Modern C++ - Classes

Classes

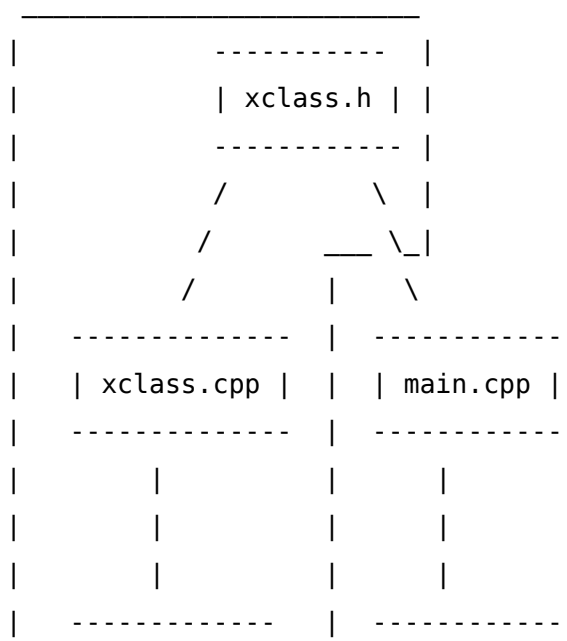
In this chapter we describe how to implement (non templated) classes in the C++ programming language. This chapter is not intended to teach basic object-oriented programming concepts,w e suppose that the reader is familiar with *class*, *encapsulation*, *interface* and *implementation*. In this chapter we teach how these can be implemented in C++.

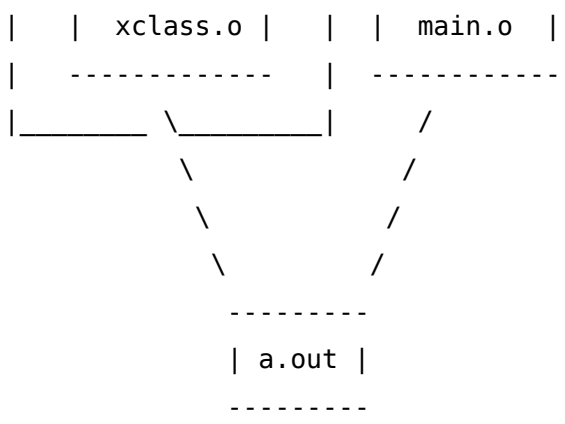
Files

In C++ a non templated class is placed into two portions: a header file represents the knowledge the clients of the class should know about the class, and a source file, where the class methods and free functions related to the class are implemented. Some of the class methods are *inline* methods and implemented in the header file.

Clients of the class should include the class's header file and they should link to the class's object file compiled from the source. The implementation can be splitted into more sources, but the usual practice is to have one header file and one source file. (Class templates have a different schema: both the interface and implementation is placed to header file(s).)

The following picture describes the files for a class called **XClass** and a client called **main**.





Members

A class is introduced by either the **class** or **struct** keyword. The only difference between using class or struct is the default visibility of the class.

Classes form a namespace with its own name. All members are declared inside this namespace. When membersof a class referred from inside a membre function the member names are visible by default (except they are hidden by some locally declared name). When members are referred outside of the class namespace (e.g. from an other class or from a namespace function) they should be qualified by the class name.

```
1 struct XClass
2 {
3   void func();
4   int  i;
5 };
6 void XClass::func()
7 {
8   i = 5;  // here inside Xclass namespace "i" means "Xclass::i"
9 }
```

Language elements declared inside a class are called *members*. Members of a C++ class can be *data members* (or in other name: attributes), *member functions* (in C++ the expression *method* is sometimes restricted to *virtual functions*), *type declarations* and some other language elements, like *constants*, *enums*, etc.

Static data members and member functions has a special behaviour, we will discuss later.

Data members

The non static data members (sometimes called as *attribute* or *field*) form the memory

area of the objects belonging to the class. Data members can be of any type visible in the class, including pointers, arrays, etc. The lifetime of the data members are the same as the object itself, the creation order is the same as the order of declaration (even when the constructor *initializer list* is different). As usual, the destruction order is the opposit of the construction order.

Class data members can be **const**. Const data members are immutable under the lifetime of the objects, but they might have different value in each objects.

Data members declared as **mutable** can be changed even for constant objects or inside contant member functions.

```
1 class XClass
2 {
3             int  ifield; // data field
4             int *iptr;
5   const     int  id;     // const, must be initialized
6   mutable mutex  mut;    // mutable field
7 };
```

A data member declaration does not denote an actual memory area: it is rather an offset inside an object. It becames a real memory area only when a real object is defined, and the data member designates a certain part inside that object. As a consequence, normal pointers can not be set to a data member itself, it is only a member of a specific object can be pointed by a usual pointer.

