**Item 13: Use objects to manage resources.**

class Investment { ... }; // root class of hierarchy of investment types

below method return ptr to dynamically allocated object in the Investment hierarchy the caller must delete it (parameters omitted for simplicity)

Investment\* createInvestment();

void f()

{

Investment \*pInv = createInvestment(); // call factory function

... // use pInv

delete pInv; // release object

}

There might be a premature return statement somewhere inside the “...” part of the function. If such a return were executed, control would never reach the delete statement. Result memory leak.

The standard library’s auto\_ptr is tailor made for this kind of situation. auto\_ptr is a pointer-like object (a smart pointer) whose destructor automatically calls delete on what it points to. Here’s how to use auto\_ptr to prevent f’s potential resource leak:

void f()

{

std::auto\_ptr<Investment> pInv(createInvestment()); // call factory function

... // use pInv as before

} // automatically delete pInv via auto\_ptr’s dtor

This simple example demonstrates the two critical aspects of using objects to manage resources:

* **Resources are acquired and immediately turned over to resource-managing objects.**

The idea of using objects to manage resources is often called *Resource Acquisition Is Initialization* (RAII).

* **Resource-managing objects use their destructors to ensure that resources are released.**

auto\_ptr automatically deletes what it points to when the auto\_ptr is destroyed, it’s important that there never be more than one auto\_ptr pointing to an object. If there were, the object would be deleted more than once, and would lead the undefined behavior of program . To prevent such problems, auto\_ptrs have an unusual characteristic:

Copying them (via copy constructor or copy assignment operator) sets them to null, and the copying pointer assumes sole ownership of the resource.

std::auto\_ptr<Investment> pInv1(createInvestment()); // pInv1 points to the object

// returned from

//createInvestment

std::auto\_ptr<Investment> pInv2(pInv1); // pInv2 now points to the object pInv1 is now null

pInv1 = pInv2; // now pInv1 points to the object,

// and pInv2 is null

Resources managed by auto\_ptrs must never have more than one auto\_ptr pointing to them.

***STL containers require that their contents exhibit “normal” copying behavior, so containers of auto\_ptr aren’t allowed.***

An alternative to auto\_ptr is a *reference-counting smart pointer* (RCSP). An RCSP is a smart pointer that keeps track of how many objects point to a particular resource and automatically deletes the resource when nobody is pointing to it any longer. As such, RCSPs offer behavior that is similar to that of garbage collection.

void f()

{

...

// pInv1 points to the object returned from createInvestment

std::tr1::shared\_ptr<Investment> pInv1(createInvestment());

std::tr1::shared\_ptr<Investment> pInv2(pInv1); // both pInv1 and pInv2 now

// point to the object.

pInv1 = pInv2; // ditto — nothing has changed

...

} //pInv1 and pInv2 are destroyed, and the object they point to is automatically deleted.

***Because copying shared\_ptrs works “as expected,” they can be used in STL containers and other contexts where auto\_ptr’s unorthodox copying behavior is inappropriate.***

Both auto\_ptr and tr1::shared\_ptr use delete in their destructors, not delete []. That means that using auto\_ptr or tr1::shared\_ptr with dynamically allocated arrays is a bad idea.

std::auto\_ptr<std::string> aps(new std::string[10]); // bad idea! the wrong

// delete form will be used

std::tr1::shared\_ptr<int> spi(new int[1024]); // same problem

That’s because vector and string can almost always replace dynamically allocated arrays in c++.

**Things to Remember**

✦ To prevent resource leaks, use RAII objects that acquire resources in their constructors and release them in their destructors.

✦ Two commonly useful RAII classes are tr1::shared\_ptr and auto\_ptr. tr1::shared\_ptr is usually the better choice, because its behavior when copied is intuitive. Copying an auto\_ptr sets it to null.

**Item 14: Think carefully about copying behavior in resource-managing classes.**

RAII is the backbone of resource-managing classes. Not all resources are heap-based, however, for such resources, smart pointers like auto\_ptr and tr1::shared\_ptr are generally inappropriate as resource handlers.

For that need to create your own resource managing classes.

For example, suppose we are using a C API to manipulate mutex objects of type Mutex offering functions lock and unlock:

void lock(Mutex \*pm); // lock mutex pointed to by pm

void unlock(Mutex \*pm); // unlock the mutex

To make sure that you never forget to unlock a Mutex you’ve locked, you’d like to create a class to manage locks. The basic structure of such a class is dictated by the RAII principle that resources are acquired during construction and released during destruction:

class Lock {

public:

explicit Lock(Mutex \*pm): mutexPtr(pm){

lock(mutexPtr); // acquire resource

}

~Lock() {

unlock(mutexPtr); // release resource

}

private:

Mutex \*mutexPtr;

};

Clients use Lock in the conventional RAII fashion:

Mutex m; // define the mutex you need to use

...

