Laboratorio di Tecnologie dell'Informazione

Ing. Marco Bertini
marco.bertini@unifi.it
http://www.micc.unifi.it/bertini/

Resource Management

Memory, smart pointers and RAII

Resource management

- The most commonly used resource in C++ programs is memory
  - there are also file handles, mutexes, database connections, etc.
- It is important to release a resource after that it has been used

An example

```
class Vehicle { ... }; // root class of a hierarchy

Vehicle* createVehicle(); /* return a pointer to root
class but may create any other object in the hierarchy.
The caller MUST delete the returned object */

void f() {
  Vehicle* pV = createVehicle();
  //... use pV
  delete pV;
}
```

An example

```
class Vehicle { ... }; // root class of a hierarchy

Vehicle* createVehicle(); /* return a pointer to root
class but may create any other object in the hierarchy.
The caller MUST delete the returned object */

void f() {
  Vehicle* pV = createVehicle();
  //... use pV
  delete pV;
}
```

If there's a premature return or an exception we may never reach the delete !

A solution

- Put the resource returned by `createVehicle` inside an object whose destructor automatically release the resource when control leaves `f()`.
  - destructor calls are automatic
- With these objects that manage resources:
  - resources are acquired and immediately turned over to resource-managing objects (RAII)
  - these objects use their destructors to ensure that resources are released

RAII

Resource **A**cquisition **I**s **I**nitialization

What is RAII

- This technique was invented by Stroustrup to deal with resource deallocation in C++ and to write exception-safe code: the only code that can be guaranteed to be executed after an exception is thrown are the destructors of objects residing on the stack.
- This technique allows to release resources before permitting exceptions to propagate (in order to avoid resource leaks)

What is RAII - cont.

- Resources are tied to the lifespan of suitable objects.
  They are acquired during initialization, when there is no chance of them being used before they are available.
  They are released with the destruction of the same objects, which is guaranteed to take place even in case of errors.

RAII example

```
#include <cstdio>
#include <stdexcept> // std::runtime_error

class file {
public:
    file (const char* filename) : file_(std::fopen(filename, "w+")) {
        if (!file_) {
            throw std::runtime_error("file open failure");
        }
    }
    ~file() {
        if (std::fclose(file_)) {
            // failed to flush latest changes.
            // handle it
        }
    }
    void write (const char* str) {
        if (EOF == std::fputs(str, file_)) {
            throw std::runtime_error("file write failure");
        }
    }
private:
    std::FILE* file_;
    // prevent copying and assignment; not implemented
    file (const file &);
    file & operator= (const file &);
};
```

RAII example

```
#include <cstdio>
#include <stdexcept> // std::runtime_error

class file {
public:
    file (const char* filename) : file_(std::fopen(filename, "w+")) {
        if (!file_) {
            throw std::runtime_error("file open failure");
        }
    }
    ~file() {
        if (std::fclose(file_)) {
            // failed to flush latest changes.
            // handle it
        }
    }
    void write (const char* str) {
        if (EOF == std::fputs(str, file_)) {
            throw std::runtime_error("file write
        }
    }
private:
    std::FILE* file_;
    // prevent copying and assignment; not implemented
    file (const file &);
    file & operator= (const file &);
};
```

```
void example_usage() {
    // open file (acquire resource)
    file logfile("logfile.txt");
    logfile.write("hello logfile!");
    // continue using logfile ...
    // throw exceptions or return without
    //  worrying about closing the log;
    // it is closed automatically when
    // logfile goes out of scope
}
```

Smart pointers

What is a smart pointer ?

- In C++, smart pointers are classes that mimic, by means of operator overloading, the behaviour of traditional (raw) pointers, (e.g. dereferencing, assignment) while providing additional memory management algorithms.
- Old C++98 provided only `auto_ptr<>`, C++11 introduces many new smart pointers:
  - Should start to use smart pointer whenever possible, and use "raw" pointers only when necessary

Raw vs. smart pointer

- With raw pointers the programmer is in charge with deleting them:

```
// Need to create the object to achieve some goal

MyObject* ptr = new MyObject();

ptr->DoSomething();// Use the object in some way.

delete ptr; // Destroy the object. Done with it.

// Wait, what if DoSomething() raises an exception....
```

- With smart pointers there is need to create the object, but its destruction is up to the policy defined by the smart pointer

