C++ Classes

2008/04/03

Soonho Kong

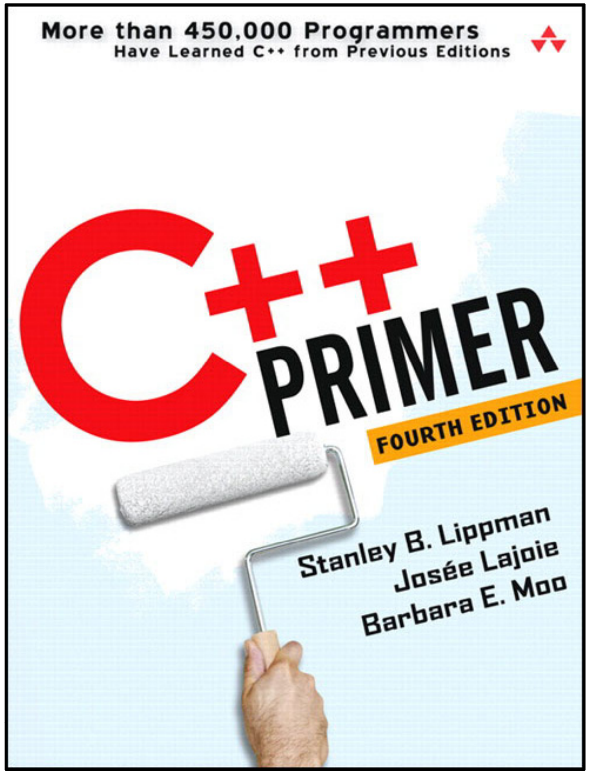
soon@ropas.snu.ac.kr

Programming Research Laboratory

Seoul National University

Most of text and examples

are excerpted from C++ Primer 4th e/d.



Contents

Class

Abstract Date Types(ADT)

Abstract Data Types

Data + Operation

ADT	C++
Data	Data Member
Operation	Function Member

12.1

Class Definitions and Declarations

12.1.1

Class Definitions: A Recap

“Most fundamentally,

A class defines a new **type** and a new **scope**”

Class Definition

(Class Members)*

(Member function)*

Constructors

* The same name as the class

* Initialize the object

* Generally should use a "*constructor initializer list*"

```
// default constructor needed to initialize members of built-in type
Sales_item(): units_sold(0), revenue(0.0) { }
```

Constructor Initializer List

```
Class A
{
    const int i;
    A(int arg);
};

A::A(int arg)
{
    i = arg;
}
```

```
Class A
{
    const int i;
    A(int arg);
};

A::A(int arg) : arg(i)
{
}
```

Functions defined *inside* the class are **inline**

```
Class A
{
    const int i;
    A(int arg) : i(arg)
    {
    }
};
```

Otherwise, it should indicate that

they are in the scope of the class

```
Class A
{
    const int i;
    A(int arg);
};

A::A(int arg) : arg(i)
{
}
```

Const member function

```
double const_member_function(…) const;
```

* May not change the data members of the object.

* “const” must appear in both the declaration and definition

12.1.2

Data Abstraction and Encapsulation

Data Abstraction:

Separation of

interface and implementation

Encapsulation:

* Combining lower-level elements

to form a new, higher-level entity.

* Information Hiding

12.1.3

More on Class Definitions

Using Typedefs to Streamline Classes

```
class A
{
public:
        typedef unsigned int index;

        index id(index i)
        {
                return i;
        }
};

int main()
{
        A a;
        A::index i = a.id(3);
}
```

Explicitly Specifying **inline** Member Functions

```
class Screen {
    public:
        typedef std::string::size_type index;
        char get() const { return contents[cursor]; }
        inline char get(index ht, index wd) const;
        index get_cursor() const;
     };

    char Screen::get(index r, index c) const
    {
        index row = r * width;
        return contents[row + c];
    }

    inline Screen::index Screen::get_cursor() const
    {
        return cursor;
    }
```

Explicitly Specifying **inline** Member Functions

The definition for an inline member function

that is not defined within the class body

ordinarily **should** be placed <u>in the same header file</u>

in which <u>the class definition appears</u>

12.1.4

Class Declarations v.s. Definitions

Forward Declaration

```
Class Screen;
```

The type “Screen” is **incomplete** type.