{ // create block to define critical section

Lock ml(&m); // lock the mutex

... // perform critical section operations

} // automatically unlock mutex at end of block

This is fine, but what should happen if a Lock object is copied?

Lock ml1(&m); // lock m

Lock ml2(ml1); // copy ml1 to ml2 — what should

what should happen when an RAII object is copied?

Most of the time, you’ll want to choose one of the following possibilities:

1. **Prohibit copying:**
2. **Reference-count the underlying resource:**

Sometimes it’s desirable to hold on to a resource until the last object using it has been destroyed like a shared\_ptr.

Consider to our example, tr1::shared\_ptr’s default behavior is to delete what it

points to when the reference count goes to zero, and that’s not what we want.

When we’re done with a Mutex, we want to unlock it, not delete it.

shared\_ptr allows specification of a “***deleter***” — a function or function object to be called when the reference count goes to zero. (This functionality does not exist for auto\_ptr, which *always* deletes its pointer.) The deleter is an optional second parameter to the shared\_ptr constructor, so the code would look like this:

class Lock {

public:

explicit Lock(Mutex \*pm) // init shared\_ptr with the Mutex

: mutexPtr(pm, unlock) // to point to and the unlock function

{ // as the deleter

lock(mutexPtr.get()); //Get the raw pinter on Mutex.

}

private:

std::tr1::shared\_ptr<Mutex> mutexPtr; // use shared\_ptr instead of raw

// pointer

};

In this example, notice how the Lock class no longer declares a destructor. That’s

because there’s no need to (It automatically invokes the destructors of the class’s

non-static data members.). mutexPtr’s destructor will automatically call the

tr1::shared\_ptr’s deleter — unlock, in this case when the mutex’s reference count

goes to zero.

1. **Copy the underlying resource:**

Sometimes we need the multiple copies of a resource, and the only reason you need a resource-managing class is to make sure that each copy is released when you’re done with it. In that case, copying the resource-managing object should also copy the resource and copying a resource-managing object performs a “deep copy.”

1. **Transfer ownership of the underlying resource:**

Sometimes it is required that only one RAII object refers to a raw resource and that when the RAII object is copied, ownership of the resource is transferred from the copied object to the copying object (Like auto\_ptr).

**Things to Remember**

✦ Copying an RAII object entails copying the resource it manages, so the copying

behavior of the resource determines the copying behavior of the RAII object.

✦ Common RAII class copying behaviors are disallowing copying and performing reference counting, but other behaviors are possible.

**Item 15: Provide access to raw resources in resource managing classes.**

std::tr1::shared\_ptr<Investment> pInv(createInvestment()); //Item 13 for Investment //class.

Suppose that a function like to use when working with Investment objects is this:

int daysHeld(const Investment \*pi); // return number of days investment has been

// held

You’d like to call it like this,

int days = daysHeld(pInv); // error!

But the above code won’t compile: daysHeld wants a raw Investment\* pointer, but we are passing an object of type shared\_ptr<Investment>

We need a way to convert an object of the RAII class (shared\_ptr) into the raw resource it contains (Investment\*). There are two general ways to do it:

1. **Explicit conversion**

Both tr1::shared\_ptr and auto\_ptr offers to get member function to perform an explicit conversion, i.e., to return (a copy of) the raw pointer inside the smart pointer object:

int days = daysHeld(pInv.get()); // fine, passes the raw pointer in pInv to

// daysHeld

Other example considers this RAII class for fonts.

FontHandle\* getFont(); // Params omitted for simplicity

void releaseFont(FontHandle \*fh);

class Font { // RAII class

public:

explicit Font(FontHandle \*fh) : f(fh){} // acquire resource

~Font() { releaseFont(f ); } // release resource

FontHandle\* get() const { return f; } // explicit conversion function

... // handle copying (see Item14)

private:

FontHandle \*f; // the raw font resource

};

void changeFontSize(FontHandle f, int newSize); // Change font method

Font getFont() // Return new Font

Font f(getFont()); // Create new font

int newFontSize;

...

changeFontSize(f.get(), newFontSize); // explicitly convert Font to FontHandle

1. **Implicit conversion**

shared\_ptr and auto\_ptr also overload the pointer dereferencing operators (operator-> and operator\*), and this allows implicit conversion to the underlying raw pointers:

class Investment { // root class for a hierarchy of investment types

public:

bool isTaxFree() const;

...

