Lab 16

Polymorphism in Employee Class Hierarchy

Introduction

In this lab you will work with a modified version of the **Employee** hierarchy to gain practice with RTTI. In the new hierarchy there is a root class called **Object**, which has a single virtual function **ToString**. The starter project provides a simple implementation of this function. You will experiment with another implementation that uses RTTI, and you will also observe the danger of storing a non-Employee object in a collection of employees. You will use RTTI to provide a more robust program.

Suggested time: 30 minutes

Instructions:

1. Create a new project then add the starter files to the project.
2. Build and run the starter code. When you run, there should be a crash. Why?
3. There is a problem in the main program when you assign a pointer to an **Object** to an array element of **Employee** pointers. This sets up a dangerous situation, which indeed flares up when you run the program. The crash occurs when you attempt to call an **Employee** method for a non-employee.
4. You can of course avoid assigning a non-employee in the first place, but the **PayReport** function, which is passed an array of "employee pointers" could be written more robustly, so that even if you are passed an array in which some of the pointers do not point to an Employee, your program won't crash. Devise a solution.
5. At first blush, you might think that the **dynamic_cast** operator would help. In **PayReport** perform a dynamic cast to **Employee*** before calling any of the methods of **Employee**. Call these methods only if the pointer that comes back is not zero. Unfortunately, this does not work, because in our main function we tricked the runtime system by casting the non-employee (a generic **Object**) into an **Employee***.
6. You can use RTTI to check that the Employee is one of **SalaryEmployee**, **WageEmployee**, or **SalesEmployee**. Implement this helper function:

```
bool ValidEmployeeType(Employee* p)
{
   if (typeid(*p) == typeid(SalaryEmployee))
      return true;
   if (typeid(*p) == typeid(WageEmployee))
      return true;
   if (typeid(*p) == typeid(SalesEmployee))
      return true;
   return false;
}
```

7. You can then modify **PayReport()** to test for a valid employee type.

```
void PayReport(Employee** pEmp, int count)
{
   for (int i = 0; i < count; ++i)
   {
      Employee *p = pEmp[i];
      if (ValidEmployeeType(p))
         Trace(p->GetName(),"   ", pEmp[i]->GetPay());
   }
}
```

8. Build and run. This time you should make it through **PayReport**, but the program crashes another place before exiting. What is the problem?

9. The problem comes in the loop where the objects in the array are deleted. The built-in destructor in **Object** is not virtual, so there is an attempt to destroy a non-Employee object through an Employee destructor. Fix this problem by adding an explicit virtual destructor to **Object**. (The implementation does not have to do anything.) Build and run. Now your program should work.

10. Our implementation of **ToString** involves explicitly printing out the name of the class. An override is placed in each derived class. You can save work in the derived classes by using RTTI in the base class, and simply let the derived classes inherit this default behavior.

Comment out the old implementations of **ToString**. In **Object** implement **ToString** by obtaining a **type_info** and using the **name** method. You will need to include **<typeinfo>**. You will get a compile error. Why?

11. In the starter project the **Employee** class did not publicly derive from **Object**. This mistake did not trip us up before, because we did not attempt to call a base class method through a derived class. Now that we commented out the derived class implementations, we hit this problem. Change the derivation in **employee.h** so that **Employee** publicly derives from **Object**. Now you should get a good build, and your program should work! (Actually, you will see displayed “class Object”, “class Employee”, etc., but that is good enough.)