Copy Control

2008/04/08

Soonho Kong

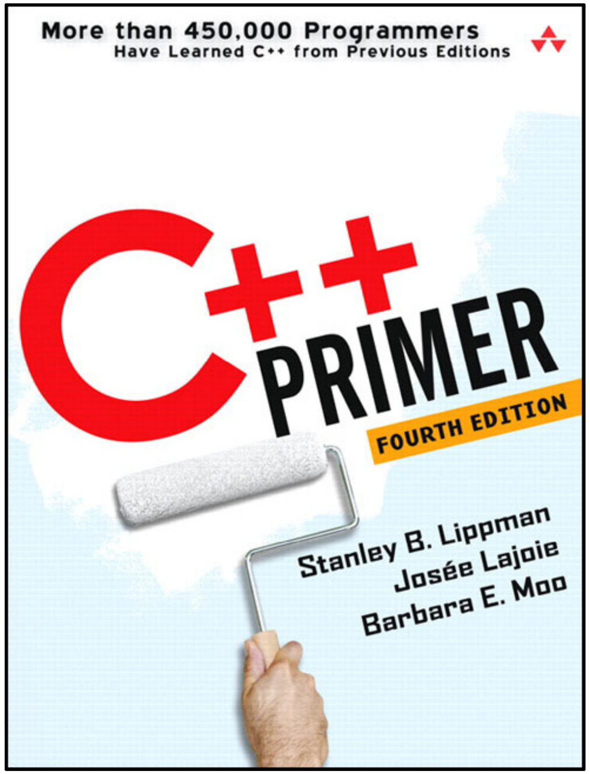
soon@ropas.snu.ac.kr

Programming Research Laboratory

Seoul National University

Most of text and examples

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



Types control what happens

when objects of the type are

1) copied, 2) assigned, or 3) destroyed.

The copy constructor,

assignment operator,

and destructor

are referred to as

copy control

Contents

13.1

The Copy Constructor

Copy Constructor

The constructor that takes a single parameter

that is a (usually const) reference

to an object of the class type itself.

```
Foo( (const) Foo&);
```

Forms of Object Definition

Two form of initialization:

Direct v.s. Copy

```
string empty_copy = string();      // copy-initialization
string empty_direct;               // direct-initialization
```

Direct-initialization directly invokes

the constructor matched by the arguments.

Copy-initialization always involves the copy constructor.

Copy-initialization first uses the indicated **constructor** to create a **temporary** object.

It then uses the **copy constructor** to copy that temporary into the one we are creating:

```
string empty_copy = string();      // copy-initialization
```

Copy-initialization can be used

only when specifying a single argument or

when we explicitly build a temporary object to copy.

```
string null_book = "9-999-99999-9"; // copy-initialization
string empty_copy = string();      // copy-initialization
```

We cannot copy objects of the IO types

For types that do **not** support copying,

or when using a constructor that is **non explicit**

the distinction can be essential:

```
ifstream file1("filename"); // ok: direct initialization
ifstream file2 = "filename"; // error: copy constructor is private

// This initialization is okay only if
// the Sales_item(const string&) constructor is not explicit
Sales_item item = string("9-999-99999-9");
```

Parameters and Return Values

When a parameter is a nonreference type

the argument is copied.

Parameters and Return Values

Similarly, a nonreference return value is returned by copying

Parameters and Return Values

```
// copy constructor used to copy the return value;
// parameters are references, so they aren't copied
string make_plural(size_t, const string&, const string&);
```

Initializing Container Elements

```
vector<string> svec(5);
```

1. Create a temporary value.

2. The copy constructor is then used to copy the temporary into each element of svec

Constructors and Array Elements

```
Sales_item primer_eds[] = { string("0-201-16487-6"),
                                 string("0-201-54848-8"),
                                 string("0-201-82470-1"),
                                 Sales_item()
                               };
```

13.1.1

The Synthesized Copy Constructor

Unlike the synthesized default constructor,

a copy constructor is **synthesized**

even if we **define** other constructors.

The behavior of the synthesized copy constructor is

to **memberwise** initialize the new object

as a copy of the original object.

Memberwise Copy

1. Built-in type => copy

2. Class type => call copy constructor

3. Array => copy each element

For many classes,

the synthesized copy constructor

does **exactly** the work that is needed.

However,

some classes **must** take control of

what happens when objects are copied

1. Have a data member that is a **pointer** or that represents **another resource** that is allocated in the constructor.

