Using classes

William Hendrix

Lecture 16

Outline

- Class usage
- Pointers and arrays
- Operator overloading
- `friend`
- `this`

Using classes

- Declaring a class:

```
Class_name obj;
Class_name obj(constructor_args);
```

- Anonymous objects

```
Class_name(constructor_args)
```

  - Destroyed immediately if not assigned

- Use . to access functions and data members

```
str.length(); complex.re
```

- Classes are constructed on declaration
  - Class data members are stored like a struct
  - All objects of a class share the functions
- Destroyed when object passes out of scope, etc.

Class pointers

- Syntax somewhat similar to dynamic array

```
Class* ptr;
ptr = new Class(args...);
delete ptr;
```

- Shortcut: `->` is like `.`, but dereferences first

```
strptr->length() //Or (*strptr).length()
```

- Constructed on `new`, destroyed on `delete`
  - Allows more control over construction/destruction
  - Generally preferable though somewhat more tedious
- Class arguments to functions are almost always pointers or references
  - Avoids making a copy of the object
    - Pointers are always 4 or 8 bytes
    - Copying an object uses a constructor

Copy constructors

- One of the following:

```
Class(Class& copy);
Class(const Class& copy);
```

- Invoked every time a function with a class argument is called
  - Copy constructor cannot be:

```
Class(Class copy);
```

- C++ provides a default copy constructor
  - Copies all data members
  - Usually sufficient unless your class allocates dynamic memory
    - Default would result in shallow copy

Class arrays

- Syntax:

```
Class* arr;
arr = new Class[numElements];
```

  - Always uses default constructor

- Elements can be referenced and modified as normal
  - E.g., `arr[i].print();`

- Deallocating the array:

```
delete[] arr;
```

  - Destroys every element, then frees the array
  - Using `delete arr;` is legal but causes memory leak

Operator overloading

- Why do we use `str1 + str2` instead of `str1.concat(str2)`?
  - C++ allows us to define functions using operators like +, –, etc.
- Two ways
  - Define operator as a member function of a class

    ```
    Class Class::operator+(Class& other);
    ```

    - Only works for unary operators or when class object on LHS
  - Define operator as a standalone function

    ```
    Class operator+(Class& lhs, Class& rhs);
    ```

  - Return type can be anything
  - Unary operators (e.g., `!`) require one fewer argument
- Nearly all operators can be overloaded
  - Exceptions: `.`, `::`, `?:`, and `.*`
  - Notable operators: `=`, `==`, `-`, `+=`, `*`, `&`, `[]`, `<<`
    - Can even overload type casting

      ```
      Class::operator bool() {}
      ```

Overriding access modifiers

- Externally-defined functions can't access private data

```
Complex operator+(Complex& lhs, Complex& rhs)
ostream& operator<<(ostream& out, Complex& c)
```

- `friend` keyword
  - Allows a function or class access to private fields and functions
  - Appears in class definition for class granting permission

- Syntax

```
friend ostream& operator<<(ostream& out, Complex& c);
friend Polynomial;
```

Classes as data members

- Data members can be any type, including classes

- Before constructor is called, every data member is constructed with default parameters
  - Member must have a default constructor
  - Inefficient if we want constructor to initialize

- Alternate constructor syntax

```
Class::Class(int args)
  :class_member1(args), class_member2(args)
{
  // ...
}
```

  - Some programmers find this syntax confusing

Self-reference

- In a member function, we can access fields like local variables
  - What if we need to refer to the entire object?

- `this` keyword
  - Pointer to the current object
  - E.g., `this->re`

- Example

```
void Complex::square()
{
  mult(this);
}
```

Tonight

Midterm is Feb. 11

Lab 4 is due Wednesday, Feb 12

Recommended reading: Sections 5.1-5.7, 14.1, 14.3, 14.5, 18.4, Advanced Topic 7.1