Chapter 9

More on C++

Peter Müller
Globewide Network Academy (GNA)
pmueller@uu-gna.mit.edu

This section concludes our introduction to C++. We introduce "real" object-oriented concepts and we answer the question, how a C++ program is actually written.

9.1 Inheritance

In our pseudo language, we formulate inheritance with "inherits from". In C++ these words are replaced by a colon. As an example let's design a class for 3D points. Of course we want to reuse our already existing class *Point*. We start designing our class as follows:

```
class Point3D : public Point {
  int _z;

public:
  Point3D() {
    setX(0);
    setY(0);
    _z = 0;
  }
  Point3D(const int x, const int y, const int z) {
    setX(x);
    setY(y);
    _z = z;
  }

  ~Point3D() { /* Nothing to do */ }

  int getZ() { return _z; }
  void setZ(const int val) { _z = val; }
};
```

9.1.1 Types of Inheritance

You might notice again the keyword `public` used in the first line of the class definition (its *signature*). This is necessary because C++ distinguishes two types of inheritance: *public* and *private*. As a default, classes are privately derived from each other. Consequently, we must explicitly tell the compiler to use public inheritance.

The type of inheritance influences the access rights to elements of the various superclasses. Using public inheritance, everything which is declared `private` in a superclass remains `private` in the subclass. Similarly, everything which is `public` remains `public`. When using private inheritance the things are quite different as is shown in table 9.1.

	Type of Inheritance	
	private	**public**
private	private	private
protected	private	protected
public	private	public

Table 9.1: Access rights and inheritance.

The leftmost column lists possible access rights for elements of classes. It also includes a third type `protected`. This type is used for elements which should be directly usable in subclasses but which should not be accessible from the outside. Thus, one could say elements of this type are between `private` and `public` elements in that they can be used within the class hierarchy rooted by the corresponding class.

The second and third column show the resulting access right of the elements of a superclass when the subclass is privately and publically derived, respectively.

9.1.2 Construction

When we create an instance of class *Point3D* its constructor is called. Since *Point3D* is derived from *Point* the constructor of class *Point* is also called. However, this constructor is called *before* the body of the constructor of class *Point3D* is executed. In general, prior to the execution of the particular constructor body, constructors of every superclass are called to initialize their part of the created object.

When we create an object with

```
Point3D point(1, 2, 3);
```

the second constructor of *Point3D* is invoked. Prior to the execution of the constructor body, the constructor *Point()* is invoked, to initialize the *point* part of object *point*. Fortunately, we have defined a constructor which takes no arguments. This constructor initializes the 2D coordinates *_x* and *_y* to 0 (zero). As *Point3D* is only derived from *Point* there are no other constructor calls and the body of *Point3D(const int, const int, const int)* is executed. Here we invoke methods *setX()* and *setY()* to explicitly override the 2D coordinates. Subsequently, the value of the third coordinate *_z* is set.

This is very unsatisfactory as we have defined a constructor *Point()* which takes two arguments to initialize its coordinates to them. Thus we must only be able to tell, that instead of using the *default constructor Point()* the paramterized *Point(const int, const int)* should be used. We can do that by specifying the desired constructors after a single colon just before the body of constructor *Point3D()*:

```
class Point3D : public Point {
  ...

public:
  Point3D() { ... }
```

```
    Point3D(
      const int x,
      const int y,
      const int z) : Point(x, y) {
        _z = z;
    }
    ...
  };
```

If we would have more superclasses we simply provide their constructor calls as a comma separated list. We also use this mechanism to create contained objects. For example, suppose that class *Part* only defines a constructor with one argument. Then to correctly create an object of class *Compound* we must invoke *Part()* with its argument:

```
  class Compound {
    Part part;
    ...

  public:
    Compound(const int partParameter) : part(partParameter) {
      ...
    }
    ...
  };
```

This dynamic initialization can also be used with built-in data types. For example, the constructors of class *Point* could be written as:

```
  Point() : _x(0), _y(0) {}
  Point(const int x, const int y) : _x(x), _y(y) {}
```

You should use this initialization method as often as possible, because it allows the compiler to create variables and objects correctly initialized instead of creating them with a default value and to use an additional assignment (or other mechanism) to set its value.

9.1.3 Destruction

If an object is destroyed, for example by leaving its definition scope, the destructor of the corresponding class is invoked. If this class is derived from other classes their destructors are also called, leading to a recursive call chain.

9.1.4 Multiple Inheritance

C++ allows a class to be derived from more than one superclass, as was already briefly mentioned in previous sections. You can easily derive from more than one class by specifying the superclasses in a comma separated list:

```
  class DrawableString : public Point, public DrawableObject {
    ...

  public:
    DrawableString(...) :
      Point(...),
```

```
    DrawableObject(...) {
      ...
  }
  ~DrawableString() { ... }
  ...
};
```

We will not use this type of inheritance in the remainder of this tutorial. Therefore we will not go into further detail here.

