EMC++ Chapter 4

Smart Pointers
C++11 / C++14 smart pointer types

auto_ptr

unique_ptr

shared_ptr

weak_ptr
C++11 / C++14 smart pointer types

auto_ptr

unique_ptr
C++11 replacement for auto_ptr. C++14 adds make_unique.

shared_ptr
C++11. Reference-counting.

weak_ptr
C++11. "Weak" references.
EMC++ Item 18

Use `std::unique_ptr` for exclusive-ownership resource management.
std::unique_ptr<T>

stack

ptr to T

heap

controlled object of type T
std::unique_ptr<T, Deleter>

- Stack
  - `ptr to T`
  - `deleter of type Deleter`

- Heap
  - `controlled object of type T`
EMC++ Item 19

Use `std::shared_ptr` for shared-ownership resource management.
std::shared_ptr<T>
Copying a `std::shared_ptr`
std::shared_ptr to base class

stack

- ptr to A
- ptr to control block

heap

- controlled object of type class D: public A, B
- reference count ++
- weak ref count
- default_delete<D>
- ptr to controlled object of type D

- ptr to B
- ptr to control block
“Shares ownership with”

#include <memory>
#include <vector>

using Vec = std::vector<int>;

std::shared_ptr<int> foo() {
  auto elts = { 0,1,2,3,4 };  
  std::shared_ptr<Vec> pvec = std::make_shared<Vec>(elts);  
  return std::shared_ptr<int>(pvec, &(*pvec)[2]);
}

int main() {
  std::shared_ptr<int> ptr = foo();
  for (auto i = -2; i < 3; ++i) {
    printf("%d\n", ptr.get()[i]);
  }
}
```
#include <memory>
#include <vector>

using Vec = std::vector<int>;

std::shared_ptr<int> foo() {
    auto elts = { 0,1,2,3,4 };  
    std::shared_ptr<Vec> pvec = std::make_shared<Vec>(elts);
    return std::shared_ptr<int>(
        pvec,
        &(*pvec)[2]
    );
}

int main() {
    std::shared_ptr<int> ptr = foo();
    for (auto i = -2; i < 3; ++i) {
        printf("%d\n", ptr.get()[i]);
    }
}
```
EMC++ Item 20

Use `std::weak_ptr` for `shared_ptr`-like pointers that can dangle.
**std::weak_ptr**

- **Heap**:
  - Reference count = 1
  - Weak ref count = 1
  - Custom deleter?
  - Pointer to controlled object

- **Stack**:
  - `shared_ptr`
  - Pointer to `T`
  - Pointer to control block

- **Weak_ptr**:
  - Pointer to `T`
  - Pointer to control block

The diagram illustrates the relationship between stack and heap, showing how `shared_ptr` and `weak_ptr` are used to manage controlled objects.
Destroy the `shared_ptr`, which also destroys the controlled object...
std::weak_ptr

- Stack:
  - weak_ptr
  - ptr to T
  - ptr to control block

- Heap:
  - reference count = 0
  - weak ref count = 1
  - custom deleter?
  - ptr to controlled object = NULL

- Weak reference count = 1

You can’t dereference a weak_ptr

You can only convert it to a shared_ptr.

```cpp
void recommended(std::weak_ptr<T> wptr) {
    std::shared_ptr<T> sptr = wptr.lock();
    if (sptr) {
        use(sptr);
    }
}

void not_recommended(std::weak_ptr<T> wptr) {
    try {
        std::shared_ptr<T> sptr { wptr };  // call the explicit constructor
        use(sptr);
    } catch (std::bad_weak_ptr) {}  
}
```
Prefer `std::make_unique` and `std::make_shared` to direct use of `new`.
EMC++ Item 21

std::make_shared is an optimization
std::make_unique is not

But! Both are useful for exception-safety

if (func(unique_ptr<T1>(new T1), unique_ptr<T2>(new T2))) {
  // “new T2” is unsequenced with respect to both
  // “new T1” and “unique_ptr<T1>(...)”
}

http://channel9.msdn.com/Events/GoingNative/2013/Don-t-Help-the-Compiler
EMC++ Item 22

When using the Pimpl idiom, define special member functions in the implementation file.
EMC++ Item 22: The Pimpl idiom

```cpp
<<<Widget.h>>>

class Widget {
public:
    Widget();
    ~Widget();

private:
    struct Impl;
    Impl *pImpl;
};

<<<Widget.cpp>>>

struct Widget::Impl {
    ...
};

Widget::Widget()
    : pImpl(new Impl) {}

Widget::~Widget() {
    delete pImpl;
}
```
EMC++ Item 22: The Pimpl idiom

```cpp
<<<Widget.h>>>
#include <memory>

class Widget {
 public:
   Widget();
   ~Widget();

 private:
   struct Impl;
   unique_ptr<Impl> pImpl;
};

<<<Widget.cpp>>>

struct Widget::Impl {
   ...
};

Widget::Widget()
   : pImpl(make_unique<Impl>())
{}

Widget::~Widget() {}
```
# EMC++ Item 22: Why not...?

```cpp
#include <memory>

class Widget {
public:
    Widget();
    ~Widget() = default;
private:
    struct Impl;
    unique_ptr<Impl> pImpl;
};

struct Widget::Impl {
    ...
};

Widget::Widget() :
    pImpl(make_unique<Impl>())
{};
```
EMC++ Item 22: Why not...? (gcc)

