Chapter 17: Nested Classes

Classes can be defined inside other classes. Classes that are defined inside other classes are called *nested classes*. Nested classes are used in situations where the nested class has a close conceptual relationship to its surrounding class. For example, with the class string a type string::iterator is available which provides all characters that are stored in the string. This string::iterator type could be defined as an object iterator, defined as nested class in the class string.

A class can be nested in every part of the surrounding class: in the `public, protected` or `private` section. Such a nested class can be considered a member of the surrounding class. The normal access and rules in classes apply to nested classes. If a class is nested in the `public` section of a class, it is visible outside the surrounding class. If it is nested in the `protected` section it is visible in subclasses, derived from the surrounding class, if it is nested in the `private` section, it is only visible for the members of the surrounding class.

The surrounding class has no special privileges towards the nested class. The nested class has full control over the accessibility of its members by the surrounding class. For example, consider the following class definition:

```
class Surround
{
   public:
      class FirstWithin
      {
         int d_variable;

         public:
            FirstWithin();
            int var() const;
      };
   private:
      class SecondWithin
      {
         int d_variable;
```

```
        public:
            SecondWithin();
            int var() const;
    };
};
inline int Surround::FirstWithin::var() const
{
    return d_variable;
}
inline int Surround::SecondWithin::var() const
{
    return d_variable;
}
```

Here access to the members is defined as follows:

- The class `FirstWithin` is visible outside and inside `Surround`. The class `FirstWithin` thus has global visibility.
- `FirstWithin`'s constructor and its member function `var` are also globally visible.
- The data member `d_variable` is only visible to the members of the class `FirstWithin`. Neither the members of `Surround` nor the members of `SecondWithin` can directly access `FirstWithin::d_variable`.
- The class `SecondWithin` is only visible inside `Surround`. The public members of the class `SecondWithin` can also be used by the members of the class `FirstWithin`, as nested classes can be considered members of their surrounding class.
- `SecondWithin`'s constructor and its member function `var` also can only be reached by the members of `Surround` (and by the members of its nested classes).
- `SecondWithin::d_variable` is only visible to `SecondWithin`'s members. Neither the members of `Surround` nor the members of `FirstWithin` can access `d_variable` of the class `SecondWithin` directly.
- As always, an object of the class type is required before its members can be called. This also holds true for nested classes.

To grant the surrounding class access rights to the private members of its nested classes or to grant nested classes access rights to the private members of the surrounding class, the classes can be defined as friend classes (see section 17.3).

Nested classes can be considered members of the surrounding class, but members of nested classes are *not* members of the surrounding class. So, a member of the class `Surround` may not access `FirstWithin::var` directly. This is understandable considering that a `Surround` object is not also a `FirstWithin` or `SecondWithin` object. In fact, nested classes are just typenames. It is not implied that objects of such classes automatically exist in the surrounding class. If a member of the surrounding class should use a (non-static) member of a nested class then the surrounding class must define a nested class object, which can thereupon be used by the members of the surrounding class to use members of the nested class.

For example, in the following class definition there is a surrounding class `Outer` and a nested class `Inner`. The class `Outer` contains a member function `caller`. The member function `caller` uses the `d_inner` object that is composed within `Outer` to call `Inner::infunction`:

```
class Outer
{
    public:
        void caller();

    private:
        class Inner
        {
            public:
                void infunction();
        };
        Inner d_inner;      // class Inner must be known
};
void Outer::caller()
{
    d_inner.infunction();
}
```

Inner::infunction can be called as part of the inline definition of Outer::caller, even though the definition of the class Inner is yet to be seen by the compiler. On the other hand, the compiler must have seen the definition of the class Inner before a data member of that class can be defined.

17.1: Defining nested class members

Member functions of nested classes may be defined as inline functions. Inline member functions can be defined as if they were defined outside of the class definition. To define the member function Outer::caller outside of the class Outer, the function's fully qualified name (starting from the outermost class scope (Outer)) must be provided to the compiler. Inline and in-class functions can be defined accordingly. They can be defined and they can use any nested class. Even if the nested class's definition appears later in the outer class's interface.