C++11 smart pointers

- `shared_ptr`: implements shared ownership. Any number of these smart pointers jointly own the object. The owned object is destroyed only when its last owning smart pointer is destroyed.
- `weak_ptr`: doesn't own an object at all, and so plays no role in when or whether the object gets deleted. Rather, a weak_ptr merely observes objects being managed by shared_ptrs, and provides facilities for determining whether the observed object still exists or not. C++11's weak_ptrs are used with shared_ptrs.
- `unique_ptr`: implements unique ownership - only one smart pointer owns the object at a time; when the owning smart pointer is destroyed, then the owned object is automatically destroyed.
- Just use: `#include <memory>`

unique_ptr

unique_ptr

- Exclusive ownership semantics, i.e., at any point of time, the resource is owned by only one unique_ptr. When unique_ptr goes out of scope, the resource is released.
- Solves the problem of transfer of ownership that are present in the (deprecated) auto_ptr
  - copy constructor and assignment operator are declared as private
- Can be used in STL containers and algorithms

unique_ptr and STL

- You can fill a Standard Container with unique_ptrs that own objects, and the ownership then effectively resides in the container.

- If you erase an item in the container, you are destroying the unique_ptr, which will then delete the object it is pointing to.

```
std::vector<std::unique_ptr<Thing>> v;
...
if(v[3]) // check that v[3] still owns an object
  v[3]->foo();!// tell object pointed to by v[3] to foo
```

unique_ptr and RAII

- Reconsider the f() function using unique_ptr:

```
void f() {
  std::unique_ptr<Vehicle>pV(createVehicle());
  // use pV as before...
} /* the magic happens here: automatically
deletes pV via the destructor of unique_ptr,
called because it's going out of scope */
```

unique_ptr: another example

- In general here's how to rewrite unsafe code in safe code:

```
// Original code
void f() {
  T* pt( new T );
  /*...more code...*/
  delete pt;
}
```

```
//Safe code, with auto_ptr
void f() {
  unique_ptr<T> pt( new T );
  /*...more code...*/
} /* pt's destructor is called
as it goes out of scope, and
the object is deleted
automatically */
```

unique_ptr vs. auto_ptr

- Consider unique_ptr an improved version of auto_ptr. It has an almost identical interface:

- ```
  #include <utility>
  using namespace std;
  unique_ptr<int> up1; //default construction
  unique_ptr<int> up2(new int(9)); //initialize with pointer
  *up2 = 23; //dereference
  up2.reset(); //reset
  ```

- The main difference between `auto_ptr` and `unique_ptr` is visible in move operations. While auto_ptr sometimes disguises move operations as copy-operations, `unique_ptr` will not let you use copy semantics when you're actually moving an lvalue `unique_ptr`:

- ```
  auto_ptr<int> ap1(new int);
  auto_ptr<int> ap2=ap1; // OK but unsafe: move
                         // operation in disguise
  unique_ptr<int> up1(new int);
  unique_ptr<int> up2=up1; // compilation error: private
                           // copy ctor inaccessible
  ```

  Instead, you must call `move()` when moving operation from an lvalue:

  ```
  unique_ptr<int> up2 = std::move(up1);//OK
  ```

shared_ptr

shared_ptr

- std::shared_ptr represents reference-counted ownership of a pointer. Each copy of the same shared_ptr owns the same pointer. That pointer will only be freed if all instances of the shared_ptr in the program are destroyed.
- Can be used in STL containers: the copied shared_ptr will increase reference count

shared_ptr - example

```
#include <iostream>
#include <memory>

using namespace std;

class Car {
public:
   Car(int eng=500) : engine(eng) {}
   void turnOn() {engineOn=true; cout <<
"Engine on\n";}
   void turnOff() {engineOn=false; cout
<< "Engine off\n";}
   int getEngine() {return engine;}

private:
   int engine;
   bool engineOn {false};
};
```

```
int main(int argc, char *argv[]) {
   std::shared_ptr<Car> p1=make_shared<Car>(1400);
   // you can use also:
    // std::shared_ptr<Car> p1(new Car(1400));
   std::shared_ptr<Car> p2 = p1; // Both now own
                                 // the memory.

   // call methods of the shared object
   (*p1).turnOn();
   p2->turnOff();

   // call methods of the smart pointer
   cout << p2.use_count() << endl;

   p1.reset(); //Memory still exists, due to p2.
   cout << "Engine: " << p2->getEngine() << endl;
   p2.reset(); // Deletes the memory, since no one
               // else owns the memory.
}
```