There are, however, *pointer to member* which can be assigned to a member, with the meaning as the offset inside the object.

Member functions

The non static member functions are functions called on an object. In every non static member function a pointer to the actual object called **this** can be used. The body of a membre function is *inside* the namespace of the class, i.e. names used inside the function body are belonging primarily to the class namespace.

Member functions are implemented as “real” functions with a hidden extra first parameter: the **this** pointer. The this pointer is declared as a no-const pointer to the class for normal member functions, and as a pointer to const in constant member functions. The actual argument of the this parameter is set to a pointer to the actual object on which the member function is called. All the members (data or function) accessed inside a member function is accessed via the this pointer.

```
1 void XClass::func(int x)  // non-const member function
```

```
 2 {
 3   // pointer this is declared here as "XClass *"
 4   ifield = 5; // same as this->ifield (if ifield is declared in XClass)
 5   id = 6;     // ERROR: id is const member, must not changed
 6   mut.lock(); // m is XClass::m if m is declared in XClass
 7 };
 8 void XClass::func(int x) const  // const member function
 9 {
10   // pointer this is declared here as "const XClass *"
11   ifield = 5; // ERROR: same as this->ifield, but this points to const
12   mut.lock(); // mut can be modified since it is mutable
13 };
```

This ensures two things:

1. only constant member functions can be called on constant objects.
2. inside a constant member function (whether it is called on a const or non-const object) no data member can be modified.

```
1 XClass obj;  // non-const object
2 const XClass cobj; // const object
3
4  obj.func(1);  // call XClass::f(int), passing this as XClass*
5 cobj.func(1);  // call XClass::f(int) const, passing this as const XClass*
```

it is usual to define the minimal necessary set of member functions on a class. The rest of the convenience operations can be defined as non member – so called – namespace function.

```
1 void print(const XClass &xc)
2 {
3   // calling public methods on XClass
4 }
```

Operators defined on classes are discussed in the next chapter.

Visibility

There are three visibility categories of class members: *public*, *protected* and *private*. Each category is denoted with the label of the same keyword. Visibility categories can be repeated and may appear in any order. When the class is introduced by the **class**

keyword, the default visibility area is *private*, in case of using **struct** it is *public*.

```
 1 class XClass
 2 {
 3   // private for class, public for stuct
 4 public:
 5   // visible from everywhere
 6 protected:
 7   // visible only for derived classes and friend
 8 private:
 9   // visible only for class members and friends
10 };
```

There is a special way to access non-public data members and calling non-public member functions: *friends*. Friend functions have the unlimited access to all data members to the class. Friend classes are classes which have all members as friend functions to the class.

```
1 class XClass
2 {
3   friend void nonMemberFunc();
4   friend class OtherClass;
5 };
6 void nonMemberFunc()
7 {
8   // can access all XClass members
9 }
```

We can place the friend declarations in any visibility sections.

Special member functions

There are special member functions in the class we can define.

Constructor

The constructor has the same name as the class. The constructor is called when a new object has been created and is responsible to initialize it. The programmer may define multiply constructors overloaded by their parameters. The one without parameters is called *default constrator*.

Copy constructor

The copy constructor has also the same name as the class but has a special signature: receives a single parameter of the same class by reference. The copy constructor is responsible to initialize the newly created object from an other object of the same class.

The signature of the copy constructor is usually *const reference* of the type to ensure that the source of the initialization is not modified, but there are exceptions: smart pointers may declare the parameter as non-const.

Assignment operator

An assignment operator is used when a new value is about to assigned to an existing object. The parameter list is optional, but it is usually the same as of the copy constructor. Although, the return value can be freely defined, the most design rules requires returning the freshly assigned object by reference.

Destructor

Destructor is an optional single method with no parameter. A destructor is called when an object goes out of life. A destructor is mainly repsonsible to automatically free allocated resources of the object.

Move constructor (since C++11)

Similar to the copy constructor but has a *non-const right value reference* parameter to initialize the object moving the value from the source (and such way stealing the resources from it).

Move assignment (since C++11)

Similar to assignment operator, but has a *right value reference* parameter to assing new value to an existing object moving the value from the source (and such way stealing the resources from it).

```
1 class XClass
2 {
3   XClass();                           // default constructor
4   XClass(const XClass &);             // copy constructor
5   XClass(XClass &&);                  // move constructor since C++11
6   XClass& operator=(const XClass &); // assingment operator
7   XClass& operator=(XClass &&);       // move assingment operator since
C++11
8   ~XClass();                          // destructor
9 };
```

Special member functions need not to be defined. When a special member function is not defined the compiler generates one by the default *memberwise copy* semantic. Generation rules of the move special member functions are a bit more complex.

