CS11 – Advanced C++

Winter 2014-2015

Lecture 2

The Ray Tracer

- This term's project will be a ray tracer
  - Very well suited to C++ language features
  - Class hierarchies and operator overloading, in particular
  - Also, basic use of STL and other good C++ practices
- Will focus on <u>simple</u> ray-tracing features
  - Simple objects like spheres, planes, cylinders, etc.
  - Little to no rendering optimization ☺
  - Simple scene description format
  - Later labs focus on additional features
- Start by implementing basic abstractions
  - Go from simple to complex

First Tasks

- Implement a 3D vector class
  - Provide all necessary math operations
  - Use operator overloads to make coding easier
- Implement an RGB color class
- Implementations need to be fully featured
- Also needs to be reasonably fast
  - Avoid unnecessary use of dynamic memory allocation
- Implementation also needs to be bulletproof!
  - Specify `const` in appropriate places
  - Use assertions everywhere they're appropriate
  - Reuse, reuse, reuse! Less code means less bugs.

Fixed-Size Vectors

- Common approach is to use an array for elements

```
// A 3D vector with float elements.
class Vector3F {
  float elems[3];
public:
  Vector3F();
  Vector3F(float x, float y, float z) { ... }
  ...
};
```

  - No dynamic allocation of elements – fast!
  - No need for a destructor.
  - Don't even need a copy constructor; C++ knows how to copy a fixed-size array member

Element Access

- Also want to provide element access
  - Could do: **`float Vector3F::getElem(int i)`**
  - Or, could overload the **`[]`** operator

```
Vector3F v;
...
v[0] = 15.3;
v[1] = v[0] * 0.65;
```

  - Simple, widely used notation
- Implementing the **`[]`** operator:
  - Implement as a member function
  - Takes *exactly* one argument (argument type is flexible)
  - Returns some value

Implementing `[]` Operator

- Actually need to provide *two* versions of `[]`
- First version is for read-only access:

```
float Vector3F::operator[](int i) const {
  assert(i >= 0);
  assert(i < 3);
  return elems[i];
}
```

  - Can't use this on LHS of an assignment

```
cout << v[2] << endl;    // OK
v[0] = 25;               // Compile error!
```

Using `[]` with Assignment

- Implement second version of `[]` for use on LHS of assignment

```
float & Vector3F::operator[](int i) {
  assert(i >= 0);
  assert(i < 3);
  return elems[i];
}
```

  - This version of `[]` isn't const. (It really *can't* be…)
  - Returns a *reference* to the element
    - Allows assignment directly to that element
- Now this works: `v[0] = 25;`
  - `v[0]` evaluates to a non-const reference to the first elem
  - Allowed to assign to a non-const reference

Assignment and Encapsulation

- Any issues with this approach?
  - Exposes internal values to users – violates encapsulation
  - Fine for a vector class – direct element access is both expected and common
  - In general, is usually a *really bad idea.*

Direct Member Access

- Example:

```
class Widget {
  double wgt;   // Weight of the widget
public:
  Widget(double w) : wgt(w) { assert(w >= 0); }

  double weight() const { return wgt; }
  double & weight() { return wgt; }
};
```

  - Widget weights should *probably* be nonnegative…
- Can use our widget like this:

```
Widget w(35);
cout << "Widget's weight is:  " << w.weight() <<
   endl;
```

Direct Member Access (2)

- Can also write this code:

```
Widget w(35);
w.weight() = -6;
```

- Uses non-**`const`** version of **`weight()`**
  - Allows direct access to widget's **`wgt`** field
- No way to check new values for validity!
  - A negative weight doesn't make any sense at all.
- If you need to check new values, write *real* accessors and mutators
  - Can include tests, assertions, etc.
- Only return non-**`const`** references to data members when it *really* makes sense

Overloading the `()` Operator

- Can use `()` instead of `[]` if desired
  - Parentheses usually denote function invocation
  - Can give them additional meanings
- Sometimes you *have to* use `()` instead of `[]`
  - `[]` takes exactly one argument
  - `()` can take any number of arguments
- Implementation:

```
float Vector3F::operator()(int i) const;
float & Vector3F::operator()(int i);
```

  - `()` overload *must* be a member function
  - Can take any number, type of arguments
  - Can return any type of value

Using () vs. []

- Example of using `()` instead of `[]`
  - A matrix class (e.g. a 4x4 square transform matrix)
  - Again, would like direct access to matrix elements
  - Can't use `[]` because we need two args: row and column
  - Use `()` instead:

```
// Version for use as target of assignment
float & SquareMatrix4F::operator()(int r, int c) {
  assert(r >= 0 && r < 4);
  assert(c >= 0 && c < 4);
  return elems[r * 4 + c];
}
```

  - Now we can write:

```
SquareMatrix4F m;
m(3, 1) = 0.975;
```

Final Note About `()`

- Normal use of `()` is for function invocations

```
double y = sin(angle);
```

- Can imitate function invocations by overloading `()` operator

```
SquareMatrix4F m;
...
double v = m(2, 2);
```

  - Syntax for element access is identical to function invocation
- This syntactic similarity is used *heavily* by C++ Standard Template Library
  - Function objects (aka "functors") emulate simple function calls using overload of `()` operator

Vector Math

- Can multiply vectors by scalars
  - Simple scaling operation
- Compound assignment operators `*=` and `/=`
  - Always implement these as member functions

```
Vector3F & Vector3F::operator*=(float factor) {
  for (int i = 0; i < 3; i++)
    elems[i] *= factor;

  return *this;
}
```

Note: many compilers can optimize this code by "unrolling the loop," since lower and upper bounds are constant.