shared_ptr - example

- Two references to the same object
- Counts the number of references
- Eliminates one of the references

```
#include <iostream>
#include <memory>

int main(int argc, char *argv[]) {
   std::shared_ptr<Car> p1=make_shared<Car>(1400);
   // you can use also:
    // std::shared_ptr<Car> p1(new Car(1400));
   std::shared_ptr<Car> p2 = p1; // Both now own
                                 // the memory.

<<
   // call methods of the shared object
t  (*p1).turnOn();
   p2->turnOff();

   // call methods of the smart pointer
   cout << p2.use_count() << endl;

   p1.reset(); //Memory still exists, due to p2.
   cout << "Engine: " << p2->getEngine() << endl;
   p2.reset(); // Deletes the memory, since no one
               // else owns the memory.
}
```

shared_ptr - example

```
#include <iostream>
#include <memory>

int main(int argc, char *argv[]) {
   std::shared_ptr<Car> p1=make_shared<Car>(1400);
   // you can use also:
    // std::shared_ptr<Car> p1(new Car(1400));
   std::shared_ptr<Car> p2 = p1; // Both now own
                                 // the memory.

<<
   // call methods of the shared object
t  (*p1).turnOn();
   p2->turnOff();

   // call methods of the smart pointer
   cout << p2.use_count() << endl;

   p1.reset(); //Memory still exists, due to p2.
   cout << "Engine: " << p2->getEngine() << endl;
   p2.reset(); // Deletes the memory, since no one
```

- Two references to the same object
- Counts the number of references
- Eliminates one of the references

Remind to use the compiler switches to activate C++11, like `-std=c++11 -stdlib=libc++`

shared_ptr+STL - example

```
#include <iostream>
#include <string>
#include <algorithm>
#include <vector>
#include <memory>

using namespace std;

class Song {
public:
    Song(string name, string t) : artist(name), title(t)
{}
    string artist, title;
};

int main(int argc, char *argv[]) {
    shared_ptr<Song> p1 = make_shared<Song>("Bob Dylan",
"The Times They Are A Changing");
    shared_ptr<Song> p2 = make_shared<Song>("Aretha
Franklin", "Bridge Over Troubled Water");
    shared_ptr<Song> p3 = make_shared<Song>("Francesco
Guccini", "Il vecchio e il bambino");
    vector<shared_ptr<Song>> v;

    v.push_back(p1);
    v.push_back(p2);
    v.push_back(p3);
    vector<shared_ptr<Song>> v2;
    // see slides on lambda functions...
    remove_copy_if(v.begin(), v.end(), back_inserter(v2),
[] (shared_ptr<Song> s)
    {
        return s->artist.compare("Francesco Guccini")
                    == 0;
    });

    for_each(v2.begin(), v2.end(), [](shared_ptr<Song> s)
    {
        cout << s->artist << ": " << s->title << endl;
    });
    // see slides on lambda functions...
    v.pop_back();
    cout << p1->artist << ": " << p1.use_count() << endl;
    cout << p2->artist << ": " << p2.use_count() << endl;
    cout << p3->artist << ": " << p3.use_count() << endl;
}
```

