CS11 Advanced C++

Lecture 3

Spring 2012-2013

Complex Numbers

- Write a complex number class:

```
class Complex {
    float real, imag;

public:
    Complex(float re, float im) :
        real(re), imag(im) { }
    ...
};
```

- Construct complex numbers like this:

```
Complex a(2, 3);  // 2 + 3i
```

- What other ways to initialize complex numbers?

```
Complex b;        // 0 + 0i
Complex c(5);     // 5 + 0i
```

Default Values

- Instead of creating multiple constructors, can specify default values for arguments

```
class Complex {
    ...
public:
    Complex(float re = 0, float im = 0) :
        real(re), imag(im) { }
    ...
};
```

- Supports all desired call patterns:

```
Complex a(2, 3);   // Both re and im specified
Complex b;         // Both re and im default to 0
Complex c(5);      // re = 5; im defaults to 0
```

Rules for Default Values

- Default values can only be specified once
  - Typically specified in declaration, not in definition
- Arguments with default values must be at <u>end</u> of argument-list
  - This is invalid:

```
void foo(int i, char *psz = 0, double x, int j = -1);
```

  - Move all default args to end:

```
void foo(int i, double x, char *psz = 0, int j = -1);
```

Rules for Default Values (2)

- When calling a function with default values, can't skip over some arguments
  - Only *trailing* arguments may be left unspecified
- Continuing previous example:

```
void foo(int i, double x, char *psz = 0, int j = -1);
```

  - This is invalid:

```
foo(3, 2.25, , 15);          // let psz default to 0
```

- These are the only valid call patterns:

```
foo(3, 2.25);                // psz = 0, j = -1
foo(3, 2.25, "hello");       // j = -1
foo(3, 2.25, "hello", 15);   // no args unspecified
```

Function Pointers and Default Values

- <u>All</u> arguments are part of a function's type
  - …even those with default values
- Example:

```
void f(int i, int j = 0);
...
void (*fp1)(int, int) = f; // OK
void (*fp2)(int) = f;      // COMPILE ERROR
```

Another Design Example

- Create a 2D point class
  - Default coordinates to (0, 0)

```
class Point {
      float x_coord, y_coord;

public:
      Point(float x = 0, float y = 0) :
            x_coord(x), y_coord(y) { }

      ...
};
```

- Do all valid call patterns make sense?

```
Point a;          // defaults to (0, 0)
Point b(15, 2);
Point c(3);       // (3, 0) ???
```

Default Arguments

- When using default arguments, make sure that <u>all</u> valid call patterns make sense.
- For 2D points, no reason to specify X coordinate and let Y default to 0.
- Create two separate constructors:

```
Point() : x_coord(0), y_coord(0) { }
Point(float x, float y) :
    x_coord(x), y_coord(y) { }
```

  - <u>Disallow</u> the call-pattern **`Point(x)`**

Class Members

- Class-members are usually associated with objects

```
class Matrix4F {
  float values[16];
public:
  ...
  bool isOrthogonal() { ... }
};
```

  - Each matrix-object has its own copy of **`values`** member
  - Calling **`isOrthogonal()`** requires a specific object

```
Matrix4F m = ... ;
if (m.isOrthogonal())
    m.transpose();  // can invert by transposing
```

Static Members

- Might want to provide constants for our class
  - Only want <u>one</u> copy of the constant for the entire class, not one per object!
- Also might want to provide general utility functions for our class
  - Functions that help with the class, but don't need to be called on a specific object
- Static class-members aren't associated with a specific object
  - The member is associated with the class itself, not with individual objects

Static Constants

- Add an **`Identity`** constant to our matrix class
- Updated declaration, in our **`matrix.hh`** file:

```
class Matrix4F {
  ...
public:
  ...
  // Identity matrix constant
  static const Matrix4F Identity;
};
```

- Static member is <u>not</u> defined or initialized in the declaration!
  - *Separately* initialized in corresponding **`matrix.cc`** file
  - (reasons are gross and involve compilation/linking issues)

Static Constants (2)

- Within the corresponding **`matrix.cc`** file:

```
const Matrix4F Matrix4F::Identity =
    Matrix4F().setIdentity();
```

  - Only use **`static`** keyword in declaration, not in definition
  - Refer to static member by its qualified name:
  `Matrix4F::Identity`
- To use default initialization for static member:

```
const Matrix4F Matrix4F::Identity;
```

  - Still must appear in the **`.cc`** file! Otherwise, linker errors…
- Other code can refer to the static constant:

```
#include "matrix.hh"

if (m == Matrix4F::Identity)
  ...
```

Static Functions

- Add a static function to convert Euler angles into corresponding transform matrix

```
class Matrix4F {
   ...
public:
   ...
   static Matrix4F getEulerTransform(
      const EulerAngles &a);
};
```