  - All assignment operators return a non-`const` reference to `*this`
  - Can only have a vector on LHS, and a scalar on RHS
    - Other order doesn't make any sense

Vector Math (2)

- Also implement simple arithmetic operators `*` and `/`
  - Need to support both (vector * scalar) and (scalar * vector)
- Problem: can't do this with member functions
  - Can do (vector * scalar), but not (scalar * vector)
- These should be implemented as non-member operator overloads
  - Operator overloads defined outside of the class

    ```
    const Vector3F operator*(const Vector3F &v, float s);
    const Vector3F operator*(float s, const Vector3F &v);
    ```
  - LHS is first argument, RHS is second argument
  - Simple arithmetic operators always return a **`const`** value
  - Implement these in terms of `*=` and `/=`, of course!

General Operator-Overload Guidelines

- Must be member-functions: **`=  ()  []  ->`**
  - Compound assignment ops *should be* member-functions
- Cannot be member functions: **>> <<**
  - (at least, not when using them for stream-output)
  - Require a stream on the LHS
- Some more guidelines:
  - If operator can be implemented using only class' public interface: non-member *strongly* recommended
  - If operator supports mixed types: non-member
  - If operator overload must be virtual: member-function
  - If none of the above, make it a member-function

Implementing Stream-Output

- C++ uses **<<** for stream output, **>>** for stream input

```
string name;
cout << "What is your name?  ";
cin >> name;
cout << "Hello, " << name << endl;
```

- Stream output operator:

```
ostream & operator<<(ostream &os, const T &value);
```

  - LHS is an output stream, RHS is value to output
  - Return the passed-in **`ostream`**, to allow operator chaining

Outputting Vectors

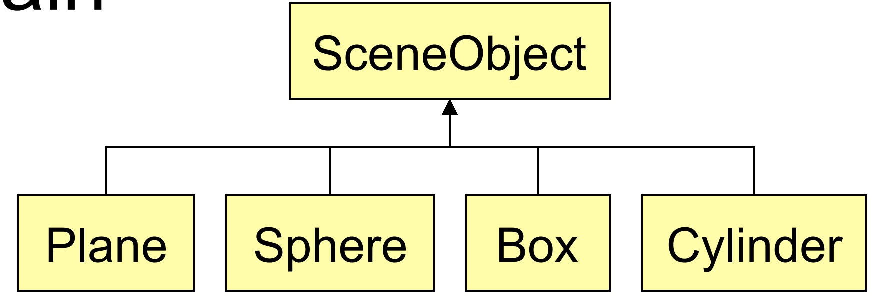
- Simple implementation for vectors:

```
ostream & operator<<(ostream &os, const Vector3F &v) {
  os << "(" << v[0] << ", "
            << v[1] << ", "
            << v[2] << ")";
  return os;
}
```

  - Build from simpler output operations to make this easy
  - Usually don't include an **`endl`**
    - The caller should get to choose whether or not **`endl`** is added
- Want to choose a clean, simple format
  - Stream input should consume same format
    - Will cover stream input in a future lecture…

Class Hierarchies and Stream Output

- Implementing stream-output for class hierarchies can be a pain



- A naïve approach:
  - One **`operator<<`** implementation for every class in the hierarchy
  - When new classes are added in future, need to add another **`operator<<`** implementation
  - Easy to leave out one of the classes by accident!

Class Hierarchies and Stream Output (2)

- What about collections of pointers to these objects?
- Example:
  - A vector of different scene-object subclasses, stored as pointers

```
vector<SceneObject *> sceneObjs;
vector<SceneObject *>::iterator iter;

iter = sceneObjs.begin();
while (iter != sceneObjs.end()) {
  cout << **iter << end;
  iter++;
}
```

- What will this print?
  - Uses **`SceneObject`** version of **`operator<<`** for all objects
  - *Can't* make **`operator<<`** virtual: it's not a member function!