shared_ptr+STL - example

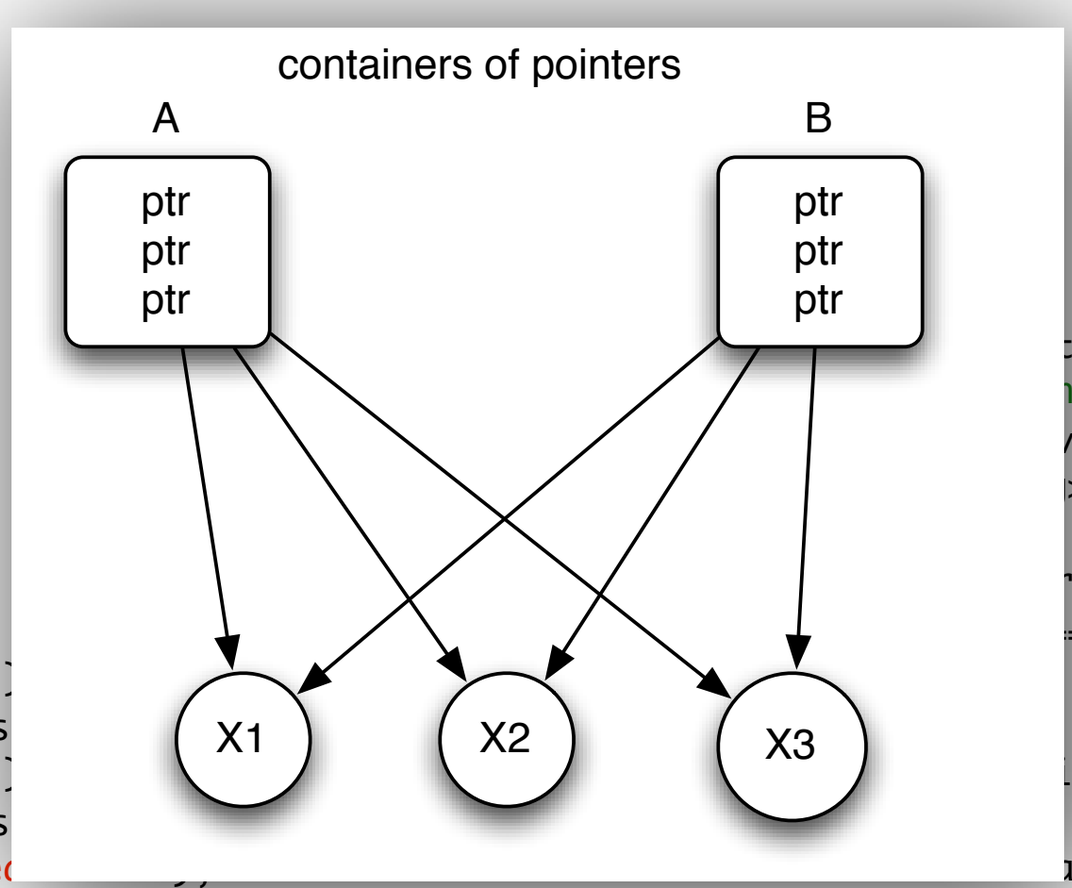
```
#include <iostream>
#include <string>
#include <algorithm>
#include <vector>
#include <memory>

using namespace std;

class Song {
public:
    Song(string name, string t)
{}
    string artist, title;
};

int main(int argc, char *argv[]
    shared_ptr<Song> p1 = make_s
"The Times They Are A Changing"
    shared_ptr<Song> p2 = make_s
Franklin", "Bridge Over Troubled
    shared_ptr<Song> p3 = make_shared<Song>("Francesco
Guccini", "Il vecchio e il bambino");
    vector<shared_ptr<Song>> v;

    v.push_back(p1);
    v.push_back(p2);
    v.push_back(p3);
```




```
r<Song>> v2;
 lambda functions...
.begin(), v.end(), back_inserter(v2),
> s)

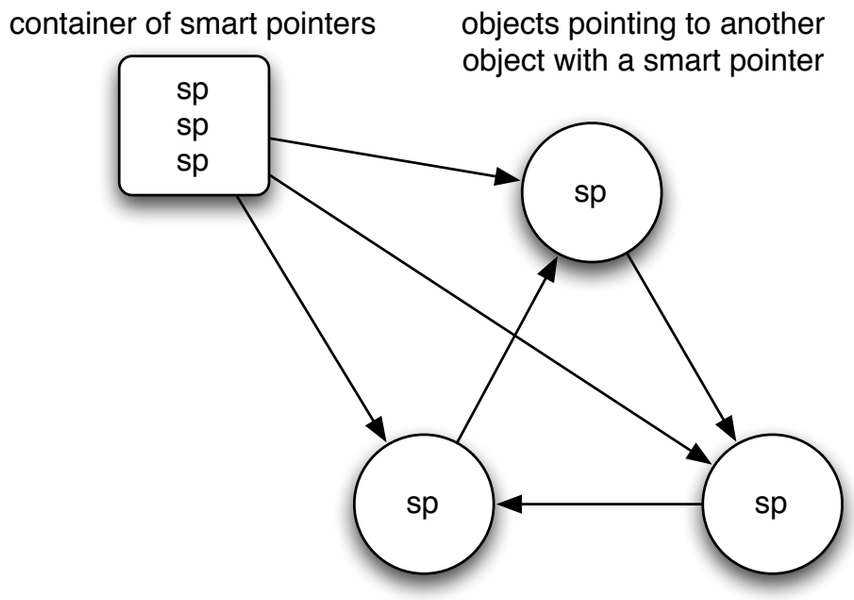
rtist.compare("Francesco Guccini")
= 0;


n(), v2.end(), [](shared_ptr<Song> s)

rtist << ": " << s->title << endl;
    });
    // see slides on lambda functions...
    v.pop_back();
    cout << p1->artist << ": " << p1.use_count() << endl;
    cout << p2->artist << ": " << p2.use_count() << endl;
    cout << p3->artist << ": " << p3.use_count() << endl;
}
```

shared_ptr: problems

- A problem with reference-counted smart pointers is that if there is a ring, or cycle, of objects that have smart pointers to each other, they keep each other “alive” - they won't get deleted even if no other objects are pointing to them from "outside" of the ring.