When (nested) member functions are defined inline, their definitions should be put below their class interface. Static nested data members are also usually defined outside of their classes. If the class `FirstWithin` would have had a `static size_t` datamember epoch, it could have been initialized as follows:

```
size_t Surround::FirstWithin::epoch = 1970;
```

Furthermore, multiple scope resolution operators are needed to refer to public static members in code outside of the surrounding class:

```
void showEpoch()
{
    cout << Surround::FirstWithin::epoch;
```

```
}
```
Within the class `Surround` only the `FirstWithin::` scope must be used; within the class `FirstWithin` there is no need to refer explicitly to the scope.

What about the members of the class `SecondWithin`? The classes `FirstWithin` and `SecondWithin` are both nested within `Surround`, and can be considered members of the surrounding class. Since members of a class may directly refer to each other, members of the class `SecondWithin` can refer to (public) members of the class `FirstWithin`. Consequently, members of the class `SecondWithin` could refer to the `epoch` member of `FirstWithin` as `FirstWithin::epoch`.

17.2: Declaring nested classes

Nested classes may be declared before they are actually defined in a surrounding class. Such forward declarations are required if a class contains multiple nested classes, and the nested classes contain pointers, references, parameters or return values to objects of the other nested classes.

For example, the following class `Outer` contains two nested classes `Inner1` and `Inner2`. The class `Inner1` contains a pointer to `Inner2` objects, and `Inner2` contains a pointer to `Inner1` objects. Cross references require forward declarations. Forward declarations must be given an access specification that is identical to the access specification of their definitions. In the following example the `Inner2` forward declaration must be given in a `private` section, as its definition is also part of the class `Outer`'s private interface:

```
class Outer
{
   private:
      class Inner2;      // forward declaration

      class Inner1
      {
         Inner2 *pi2;    // points to Inner2 objects
      };
      class Inner2
      {
         Inner1 *pi1;    // points to Inner1 objects
      };
};
```

17.3: Accessing private members in nested classes

To grant nested classes access rights to the private members of other nested classes, or to grant a surrounding class access to the private members of its nested classes the `friend` keyword must be used.

Note that no friend declaration is required to grant a nested class access to the private members of its surrounding class. After all, a nested class is a type defined by its surrounding class and as such objects of the nested class are members of the outer class and thus can access all the outer class's members. Here is an example showing this principle. The example won't compile as members of the class `Extern` are denied access to `Outer`'s private members, but `Outer::Inner`'s members *can* access `Outer`'s private memebrs:

```
class Outer
{
    int d_value;
    static int s_value;

    public:
        Outer()
        :
            d_value(12)
        {}
        class Inner
        {
            public:
                Inner()
                {
                    cout << "Outer's static value: " << s_value << '\n';
                }
                Inner(Outer &outer)
                {
                    cout << "Outer's value: " << outer.d_value << '\n';
                }
        };
};
class Extern            // won't compile!
{
    public:
        Extern(Outer &outer)
        {
            cout << "Outer's value: " << outer.d_value << '\n';
        }

        Extern()
        {
            cout << "Outer's static value: " << Outer::s_value << '\n';
        }
};

int Outer::s_value = 123;
int main()
{
    Outer outer;
    Outer::Inner in1;
    Outer::Inner in2{ outer };
}
```

Now consider the situation where a class `Surround` has two nested classes `FirstWithin` and `SecondWithin`. Each of the three classes has a static data member `int s_variable`:

```
class Surround
{
    static int s_variable;
    public:
        class FirstWithin
        {
            static int s_variable;
            public:
                int value();
        };
        int value();
    private:
        class SecondWithin
        {
            static int s_variable;
            public:
                int value();
        };
};
```

If the class Surround should be able to access FirstWithin and SecondWithin's private members, these latter two classes must declare Surround to be their friend. The function Surround::value can thereupon access the private members of its nested classes. For example (note the friend declarations in the two nested classes):

```
class Surround
{
    static int s_variable;
    public:
        class FirstWithin
        {
            friend class Surround;
            static int s_variable;
            public:
                int value();
        };
        int value();
    private:
        class SecondWithin
        {
            friend class Surround;
            static int s_variable;
            public:
                int value();
        };
};
inline int Surround::FirstWithin::value()
{
    FirstWithin::s_variable = SecondWithin::s_variable;
    return (s_variable);
}
```

Friend declarations may be provided *beyond* the definition of the entity that is to be considered a friend. So a class can be declared a friend *beyond* its definition. In that situation in-class code may already use the fact that it is going to be declared a friend by the upcoming class. As an example, consider an in-class implementation of the function `Surround::FirstWithin::value`. The required `friend` declaration can also be inserted *after* the implementation of the function `value`:

```
class Surround
{
   public:
      class FirstWithin
      {
         static int s_variable;
         public:
            int value();
            {
               FirstWithin::s_variable = SecondWithin::s_variable;
               return s_variable;
            }
         friend class Surround;
      };
   private:
      class SecondWithin
      {
         friend class Surround;
         static int s_variable;
      };
};
```