2. Have **bookkeeping** that must be done whenever a new object is created

13.1.2

Defining Our Own Copy Constructor

```
class Foo {
    public:
       Foo();            // default constructor
       Foo(const Foo&); // copy constructor
       // ...
    };
```

13.1.2

Preventing Copies

Some classes need to **prevent copies** from being made at all.

For example, the **iostream** classes do **not permit copying.**

Why?

```
Iostream1 = iostream2;
Iostream1.read();
Iostream2.read();
Iostream1.write();
Iostream2.write();
.
.
.
```

How to prevent?

1. Declare copy constructor private

Member function and friends

are still able to use "copy constructor"

2. Do not define it!

You'll have link-error when you use it.

13.2

The Assignment Operator

```
Sales_item trans, accum;

trans = accum;
```

The compiler synthesizes an assignment operator

if the class does not define its own.

Introducing Overloaded Assignment

```
class Sales_item {
    public:
        // other members as before
        // equivalent to the synthesized assignment operator
        Sales_item& operator=(const Sales_item &);
    };
```

The Synthesized Assignment Operator

Similar with the synthesized copy constructor

“Memberwise assignment”

The Synthesized Assignment Operator

```
// equivalent to the synthesized assignment operator
Sales_item& Sales_item::operator=(const Sales_item &rhs)
     {
        isbn = rhs.isbn;              // calls string::operator=
        units_sold = rhs.units_sold;  // uses built-in int assignment
        revenue = rhs.revenue;        // uses built-in double assignment
        return *this;
     }
```

Copy and Assign Usually Go Together

13.3

The Destructor

The **destructor** is a special member function

that can be used to do whatever **resource deallocation** is needed.

The destructor is called **automatically**

whenever an object of its class is destroyed:

Destructors are also run

on the elements of class type in a container

whether a library container or built-in array

when the container is destroyed:

```
{
      Sales_item *p = new Sales_item[10]; // dynamically allocated
      vector<Sales_item> vec(p, p + 10);  // local object
      // ...
      delete [] p; // array is freed; destructor run on each element
}   // vec goes out of scope; destructor run on each element
```

When to Write an Explicit Destructor:

The Rule of Three:

if you need a **destructor**,

then you need all three copy-control members.

The Synthesized Destructor

The synthesized destructor destroys

each **nonstatic member**

in the **reverse** order

from that in which the object was created.

The Synthesized Destructor

The synthesized destructor destroys

each **nonstatic member**

in the **reverse** order

from that in which the object was created.

!

The synthesized destructor

does not delete the object pointed to by a pointer member.

How to Write a Destructor

```
class Sales_item {
     public:
        // empty; no work to do other than destroying the members,
        // which happens automatically
         ~Sales_item() { }
        // other members as before
     };
```

13.5

Managing Pointer Members

Designing a class with a pointer member

?

“What behavior that pointer should provide”

1

The pointer member can be given normal **pointerlike** behavior.

2

The class can implement so-called "**smart pointer**" behavior.

3

The class can be given **valuelike** behavior.

1

A Simple Class with a Pointer Member

A Simple Class with a Pointer Member

```
// class that has a pointer member that behaves like a plain pointer
    class HasPtr {
    public:
       // copy of the values we're given
       HasPtr(int *p, int i): ptr(p), val(i) { }
       // const members to return the value of the indicated data member
         int *get_ptr() const { return ptr; }
         int get_int() const { return val; }

         // non const members to change the indicated data member
         void set_ptr(int *p) { ptr = p; }
         void set_int(int i) { val = i; }

       // return or change the value pointed to, so ok for const objects
         int get_ptr_val() const { return *ptr; }
         void set_ptr_val(int val) const { *ptr = val; }

     private:
         int *ptr;
         int val;
     };
```

A Simple Class with a Pointer Member

```
int obj = 0;
HasPtr ptr1(&obj, 42); // int* member points to obj, val is 42
HasPtr ptr2(ptr1);     // int* member points to obj, val is 42

ptr1.set_int(0); // changes val member only in ptr1
ptr2.get_int();  // returns 42
ptr1.get_int();  // returns 0

// sets object to which both ptr1 and ptr2 point
ptr1.set_ptr_val(42);
ptr2.get_ptr_val();   // returns 42
```

Dangling Pointers Are Possible

```
// dynamically allocated int initialized to 42
int *ip = new int(42);

HasPtr ptr(ip, 10);    // Has Ptr points to same object as ip does
delete ip;             // object pointed to by ip is freed

// disaster: The object to which Has Ptr points was freed!
ptr.set_ptr_val(0);
```

2

Defining Smart Pointer Classes

Smart Pointer

Need “Use Count”.

