop == nullptr;  }  int StackTwo::size()  {  return count;  } |
| **Important Concepts - Generic**  **Generic**  Ability to specify the data type associated with a feature of a Class. For example, a Stack class may allow specification of the type of data stacked.  For **templates** in C++, do **NOT** use separation of header file and cpp. They must be included together since the compiler does inline substitution for the specified parameter types. (In earlier implementations of C++, they were separated.)  Consumer for the example:  StackThree<int> stack3;  stack3.push(10);  stack3.push(20);  stack3.push(stack3.pop() + stack3.pop());  try  {  printf("1. %d\n", stack3.pop());  printf("2. %d\n", stack3.pop());  }  catch (std::exception & e)  {  printf("%s\n", e.what());  } | // StackThree.h  #ifndef STACKTHREE\_H  #define STACKTHREE\_H  #define STACK\_MAX 10  #include "StackOverflow.h"  #include "StackUnderflow.h"  //once it sees template it expects code here, not in its own cpp  template <typename T> // typename is our name for the type  class StackThree  {  private:  T dataM[STACK\_MAX];  int count;  public:  // constructor  StackThree<T>()  {  count = 0;  }  void push(T item) throw (StackOverflow)  {  if (count == STACK\_MAX)  {  throw StackOverflow(STACK\_MAX);  }  dataM[count] = item;  count++;  }  T pop() throw (StackUnderflow)  {  if (count == 0)  throw StackUnderflow();  count--;  return dataM[count];  }  bool isEmpty()  {  return count == 0;  }  int size()  {  return count;  }  };  #endif |
| **Exception Classes**  The examples show the StackOverflow and StackUnderFlow exception classes. This was done to illustrate how you can create exception classes; however, due to the simplicity of these exceptions, it would be wiser to simply reuse existing classes (which is shown below).  Exception classes should provide a what() method which returns a C string representing the exception. String containing any information on the exception | // StackOverflow.h  #ifndef STACKOVERFLOW\_H  #define STACKOVERFLOW\_H  #include <exception>  class StackOverflow : public std::exception  {  private:  int capacity;  public:  StackOverflow(const int capacity);  const char \* what() const throw();  };  #endif |
|  | // StackOverflow.cpp  #include "StackOverflow.h"  #include <exception>  #include <string>  StackOverflow::StackOverflow(const int capacity)  {  this->capacity = capacity;  }  const char \*StackOverflow::what()const throw()  {  std::string s = "Stack Overflow, capacity = " + std::to\_string(capacity);  return s.c\_str();  }; |
|  | // StackUnderflow.h  #ifndef STACKUNDERFLOW\_H  #define STACKUNDERFLOW\_H  #include <exception>  class StackUnderflow : public std::exception  {  private:  public:  StackUnderflow();  const char \* what() const throw();  };  #endif |
|  | // StackUnderflow.cpp  #include "StackUnderflow.h"  #include <exception>  #include <string>  StackUnderflow::StackUnderflow()  {  }  const char \*StackUnderflow::what()const throw()  {  return "Stack Underflow";  }; |
| **Simplifying by Using the Existing out\_of\_range Class**  Since our StackUnderflow and StackOverflow where not interesting, we could have simply used existing exception classes:  std::invalid\_argument  Thrown due to bad arguments  std::out\_of\_range  Thrown when a value is out of range  std::runtime\_error  Thrown when a runtime error (which cannot be detected by simply reading the code) happens  #include "StackFour.h"  #include <exception>  #include <string>  #include <stdio.h>  int main (int argc, char\*argv[])  {  StackFour<int> stack4;  stack4.push(10);  stack4.push(20);  stack4.push(stack4.pop() + stack4.pop());  try  {  printf("1. %d\n", stack4.pop());  printf("2. %d\n", stack4.pop());  }  catch (std::out\_of\_range & e)  {  printf("%s\n", e.what());  }  return 0;  }  **Output:**  1. 30  Stack Underflow | // StackFour.h  #ifndef STACKFOUR\_H  #define STACKFOUR\_H  #define STACK\_MAX 10  #include <stdexcept>  template <typename T> // typename is our name for the type  class StackFour  {  private:  T dataM[STACK\_MAX];  int count;  public:  // constructor  StackFour<T>()  {  count = 0;  }  void push(T item) throw (std::out\_of\_range) // throw()  {  if (count == STACK\_MAX)  {  throw std::out\_of\_range("Stack Overflow");  }  dataM[count] = item;  count++;  }  T pop() throw (std::out\_of\_range) // throw()  {  if (count == 0)  throw std::out\_of\_range("Stack Underflow");  count--;  return dataM[count];  }  bool isEmpty()  {  return count == 0;  }  int size()  {  return count;  }  };  #endif |
| **Destructor**  If an object creates other objects and holds on to them, it will be necessary to have a **destructor** to free up those other objects.  For simple classes, a destructor is not needed.  StackTwo has a linked list which might not be empty when code completes.  When a StackTwo object is deleted (explicitly for dynamic memory and implicitly for automatic memory), its destructor will be called. Destructors are named  ~*className*() | // StackTwo.h using a linked list  #ifndef STACKTWO\_H  #define STACKTWO\_H  #include "StackUnderflow.h"  class StackTwo  {  private:  // private nested class for the linked list nodes  class Node  {  public:  int data;  Node \*next;  Node(int data, Node \*next)  {  this->data = data;  this->next = next;  }  };  Node \*pTop;  int count;  public:  // constructor  StackTwo();  // destructor  ~StackTwo();  void push(int item);  int pop() throw (StackUnderflow);  bool isEmpty();  int size();  };  #endif |
| #include "StackTwo.h"  #include <exception>  #include <string>  #include <stdio.h>  int main (int argc, char\*argv[])  {  StackTwo stack2;//automaic  stack2.push(10);  stack2.push(20);  stack2.push(50);  stack2.push(60);  stack2.push(70);  stack2.push(stack2.pop() + stack2.pop());  try  {  printf("1. %d\n", stack2.pop());  printf("2. %d\n", stack2.pop());  }  catch (std::exception & e)  {  printf("%s\n", e.what());  } return 0;  }  **Output:**  Deleting Node 70  Deleting Node 60  Deleting Node 130  1. 130  Deleting Node 50  2. 50  destructor called  Deleting Node 20  Deleting Node 10  How was the destructor called?  Whenever StackTwo was freed, in this case, automatic would have freed. Without the destructor’s pop() , there would be a memory leak. Destructor is for freeing mem. When return out of function. | // StackTwo.cpp using a linked list  #include "StackTwo.h"  #include "StackUnderflow.h"  #include <stdio.h>  //#define nullptr \_\_null // g++ didn't have it  // constructor  StackTwo::StackTwo()  {  pTop = nullptr;  count = 0;  }  StackTwo::~StackTwo()  {  printf("destructor called\n");  while (!isEmpty())  {  pop();  }  }  void StackTwo::push(int item)  {  pTop = new Node(item, pTop);  count++;  }  int StackTwo::pop() throw (StackUnderflow)  {  if (pTop == nullptr)  throw StackUnderflow();  int item = pTop->data;  Node \*pDelete = pTop;  pTop = pTop->next;  count--;  printf("Deleting Node %d\n", pDelete->data);  delete pDelete; // free it as we pop  return item;  }  bool StackTwo::isEmpty()  {  return pTop == nullptr;  }  int StackTwo::size()  {  return count;  } |
|  |  |
|  |  |
|  |  |
|  |  |