* Know that it is a type.

* Do not know what members that type contains.

Incomplete Type

An incomplete type may be used to define **only**

1. Pointers to the type

```
Screen* p;
```

2. References to the type

```
Screen& r;
```

3. Parameter or return type

```
void foo(Screen* p);
```

in function declaration(not definition!)

Incomplete Type

A class cannot have data members of its own type.

Because it is not defined until its class body is complete.

```
class LinkedListNode
{
        int x;
        LinkedListNode* next;
};
```

12.1.5

Class Objects

Defining Objects of Class Type

```
1.       Sales_item item1;
2. class Sales_item item1;
```

Both of them are equivalent.

Why a Class Definition Ends in a Semicolon

```
int x;
class A { ; … ; };
```

12.2

The Implicit **this** Pointer

You don’t have to use **this** in general.

The compiler treats

an unqualified reference to a class member

as if it had been made through the this pointer

```
classs A
{
        int x;
        void foo()
        {
                x = 3;
        }
};
```

```
classs A
{
        int x;
        void foo()
        {
                this.x = 3;
        }
};
```

When to use the this pointer

When we need to refer to the object as **a whole**

rather than to a member of the object.

Returning *this

```
class Screen {
public:
     // interface member functions
     Screen& move(index r, index c);
     Screen& set(char);
     Screen& set(index, index, char);
     // other members as before
};
```

```
Screen& Screen::set(char c)
{
    contents[cursor] = c;
    return *this;
}


Screen& Screen::move(index r, index c)
{
    index row = r * width;
    cursor = row + c;
    return *this;
}
```

Returning ***this** from a **const** Member Function

In an ordinary non-const member function, the type of **this** is a const pointer to the class type.

In a const member function, the type of **this** is a const pointer to a const class-type object.

	Type(this)
Non-const Member Function	T * const
Const Member Function	**const T* const**

Returning ***this** from a **const** Member Function

```
// move cursor to given position, set that character and display the screen
myScreen.move(4,0).set('#').display(cout);
```

```
Screen myScreen;
// this code fails if display is a const member function
// display return a const reference; we cannot call set on a const
myScreen.display().set('*');
```

Overloading Based on **const**

```
class Screen {
public:
    // interface member functions
    // display overloaded on whether the object is const or not
    Screen& display(std::ostream &os)
        { do_display(os); return *this; }
    const Screen& display(std::ostream &os) const
        { do_display(os); return *this; }
private:
    // single function to do the work of displaying a Screen,
    // will be called by the display operations
    void do_display(std::ostream &os) const
        { os << contents; }
    // as before
 };
```

```
Screen myScreen(5,3);
const Screen blank(5, 3);
myScreen.set('#').display(cout); // calls nonconst version
blank.display(cout);             // calls const version
```

Mutable Data Members

A **mutable** data member is a member that is **never const**,

even when it is a member of a const object.

```
class Screen {
public:
    // interface member functions
private:
    mutable size_t access_ctr; // may change in a const members
    // other data members as before
 };

void Screen::do_display(std::ostream& os) const
{
    ++access_ctr; // keep count of calls to any member function
    os << contents;
}
```

12.3

Class Scope

Class Scope

Every class defines

its own new scope and a unique type

12.3.1

Name Lookup in Class Scope

Using a Class Member

may be accessed only through an **object** or a **pointer**

using member access operators **dot** or **arrow**, respectively.

```
Class obj;      // Class is some class type
Class *ptr = &obj;

// member is a data member of that class
ptr->member;    // fetches member from the object to which ptr points
obj.member;     // fetches member from the object named obj

// memfcn is a function member of that class
ptr->memfcn(); // runs memfcn on the object to which ptr points
obj.memfcn();  // runs memfcn on the object named obj
```

Scope and Member Definitions

Member definitions behave as if they are in the scope of the class,

even if the member is defined outside the class body.

```
double Sales_item::avg_price() const
    {
        if (units_sold)
            return revenue/units_sold;
        else
            return 0;
    }
```