In file included from /usr/include/c++/4.7/memory:86:0,
    from Widget.h:1,
    from test.cc:1:
/usr/include/c++/4.7/bits/unique_ptr.h: In instantiation of ‘void std::default_delete<_Tp>::operator()(_Tp*) const [with _Tp = Widget::Impl]’:
/usr/include/c++/4.7/bits/unique_ptr.h:173:4:   required from ‘std::unique_ptr<_Tp, _Dp>::~unique_ptr() [with _Tp = Widget::Impl; _Dp = std::default_delete<Widget::Impl>]’
Widget.h:6:3:   required from here
/usr/include/c++/4.7/bits/unique_ptr.h:63:14: error: invalid application of ‘sizeof’ to incomplete type ‘Widget::Impl’
EMC++ Item 22: Why not...? (clang)

/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/memory:2424:27: error: invalid application of 'sizeof' to an incomplete type 'Widget::Impl'
    static_assert(sizeof(_Tp) > 0, "default_delete can not delete incomplete type");
    ^~~~~~~~~~~
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/memory:2625:13: note: in instantiation of member function 'std::__1::default_delete<Widget::Impl>::operator()' requested here
    __ptr_.second(__tmp);
    ^
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/memory:2593:46: note: in instantiation of member function 'std::__1::unique_ptr<Widget::Impl, std::__1::default_delete<Widget::Impl> >::reset' requested here
    _LIBCPP_INLINE_VISIBILITY ~unique_ptr() {reset();}
    ^
./Widget.h:6:3: note: in instantiation of member function 'std::__1::unique_ptr<Widget::Impl, std::__1::default_delete<Widget::Impl> >::~unique_ptr' requested here
    ~Widget() = default;
    ^
./Widget.h:8:10: note: forward declaration of 'Widget::Impl'
    struct Impl;
    ^
1 error generated.
~Widget doesn’t know how to delete *pImpl

```cpp
#include <memory>
class Widget {
    public:
        Widget();
        ~Widget() = default;
    private:
        struct Impl;
        unique_ptr<Impl> pImpl;
};

struct Widget::Impl {
    ...
};

Widget::Widget()
    : pImpl(make_unique<Impl>())
{};
```
#include <memory>

struct Puzzle {
    struct Impl;
    std::unique_ptr<Impl> pImpl;
    Puzzle();
};

Puzzle foo() {
    return Puzzle();
}

1 error generated.
Sidebar: The Rule of Five

```cpp
#include <memory>

struct Puzzle {
    struct Impl;
    std::unique_ptr<Impl> pImpl;
    Puzzle();
    ~Puzzle(); // right?
};

Puzzle foo() {
    return Puzzle();
}
```

test.cc:11:10: error: call to implicitly-deleted copy constructor of 'Puzzle'
    return Puzzle();
         ^~~~~~~~~~

test.cc:5:25: note: copy constructor of 'Puzzle' is implicitly deleted because field 'pImpl' has a deleted copy constructor
    std::unique_ptr<Impl> pImpl;
          ^

/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/memory:2510:31: note: copy constructor is implicitly deleted because 'unique_ptr<Puzzle::Impl, std::__1::default_delete<Puzzle::Impl> >' has a user-declared move constructor
_LIBCPP_INLINE_VISIBILITY unique_ptr(unique_ptr&& __u) _NOEXCEPT
```

1 error generated.
Sidebar: The Rule of Five

12.8 [class.copy] 7

If the class definition does not explicitly declare a copy constructor, one is declared *implicitly*. If the class definition declares a move constructor or move assignment operator, the implicitly declared copy constructor is defined as deleted; otherwise, it is defined as defaulted. The latter case is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor.

Thus class *Puzzle*’s copy constructor is implicitly declared as defaulted; but since member *m* is uncopiable, the defaulted copy constructor is defined as deleted. (12.8 [class.copy] 11)
If the definition of a class $X$ does not explicitly declare a move constructor, one will be implicitly declared as defaulted if and only if

— $X$ does not have a user-declared copy constructor,
— $X$ does not have a user-declared copy assignment operator,
— $X$ does not have a user-declared move assignment operator, and
— $X$ does not have a user-declared destructor.

\begin{quote}
\textbf{class Puzzle has a user-declared destructor, so no move constructor is implicitly declared.}
\end{quote}
Note: When the move constructor is not implicitly declared or explicitly supplied, expressions that otherwise would have invoked the move constructor may instead invoke a copy constructor.
— end note]
Sidebar: The Rule of Five

```cpp
struct Widget {
    Widget(Widget&&);                  // move construction
    Widget(const Widget&);             // copy construction
    Widget& operator=(Widget&&);       // move assignment
    Widget& operator=(const Widget&);  // copy assignment
    ~Widget();                         // destructor
};
```

– If you declare any one of these, you **should** declare them all.
– Any of these may be declared =default or =delete
Sidebar: The Rule of Five

struct Widget {
    Widget(Widget&&);                  // move construction
    Widget(const Widget&);             // copy construction
    Widget& operator=(Widget&&);       // move assignment
    Widget& operator=(const Widget&);  // copy assignment
    ~Widget();                         // destructor
};

– If you declare any one of these, you should declare them all.
– Any of these may be declared =default or =delete, but...
– watch out for cases in which =default is not equivalent to {}