Smart Pointer

Responsible for deleting the shared object.

Use-Count Class

```
// private class for use by HasPtr only
     class U_Ptr {
         friend class HasPtr;
         int *ip;
         size_t use;
         U_Ptr(int *p): ip(p), use(1) { }
         ~U_Ptr() { delete ip; }
     };
```

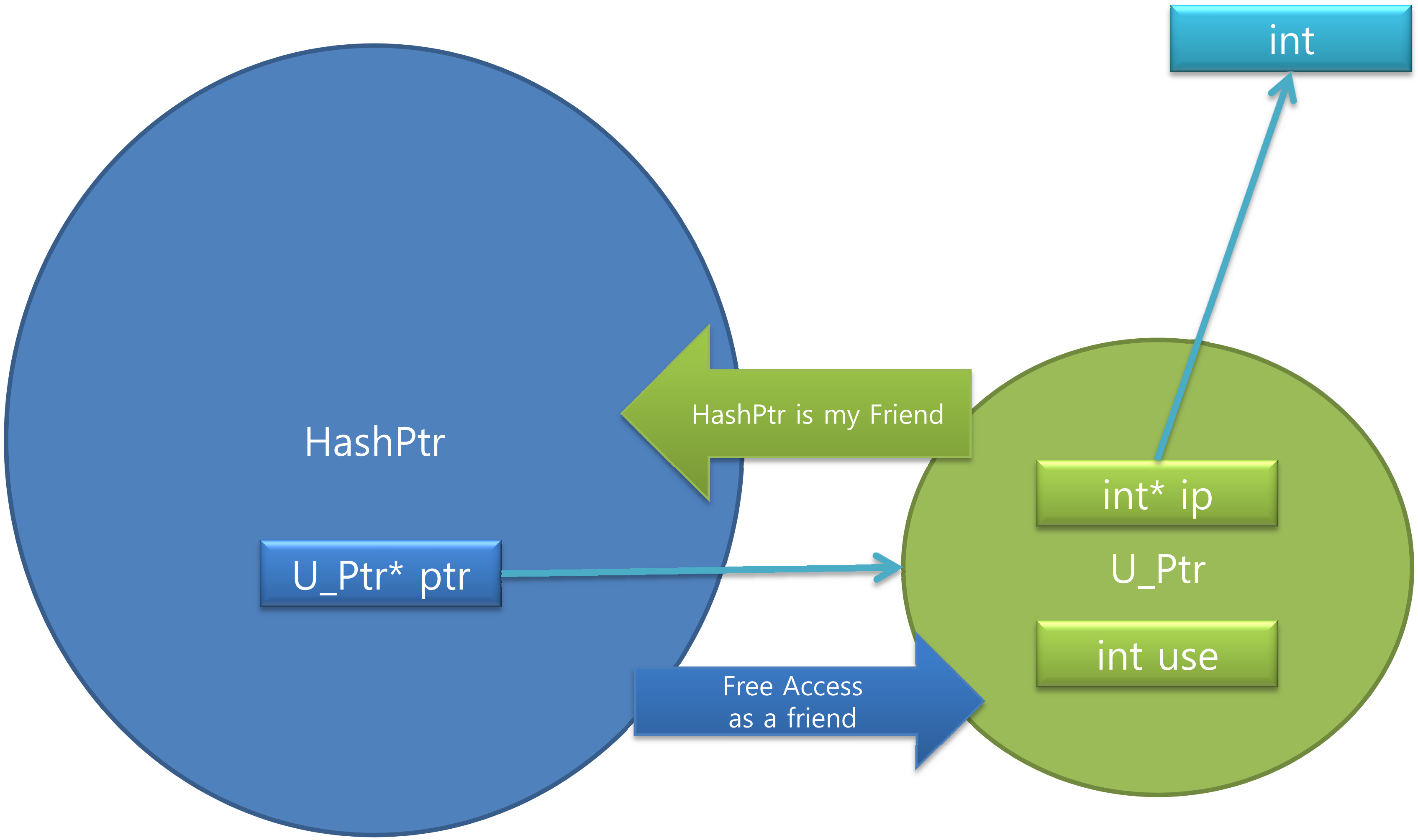
Using the Use-Counted Class

```
/* smart pointer class: takes ownership of the dynamically allocated
      *          object to which it is bound
      * User code must dynamically allocate an object to initialize a HasPtr
      * and must not delete that object; the HasPtr class will delete it
      */
     class HasPtr {
     public:
         // HasPtr owns the pointer; pmust have been dynamically allocated
         HasPtr(int *p, int i): ptr(new U_Ptr(p)), val(i) { }

         // copy members and increment the use count
         HasPtr(const HasPtr &orig):
            ptr(orig.ptr), val(orig.val) { ++ptr->use; }
         HasPtr& operator=(const HasPtr&);

         // if use count goes to zero, delete the U_Ptr object
         ~HasPtr() { if (--ptr->use == 0) delete ptr; }
     private:
         U_Ptr *ptr;        // points to use-counted U_Ptr class
         int val;
     };
```

