C++ for C programmers

CMSC 435/634

Incremental improvements to C

Well, really many significant changes

- But most developers choose a useful subset
- Start by writing C

File naming conventions:

- Source files: .C, .c++, .cc, .cpp, .cxx
- Header files: .H, .h++, .hh, .hpp, .hxx, .h, nothing
- I usually use .hpp and .cpp

I'll just highlight a few C++ features I find useful

Comments

C comments still work

```
not comment /* comment */ not comment
```

C++ comments from // to end of line

```
not comment // comment
```

Pointers and References

Pointer (like C)

```
StructType x;
StructType *y = &x; // pointer to a StructType
// use x.a or y->a
```

Reference

- Acts like a regular variable
- But refers to another location like a pointer
- Can *never* be NULL

```
StructType &z = x;
// use x.a or z.a
```

Function overloading

Can *overload* functions based on parameter types

```
// different functions, based on type of a
int f(int a, int b);
int f(float a, int b);
```

Can't overload on return type

```
// can't do this
int f(int a, int b);
float f(int a, int b);
```

Functions and C

Overloading messes with ability to call/be called by C code

- Use *extern "C"*

```
// C function to call in C++
extern "C" int fn();

// C++ function to call from C
extern "C" int fn() { ... }
```

Can also do blocks of code (standard includes already have this)

```
extern "C" {
   ... bunch of definitions ...
}
```

Operator Functions

Lets you declare new operators +,-,*,/,[],...

```
// declare
Type operator*(Type &a, Type &b);

// use
c = a*b;
```

Best to avoid unexpected operator behavior

- a*b = vector multiply is obvious
- a*b = max would confuse anyone reading the code

Namespaces

"namespace" manages functions with the same name

```
// all symbols declared in namespace myPackage
namespace myPackage {
  double sqrt(double a);
  // sqrt() means this one, not the global one
};
```

Using functions declared in a namespace

```
sqrt(a);                  // global one
myPackage::sqrt(a); // mine
```

Namespaces can nest

```
package::subpackage::sqrt();
```

Using

Can default to use a specific namespace

```
sqrt(a);      // global sqrt()

using namespace myPackage;
sqrt(a);      // myPackage::sqrt()
::sqrt(a);    // global sqrt()
```

Can also be more controlled

```
// just default to myPackage::sqrt
// not the rest of myPackage
using myPackage::sqrt;
```

Best to only use **using** in source, not headers!

- Otherwise could change which unrelated functions are used
- There is no unusing!

Implicit typedef

Built-in typedef by putting name after enum or struct

```
enum ColorChannel {RED, GREEN, BLUE};
struct Color {
  float red, green, blue;
};
```

C equivalents

```
typedef enum {RED, GREEN, BLUE} ColorChannel;
typedef struct {
  float red, green, blue;
} Color;
```

Struct as Class

Structs can contain functions

```
struct Color {
  float red, green, blue;
  float luminance() {
    return 0.2126*red+0.7152*green+0.0722*blue;
  };
};
...
Color col;
...
float lum = col.luminance();
```

Inside function, **this** is a pointer to the current struct

Public, Private and Class

Limit who can use data and functions

```
struct Color {
  private: // only usable by functions inside Color
    float red, green, blue;

  public:  // usable by anyone
    float luminance() {
      return 0.2126*red+0.7152*green+0.0722*blue;
    };
};
```

A **class** is just a **struct** that starts in private mode instead of public

Inheritance

Can extend any class (or struct) with inheritance

```
class Sphere : public Object {
  // Sphere has everything in Object, plus..
};
```

Add **protected:**

- Like **public**/**private**, but only accessible inside child classes.
- Declare protected in parent/base class

Class namespace

All classes act like a namespace for anything inside

- Member data, functions, enums, typedefs

Use to separate declaration from code

```
class Color {
  float red, green, blue;  // note: this is private
public:
  float luminance();       // just declared
};

float Color::luminance() { // need Color::
  // now inside class, don't need Color::
  // also can access private data
  return 0.2126*red+0.7152*green+0.0722*blue;
};
```

Constructor and destructor

Special function with same name of class is *constructor*

- Used to initialize class data
- No return type
- Can have multiple constructors with different arguments

Special function with name ~ClassName is *destructor*

- Used to clean up (especially allocated memory)
- No return type, no arguments

Constructor and destructor

```
class Color {
  float red, green, blue;
public:
  Color();                                  // constructor 1
  Color(float r, float g, float b); // constructor 2
  ~Color();                                 // destructor
};

void someFunction() {
  Color black;                          // uses constructor1
  Color skyblue(0.5, 0.7, 0.9); // uses constructor 2
  ...
} // destructor called for black and skyblue
```