shared_ptr: problems - example

```
class B;
class A {
public:
    A() : m_sptrB(nullptr) { };
    ~A() {
     cout<<" A is destroyed"<<endl;
    }
    shared_ptr<B> m_sptrB;
};

class B {
public:
    B() : m_sptrA(nullptr) { };
    ~B() {
     cout<<" B is destroyed"<<endl;
    }
    shared_ptr<A> m_sptrA;
};
```

```
void main( ) {
 shared_ptr<B> sptrB( new B );
 shared_ptr<A> sptrA( new A );
 sptrB->m_sptrA = sptrA;
 sptrA->m_sptrB = sptrB;
}
```

shared_ptr: problems - example

```
class B;
class A {
public:
    A() : m_sptrB(nullptr) { };
    ~A() {
     cout<<" A is destroyed"<<endl;
    }
    shared_ptr<B> m_sptrB;
};

class B {
public:
    B() : m_sptrA(nullptr) { };
    ~B() {
     cout<<" B is destroyed"<<endl;
    }
    shared_ptr<A> m_sptrA;
};
```

	Reference count
shared_ptr<B> sptrB(new B)	1
shared_ptr<A> sptrA(new A)	1
sptrB->m_sptrA = sptrA	Ref count of sptrA -> 2
sptrA->m_sptrB = sptrB	Ref count of sptrB ->2
main ends -> sptrA goes out of scope	Ref count of sptrA ->1
main ends -> sptrB goes out of scope	Ref count of sptrB ->1

```
void main( ) {
 shared_ptr<B> sptrB( new B );
 shared_ptr<A> sptrA( new A );
 sptrB->m_sptrA = sptrA;
 sptrA->m_sptrB = sptrB;
}
```

weak_ptr

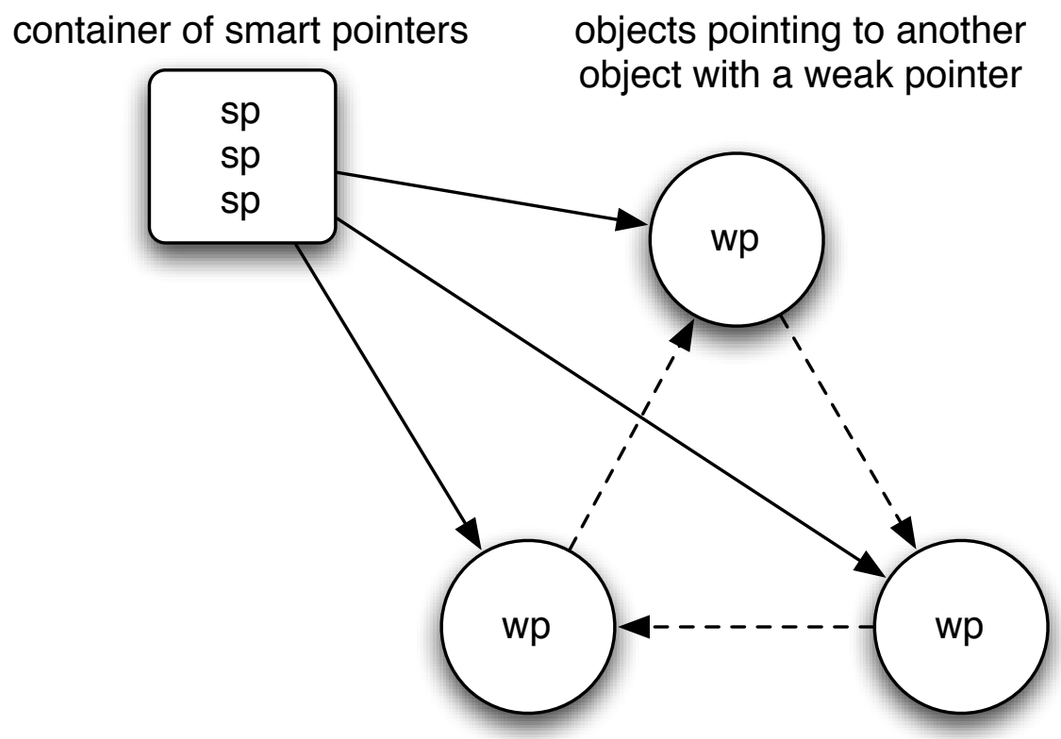
weak_ptr: helping shared_ptr

- `std::weak_ptr`: a weak pointer provides sharing semantics and not owning semantics.
- This means a weak pointer can share a resource held by a `shared_ptr`.
  - So to create a weak pointer, somebody should already own the resource which is nothing but a shared pointer.
  - To break up cycles, `weak_ptr` can be used to access the stored object. The stored object will be deleted if the only references to the object are `weak_ptr` references.
- `weak_ptr` therefore does not ensure that the object will continue to exist, but it can ask for the resource.

