CS 246 - Object Oriented Programming **Fall 2016**

Lecture 11: November 24, 2016

Lecturer: Brad Lushman *Notes By: Harsh Mistry*

Recall :

```
void f() {
  auto p = std::make_unique<MyClass> ();
  MyClass mc;
  g();
}
```

Now Consider :

```
class c { ... };

unique_ptr<c> p { new C {...}};
unique_ptr<c> q = p
```

When a unique pointer is copied and then subsequently deleted, you will have a double free. To prevent such a incident, copying of pointers is disabled and prohibited by the compiler. But moving is allowed.

Sample Implementation of unique pointer :

```
template <typename T> class unique_ptr {
    T *ptr;
  public :
    unique_ptr( T *p) : ptr {p} {}

    ~unique_ptr() {delete ptr;}

    unique_ptr(const unique_ptr <T> &other ) = delete; //disables copying

    unique_ptr<T> &operator=(const unique_ptr<T> &other) = delete; //disables copying

    unique_ptr(const uniquie_ptr <t> &&other) : ptr{other.ptr} {
      //Set other to null to prevent loss of data
      other.ptr = nullptr;
    }

    unique_ptr<T> &operator=(const unique_ptr<T> &&other){
      std::swap(ptr, other.ptr);
      return *this;
    }

    //
    T &operator*() {return *ptr;}
}
```

If you need to able to copy smart pointers use `std::shared_ptr`. Shared pointer will ensure when both smart pointers go out of scope, only the last one will be deleted. This happens because shared pointers maintain a **reference count**. In essence, the pointer keeps track of how many pointers point the same object. Then memory is only freed when the internal count reaches 0.

With all this in mind, it is recommended that you use smart pointers, as it dramatically decreases chances of memory leaks.

11.1 3 levels of Exception Safety for a function f

1. Basic Guarantee - If an exception occurs, the program will still remain in valid state
2. Strong Guarantee - If an exception is raised while executing f, the state of the program will be as it was before f was called.
3. No throw guarantee - f will never throw an exception, and always accomplishes its task.

Now Consider :

```
Class A {...}
Class B {...}
Class C {...}

A a;
B b;

void f() {
  a.method1();  // Might Throw (Strong Guarantee)
  b.method2();  // Might Throw
}
```

If method one throws nothing happens, but if method 2 throws, the effects of method 1 must be undone to offer the strong guarantee. This can be very difficult if method 1 has non-local side effects. Therefore, the function is not exception safe.

If method1 and method2 do not have non-local side-effects, we can use copy/swap idiom to make f() exception safe.

```
Class A {...}
Class B {...}
Class C {...}

A a;
B b;

void f() {
  A aTemp = a; //If this throw, its ok, as nothing has happened yet.
  B bTemp = b;

  //If these throw, original version of A and B are still intact.
  aTemp.method1();
  bTemp.method2();

  //Copy Temp back to orginal
  a = aTemp;
  b = btemp;
}
```

This works to a certain degree, but if copy throws during the last step, then the function would not be considered exception safe. To fix this we can utilize the pImpl idiom to ensure the sawp can not throw errors by using pointers.

```
struct CImpl {
  A a;
  B b;
};

Class C {
    ...
    unique_ptr<CImp>  (* pImpl);
    ...
    void f() {
      auto temp = make_unique<Cimpl> (*pImpl);
      temp ->a.method1();
      temp->b.method2();
      std::swap(temp, pImpl); //Does not throw, thus strong guarantee.
    }
};
```

Note : If method1 or method2 offer no exception safety guarantee, then neither can f.

Warning

Never let a destructor emit an exception.
If the destructor was executed during stack unwinding while dealing with another exception, you now have two active, unhandled exceptions, and the program will abort immediately.