9.2 Polymorphism

In our pseudo language we are able to declare methods of classes to be `virtual`, to force their evaluation to be based on object content rather than object type. We can also use this in C++:

```
class DrawableObject {
public:
  virtual void print();
};
```

Class *DrawableObject* defines a method *print()* which is virtual. We can derive from this class other classes:

```
class Point : public DrawableObject {
  ...
public:
  ...
  void print() { ... }
};
```

Again, *print()* is a virtual method, because it inherits this property from *DrawableObject*. The function *display()* which is able to display any kind of drawable object, can then be defined as:

```
void display(const DrawableObject &obj) {
  // prepare anything necessary
  obj.print();
}
```

When using virtual methods some compilers complain if the corresponding class destructor is not declared virtual as well. This is necessary when using pointers to (virtual) subclasses when it is time to destroy them. As the pointer is declared as superclass normally its destructor would be called. If the destructor is virtual, the destructor of the actual referenced object is called (and then, recursively, all destructors of its superclasses). Here is an example adopted from [1]:

```
class Colour {
public:
  virtual ~Colour();
};

class Red : public Colour {
public:
  ~Red();      // Virtuality inherited from Colour
```

```
};

class LightRed : public Red {
public:
  ~LightRed();
};
```

Using these classes, we can define a *palette* as follows:

```
Colour *palette[3];
palette[0] = new Red;   // Dynamically create a new Red object
palette[1] = new LightRed;
palette[2] = new Colour;
```

The newly introduced operator `new` creates a new object of the specified type in dynamic memory and returns a pointer to it. Thus, the first `new` returns a pointer to an allocated object of class *Red* and assigns it to the first element of array *palette*. The elements of *palette* are pointers to *Colour* and, because *Red* is-a *Colour* the assignment is valid.

The contrary operator to `new` is `delete` which explicitly destroys an object referenced by the provided pointer. If we apply `delete` to the elements of *palette* the following destructor calls happen:

```
delete palette[0];
// Call destructor ~Red() followed by ~Colour()
delete palette[1];
// Call ~LightRed(), ~Red() and ~Colour()
delete palette[2];
// Call ~Colour()
```

The various destructor calls only happen, because of the use of virtual destructors. If we would have not declared them virtual, each `delete` would have only called *~Colour()* (because *palette[i]* is of type pointer to *Colour*).

9.3 Abstract Classes

Abstract classes are defined just as ordinary classes. However, some of their methods are designated to be necessarily defined by subclasses. We just mention their *signature* including their return type, name and parameters but not a definition. One could say, we omit the method body or, in other words, specify "nothing". This is expressed by appending "= 0" after the method signatures:

```
class DrawableObject {
  ...
public:
  ...
  virtual void print() = 0;
};
```

This class definition would force every derived class from which objects should be created to define a method *print()*. These method declarations are also called *pure methods*.

Pure methods must also be declared `virtual`, because we only want to use objects from derived classes. Classes which define pure methods are called *abstract classes*.

9.4 Operator Overloading

If we recall the abstract data type for complex numbers, *Complex*, we could create a C++ class as follows:

```
class Complex {
  double _real,
         _imag;

  public:
    Complex() : _real(0.0), _imag(0.0) {}
    Complex(const double real, const double imag) :
      _real(real), _imag(imag) {}

    Complex add(const Complex op);
    Complex mul(const Complex op);
    ...
  };
```

We would then be able to use complex numbers and to "calculate" with them:

```
Complex a(1.0, 2.0), b(3.5, 1.2), c;

c = a.add(b);
```

Here we assign c the sum of a and b. Although absolutely correct, it does not provide a convenient way of expression. What we would rather like to use is the well-known "+" to express addition of two complex numbers. Fortunately, C++ allows us to *overload* almost all of its operators for newly created types. For example, we could define a "+" operator for our class *Complex*:

```
class Complex {
  ...

public:
  ...

  Complex operator +(const Complex &op) {
    double real = _real + op._real,
           imag = _imag + op._imag;
    return(Complex(real, imag));
  }

  ...
};
```

In this case, we have made operator `+` a member of class *Complex*. An expression of the form

```
c = a + b;
```

is translated into a method call

```
c = a.operator +(b);
```

Thus, the binary operator + only needs one argument. The first argument is implicitly provided by the invoking object (in this case *a*).

However, an operator call can also be interpreted as a usual function call, as in

```
c = operator +(a, b);
```

In this case, the overloaded operator is **not** a member of a class. It is rather defined outside as a normal overloaded function. For example, we could define operator + in this way:

```
class Complex {
  ...

public:
  ...

  double real() { return _real; }
  double imag() { return _imag; }

  // No need to define operator here!
};

Complex operator +(Complex &op1, Complex &op2) {
  double real = op1.real() + op2.real(),
         imag = op1.imag() + op2.imag();
  return(Complex(real, imag));
}
```

In this case we must define access methods for the real and imaginary parts because the operator is defined outside of the class's scope. However, the operator is so closely related to the class, that it would make sense to allow the operator to access the private members. This can be done by declaring it to be a *friend* of class *Complex*.