weak_ptr: how to use it

- A weak pointer does not allow normal interfaces supported by a pointer, like calling *, ->. Because it is not the owner of the resource and hence it does not give any chance for the programmer to mishandle it.
- Create a `shared_ptr` out of a `weak _ptr` and use it. This makes sure that the resource won't be destroyed while using it by incrementing the strong reference count.
  - As the reference count is incremented, it is sure that the count will be at least 1 till you complete using the shared_ptr created out of the weak_ptr. Otherwise what may happen is while using the weak_ptr, the resource held by the shared_ptr goes out of scope and the memory is released which creates chaos.
- To get a `shared_ptr` from a weak_ptr call `lock()` or directly casting the `weak_ptr` to `shared_ptr`.

weak_ptr and shared_ptr



- If the container of smart pointers is emptied, the three objects in the ring will get automatically deleted because no other smart pointers are pointing to them; like raw pointers, the weak pointers don't keep the pointed-to object "alive".

- The cycle problem is solved. But unlike raw pointers, the weak pointers "know" whether the pointed-to object is still there or not and can be interrogated about it, making them much more useful than a simple raw pointer would be.

weak_ptr: example

```
#include <iostream>
#include <memory>
using namespace std;

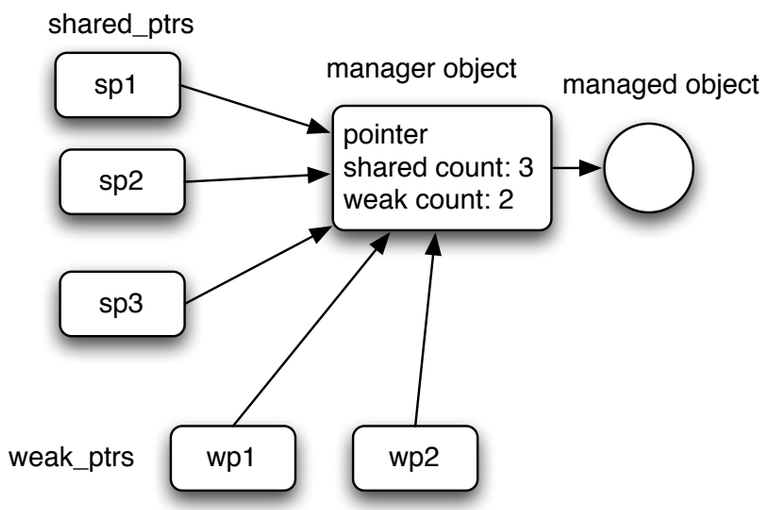
int main() {
   std::shared_ptr<int> p1(new int(5));
   std::weak_ptr<int> wp1 = p1; //p1 owns the memory.

   { // scope of p2
     std::shared_ptr<int> p2 = wp1.lock(); //Now p1 and p2 own the memory.
     if(p2) //Always check to see if the memory still exists
     {
       //Do something with p2
     }
   } //p2 is destroyed. Memory is owned by p1.

   p1.reset(); //Memory is deleted.

   std::shared_ptr<int> p3 = wp1.lock(); //Memory is gone, so we get an empty shared_ptr.
   if(p3)
   {
     //Will not execute this.
   }
}
```

weak_ptr: how it works



shared_ptr creates the manager object that contains a pointer to the managed object
The manager object counts the references to the managed object

- Whenever a shared_ptr is destroyed, or reassigned to point to a different object, the shared_ptr destructor or assignment operator decrements the shared count.
- Similarly, destroying or reassigning a weak_ptr will decrement the weak count.
- When the shared count reaches zero, the shared_ptr destructor deletes the managed object and sets the pointer to 0.
- If the weak count is also zero, then the manager object is deleted also, and nothing remains.
- But if the weak count is greater than zero, the manager object is kept. If the weak count is decremented to zero, and the shared count is also zero, the weak_ptr destructor deletes the manager object.
- Thus the managed object stays around as long as there are shared_ptrs pointing to it, and the manager object stays around as long as there are either shared_ptrs or weak_ptrs referring to it.