};

Investment\* createInvestment(); // factory function

// have tr1::shared\_ptr manage a resource

std::tr1::shared\_ptr<Investment> pi1(createInvestment());

bool taxable1 = !(pi1->isTaxFree()); // access resource via operator->, Implicit

//conversion shared\_ptr to raw pointer.

...

std::auto\_ptr<Investment> pi2(createInvestment()); // have auto\_ptr manage a

// resource

bool taxable2 = !((\*pi2).isTaxFree()); // access resource via operator\*

**Things to Remember**

✦ APIs often require access to raw resources, so each RAII class should offer a way to get at the resource it manages.

✦ Access may be via explicit conversion or implicit conversion. In general, explicit conversion is safer, but implicit conversion is more convenient for clients.

**Item 16: Use the same form in corresponding uses of new and delete.**

std::string \*stringArray = new std::string[100];

...

delete stringArray;

The program’s behavior is undefined. At least 99 of the 100 string objects pointed to by stringArray are not properly destroyed, because their destructors will probably never be called.

When we employ a new *expression,* two things happen:

First, memory is allocated (via a function named operator new) and second, one or more constructors are called for that memory.

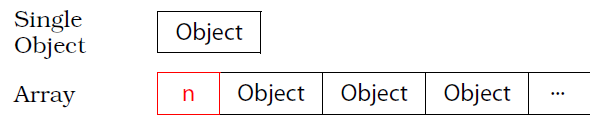
When you employ a delete *expression*, similarly two other things happen:

One or more destructors are called for the memory, then the memory is deallocated (via a function named operator delete).

The big question for delete is this: *how many* objects reside in the memory being deleted?

The answer to that determines how many destructors must be called (pointer being deleted point to a single object or to an array of objects). It’s a critical question, because the memory layout for single objects is generally different from the memory layout for arrays.

The memory for an array usually includes the size of the array, thus making it easy for delete to know how many destructors to call. The memory for a single object lacks this information.



Widget \*w= new Widget();

….

Delete [] w;

The result is undefined. Assuming the layout above, delete would read some memory and interpret what it read as an array size, then start invoking that many destructors which is not in this case at least.

**Things to Remember**

✦ If you use [] in a new expression, you must use [] in the corresponding delete expression. If you don’t use [] in a new expression, you mustn’t use [] in the corresponding delete expression.

**Item 17: Store newed objects in smart pointers in standalone statements.**

Suppose we have two functions:

int priority(); // Reveal our processing priority.

void processWidget(shared\_ptr<Widget> pw, int priority); // Do some processing on a

//dynamically allocated

// Widget in accord with a

//priority:

processWidget uses a smart pointer (shared\_ptr) for the dynamically allocated Widget it processes. Now call like:

processWidget(new Widget, priority());

It won’t compile. shared\_ptr’s constructor taking a raw pointer is explicit, so there’s no implicit conversion from the raw pointer returned by the expression “new Widget” to the shared\_ptr required by processWidget.

The following code, however, it will compile:

processWidget(std::tr1::shared\_ptr<Widget>(new Widget), priority());

Although, we are using RAII approach still there is a chance to memory leak.

Before compilers can generate a call to processWidget, first it evaluates the arguments being passed as its parameters. The second argument is just a call to the function priority, but the first argument consists of two parts:

■ Execution of the expression “new Widget”.

■ A call to the tr1::shared\_ptr constructor.

Before processWidget can be called, then, compilers must generate code to do these three things:

■ Call priority.

■ Execute “new Widget”.

■ Call the tr1::shared\_ptr constructor.

But order of parameter processing is fully depending on compiler. Here “new Widget” expression must be executed before the tr1::shared\_ptr constructor can be called. But the call to priority can be performed first, second, or third. But still order of execution is compiler dependent, so let’s suppose below sequence:

1. Execute “new Widget”.

2. Call priority.

3. Call the tr1::shared\_ptr constructor

But consider what will happen if the call to priority yields an exception. In that case, the pointer returned from “new Widget” will be lost.

The way to avoid problems like this is simple: use a separate statement to create the Widget and store it in a smart pointer, then pass the smart pointer to processWidget:

std::tr1::shared\_ptr<Widget> pw(new Widget); // store newed object in a smart

// pointer in a standalone

// statement

processWidget(pw, priority()); // this call won’t leak

**Things to Remember**

✦ Store newer objects in smart pointers in standalone statements. Failure to do this can lead to subtle resource leaks when exceptions are thrown.