Note that members named identically in outer and inner classes (e.g., `` `s_variable' ``) may be accessed using the proper scope resolution expressions, as illustrated below:

```
class Surround
{
   static int s_variable;
   public:
      class FirstWithin
      {
         friend class Surround;
         static int s_variable;  // identically named
         public:
            int value();
      };
      int value();

   private:
      class SecondWithin
      {
         friend class Surround;
         static int s_variable;  // identically named
         public:
            int value();
      };
      static void classMember();
};
```

```
inline int Surround::value()
{                               // scope resolution expression
    FirstWithin::s_variable = SecondWithin::s_variable;
    return s_variable;
}
inline int Surround::FirstWithin::value()
{
    Surround::s_variable = 4;       // scope resolution expressions
    Surround::classMember();
    return s_variable;
}
inline int Surround::SecondWithin::value()
{
    Surround::s_variable = 40;      // scope resolution expression
    return s_variable;
}
```

Nested classes aren't automatically each other's friends. Here `friend` declarations must be provided to grant one nested classes access to another nested class's private members.

To grant `FirstWithin` access to `SecondWithin`'s private members, `SecondWithin` must contain a `friend` declaration.

Likewise, the class `FirstWithin` simply uses `friend class SecondWithin` to grant `SecondWithin` access to `FirstWithin`'s private members. Even though the compiler hasn't seen `SecondWithin` yet, a friend declaration is also considered a forward declaration.

Note that `SecondWithin`'s forward declaration cannot be specified inside `FirstWithin` by using `` `class Surround::SecondWithin;' ``, as this would generate an error message like:

`Surround' does not have a nested type named `SecondWithin'

Now assume that in addition to the nested class `SecondWithin` there also exists an outer-level class `SecondWithin`. To declare that class a friend of `FirstWithin's` declare `friend ::SecondWithin` inside `class FirstWithin`. In that case, an outer level class declaration of `FirstWithin` must be provided before the compiler encounters the `friend ::SecondWithin` declaration.

Here is an example in which all classes have full access to all private members of all involved classes: , and a outer level `FirstWithin` has also been declared:

```
class SecondWithin;

class Surround
{
    // class SecondWithin;      not required (but no error either):
    //                          friend declarations (see below)
    //                          are also forward declarations

    static int s_variable;
```

```
    public:
        class FirstWithin
        {
            friend class Surround;
            friend class SecondWithin;
            friend class ::SecondWithin;

            static int s_variable;
            public:
                int value();
        };
        int value();            // implementation given above
    private:
        class SecondWithin
        {
            friend class Surround;
            friend class FirstWithin;

            static int s_variable;
            public:
                int value();
        };
};
inline int Surround::FirstWithin::value()
{
    Surround::s_variable = SecondWithin::s_variable;
    return s_variable;
}
inline int Surround::SecondWithin::value()
{
    Surround::s_variable = FirstWithin::s_variable;
    return s_variable;
}
```

17.4: Nesting enumerations

Enumerations may also be nested in classes. Nesting enumerations is a good way to show the close connection between the enumeration and its class. Nested enumerations have the same controlled visibility as other class members. They may be defined in the private, protected or public sections of classes and are inherited by derived classes. In the class ios we've seen values like ios::beg and ios::cur. In the current Gnu **C++** implementation these values are defined as values of the seek_dir enumeration:

```
class ios: public _ios_fields
{
    public:
        enum seek_dir
        {
            beg,
            cur,
            end
        };
};
```

As an illustration assume that a class DataStructure represents a data structure that may be traversed in a forward or backward direction. Such a class can define an enumeration Traversal having the values FORWARD and BACKWARD. Furthermore, a member function setTraversal can be defined requiring a Traversal type of argument. The class can be defined as follows:

```
class DataStructure
{
    public:
        enum Traversal
        {
            FORWARD,
            BACKWARD
        };
        setTraversal(Traversal mode);
    private:
        Traversal
            d_mode;
};
```