Parameter List and Function Bodies Are in Class Scope

Function Return Types Aren’t Always in Class Scope

```
class A
{
public:
    typedef int my_int;
    my_int bar(my_int arg);
};

my_int A::bar(my_int arg)
{
    return arg;
}
```

Class Members Follow Normal Block-Scope Name Lookup

```
int height;
     class Screen {
     public:
         void dummy_fcn(index height) {
             cursor = width * height; // which height? The parameter
         }

          void dummy_fcn(index height) {
             cursor = width * this->height;   // member height
             // alternative way to indicate the member
            cursor = width * Screen::height; // member height
         }

     private:
         index cursor;
         index height, width;
     };
```

Names Are Resolved Where They Appear within the File

```
class Screen {
    public:
        // ...
        void setHeight(index);
    private:
        index height;
    };

    Screen::index verify (Screen::index);

    void Screen::setHeight(index var) {
        // var: refers to the parameter
        // height: refers to the class member
        // verify: refers to the global function
        height = verify(var);
    }
```

12.3

Constructors

Constructor

Special member functions that are executed

whenever we create new objects of a class type

Constructor

The **job** of a constructor is

to **ensure** that the data members of each object

start out with **sensible initial values**

The job of a constructor is to ensure that the data members of each object start out with sensible i nitial values

Constructors May Be **Overloaded**

and

Arguments Determine Which Constructor to Use

```
class Sales_item;
    // other members as before
    public:
        // added constructors to initialize from a string or an istream
        Sales_item(const std::string&);
        Sales_item (std::istream&);
        Sales_item ();
    };
```

A constructor may not be declared as const

```
class Sales_item {
    public:
        Sales_item() const;    // error
    };
```

The job of the constructor is to initialize an object.

12.4.1

The Constructor Initializer

“**The constructor initializer** is a feature

that many reasonably experienced C++ programmers

have **not** mastered.”

```
Sales_item::Sales_item
(const string &book) :
  isbn(book),
  units_sold(0),
  revenue(0.0)
{
}
```

```
Sales_item::Sales_item
(const string &book)
{
          isbn = book;
          units_sold = 0;
          revenue = 0.0;
}
```

Constructor Initializers Are Sometimes Required

```
class ConstRef {
     public:
         ConstRef(int ii);
     private:
         int i;
         const int ci;
         int &ri;
     };
     // no explicit constructor initializer: error ri is uninitialized
     ConstRef::ConstRef(int ii)
     {              // assignments:
          i = ii;   // ok
          ci = ii;  // error: cannot assign to a const
          ri = i;   // assigns to ri which was not bound to an object
     }
```

Constructor Initializers Are Sometimes Required

Members of a class type that

1) do not have a default constructor and

members that are 2) const or 3) reference types

must be initialized in the constructor initializer regardless of type.

Order of Member Initialization

```
 class X {
        int i;
        int j;
    public:
         X(int val): j(val), i(j) { }
    };

int main()
{
         X theX(10);
         x.i?
         x.j?
}
```

Order of Member Initialization

The order in which members are **initialized** is

the order in which the members are **defined**.

12.4.2

Default Arguments and Constructors

Default Arguments and Constructors

```
class Sales_item {
     public:
         // default argument for book is the empty string
         Sales_item(const std::string &book = ""):
                   isbn(book), units_sold(0), revenue(0.0) { }
         Sales_item(std::istream &is);
         // as before
};

     Sales_item empty;
     Sales_item Primer_3rd_Ed("0-201-82470-1");
```

12.4.3

The Default Constructor

“If a class **defines** *even one constructor*,

then the compiler will **not** generate <u>the default constructor</u>.”

Classes Should Usually Define a Default Constructor

The fact that

“NoDefault has no default constructor”

means

1.

Every constructor for every class that has a NoDefault member

must explicitly initialize the NoDefault member

by passing an initial string value to the NoDefault constructor.

The fact that

“NoDefault has no default constructor”

means

2.