  - Function definition can appear in class declaration or in definition
  - (only static member-variables are finicky)

Static Functions (2)

- Can call static function without requiring an object

```
#include "matrix.hh"

Matrix4F m;
EulerAngles a;
...
m = Matrix4F::getEulerTransform(a);
```

  - Use qualified name to refer to static member-function
- Within the **`Matrix4F`** class definition, don't need fully qualified name to refer to static member

C++ Templates

- C++ supports both class templates and function templates
- Example class template:

```
template<typename T> class Point {
     T x_coord, y_coord;

public:
     Point() : x_coord(0), y_coord(0) { }
     Point(T x, T y) : x_coord(x), y_coord(y) { }
     ...
};
```

  - Instantiate template as follows:

```
Point<float> p1(3.31, 2.67);
Point<int> p2(15, -6);
```

Function Templates

- Function templates are declared in a similar way:

```
template <typename T>
T square(T val) {
    return val * val;
}
```

  - Squares its input
  - Works on any type that supports multiplication operator `*`
- To use:

```
int i = 15;
cout << i << " squared = " << square(i) << endl;

Matrix m(3, 3);    // Matrix provides operator*
Matrix m2 = square(m);
```

Function Templates (2)

- Function templates don't require the template parameters to be specified!

```
int i = 15;
cout << i << " squared = " << square(i) << endl;
Matrix m(3, 3);
Matrix m2 = square(m);
```

  - Compiler figures it out from the function's context
- Function-template declaration:

```
template <typename T> T square(T val)
```

  - When arg is an **`int`**, **`square`** is instantiated with **`T`** = **`int`**
  - When arg is a **`Matrix`**, **`square`** is instantiated with **`T`** = **`Matrix`**

Function Templates (3)

- Can specify template parameters if you want:

  ```
  double result = square<double>(3);
  ```

  - 3 is an **int** by default, so **square(3)** would instantiate **square<int>()** …
  - …but that wouldn’t actually change the answer.
  - Explicitly specifying function-template params is rarely necessary
    - Primarily needed when passing a function-template instantiation to another function (or function-template)

Functor Adapters

- STL provides “function object adaptors”
  - **`not1`**, **`not2`** – negate a unary or binary predicate
  - **`bind1st`**, **`bind2nd`** – turn a binary functor into a unary functor
    - Bind a value into the first or second argument
  - **`compose1`**, **`compose2`** – compose unary or binary functions
    - e.g. compose f(x) and g(x) to produce f(g(x))
- These are actually <u>functions</u> that create function-object adaptors for you
  - Functor adapters are <u>structs</u> that implement **`operator()`**
  - See STL docs for more details…

Adaptable Function Objects

- Functor adapters only work on Adaptable Function Objects
  - Another concept in the STL functor specification
  - Functor adapters also produce Adaptable Function Objects
- Adaptable Function Objects are:
  - Function objects that specify the types of their arguments and return-value as nested **`typedef`**s…
  - Adaptable function objects are classes or structs that implement **`operator()`**
- STL provides:

```
unary_function<class Arg, class Result>
binary_function<class Arg1, class Arg2, class Result>
```

Adaptable Function Objects (2)

- Example implementation of STL **`unary_function`**:

```
typedef template <class Arg, class Result>
struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
} unary_function;
```

- Our adder functor from lecture 1:

```
struct adder : public unary_function<int, void> {
    int sum;
    adder() : sum(0) { }
    void operator()(int x) { sum += x; }
};
```

  - Derived from **`unary_function`** so that our adder is <u>adaptable</u>
  - Can use our adder functor with the functor adapters

Function Pointers and Adapters

- Simple function-pointers aren't adaptable
  - Adaptable functors must provide nested **`typedef`**s
  - Simple function pointers can't provide this!
- STL provides wrapper-structs for function pointers

  `pointer_to_unary_function<Arg, Result>`
    - Derives from **`unary_function`**

  `pointer_to_binary_function<Arg1, Arg2, Result>`
    - Derives from **`binary_function`**
- Wrapper struct stores the function-pointer
  - Necessary **`typedef`**s are also provided
  - Wrapper's **`operator()`** calls the function-pointer

Function Pointers and Adapters (2)

- Typing long class-template names is lame…
  - Plus you have to specify the template parameters since it's a class-template
- STL also provides **`ptr_fun()`** function-template to generate an appropriate wrapper
- Example implementation of **`ptr_fun()`**:

```
template <class Arg, class Result>
pointer_to_unary_function<Arg, Result> // Return type
ptr_fun(Result (*func)(Arg)) {
  // Instantiate the wrapper struct, then return it.
  return pointer_to_unary_function<Arg,Result>(func);
}
```