Constructor initialization list

Special constructor syntax can give a list of initial values

- *Watch out!* Called in class order, **not** list order
- Only way to specify constructor for parent class

```
class Sphere : public Object {
  float radius;
public:
  Sphere(float x, float y, float z, float r);
};

Sphere::Sphere(float x, float y, float z, float r)
  : Object(x, y, z), // which Object constructor
    radius(r)        // also member data
{ // constructor code could do member data too
}
```

Class pointers

Allocate/free classes with **new** and **delete**

```
MyClass *c = new MyClass(constructorArgs);
delete c;
```

Free arrays of class data with **delete[]**

```
MyClass *array = new MyClass[size]; // no args
delete[] array;
```

Can always use 0 instead of NULL for any pointer type

delete and **delete[]** can take NULL

- Good practice to initialize unused pointers to NULL or 0

Memory management

Simple:

- Every pointer is “owned” by one class
- That class should outlive other uses of the pointer
- That class should delete the data in its destructor

Memory management

Simple:

- Every pointer is “owned” by one class
- That class should outlive other uses of the pointer
- That class should delete the data in its destructor

More complex:

- Track and transfer pointer ownership (see autoptr)

Memory management

Simple:

- Every pointer is “owned” by one class
- That class should outlive other uses of the pointer
- That class should delete the data in its destructor

More complex:

- Track and transfer pointer ownership (see autoptr)

More complex still:

- Add reference counting to classes
- Destructor changes count, only delete if count is 0

Memory management

Simple:

- Every pointer is "owned" by one class
- That class should outlive other uses of the pointer
- That class should delete the data in its destructor

More complex:

- Track and transfer pointer ownership (see autoptr)

More complex still:

- Add reference counting to classes
- Destructor changes count, only delete if count is 0

Heavy-weight:

- Overload new / use *placement new*
- Allows one or more custom memory allocators in a single application

Virtual methods

virtual calls based on the type the class *is*, not just the type the pointer you *have*

```
struct Object {
  void reset();
  virtual void draw();
};
struct Sphere : public Object {
  void reset();
  virtual void draw(); // MUST match
};

Object *obj = new Sphere;
obj->reset();      // uses Object::reset()
obj->draw();       // uses Sphere::draw()
```

More virtual

If you use class overloading, make the destructor virtual too!

- Makes sure the destructor of the *real* class is used

Pure virtual base class

- Only sub-classes can actually exist, but base class defines common interface

```
struct Object {
  virtual void draw() = 0;
};
```

Plain Old Data (POD)

Opposite end of the spectrum

- Simple class or struct
- Only POD member data
- No or simple constructor
- No destructor
- No virtual functions

Why?

- Only the additions that work with C classes
- *Guarantees on how it'll map to memory*

Casting

C-style casting still works

```
Sphere *s = (Sphere*)object;
```

New functional form

```
int x = int(f); // same as x = (int)f
```

New forms that limit kinds of cast changes

```
// ONLY add or remove const
const_cast<Sphere*>(var);
// if class of var is a parent or child of Sphere
static_cast<Sphere*>(var);
// if run-time var object is really a Sphere
dynamic_cast<Sphere*>(var);
// just do it, equivalent to (Sphere*)var
reinterpret_cast<Sphere*>(var);
```

Templates

Class or function that can work with multiple types
Can template over types or numbers
Functions should appear in a header

- But be declared **inline**
- Often use a separate header (.inl, .tpp, .txx)

Template declarations

Template class

```
template <typename Type, int Size>
struct Vector {
  Type data[Size];
};

Vector<float, 3> vec3;
```

Template function

```
template <typename Type, int Size>
Vector<Type,Size> length(Vector<Type,Size> &v);
```

Template specialization

Can declare special versions for certain parameter choices

```
template <typename Type>
struct Vector<Type, 2> {
  union {
    Type data[Size];
    struct { Type x, y; };
  };
};

// special version with vec2.x and vec2.y
Vector<float, 2> vec2;
```

Full specialization

```
template <>
struct Vector<float, 3> {...};
```

Standard Template Library

Template classes for many standard data structures

- string (dynamically resizes to data)
- vector<type>(dynamically resizing array)
- list<type>(doubly linked list)
- map<key, value>(associative array)
- ...

Common features

- in "std" namespace, use std::class or "using namespace std'
- include file is name of class (#include <string>)
- standard functions (begin, end, size, find, ...)
- *iterators* to loop over elements