CS32 Summer 2013

Object-Oriented Programming in C++

Advanced Topics

Victor Amelkin

August 21, 2013

Plan for Today

- Namespaces
- More on Constructors
- Type Conversion (from)
- Automatic Destruction and RAII
- Const, Static
- Operators
- Type Conversion (to)

Namespaces

- Global names may collide

```
// window.cpp
class Menu { /* GUI Window's Menu */ };

// restaurant.cpp
class Menu { /* Restaurant Menu */ };
```

- Separate code into namespaces

```
namespace GUI {
   class Window { … };
   class Menu { … };
}

namespace Food {
   class ClamChowder { … };
   namespace Organisation {
        class Menu { … };
   }
}

// Using namespaces ( :: - scope resolution operator )
GUI::Menu m1;
Food::Organisation::Menu m2;
```

Namespaces

- Extend any namespace at any time

```
// prefer not to extend standard namespaces
namespace std { class super_string { … }; }
std::string str1;
std::super_string str2;
```

- Save coding effort – import name(s) from a namespace

```
void generous() {
  using namespace std; // imports everything from std
  string str;
}
void conservative() {
  using std::cout; // imports only cout
  cout << "hello";
}
```

- Do not import the *entire* namespace to *global* scope
- Alias long composite namespaces

```
namespace org = Food::Organisation;
org::Menu m;
```

More on Constructors

- Calling default constructor

```
xstring str1;
xstring *pstr2 = new xstring;

xstring nstr[10];
xstring *pnstr = new xstring[10];
// Cleanup:
// delete [] pnstr;
```

- Calling non-default constructor

```
xstring str1("hello");
xstring *pstr2 = new xstring("hello");

xstring **ppstr = new xstring*[10];
for(int i = 0; i < 10; i++)
  ppstr[i] = new xstring("hello");

// Cleanup:
// for(int i = 0; i < 10; i++) { delete ppstr[i]; }
// delete [] ppstr;
```

More on Constructors

- Calling copy constructor

```
// copy on the stack
xstring str;
xstring str2(str);


// copy on the heap
xstring str3;
xstring *pstr4 = new xstring(str3);

// passing object by value
void byval(const xstring str) { … }
void byref{const xstring &str) { … }

byval(str); // copy ctor is called
byref(str); // copy ctor is not called
```

Type Conversion

- Type conversion for plain types

```
double pi = 3.14159265;
int n = (int)pi; // n = 3
```

- C++ type conversion

```
class xstring {
    xstring(const char *p) { … }
};
xstring str = (xstring)”hello”; // explicit
xstring str2 = “world”; // implicit (evil?)
```

- One-argument constructors – type converters

- To prevent implicit conversion (by mistake)

```
class xstring {
  explicit xstring(const char *p) { … }
}
```

RAII

- Background: automatic destruction on the stack

```
int func() {
   MyClass obj;
} // ~MyClass() is automatically called here
```

- Problem: programmers forget to manually release acquired resources (memory, files, network or database connections, …)
- Solution: use automatic destruction on the stack to *automatically* and *reliably* release resources

```
class resource_container {
   resource_container(...args...) {
       // allocate memory on the heap
       // acquire file handles
       // create network connections
   }
   ~resource_container() {
       // release allocated memory on the heap
       // release file handles
       // close network connections
   }
};
```

RAII Use Case

- Smart Pointer – resource container for one object allocated on the heap

```
#include <memory>

using std::auto_ptr;

auto_ptr<MyClass> pobj(new MyClass(...));
// no need to manually delete pobj; it will
// be done automatically by enclosing auto_ptr
```

- In C++TR1 and C++11:

```
auto_ptr<T> // obsolete
unique_ptr<T> // owns object
shared_ptr<T> // shares with others
weak_ptr<T> // sees but does not own
```

Const

- const – “cannot be changed”

```
const int n;
n = 1; // error
```

- Accessing *fields* of a constant object

```
void func(const MyClass &obj) {
  int x = obj.field; // read - ok
  obj.field = 1; // write - error
}
```

- Accessing *method*

```
void func(const MyClass &obj) {
  // what is method() changes obj's state?
  obj.method();
}
```

Const and Methods

- Can call only const methods on const objects

```
class MyClass {
   void method1(); // potential mutator
   void method2() const; // const-method
   void method3() const { field = 1; } // error
};

const MyClass obj;
obj.method1(); // error; method1 can change obj
obj.method2(); // ok
               // method3 cannot exist
```

Const and Copy Ctors

- Why do we have *two* copy constructors?

```
MyClass(MyClass &other);
MyClass(const MyClass &other);
```

- Non-const copy ctor cannot accept constants

```
const MyClass obj;
MyClass obj2(obj); // error (if only non-const copy ctor exists)
```

- Non-const copy ctor can implement *move semantics* (auto_ptr)

```
MyClass(MyClass &other) {
    // steal contents from other and save it
    // in the current object's state;
     // move semantics is used in basic smart pointer
}
```

- Const copy ctor can accept both const and non-const objects
  - in 99 cases out of 100 you want *const copy constructor* to “copy”

- Everything said also applies to (two) assignment operator(s)

Static at C Level

- Static globals in C-files – hidden inside their translation units (“internal linkage”)
  - Translation unit – .c-file and everything `#include`'d

```
// file.c
static int global_var;
static int global_func() { … }
extern int global_var2; // visible to others (“external linkage”)
```