9.5 Friends

We can define functions or classes to be friends of a class to allow them direct access to its private data members. For example, in the previous section we would like to have the function for operator + to have access to the private data members *_real* and *_imag* of class *Complex*. Therefore we declare operator + to be a friend of class *Complex*:

```
class Complex {
  ...

public:
  ...

  friend Complex operator +(
    const Complex &,
    const Complex &
  );
};
```

```
Complex operator +(const Complex &op1, const Complex &op2) {
  double real = op1._real + op2._real,
         imag = op1._imag + op2._imag;
  return(Complex(real, imag));
}
```

You should not use friends very often because they break the data hiding principle in its fundamentals. If you have to use friends very often it is always a sign that it is time to restructure your inheritance graph.

9.6 How to Write a Program

Until now, we have only presented parts of or very small programs which could easily be handled in one file. However, greater projects, say, a calendar program, should be split into manageable pieces, often called *modules*. Modules are implemented in separate files and we will now briefly discuss how modularization is done in C and C++. This discussion is based on UNIX and the GNU C++ compiler. If you are using other constellations the following might vary on your side. This is especially important for those who are using integrated development environments (IDEs), for example, Borland C++.

Roughly speaking, modules consist of two file types: *interface descriptions* and *implementation files*. To distinguish these types, a set of suffixes are used when compiling C and C++ programs. Table 9.2 shows some of them.

Extension(s)	**File Type**
`.h`, `.hxx`, `.hpp`	interface descriptions ("header" or "include files")
`.c`	implementation files of C
`.cc`, `.C`, `.cxx`, `.cpp`, `.c++`	implementation files of C++
`.tpl`	interface description (templates)

Table 9.2: Extensions and file types.

In this tutorial we will use `.h` for header files, `.cc` for C++ files and `.tpl` for template definition files. Even if we are writing "only" C code, it makes sense to use `.cc` to force the compiler to treat it as C++. This simplifies combination of both, since the internal mechanism of how the compiler arrange names in the program differs between both languages[1].

9.6.1 Compilation Steps

The compilation process takes `.cc` files, preprocess them (removing comments, add header files)[2] and translates them into *object files*[3]. Typical suffixes for that file type are `.o` or `.obj`.

After successful compilation the set of object files is processed by a *linker*. This program combine the files, add necessary libraries[4] and creates an executable. Under UNIX this file is called *a.out* if not other specified. These steps are illustrated in Figure 9.1.

[1]This is due to the fact that C++ supports function polymorphism. Therefore the *name mangling* must take function parameters into account.
[2]This also creates an intermediary preprocessed raw C++ file. A typical suffix is `.i`.
[3]This has nothing to do with objects in the object-oriented sense.
[4]For example, standard functions such as *printf()* are provided this way.

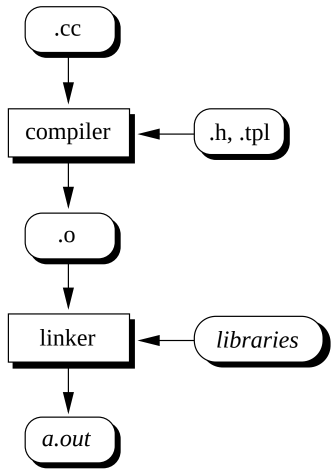

Figure 9.1: Compilation steps.

With modern compilers both steps can be combined. For example, our small example programs can be compiled and linked with the GNU C++ compiler as follows ("example.cc" is just an example name, of course):

```
gcc example.cc
```

9.6.2 A Note about Style

Header files are used to describe the interface of implementation files. Consequently, they are included in each implementation file which uses the interface of the particular implementation file. As mentioned in previous sections this inclusion is achieved by a copy of the content of the header file at each preprocessor `#include` statement, leading to a "huge" raw C++ file.

To avoid the inclusion of multiple copies caused by mutual dependencies we use *conditional coding*. The preprocessor also defines conditional statements to check for various aspects of its processing. For example, we can check if a macro is already defined:

```
#ifndef MACRO
#define MACRO /* define MACRO */
...
#endif
```

The lines between `#ifndef` and `#endif` are only included, if `MACRO` is not already defined. We can use this mechanism to prevent multiple copies:

```
/*
** Example for a header file which 'checks' if it is
** already included. Assume, the name of the header file
** is 'myheader.h'
*/

#ifndef __MYHEADER_H
#define __MYHEADER_H

/*
```

```
** Interface declarations go here
*/

#endif /* __MYHEADER_H */
```

__MYHEADER_H is a unique name for each header file. You might want to follow the convention of using the name of the file prefixed with two underbars. The first time the file is included, __MYHEADER_H is not defined, thus every line is included and processed. The first line just defines a macro called __MYHEADER_H. If accidentally the file should be included a second time (while processing the same input file), __MYHEADER_H is defined, thus everything leading up to the `#endif` is skipped.

9.7 Excercises

1. *Polymorphism.* Explain why

   ```
   void display(const DrawableObject obj);
   ```

   does not produce the desired output.