No silver bullet

- You can only use these smart pointers to refer to objects allocated with new and that can be deleted with delete. No pointing to objects on the function call stack!
- If you want to get the full benefit of smart pointers, your code should avoid using raw pointers to refer to the same objects; otherwise it is too easy to have problems with dangling pointers or double deletions.
  - Leave the raw pointers inside the smart pointers and use only the smart pointers.
- You must ensure that there is only one manager object for each managed object. You do this by writing your code so that when an object is first created, it is immediately given to a shared_ptr to manage:

```
void main( ) {
 int* p = new int;
 shared_ptr<int> sptr1( p);
 shared_ptr<int> sptr2( p );
}
```

Crashes as soon as `sptr2` goes out of scope...
`p` was destroyed when `sptr1` went out of scope !

Which smart pointer should we use ?

Some guidelines

- It purely depends upon how you want to own a resource?
- If shared ownership is needed then go for `shared_ptr`, otherwise `unique_ptr` (i.e. `unique_ptr` is your default choice).
- If cyclic references are unavoidable in shared ownership, use `weak_ptr` to give one or more of the owners a weak reference to another `shared_ptr`.

Boost smart pointers

Boost smart pointers

- The Boost libraries provide a set of alternative smart pointers
- many have been selected for introduction in C++11... use the Boost library if your compiler still does not support C++11 smart pointers
- designed to complement auto_ptr

Boost smart pointers

- Four of the Boost smart pointers:
  - scoped_ptr defined in <boost/scoped_ptr.hpp>
    Simple sole ownership of single objects. Non-copyable.
  - scoped_array defined in <boost/scoped_array.hpp>
    Simple sole ownership of arrays. Non-copyable.
  - shared_ptr defined in <boost/shared_ptr.hpp>
    Object ownership shared among multiple pointers. std::shared_ptr represents reference counted ownership of a pointer. Each copy of the same shared_ptr owns the same pointer. That pointer will only be freed if all instances of the shared_ptr in the program are destroyed.
  - weak_ptr defined in <boost/weak_ptr.hpp>
    Non-owning observers of an object owned by shared_ptr. It is designed for use with shared_ptr.

Boost smart pointers

- Four of the Boost smart pointers:
  - scoped_ptr defined in <boost/scoped_ptr.hpp>
    Simple sole ownership of single objects. Non-copyable
  - scoped_array defined in <boost/scoped_array.hpp>
    Simple sole ownership of arrays. Non-copyable.
  - shared_ptr defined in <boost/shared_ptr.hpp>
    Object ownership shared among multiple pointers. std::shared_ptr represents reference counted ownership of a pointer. Each copy of the same shared_ptr owns the same pointer. That pointer will only be freed if all instances of the shared_ptr in the program are destroyed.
  - weak_ptr defined in <boost/weak_ptr.hpp>
    Non-owning observers of an object owned by shared_ptr. It is designed for use with shared_ptr.

Similar to unique_ptr (but no transfer of ownership)

Boost smart pointers

- Four of the Boost smart pointers:
  - scoped_ptr defined in <boost/scoped_ptr.hpp>
    Simple sole ownership of single objects. Non-copyable
  - scoped_array defined in <boost/scoped_array.hpp>
    Simple sole ownership of arrays. Non-copyable.
  - shared_ptr defined in <boost/shared_ptr.hpp>
    Object ownership shared among multiple pointers. std::shared_ptr represents reference counted ownership of a pointer. Each copy of the same shared_ptr owns the same pointer. That pointer will only be freed if all instances of the shared_ptr in the program are destroyed.
  - weak_ptr defined in <boost/weak_ptr.hpp>
    Non-owning observers of an object owned by shared_ptr. It is designed for use with shared_ptr.

Similar to unique_ptr (but no transfer of ownership)

Included in C++11

auto_ptr

deprecate |ˈdepriˌkāt|
verb [with obj.]
1 express disapproval of: (as adj. **deprecating**) *: he sniffed in a deprecating way.*

auto_ptr

- `auto_ptr` is a pointer-like object (a *smart pointer*) whose destructor automatically calls delete on what it points to
- it's in the C++ standard library:
  `#include <memory>`
- `auto_ptr` has been deprecated in C++11:
  use Boost or the new C++11 smart pointers

auto_ptr: an example

- Reconsider the f() function using auto_ptr:

```
void f() {
  std::auto_ptr<Vehicle> pV(createVehicle());
  // use pV as before...
} /* the magic happens here: automatically
deletes pV via the destructor of auto_ptr,
called because it's going out of scope */
```

auto_ptr: another example

- In general here's how to rewrite unsafe code in safe code:

```
// Original code
void f() {
  T* pt( new T );
  /*...more code...*/
  delete pt;
}
```

```
//Safe code, with auto_ptr
void f() {
  auto_ptr<T> pt( new T );
  /*...more code...*/
} /* pt's destructor is called
as it goes out of scope, and
the object is deleted
automatically */
```

