CSE 374: Lecture 24

C++ Classes

Header File Review

Headers exist to define an interface to a module.

Headers contain

- #includes that are needed for the header to compile
- #macros that may be needed by calling code
- Datatype (struct, typedef) that are needed for the interface definition
- Function declarations that define the interface

Headers do not contain

- #includes that are only needed within the module code
- Datatypes that are only needed within module
- ‘Private’ function declarations
- Function definitions

Header File Review

Don’t ‘over-include’

- Only include the headers that are necessary for the code.
  - Well defined headers control access to modules - mimic encapsulation
  - mem_impl.h is for ‘internal’ stuff
  - Only include mem.h in bench (calling module)

- Don’t forget

```
#ifndef HGUARD
#define HGUARD
...
#endif
```

mem_impl.h

1. Includes:
    a. Any functions only used in the implementation of the memory program.
    b. Check_heap?
    c. Print_heap?
    d. Definition of a listNode struct
    e. Extern global freeList declaration ok
2. Included in:
    a. Memory management modules, but not bench

Where do you 'define' freeList?

Other things to do next for HW6

1. Run clint.py; Fix stuff
2. Work on your ReadMe
    a. Tell us who did what parts of the project
    b. Do it early to help coordinate with partner
3. Work on ‘bench’
    a. Do it first so you can test your program as you go
4. Make sure make works
    a. Build a test target that does things like run valgrind and call bench

C Structs: Not object-oriented

```
typedef struct person {
  char* name;
  int age;
} person;
```

```
person *p2;
char name[MAX_NAME];
int  age;
// fill name, age
p2 = makePerson (name, age);
```

```
person* makePerson (char *name, int a) {
  person* p = (person*) malloc (sizeof (person));
  p->name = (char*) malloc (MAX_NAME+1);
  strncpy (p->name, name, MAX_NAME);
  p->age = a;
  return p;
}
```

Notes:
Not self contained
need to allocate heap memory so object will persist
need to allocate memory for the string
Unless you statically declare (char name[MAX_NAME])

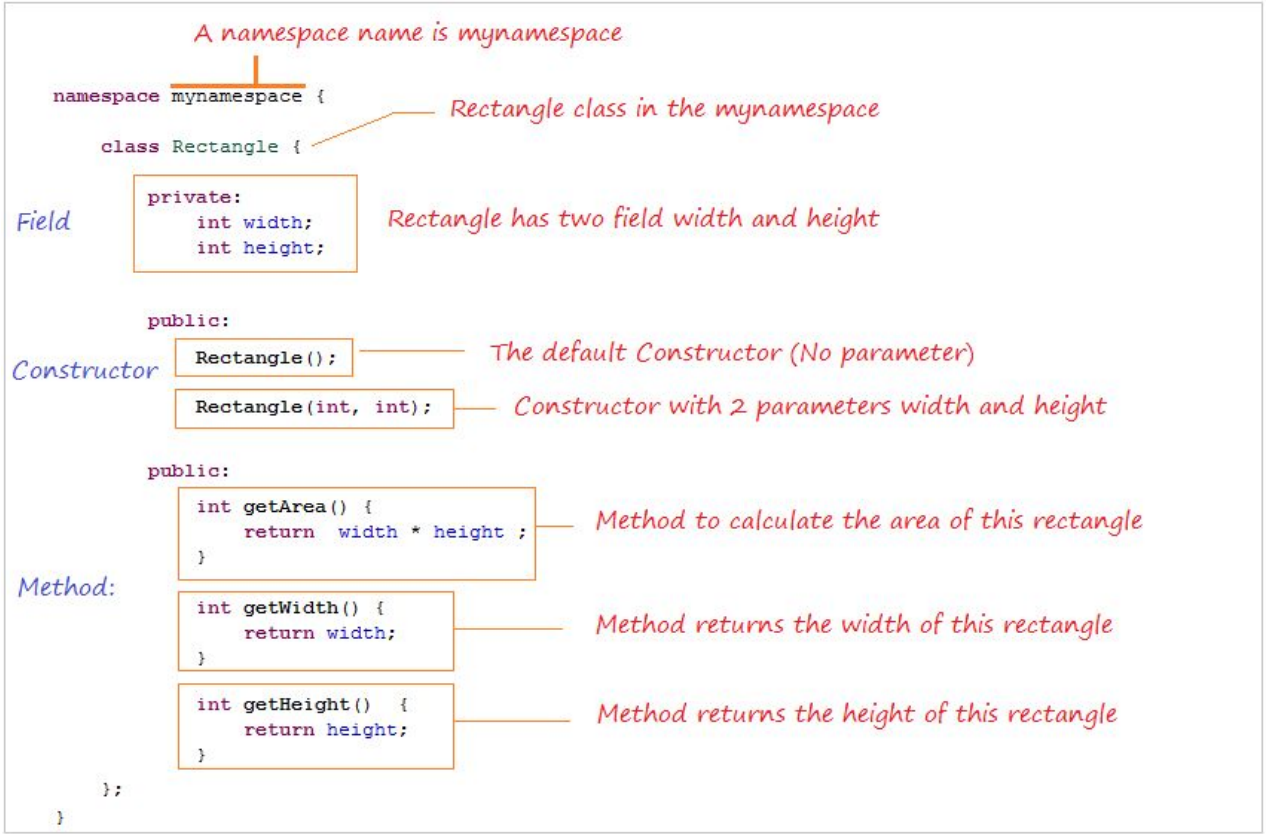
C++ classes: object-oriented

```
class String {
        public:
        String();
        String(const String& other);
        String(const char* raw);
        virtual ~String();
        String& operator=(const String& other);
        size_t length() const;
        void append(const String& other);
        void clear();
        friend std::ostream&
operator<<(std::ostream& out, const String& s);

        private:
        void makeNewRaw(size_t length);
        char* raw_;
};
```