- Static globals in C-headers – each `#includ`'ing file gets its own copy

```
// header.h
static int another_var; // individual copy per inclusion (despite #include guards)
```

- Static local vars (“local global vars”) – retain value between calls

```
int func() {
  static int n = 0; // not the same as static int n; n = 0;
  return ++n;
}
func(); // returns 1
func(); // returns 2
func(); // returns 3
```

- In C, global vars are extern by default; in C++ – static, but const are extern

Static Class Fields

- Static field – belongs to class, not to object

```
// myclass.h
class MyClass {
private:
   char byte1;
   char byte2;
   static char my_static_field;

public:
   // instance members see static fields
   void print() { cout << my_static_field; }
};

// myclass.cpp
// must define class' static field:
char MyClass::my_static_field = 'x';

// main.cpp
MyClass obj;
cout << sizeof(obj); // prints 2
MyClass::my_static_field = 'y'; // error; var is private
```

Static Class Methods

- Static method – sees only static fields and other static methods

```
// myclass.h
class MyClass {
private:
   char nonstatic_field;
   static char static_field;
public:
   void method1() { cout << static_field; } // ok
   static void method2() { cout << static_field; } // ok
   static void method3() { method2(); } // ok
   static void method4() { method1(); } // error
   static void method5() { nonstatic_field = 'a'; } // error
};

// myclass.cpp
char MyClass::static_field = 'x';

// main.cpp
MyClass::method3();
```

Static Methods and (*this)

- Instance methods receive a pointer to the object

```
class MyClass {
   void instance_method(...args...);
};
obj.instance_method('x', 3);
```

translates into

```
class MyClass {
   void instance_method(MyClass *this, ...args...);
};
obj.instance_method(&obj, 'x', 3);
```

- Static methods do not

Operators in C++

- Most operators in C++ are just regular methods

```
class supernum {
private:
   int _val;
public:
   supernum(int val) { _val = val; }
   supernum operator+(const supernum &right) const {
      return supernum(this->_val + right._val);
   }
   void print() { cout << _val; }
};

supernum x(99);
supernum y(1);
(x + y).print(); // prints 100 (operator-style call)
(x.operator+(y)).print(); // (function-style call)
```

Operators in C++

- Another example – assignment operator

```
class supernum {
private:
   int _val;
public:
   supernum(int val) { _val = val; }
   supernum& operator=(const supernum &right) {
      _val = right._val;
      return *this;
   }
   void print() { cout << _val; }
};

supernum x(1), y(2), z(3); // x holds 1, y holds 2, z holds 3
x = y = z; // evaluation from right to left (x = (y = z))
x.print(); // prints 3
y.print(); // prints 3

// but of course operators are just methods:
x.operator=(y.operator=(z)); // same as x = y = z
```

Member Operators vs. Commutativity

- Member operators are always called for the **left** operand

```
class supernum {
    ...
    supernum operator+(const int x) {
        return supernum(_val + x);
    }
};

supernum num(7);
(num + 3).print(); // prints 10
(num.operator+(3)).print(); // interpretation

(3 + num).print(); // compilation error
(3.operator+(num)).print(); // interpretation

// int has no operator+(const supernum &num)
```

Member Operators vs. Commutativity

- For commutativity, use external operators

```
class supernum {
   private:
        int _val;
   public:
        ...
        int get_val() const { return _val; } // “getter”
};

supernum operator+(const supernum &left, const int right) {
   return supernum(left.get_val() + right);
}

supernum operator+(const int left, const supernum &right) {
   return right + left; // just calling another +
}

// usage
supernum num(7);
(num + 3).print(); // prints 10
(3 + num).print(); // prints 10
```

Member Operators vs. Commutativity

- Operator can be declared a **friend** of a class, which grants it access to this class' **private** members:

```
class supernum {
   private:
      int _val;
   public:
      friend supernum operator+(...args as below...);
};

supernum operator+(const supernum &left, const int right) {
   return supernum(left->_val + right);
}
```

- Introduce friends *only if necessary*
  - The more friends a class has, the greater is the chance some of them will mess up the state

Operators Overview

- In C++, there is a customizable operator for almost everything

```
T& operator=(...)          // x = y
T  operator+(...)          // x + y
T  operator-(...)          // x - y
T  operator+()             // +x (unary plus)
T  operator-()             // -x (unary minus)
T  operator*(...)          // x * y
T  operator/(...)          // x / y
T  operator%(...)          // x % y (modulo)
T& operator++()            // ++x (prefix increment)
T  operator++(int dummy)   // x++ (suffix increment)
T& operator--()            // --x (prefix decrement)
T  operator--(int dummy)   // x-- (suffix decrement)
T  operator==(...)         // x == y
T  operator>(...)          // x > y
…
logical operators (!, &&, ||)
bitwise operators (~, &, |, ^, <<, >>)
member and pointer operators ([], *, ->, ...)
type conversion operators
```

Type Conversion Operators

- One-argument ctors act like type converters from foreign types

```
class supernum {
   supernum(const int x) { … } // converter from int
   supernum operator+(const supernum &right) { … }
}

supernum num(3);
(7 + num).print(); // implicit conversion from int
(((supernum)7).operator+(num)).print(); // interpretation
```

- Type-conversion operators convert to foreign types

```
class supernum {
   operator int() { return _val; } // converter to int
   ...
}

supernum num(3);
cout << 1 + 2 + num + 4 + 5; // implicit conversion to int
```