auto_ptr characteristics

- Since auto_ptr automatically deletes what it points to when it is destroyed, there should not be two auto_ptr pointing to an object
  - or the object may be deleted twice: it's an undefined behaviour, if we are lucky the program just crashes
- To avoid this auto_ptr have a special feature: copying them (e.g. copy constructor or assignment operator) sets them to null and copying pointer assumes the ownership of the object

auto_ptr characteristics: example

```
// pV1 points to the created object
std::auto_ptr<Vehicle> pV1(createVehicle());

std::auto_ptr<Vehicle> pV2( pV1 );
/* now pV2 points to the object and pV1 is
null !  */

pV1 = pV2;
/* now pV1 points to the object and pV2 is
null !  */
```

auto_ptr characteristics - cont.

- If the target auto_ptr holds some object, it is freed
- This copy behaviour means that you can't create an STL container of auto_ptr !
  - Remind: STL containers want objects with normal copy behaviours
  - Modern compilers (with modern STL) issue compile errors

auto_ptr characteristics - cont.

- If you do not want to loose ownership use the const auto_ptr idiom:

```
const auto_ptr<T> pt1( new T );
        // making pt1 const guarantees that pt1 can
        // never be copied to another auto_ptr, and
        // so is guaranteed to never lose ownership

auto_ptr<T> pt2( pt1 ); // illegal
auto_ptr<T> pt3;
pt3 = pt1;              // illegal
pt1.release();          // illegal
pt1.reset( new T );     // illegal
```

- it just allows dereferencing

auto_ptr characteristics - cont.

- auto_ptr use delete in its destructor so do NOT use it with dynamically allocated arrays:

```
std::auto_ptr<std::string>
aPS(new std::string[10]);
```

- use a vector instead of an array

auto_ptr methods

- use get() to get a pointer to the object managed by auto_ptr, or get 0 if it's pointing to nothing
- use release() to set the auto_ptr internal pointer to null pointer (which indicates it points to no object) without destructing the object currently pointed by the auto_ptr.
- use reset() to deallocate the object pointed and set a new value (it's like creating a new auto_ptr)

auto_ptr methods

```
auto_ptr<int> p (new int);
*p.get() = 100;
cout << "p points to " << *p.get() << endl;
```

- use release() to set the auto_ptr internal pointer to null pointer (which indicates it points to no object) without destructing the object currently pointed by the auto_ptr.
- use reset() to deallocate the object pointed and set a new value (it's like creating a new auto_ptr)

auto_ptr methods

```
auto_ptr<int> auto_pointer (new int);
int * manual_pointer;
*auto_pointer=10;
manual_pointer = auto_pointer.release();
cout << "manual_pointer points to " <<
*manual_pointer << "\n";
// (auto_pointer is now null-pointer auto_ptr)
delete manual_pointer;
```

- use reset() to deallocate the object pointed and set a new value (it's like creating a new auto_ptr)

auto_ptr methods

```
auto_ptr<int> p;
p.reset (new int);
*p=5;
cout << *p << endl;


p.reset (new int);
*p=10;
cout << *p << endl;
```

- use reset() to deallocate the object pointed and set a new value (it's like creating a new auto_ptr)

auto_ptr methods - cont.

- operator*() and operator->() have been overloaded and return the element pointed by the auto_ptr object in order to access one of its members.

```
auto_ptr<Car> c(new Car);
c->startEngine();
(*c).getOwner();
```

Scope guard

- Sometime we want to release resources if an exception is thrown, but we do NOT want to release them if no exception is thrown. The "Scope guard" is a variation of RAII

- ```
  Foo* createAndInit() {
    Foo* f = new Foo;
    auto_ptr<Foo> p(f);
    init(f); // may throw
             // exception
    p.release();
    return f;
  }
  ```

- ```
  int run () {
    try {
      Foo *d = createAndInit();
      return 0;
    } catch (...) {
      return 1;
    }
  }
  ```

Scope guard

- Sometime we want to release resources if an exception is thrown, but we do NOT want to release them if no exception is thrown. The "Scope guard" is a

- ```
  Foo* createAndInit() {
    Foo* f = new Foo;
    auto_ptr<Foo> p(f);
    init(f); // may throw
             // exception
    p.release();
    return f;
  }
  ```

Use auto_ptr to guarantee that an exception does not leak the resource.

When we are safe, we release the auto_ptr and return the pointer.
  ```

Credits

- These slides are (heavily) based on the material of:
  - Scott Meyers, "Effective C++, 3rd ed."
  - Wikipedia
  - Herb Sutter, "Exceptional C++"
  - David Kieras, University of Michigan