Classes - can define fields and methods

Class layout



Classes

- Like Java
  - Fields vs. methods, static vs. instance, constructors
  - Method overloading (functions, operators, and constructors too)
- Not quite like Java
  - access-modifier (e.g., private) syntax and default
  - declaration separate from implementation (like C)
  - funny constructor syntax, default parameters (e.g., ... = 0)
- Nothing like Java
  - Objects vs. pointers to objects
  - Destructors and copy-constructors
  - virtual vs. non-virtual (to be discussed)

Class Constructors (4 types)

- A *default constructor* takes zero arguments. If you don't define any constructors for your class, the compiler will generate one of these constructors for you.
- A *copy constructor* takes a single parameter which is a *const reference* (const T&) to another object of the same type, and initializes the fields of the new object with a COPY of the fields in the referenced object.
- *User-defined constructors* initialize fields and take whatever arguments you like.
- *Conversion constructors* are constructors that take a single argument. For our string example this is like:

```
String(const char* raw);
String s = "foo";
```

Copy Constructors

- In C, we know x=y or f(y) copies y (if a struct, then member-wise copy)
- Same in C++, unless a copy-constructor is defined, then do whatever the copy-constructor says
- A copy-constructor by definition takes a reference parameter (else we'd need to copy, but that's what we're defining) of the same type
- Copy constructor vs. assignment
  - Copy constructor initializes a new bag of bits (new variable or parameter)
  - Assignment (=) replaces an existing value with a new one
    - may need to clean up old state (free heap data?)

Implicit constructors & destructors

Conversion constructors are implicit: automatically applied when a constructor is called with one argument.

If you want a single argument constructor that is not implicit, must use

```
explicit String(const
char* raw);
```

Destructors are used by ‘delete’ to clean up when freeing memory.

```
Virtual ~String();
```

You do not call destructors explicitly

Stack v. Heap

Java: cannot stack-allocate an object (only a pointer to one; all objects are dynamically allocated on the heap - all objects are pointers to objects)

C: can stack-allocate a struct, then initialize it (An actual object)
C++: stack-allocate and call a constructor (where this is the object's address, as always, except this is a pointer) Thing t(10000);

Java: new Thing(...) calls constructor, returns heap allocated pointer
C: Use malloc and then initialized, must free exactly once later, untyped pointers
C++: Like Java, new Thing(…), but can also do new int(42). Like C must deallocate, but must use delete instead of free. (never mix malloc/free with new/ delete!)

Subclasses

- **Polymorphism.** In essence, polymorphism is the ability access different objects through the same *interface*. For instance, if you have an interface that represents an electronic device, that interface would have the ability to turn the device on and off. You can use the actual physical types - computer, phone, television, etc - as if they were an electronic device, because they all have the on/off capability.
- **Inheritance.** This is one of the meatiest pieces of OO programming. Inheritance allows the sharing of BEHAVIORS. For instance, a Square is a type of Rectangle, and has the same way to compute its area (width times height) - therefore by make Square inherit from Rectangle, we can share that behavior and avoid duplicating the code.