



# Laboratorio di Tecnologie dell'Informazione

Ing. Marco Bertini  
[bertini@dsi.unifi.it](mailto:bertini@dsi.unifi.it)  
<http://www.dsi.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;  
}
```



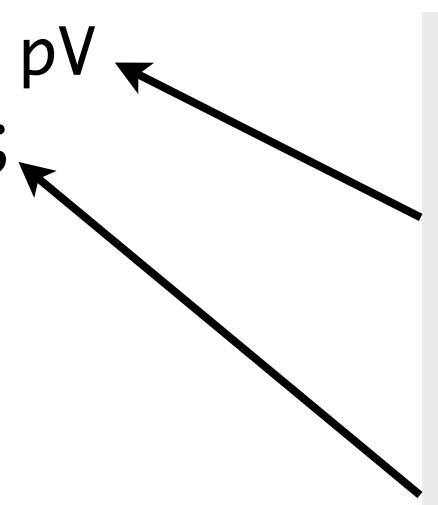
# An example

```
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 Acquisition Is Initialization



# 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
}
```



# auto\_ptr

deprecate |'depri,kāt|

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>`
- other smart pointers exist:  
e.g. Boost or the new C++11 smart pointers
- `auto_ptr` has been deprecated in C++11



# 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.



# unique\_ptr



# unique\_ptr

- Introduced in C++11
- 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 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`



# 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 those 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. Noncopyable.
  - `scoped_array` defined in `<boost/scoped_array.hpp>`  
Simple sole ownership of arrays. Noncopyable.
  - `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. Noncopyable.
  - `scoped_array` defined in `<boost/scoped_array.hpp>`  
Simple sole ownership of arrays. Noncopyable.
  - `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:

Similar to `unique_ptr`  
(but no transfer of  
ownership)

- `scoped_ptr` defined in `<boost/scoped_ptr.hpp>`  
Simple sole ownership of single objects. Noncopyable.

- `scoped_array` defined in `<boost/scoped_array.hpp>`  
Simple sole ownership of arrays. Noncopyable.

- `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.

Included in C++11

- `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`.



# Credits

- These slides are (heavily) based on the material of:
  - Scott Meyers, “Effective C++, 3rd ed.”
  - Wikipedia
  - Herb Sutter, “Exceptional C++”