Constructors must not be declared as *virtual function*. Destructor can be, and in certain design must be defined as virtual function. We will discuss copy constructor and destructor in the chapter on POD classes.

Static members

Static members are global variables with *static* lifetime. They are placed into the namespace of a class for only logical purposes. Static data members are not phisically part of any objects: they are allocated outside of any objects. As their lifetime is static, they can be referred even when no object is exists of their class.

Inside the class deficition using the **static** keyword only *declares* the static data member. We must *define* it in exactly one source file (this is usually the one we implement the class methods).

Static member functions are global functions placed the namespace of the class. They also can be referred without existing objects of the class. Static member functions do not receive **this** parameter, therefore they must not refer to other members (data or function) without specifying the actual object.

```
 1 class XClass
 2 {
 3 public:
 4   XClass *create();
 5 private:
 6   XClass(int i) : id(nid++) { }  // not thread safe
 7       const int id;
```

```
 8      static int nid;  // only declaration of static
 9    mutable mutex mut;
10 };
11 int XClass::nid = 0;  // definition of static
12
13 XClass *create()  // factory method: thread safe
14 {
15   lock_guard<mutex> guard(mut);  // lock
16   return new XClass();
17 } // unlock on dectructing guard
```

Even if static members are not phisically part of the objects can be declared *public*, *private* or *protected*. An example for public static member functon is a *factory* function, which role is to create objects of the class based on the *_factory* or *abstract factory* patterns. As static functions are members, they can access private members of their class.

An example class

Here we define a simple class to represent Date information storing *year*, *month* and *day* as integers and providing basic access and modification functions. Naturally, this class is just for demonstrational purposes: for real programs one should use **std::chrono** from the standard C++11 **<chrono>**.

First we place the declaration of the class into *date.h* header file:

```
 1 #ifndef DATE_H
 2 #define DATE_H
 3
 4 #include <iostream>
 5 class Date
 6 {
 7 public:
 8   Date( int y, int m=1, int d=1); // constructor with default params
 9
10   int   getYear()  const;  // get year
11   int   getMonth() const;  // get month
12   int   getDay()   const;  // get day
13   void  setDate( int y, int m, int d);   // set a date
14
15   date&  next();       // set next date
```

```
16   date&  add( int n);  // add n dates
17
18   void get( std::istream& is);       // read a Date from stream
19   void put( std::ostream& os) const; // write a Date to stream
20 private:
21   void checkDate(int y, int m, int d); // check date, exit on fail
22   int year;    // data members
23   int month;   // to represent
24   int day;     // a date
25 };
26 #endif // DATE_H
```

Then we define the member functions in *date.cpp* source file:

```
 1 #include <iostream>
 2 #include <cstdlib>
 3
 4 #include "date.h"
 5
 6 namespace /* anonym namespace, visible only in this source */
 7 {
 8   const int day_in_month[] = {31,28,31,30,31,30,31,31,30,31,30,31};
 9 }
10
11 Date( int y, int m, int d) : year(y), month(m), day(d) { }
12
13 void Date::setDate( int y, int m, int d)
14 {
15   checkDate( y, m, d);
16   year  = y;
17   month = m;
18   day   = d;
19 }
20 Date& Date::next()
21 {
22   ++day;
23   /* TODO: leap year */
24   if ( day-1 == day_in_month[month-1])
25   {
26     day = 1;
```

```
27     ++month;
28   }
29   if ( 13 == month )
30   {
31     month = 1;
32     ++year;
33   }
34   return *this; // return reference to *this object
35 }
36 Date& Date::add( int n)
37 {
38   for (int i = 0; i < n; ++i)
39   {
40     next(); /* KISS */
41   }
42   return *this; // return reference to *this object
43 }
44 void Date::get( std::istream& is)
45 {
46   int y, m, d;
47   if ( is >> y >> m >> d ) // have to check for success
48   {
49     setDate( y, m, d);
50   }
51 }
52 void Date::put( std::ostream& os) const
53 {
54   // better to build on accessors;
55   os << "[ " << getYear()  << "."
56              << getMonth() << "."
57              << getDay()   << " ]";
58 }
59 void Date::checkDate( int y, int m, int d)
60 {
61   // TODO: leap year
62   if ( 0 == y )                          exit(1);
63   if ( m < 1 || m > 12 )                 exit(1);
64   if ( d < 1 || d > day_in_month[m-1] ) exit(1);
65 }
```

To implement a correct version for the *leap year* problem is let to the reader as an excercise.