* The compiler will ***not*** synthesize the default constructor

for classes that have members of type NoDefault.

* If such classes want to provide a default,

they ***must define*** one ***explicitly***,

and that constructor ***must explicitly*** initialize their NoDefault member.

The fact that

"NoDefault has no default constructor"

means

3.

The NoDefault type **may not be used**

as the element type for a dynamically allocated array.

The fact that

“NoDefault has no default constructor”

means

4.

Statically allocated arrays of type NoDefault

must provide an **explicit initializer** for each element.

The fact that

“NoDefault has no default constructor”

means

5.

If we have a **container** such as ***vector*** that holds NoDefault objects,

we **cannot use** the constructor that takes a size

without also supplying an element **initializer**.

Using the Default Constructor

(X)

```
// oops! declares a function, not an object
Sales_item myobj();
```

(O)

```
// ok: create an unnamed, empty Sales_itemand use to initialize myobj
Sales_item myobj = Sales_item();
```

12.4.4

Implicit Class-Type Conversions

Implicit Class-Type Conversions

```
class Sales_item {
public:
    // default argument for book is the empty string
    Sales_item(const std::string &book = ""):
              isbn(book), units_sold(0), revenue(0.0) { }
    Sales_item(std::istream &is);
    // as before
 };

string null_book = "9-999-99999-9";
// ok: builds a Sales_itemwith 0 units_soldand revenue from
// and isbn equal to null_book
item.same_isbn(null_book);


// ok: uses the Sales_item istream constructor to build an object
 // to pass to same_isbn
item.same_isbn(cin);
```

“Whether this behavior is desired depends on

how we think our users will use the conversion.”

Supressing Implicit Conversions Defined by Constructors

```
class Sales_item {
    public:
        // default argument for book is the empty string
        explicit Sales_item(const std::string &book = ""):
                  isbn(book), units_sold(0), revenue(0.0) { }
        explicit Sales_item(std::istream &is);
        // as before
    };
```

Supressing Implicit Conversions Defined by Constructors

(X)

```
// error: explicit allowed only on constructor declaration in class header
     explicit Sales_item::Sales_item(istream& is)
     {
         is >> *this; // uses Sales_iteminput operator to read the members
     }
```

(O)

```
     Sales_item::Sales_item(istream& is)
     {
         is >> *this; // uses Sales_iteminput operator to read the members
     }
```

Supressing Implicit Conversions Defined by Constructors

```
item.same_isbn(null_book); // error: string constructor is explicit
item.same_isbn(cin);       // error : istream constructor is explicit
```

"Making a constructor explicit turns off only the use of the constructor implicitly."

Explicitly Using Constructors for Conversions

```
string null_book = "9-999-99999-9";
     // ok: builds a Sales_itemwith 0 units_soldand revenue from
     // and isbn equal to null_book
     item.same_isbn(Sales_item(null_book));
```

12.4.5

Explicit Initialization of Class Members

“Members of classes

that 1) define **no constructors** and

2) all of whose data members are **public**

may be initialized

in the same way that we initialize **array elements**”

```
struct Data {
    int ival;
    char *ptr;
};
// val1.ival = 0; val1.ptr = 0
Data val1 = { 0, 0 };

// val2.ival = 1024;
// val2.ptr = "Anna Livia Plurabelle"
Data val2 = { 1024, "Anna Livia Plurabelle" };
```

Exercise 12.31: The data members of pair are public, yet this code doesn't compile. Why?

pair<int, int> p2 = {0, 42}; // **doesn't** compile, why?

12.5

Friends

“It is convenient to let

specific **nonmember functions**

access

the **private** members of a class

while still preventing general access.”

