CSE 374: Lecture 24

C++ Classes

Week 9

Finish Test 4

Resubmit HW5 if necessary

HW6:

- Partner git repositories made
- Get started soon

Check Canvas and grades

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
```

C++ Hello, World!

```
#include <cstdlib>
#include <iostream>

const int CURRENT_YEAR = 2019;

using namespace std;

// REFERENCE
void pig(string& s) {
  char first = s[0];
  s = s.substr(1);
  s += first;
  s += "ay";
}
```

```
int main() {
  // stack-allocated array: int arr[100];
  // C++ style heap allocation:
  int* arr = new int[100];

  // C++ style array deletion:
  delete [] arr;
  // Use "delete x;" for things non-arrays.

  cout << "What is your name? ";
  string name;
  cin >> name;
  pig(name);
  cout << "What year were you born? ";
  int year;
  cin >> year;
  const int age = CURRENT_YEAR - year;
  cout << "Hello, " << name << "!" << endl;
  cout <<"You're "<<age<< " years old"<< endl;
  return EXIT_SUCCESS;
}
```

So, what different with C++?

- File Names (instead of *.c)
  - *.cc or *.cpp or *.cxx
- Compiler (instead of gcc)
  - $g++
- Preprocessor (still uses C preprocessor)
  - But #include <cstdlib>

- Still use *.h for header files

- Basically does the same thing as <stdlib.h>

Namespaces

- Group code logically
- Can re-use names for each namespace
- Can nest namespaces
- Disambiguate with :: syntax
- Can avoid using the prefix with `using namespace foo`
  `doSomething(3)`
- If you are using a namespace in a header, you must also use the namespace in the source code (.cpp)

```
namespace foo {
    int doSomething(int x);
}

namespace bar {
    int doSomething(int x);
}

int main() {
    foo::doSomething(3);
    bar::doSomething(3);
}
```

I/O in CPP

Std library include a `cout` and a `cin` function

Operators ‘>>’ and ‘<<’ act like shell redirection

Operators ‘>>’ and ‘<<’ take left and right operands and return a stream

Use namespace std or

use `std::cout` & `std::cin`

```
using namespace std

cout << "What is your name? ";
string name;
cin >> name;

cout << "When were you born? ";
int year;
cin >> year;
```

Pass by reference

- In C: all function arguments are copies
  - Pointer arguments pass a copy of the address value
- In C++: Can do the above
  - but can also use a “reference parameter” (& character before var name)
  - As though the calling line wrote pig(&name) and in ‘pig’ every ‘s’ is a ‘*s’

```
void pig(string& s) {
  char first = s[0];
  s = s.substr(1);
  s += first;
  s += "ay";
}
```

```
string name;
cin >> name;
pig(name);
```

Const

In C++ we also have the new "const" keyword, which says "this thing must not change". We can use this to declare global constants:

```
const int CURRENT_YEAR = 2018;
```

Global constants look a lot like global variables, but they are OK stylistically whereas regular global variables are not because the "const" keyword says that this value CANNOT CHANGE.

```
// This won't compile.
CURRENT_YEAR = 2038;
```

New / delete

In C:

```
int* arr = (int*) malloc(sizeof(int) * 100);
free(arr);
```

In C++, we have a nicer syntax for this that does the same thing:

```
int* arr = new int[100];
delete [] arr;
```

We can also do this for non-array types:

```
int* x = new int(4);        // x stores the value 4.
delete x;
```

New / delete

In C:

```
int
fre
```

In C++, w

```
int
del
```

We can a

```
int                                          ue 4.
delete x;
```

‘ new ‘ is an operator, not a function. The operator allocates memory, and then calls a constructor if appropriate.

- Can even initialize primitive data types
- Throws an exception if it fails (not does return NULL)
- Returns memory of the desired type, not an untype pointer
- Required size calculated by compiler, not calculated
- ‘ malloc ‘ does not all a constructor

Arrays

- Create a heap-allocated array of objects: `new A[10];`
  - Calls default (zero-argument) constructor for each element
  - Convenient if there's a good default initialization
- Create heap-allocated array of pointers to objects: `new A*[10];`
  - More like Java (but not initialized?)
- As in C, `new A()` and `new A[10]` have type `A*`
- `new A*` and `new A*[10]` both have type `A**`
- Unlike C, to delete a non-array, you must write `delete e`
- Unlike C, to delete an array, you must write `delete [] e`

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])

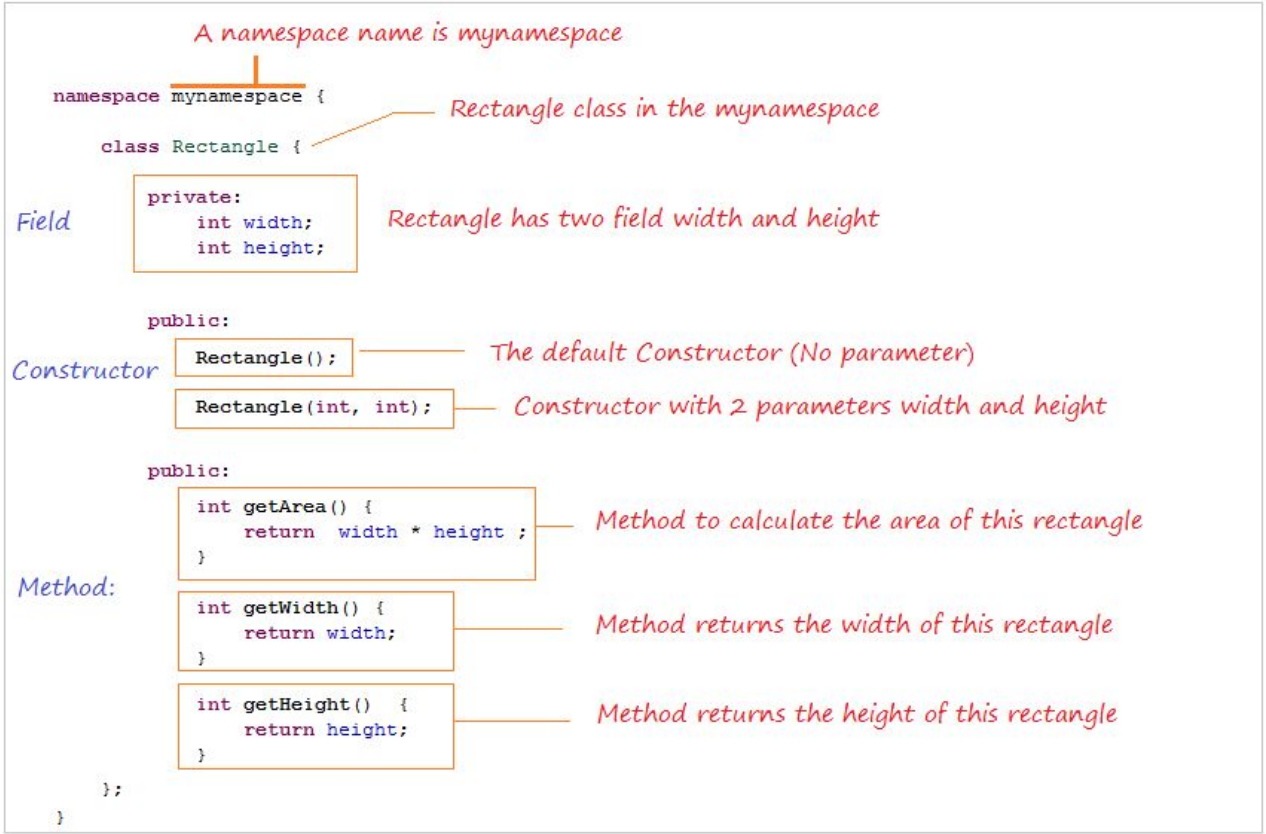
C++ classes: object-oriented

```
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.