11.2 Exception safety and the STL

Vector encapsulates a heap array and it follows RAII. Thus, the internal heap array is freed when the vector goes out of scope

Example 11.1 *Example of straight forward memory management*

```
void f() {
  vector <MyClass>  v; //All Classes are deleted when the vector goes out of scope and no
      exceptions are raised
}
```

Example 11.2 *Vectors with regular pointers.*

```
void g() {
  vector <MyClass *> v; //Pointers are not freed when the vector goes out of scope, as its
      not sure if the pointers own the data or if they point to the heap.

  //So, You must delete the pointers yourself
  for (auto x : v) delete x;
}
```

Example 11.3 *Vectors with shared pointers*

```
void h() {
  vector<shared_ptr<MyClass>> v; //Objects in vector are freed if no other shared pointer
      points at them, so there is no explicit deallocation.
}
```

Example 11.4 *Vectors with emplace back*

```
void h() {
  v.emplace_back () ; //Offers the strong guarantee.
}
```

Emplace is a strong guarantee because if the copy assignment fails when vector makes a new array, it is simply deleted and the previous state is retained.

Example 11.4 is great, but copying is expensive and old data will be thrown away. Moving objects would be more efficient. Which is why if the move constructor offers no-throw guarantee, `emplace_back` will use the move constructor, otherwise it will use the copy constructor, which is slower.

So. your mover operations should provide a no throw guarantee. To do this, you need indicate they provide such a guarantee by using the keywords `no except`

```
class MyClass {
  public :
    MyClass (MyClass &&other) no except {}
    MyClass &operator=(Myclass &&other) no except;
};
```

If you know that a function will never throw or propagate an exception, declare it no except as it facilitates optimization. At minimum, you should declare swaps and moves should be no except.

11.3 Casting

In C :

```
Node n;
int *x = (int *)(&n) //Forces compiler to treat node* as int
```

The behaviour is undefined. If you must cast, you should use C++ style casts, of which there are 4.

1. `static_cast` - Used for what can be considered a sensible cast. So the casts have well defined meaning
   - Casting for overloaded function.

   ```
   double d;

   void f(int i);
   void f(double d);

   f(static_cast<int>(d));
   ```

   - Converting superclass pointer to subclass pointer

   ```
   Book *b = new Text{ ... };
   Text *t = static_cast<Text *> (b);
   ```

2. `reinterpret_cast` - Used for unsafe, implementation specific, "weird" casts
   - Treat a student as a turtle

   ```
   Student S;
   Turtle *t = reinterpret_cast<Turtle *> (&s);
   ```

3. `const_cast` - Used for converting const/non-const and is the only C++ cast that can "cast away const"
   - Passing constant pointer to function without a const in its signature

```
void g(int *p); // Given and supposed g doesn't change *p, but was left out of
    signature

void f(const int *p) {
  ...
  g(const.cast<int *>(p));
  ...
}
```

4. `dynamic_cast` - Used to cast in cases where you want to proceed only if the cast succeeds

   - Casting Text to Book in cases where P is really a text or subclass of Text. If the cast fails, the pointer will be set to null.

```
Book *p;
Text *pt = dynamic_cast<Text *>(pb);
```

If you wish to cast upon smart pointers you must use the following which cast shared pointers to shared pointers :

1. `static_pointer_cast`
2. `const_pointer_cast`
3. `dynamic_pointer_cast`

Given dynamic casting, we can make decisions based on an object's **RTTI (run-time type information)**.

```
void whatType (shared_ptr<Book> b){
  if (dynamic_pointer_cast<comic>(b)) cout << "Comic";
  else if (dynamic_pointer_cast<Text>(b) cout << "Text";
  else cout << "A regular book"
}
```

Code like the example above is tightly coupled to the hierarchy of the classes and may indicate bad design, so before using dynamic cast consider using Virtual Methods or Visitors.

Additionally, dynamic casting also works with references :

```
Text t {...}'
Book &b = t;

Text &t2 = dynamic_cast<Text &> (b);
//If B "points to " a Text, then t2 is a reference to the same text.
//If not, then a exception bad_cast is raised
```

In general: Dynamic casting only works on classes with at least one virtual method.