“Over-loaded operators, such as the input or output operators,

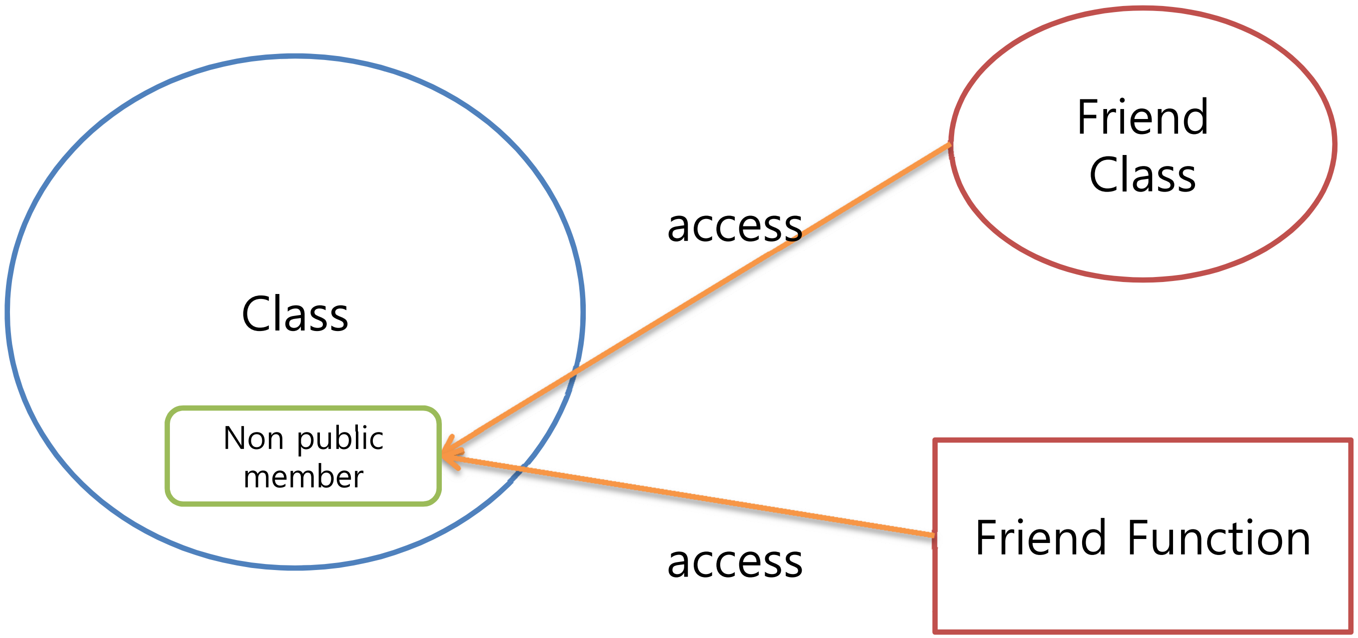
often need access to the private data members of a class.”

“Yet, even if they are **not** members of the class,

they are "**part of the interface**" to the class.”

The **friend** mechanism allows a class

to grant access to its nonpublic members to specified functions or classes.



Example

Screen says "class **Window** is my friend"

```
class Screen {
    // Window_Mgr members can access private parts of class Screen
    friend class Window_Mgr;
    // ...restofthe Screen class
};

Window_Mgr&
Window_Mgr::relocate(Screen::index r, Screen::index c,
                     Screen& s)
{
     // ok to refer to height and width
     s.height += r;
     s.width += c;

     return *this;
}
```

Making Another Class' Member Function a Friend

Screen says “ Relocate method in Window_Mgr is my friend”

```
class Screen {
    // Window_Mgrmust be defined before class Screen
    friend Window_Mgr&
        Window_Mgr::relocate(Window_Mgr::index,
                             Window_Mgr::index,
                             Screen&);
    // ...restofthe Screen class
};
```

Friend Declarations and Scope

To make a member function a friend,

the class containing that member must have been defined.

On the other hand,

a class or nonmember function need not have been declared to be made a friend.

```
class X {
    friend class Y;
    friend void f() { /* ok to define friend function in the class body */
    }
};
class Z {
    Y *ymem; // ok: declaration for class Y introduced by friend in X
    void g() { return ::f(); } // ok: declaration of f introduced by X
};
```

Overloaded Functions and Friendship

```
// overloaded storeOn functions
extern std::ostream& storeOn (std::ostream &, Screen &);
extern BitMap& storeOn (BitMap &, Screen &);

class Screen {
  // ostream version of storeOn may access private parts of Screen objects
  friend std::ostream& storeOn(std::ostream &, Screen &);
  // ...
};
```

12.6

static Class Members

“Making the object global violates encapsulation.”