Class Hierarchies and Stream Output (3)

- Need to leverage virtual functions for this problem
- Solution:
  - Make a virtual **`SceneObject::print(ostream &)`** function
    - Might even want to make it pure-virtual
  - Create <u>one</u> stream-output operator, for **`SceneObject`**

```
ostream & operator<<(ostream &os,
                     const SceneObject &so) {
  so.print(os);
  return os;
}
```

  - Every subclass provides its own implementation of **`print()`**
    - Base class can *force* subclasses to implement **`print()`** themselves, by declaring **`print()`** pure-virtual

Increment and Decrement Operators

- C and C++ include increment (**++**) and decrement (**--**) operators

```
int i = 5;
int j = i++;      // post-increment
```

  - `i = 6, j = 5`

```
int k = ++i;      // pre-increment
```

  - `i = 7, k = 7`

- Can overload these operators as well
  - e.g. for user-defined numeric types, iterator implementations, etc.

Overloading Increment/Decrement

- Need to distinguish between pre-increment and post-increment in function signature!
- Pre-increment takes no argument:
  - **`T& T::operator++();`**
  - Returns a reference to variable after it has been incremented
- Post-increment takes a dummy **`int`** arg:
  - **`const T T::operator++(int);`**
  - Argument-value is meaningless! Don't use it! ☺
  - Returns a copy of the value before incrementing
- Decrement overloads follow same pattern

Overloading Increment (2)

- Usually implement post-increment in terms of pre-increment

```
const T T::operator++(int) {
  const T old(*this);
  ++(*this);    // reuse!
  return old;
}
```

- Could also specify full name of operator:

```
this->operator++();
```

Post-Increment and **`const`**

- Post-increment returns a **`const`** object for same reason as simple arithmetic operators
  - Prevent operator chaining!
- A **`BigInt`** class that represents arbitrarily large integers
  - Defines prefix/postfix **`++`** and **`--`** operators
  - Postfix operators *don't* return **`const`** objects
- What is value of **`n`** after this code?

```
BigInt n(3);
...
n++++;
```

Post-Increment and **`const`** (2)

- What is value of **`n`** after this code?

```
BigInt n(3);
...
n++++;
```

- What does the compiler see?
  - **`n.operator++(0).operator++(0);`**
  - (assume compiler passes 0 for dummy value)
- First **`operator++(int)`** returns a temp object
- Second **`operator++(int)`** is called on that temp object!
- **`n`** only becomes 4, not 5!

Post-Increment and `const` (3)

- If post-increment operator returns **`const`** object, this code becomes invalid:
  - **`n.operator++(0).operator++(0);`**
  - Compiler won't allow a **`const`** object to be mutated

This Week's Lab

- Write up classes for vectors and RGB colors
  - Required operations are listed in assignment
  - Use operator overloads to make vector math easy
  - Follow member/nonmember overload guidelines!
- Focus on:
  - Correctness – use assertions and **`const`**!
  - Good documentation
  - Clean, consistent coding style
  - Performance:
    - Avoid dynamic allocation

Ray Tracer Components

- Building a raytracer requires a lot of work
  - This week: 3D vectors, RGB colors (along with operator overloads)
  - Future weeks: rays, shape objects, etc.
- Really won't have a complete program to test for several weeks
- Instead, can use *unit testing* to verify the code you create

Unit Testing

- *Unit testing* is focused on exercising minimal units of the program
  - For most languages, “minimal unit” is a function
  - Individual tests focus on exercising functionality of one or a few functions
- Other kinds of tests as well:
  - *Integration tests* focus on verifying behavior across multiple modules within the program
  - *Regression tests* focus on verifying that bugfixes and feature additions do not break other features in code
    - (usually include a large suite of both unit and integration tests)

Unit Testing (2)

- Generally, unit tests are automated
  - Could certainly create programs that perform ad-hoc testing, but becomes a nightmare to manage
- Use a *unit-testing framework* to provide common functionality:
  - A mechanism for determining what tests to run, possibly as specified by the developer
  - A way to record test successes and failures, plus details output by individual tests
  - A set of helper functions to perform common checks (e.g. verify that two **`char*`** values are the same)

Unit Testing Frameworks

- Many unit-testing frameworks provide several levels of abstraction
- *Test cases*: minimal unit of testing
  - Individual test cases are run, and succeed or fail
- *Test fixtures*: the state or context required for one or more test cases
  - Example: testing a database application
    - Fixture may require loading specific data into the DB for exercising the functionality being tested
  - Frameworks often provide set-up/tear-down hooks to properly initialize and clean up the required fixtures

Unit Testing Frameworks (2)

- *Test suites*: collections of tests that share the same fixture
  - Test cases in suite can be run in any order
  - (Need to re-initialize fixture before each test)
- Testing frameworks often use classes to represent test suites:
  - Individual test cases are methods on suite-class
  - Special methods (e.g. setUp() and tearDown()) initialize/clean up any fixtures needed for tests

Unit Testing Frameworks (2)

- Several C++ unit-testing frameworks
  - CppUnit (a port of the very popular JUnit Java unit-testing framework)
  - CppUnitLite, NanoCppUnit – lighter-weight versions of CppUnit
  - Boost.Test library (more on Boost in a few weeks)
  - Google C++ Testing Framework
  - …and many, many more
- We will use Google C++ Testing Framework