  - Template parameters are inferred from the context!
  - A second version of **`ptr_fun()`** for binary functions, too

Function Pointer Example

- Predicate for identifying interesting widgets

```
bool isInteresting(const Widget *pw) { ... }
```

- Find the first interesting widget:

```
list<Widget *> widgetPs;
...
list<Widget *>::iterator iter = find_if(
    widgetPs.start(), widgetPs.end(), isInteresting);
```

  - `find_if` doesn't require adaptable functors

- Find first <u>uninteresting</u> widget:

```
list<Widget*>::iterator iter = find_if(
    widgetPs.start(), widgetPs.end(),
    not1(ptr_fun(isInteresting)));
```

  - `not1` requires an adaptable functor…
  - `ptr_fun` turns `isInteresting` into an adaptable functor.

Functor Tips

- STL containers and algorithms don't actually *require* Adaptable Function Objects
  - It's a refinement of the simple Function Objects, only used by the Function Object Adapters
- When you write your own function objects, consider making them adaptable
  - i.e. structs or classes that provide **`operator()`**
  - Then you and others can use functor adapters

STL and Copying Objects

- When you put a value into an STL container:
  - STL stores a copy of the value.
- When you get a value out of STL container:
  - STL returns a copy of the object.
- Similarly, if you permute, sort, … an STL container: all kinds of copying is going on!

- “Copy in, copy out. That’s the STL way.”

How Does STL Copy This Stuff?

- STL relies on two things:
  - Copy constructor:
    - `Widget(const Widget&);`
  - Copy-assignment operator:
    - `Widget & operator=(const Widget &);`
- "Make copying cheap and correct for objects in containers." - <u>Effective STL</u>, Item 3.

- Slow copy + containers → your program will *crawl*
- If copying has a nonstandard meaning, things break
  - For a few classes, copying actually <u>changes</u> what is copied

It Slices, It Dices!

- STL containers mixed with inheritance can lead to slicing

```
class Widget {...};
class SpecialWidget : public Widget {...};

vector<Widget> widgets;
SpecialWidget sw(...);
widgets.push_back(sw);
```

- **`SpecialWidget`** is copied as a **`Widget`**!
  - All **`SpecialWidget`** data-members are sliced off
- Be careful mixing inheritance and containers

Avoiding Copy Problems

- Instead of storing instances, store pointers
  - …but now you have to clean things up

```
vector<Widget *> widgetPs;

...  // Do Widget work.

// Clean up!
vector<Widget *>::iterator iter;
for (iter = widgetPs.begin();
     iter != widgetPs.end(); iter++) {
    delete *iter;
}
```

Using **`for_each`** to Clean Up

- Turn **`delete`** into a functor

```
template<typename T> struct DeleteObject :
    public unary_function<const T *, void> {

    void operator()(const T *ptr) const {
        delete ptr;
    }
};
```

- Now you can clean up like this:

```
for_each(widgetPs.begin(), widgetPs.end(),
    DeleteObject<Widget>());
```

  - **`DeleteObject`** template requires the type of object that's pointed to

That's Nice, But…

- This is still prone to error!
- You have to make sure your types match

```
vector<SpecialWidget *> specialPs;
...
for_each(specialPs.begin(), specialPs.end(),
    DeleteObject<Widget>());
```

- If **`Widget`** doesn't have a virtual destructor, you're in trouble again!
  - "Deletion of derived object via base-class pointer, where there is no virtual destructor…"

The Best Way…

- Change to using a function-template within the delete functor

```
struct DeleteObject {
    template<typename T>
    void operator()(const T *ptr) const {
        delete ptr;
    }
};
```

- Now it’s type-safe. And you type less.

```
vector<SpecialWidget *> specialPs;
...
for_each(specialPs.begin(), specialPs.end(),
    DeleteObject()); // Create temporary object
```

  - Now the compiler will infer the type information

Adaptable Delete Functors?!

- First delete functor:

```
template<typename T> struct DeleteObject :
    public unary_function<const T *, void> {

    void operator()(const T *ptr) const {
        delete ptr;
    }
};
```

  - It's adaptable; derives from **`unary_function`**
  - …but <u>no real value</u> in making delete functors adaptable!
- 2nd delete functor isn't adaptable, but easier to use
  - Doesn't always make sense to create adaptable functors!

This Week's Lab

- Continue implementing ray tracer classes
- Scene-object class hierarchy
  - Sphere and plane, for now
- Simple representation for point-lights
- Object to represent the whole scene
  - Use STL vectors to store collections of scene objects and lights
    - (specifically, pointers to these objects)
- Object to represent a ray being traced