Advanced Programming in C++

The Curiously Recurring Template Pattern (CRTP)

This exercise is a follow-up on the P-E-M-C example used for lecture 6-7, and the polymorphic cloning pattern used to copy objects.

The Cloning Pattern

The cloning pattern is simple, as shown in code example P-E-M-C (lecture 6–7). A virtual function, typically named clone() and pure, is declared in the topmost base class (Person), and then overridden in every concrete subclass. clone() shall dynamically create a new object of the type in question, by using the copy constructor, and return a pointer to the new object. The type of the returned pointer can be either a pointer to the topmost base class or of the class in question, such pointer types are covariant.

```
class Person
{
public:
   virtual Base* clone() const = 0;
   …
};

class Employee : public Person
{
public:
   virtual Derived* clone() const override { return new Derived{*this}; }
   …
};

class Manager : public Employee
{
public:
   virtual Manager* clone() const override { return new Manager{*this}; }
   …
};
```

If clone() is to be the only public way to copy objects, the copy constructor should be protected, and the copy assignment operator should be deleted.

One problem is that you must remember to override clone() in every concrete subclass. If you forget to do that in Manager, clone() in Employee will be inherited and cloning a Manager will instead give you an Employee, initialized with the Employee part of the Manager object.

The Curiously Recurring Template Pattern (CRTP)

This pattern have been noticed to occur now and then in templates, and is regarded as a C++ design pattern. The pattern is a template class having a template type parameter derives from that type.

```
class Derived : public Base<Derived> { … };
```

Derived could also be a template. CRTP is powerful because of the way template instantiation works in C++. Declarations in the base class are instantiated when the derived class is declared (or instantiated if the base class is a template), but bodies of member functions in the base class are instantiated first after the complete declaration of the derived class is known to the compiler. Therefore the member functions in the base class can use details from the derived class.

Step 1

Keep class Person as it is given in the lecture example.

- Derive a class template class named Person_Cloneable from Person, having one template type parameter named Derived.
- Override clone() to dynamically create a Derived object, initialize it with a copy of ***this**, and return a pointer to the new object as Person*.

  this has type **const** Person_Cloneable* **const** in clone(), but we need to pass a Derived to the Derived copy constructor call in the **new** expression. This is solved with an ordinary type conversion, but be sure to get the details correct!

  A typical choice for the return type of clone() would be Derived*, but that's not possible here, why?
- Modify Employee, given in the lecture example, to derive from Person_Cloneable instead of Person. CRTP is used to inject clone(), defined in Person_Cloneable, into Employee.

Test, correct any errors, and then have a look at the given solution.

Step 2

Manager is now to be derived from Employee. In the lecture example this is straight forward:

```
class Manager : public Employee { … };
```

We now discover that our experiment with CRTP in Step 1 works only for one-level inheritance:

```
class Manager : public Person_Cloneable< What? > { … };
```

What? should be Employee, but CRTP require Manager. To get around this, let Person_Cloneable have *two* template type parameters, one for the direct base class, one for the class to be derived.

- Redefine Person_Cloneable to have two template type parameter, named Base and Derived.
- Modify Manager given in the lecture example, to derive from Person_Cloneable<Employee, Manager> instead of Employee.

If we compile at this stage, we will run into problems with constructors. All subclasses will derive from Person_Cloneable, but Person_Cloneable will derive from different base classes, with different constructors. In the Employee case Person_Cloneable will derive from Person, in the Manager case from Employee and these have different constructors, especially the public ones.

- Fortunately constructors can be inherited in C++11, so remove the explicitly declared public constructor in Person_Cloneable, and instead declare Person_Cloneable to inherit constructors from its direct base class.

Step 3

Modify Consultant.