Within the class DataStructure the values of the Traversal enumeration can be used directly. For example:

```
void DataStructure::setTraversal(Traversal mode)
{
    d_mode = mode;
    switch (d_mode)
    {
        FORWARD:
            // ... do something
        break;

        BACKWARD:
            // ... do something else
        break;
    }
}
```

Ouside of the class DataStructure the name of the enumeration type is not used to refer to the values of the enumeration. Here the classname is sufficient. Only if a variable of the enumeration type is required the name of the enumeration type is needed, as illustrated by the following piece of code:

```
void fun()
{
    DataStructure::Traversal                // enum typename required
        localMode = DataStructure::FORWARD; // enum typename not required

    DataStructure ds;
                                        // enum typename not required
    ds.setTraversal(DataStructure::BACKWARD);
}
```

In the above example the constant DataStructure;:FORWARD was used to specify a value of an enum defined in the class DataStructure. Instead of DataStructure::FORWARD the construction ds.FORWARD is also accepted. In my opinion this syntactic liberty is ugly: FORWARD is a symbolic value that is defined at the class level; it's not a member of ds, which is suggested by the use of the member selector operator.

Only if `DataStructure` defines a nested class `Nested`, in turn defining the enumeration `Traversal`, the two class scopes are required. In that case the latter example should have been coded as follows:

```
void fun()
{
   DataStructure::Nested::Traversal
      localMode = DataStructure::Nested::FORWARD;

   DataStructure ds;

   ds.setTraversal(DataStructure::Nested::BACKWARD);
}
```

Here a construction like DataStructure::Nested::Traversal localMode = ds.Nested::FORWARD could also have been used, although I personally would avoid it, as FORWARD is not a member of ds but rather a symbol that is defined in DataStructure.

17.4.1: Empty enumerations

Enum types usually define symbolic values. However, this is not required. In section 14.6.1 the std::bad_cast type was introduced. A bad_cast is thrown by the dynamic_cast<> operator when a reference to a base class object cannot be cast to a derived class reference. The bad_cast could be caught as type, irrespective of any value it might represent.

Types may be defined without any associated values. An *empty enum* can be defined which is an `enum` not defining any values. The empty enum's type name may thereupon be used as a legitimate type in, e.g. a `catch` clause.

The example shows how an empty enum is defined (often, but not necessarily within a `class`) and how it may be thrown (and caught) as exceptions:

```
#include <iostream>

enum EmptyEnum
{};

int main()
try
{
   throw EmptyEnum();
}
catch (EmptyEnum)
{
   std::cout << "Caught empty enum\n";
}
```

17.5: Revisiting virtual constructors

In section 14.13 the notion of virtual constructors was introduced. In that section a class Base was defined as an abstract base class. A class Clonable was defined to manage Base class pointers in containers like vectors.

As the class Base is a minute class, hardly requiring any implementation, it can very well be defined as a nested class in Clonable. This emphasizes the close relationship between Clonable and Base. Nesting Base under Clonable changes

```
    class Derived: public Base
```

into:

```
    class Derived: public Clonable::Base
```

Apart from defining Base as a nested class and deriving from Clonable::Base rather than from Base (and providing Base members with the proper Clonable:: prefix to complete their fully qualified names), no further modifications are required. Here are the modified parts of the program shown earlier (cf. section 14.13), now using Base nested under Clonable:

```
// Clonable and nested Base, including their inline members:
    class Clonable
    {
        public:
            class Base;
        private:
            Base *d_bp;
        public:
            class Base
            {
                public:
                    virtual ~Base();
                    Base *clone() const;
                private:
                    virtual Base *newCopy() const = 0;
            };
            Clonable();
            explicit Clonable(Base *base);
            ~Clonable();
            Clonable(Clonable const &other);
            Clonable(Clonable &&tmp);
            Clonable &operator=(Clonable const &other);
            Clonable &operator=(Clonable &&tmp);

            Base &base() const;
    };
    inline Clonable::Base *Clonable::Base::clone() const
    {
        return newCopy();
    }
    inline Clonable::Base &Clonable::base() const
    {
        return *d_bp;
    }

// Derived and its inline member:
    class Derived1: public Clonable::Base
    {
```

```
        public:
            ~Derived1();
        private:
            virtual Clonable::Base *newCopy() const;
    };
    inline Clonable::Base *Derived1::newCopy() const
    {
        return new Derived1(*this);
    }

// Members not implemented inline:
    Clonable::Base::~Base()
    {}
```


- Table of Contents
- Previous Chapter
- Next Chapter