"It is sometimes necessary for all the objects of a particular class type to access a global object."

"Making the object global violates encapsulation."

Use class static member

Advantages of Using Class static Members

1. The name of a **static member** is in the scope of the **class**, thereby avoiding name collisions with members of other classes or global objects.

2. **Encapsulation** can be enforced. A static member can be a private member; a global object cannot.

3. It is **easy to see** by reading the program that a static member is associated with a particular class. This **visibility** clarifies the programmer's intentions.

Defining static Members

```
class Account {
    public:
        // interface functions here
        void applyint() { amount += amount * interestRate; }
        static double rate() { return interestRate; }
        static void rate(double); // sets a new rate
    private:
        std::string owner;
        double amount;
        static double interestRate;
        static double initRate();
    };
```

Using a Class static Member

```
Account ac1;
Account *ac2 = &ac1;
// equivalent ways to call the static member rate function
double rate;
rate = ac1.rate();     // through an Account object or reference
rate = ac2->rate();    // through a pointer to an Account object
rate = Account::rate();// directly from the class using the scope operator
```

12.6.1

static Class Member Functions

When we **define** a static member outside the class,

we do not **respecify** the **static** keyword.

The keyword appears **only** with the declaration inside the class body:

```
void Account::rate(double newRate)
{
    interestRate = newRate;
}
```

static Functions Have No **this** Pointer

A static member is part of its **class** but not part of any **object**.

12.6.2

static Class Data Membes

“static data members must be defined

exactly once outside the class body.”

Unlike ordinary data members,

static members are **not initialized**

through the class constructor(s)

and **instead should** be initialized when they are defined.

The static keyword, however, is used

only on the declaration inside the class body.

Definitions are not labeled static.

```
// define and initialize static class member
double Account::interestRate = initRate();
```

Integral const static Members Are Special

```
class Account {
     public:
         static double rate() { return interestRate; }
         static void rate(double);  // sets a new rate
     private:
         static const int period = 30; // interest posted every 30 days
         double daily_tbl[period]; // ok: period is constant expression
};
```

a const static data member of integral type

can be initialized within the class body

as long as the initializer is a **constant expression**:

Integral const static Members Are Special

```
// definition of static member with no initializer;
// the initial value is specified inside the class definition
const int Account::period;
```

When a const static data member is initialized in the class body,

the data member **must still be defined** outside the class definition.

static Members Are Not Part of Class Objects

Because static data members are **not** part of any object,

they can be used in ways

that would be **illegal** for **nonstatic** data members

static Members Are Not Part of Class Objects

```
class Bar {
     public:
         // ...
     private:
         static Bar mem1; // ok
         Bar *mem2;       // ok
         Bar mem3;        // error
};
```

static Members Are Not Part of Class Objects

```
class Screen {
    public:
        // bkground refers to the static member
        // declared later in the class definition
        Screen& clear(char = bkground);
    private:
        static const char bkground = '#';
    };
```

Chapter 12

Classes

1. **Classes** are **the most fundamental feature** in C++. Classes let us define new types that are tailored to our own applications, making our programs shorter and easier to modify.

2. **Data abstraction -** the ability to define both data and function members - and **encapsulation -** the ability to protect class members from general access - are **fundamental to classes**. Member functions define the interface to the class. We encapsulate the class by making the data and functions used by the implementation of a class private.

3. Classes may define **constructors**, which are special member functions that control how objects of the class are initialized. Constructors may be **overloaded**. Every constructor should initialize every data member. Constructors should use a **constructor initializer list** to initialize the data members. Initializer lists are lists of name value pairs where the name is a member and the value is an initial value for that member.

4. Classes may grant access to their nonpublic members to other classes or functions. A class grants access by making the class or function a **friend**.

5. Classes may also define mutable or **static** members. A mutable member is a data member that is never const; its value may be changed inside a const member function. A static member can be either function or data; static members exist independently of the objects of the class type.