Constructor

Copy Con& Assignment

Destructor


int
HashPtr is my Friend
HashPtr
int* ip
U_Ptr* ptr
U_Ptr
int use
Free Access
as a friend

Using the Use-Counted Class

Assignment Operator

```
HasPtr& HasPtr::operator=(const HasPtr &rhs)
     {
         ++rhs.ptr->use;     // increment use count on rhs first
         if (--ptr->use == 0)
             delete ptr;    // if use count goes to 0 on this object, delete it

         ptr = rhs.ptr;      // copy the U_Ptr object
         val = rhs.val;      // copy the int member
         return *this;
     }
```

Using the Use-Counted Class

Get Method

Set Method

```
class HasPtr {
    public:
        // copy control and constructors as before
        // accessors must change to fetch value from U_Ptr object
        int *get_ptr() const { return ptr->ip; }
        int get_int() const { return val; }

        // change the appropriate data member
        void set_ptr(int *p) { ptr->ip = p; }
        void set_int(int i) { val = i; }

        // return or change the value pointed to, so ok for const objects
        // Note: *ptr->ip is equivalent to *(ptr->ip)
        int get_ptr_val() const { return *ptr->ip; }
        void set_ptr_val(int i) { *ptr->ip = i; }
    private:
        U_Ptr *ptr;        // points to use-counted U_Ptr class
        int val;
    };
```

Using the Use-Counted Class

```
/* smart pointer class: takes ownership of the dynamically allocated
      *          object to which it is bound
      * User code must dynamically allocate an object to initialize a HasPtr
      * and must not delete that object; the HasPtr class will delete it
      */
     class HasPtr {
     public:
         // HasPtr owns the pointer; pmust have been dynamically allocated
         HasPtr(int *p, int i): ptr(new U_Ptr(p)), val(i) { }

         // copy members and increment the use count
         HasPtr(const HasPtr &orig):
            ptr(orig.ptr), val(orig.val) { ++ptr->use; }
         HasPtr& operator=(const HasPtr&);

         // if use count goes to zero, delete the U_Ptr object
         ~HasPtr() { if (--ptr->use == 0) delete ptr; }
     private:
         U_Ptr *ptr;        // points to use-counted U_Ptr class
         int val;
     };
```

3

Defining Valuelike Classes

Defining Valuelike Classes

```
/*
     * Valuelike behavior even though HasPtr has a pointer member:
     * Each time we copy a HasPtr object, we make a new copy of the
     * underlying int object to which ptr points.
     */
    class HasPtr {
    public:
        // no point to passing a pointer if we're going to copy it anyw
        // store pointer to a copy of the object we're given
        HasPtr(const int &p, int i): ptr(new int(p)), val(i) {}

        // copy members and increment the use count
        HasPtr(const HasPtr &orig):
           ptr(new int (*orig.ptr)), val(orig.val) { }

        HasPtr& operator=(const HasPtr&);
        ~HasPtr() { delete ptr; }
        // accessors must change to fetch value from Ptr object
        int get_ptr_val() const { return *ptr; }
        int get_int() const { return val; }

        // change the appropriate data member
        void set_ptr(int *p) { ptr = p; }
        void set_int(int i) { val = i; }

        // return or change the value pointed to, so ok for const objects
        int *get_ptr() const { return ptr; }
        void set_ptr_val(int p) const { *ptr = p; }
    private:
        int *ptr;        // points to an int
        int val;
    };
```

Constructor

Copy Constructor

Assignment

Destructor

Get Method

Set Method

Defining Valuelike Classes

Assignment Operator

```
HasPtr& HasPtr::operator=(const HasPtr &rhs)
     {
         // Note: Every HasPtr is guaranteed to point at an actual int;
         //    We know that ptr cannot be a zero pointer
         *ptr = *rhs.ptr;       // copy the value pointed to
         val = rhs.val;         // copy the int
         return *this;
     }
```

Thank you.