C++ Class Details, Heap CSE 333 Spring 2020

Instructor: Hal Perkins

Teaching Assistants:

Ramya Challa Greg Guo Travis McGaha Cosmo Wang Haoran Yu

Mengqui Chen Zachary Keyes Arjun Singh Yifan Xu Velocity Yu John Depaszthory CJ Lin Guramrit Singh Robin Yang

Administrivia

- Yet another exercise released today, due Wed.
 - Rework exercise 10 but with dynamic memory this time
 - Fine to use ex10 solution as a starting point for ex11
- …Homework 2 due Thursday night
 - File system crawler, indexer, and search engine

(no exercise due Friday.... 🙂)

We're starting week 5!

- Congratulations on sticking with it and staying focused
 - And yes, it's harder than usual please speak up if we can help
- So we're reaching midterm season
 - But no traditional midterm this quarter (already announced)
- So what do we want to do about other tests/quizzes?
 - Answer for this quarter based on our situation: none
 - We'll have the usual collection of exercises and projects and will focus on those

Lecture Outline

- ***** Class Details
 - Filling in some gaps from last time
- Using the Heap
 - new/delete/delete[]

Rule of Three

- If you define any of:
 - 1) Destructor
 - 2) Copy Constructor
 - 3) Assignment (operator=)
- Then you should normally define all three
 - Can explicitly ask for default synthesized versions (C++11):

Dealing with the instanity

- C++ style guide tip:
 - If possible, disable the copy constructor and assignment operator.
 C++11 has direct syntax to indicate this:

Point_2011.h

```
class Point {
  public:
    Point(const int x, const int y) : x_(x), y_(y) { } // ctor
    ...
    Point(const Point& copyme) = delete; // declare cctor and "=" as
    Point& operator=(const Point& rhs) = delete; // as deleted (C++11)
    private:
    ...
  }; // class Point
Point w; // compiler error (no default constructor)
Point x(1, 2); // OK!
Point y = w; // compiler error (no copy constructor)
  y = x; // compiler error (no assignment operator)
```

If you're dealing with old code...

 In pre-C++11 code the copy constructor and assignment were often disabled by making them private and not implementing them...

```
Point.h
```

```
class Point {
  public:
    Point(const int x, const int y) : x_(x), y_(y) { } // ctor
    ...
  private:
    Point(const Point& copyme); // disable cctor (no def.)
    Point& operator=(const Point& rhs); // disable "=" (no def.)
    ...
  }; // class Point
Point w; // compiler error (no default constructor)
Point x(1, 2); // OK!
Point y = w; // compiler error (no copy constructor)
  y = x; // compiler error (no assignment operator)
```

struct vs. class

- In C, a struct can only contain data fields
 - Has no methods and all fields are always accessible
- * In C++, struct and class are (nearly) the same!
 - Both define a new type (the struct or class name)
 - Both can have methods and member visibility (public/private/protected)
 - Only real (minor) difference: members are default public in a struct and default private in a class
- Common style/usage convention:
 - Use struct for simple bundles of data
 - Use class for abstractions with data + functions

Access Control

* Access modifiers for members:

- public: accessible to all parts of the program
- private: accessible to the member functions of the class
 - Private to *class*, not object instances
- protected: accessible to member functions of the class and any *derived* classes (subclasses – more to come, later)

Reminders:

- Access modifiers apply to *all* members that follow until another access modifier is reached
- If no access modifier is specified, struct members default to public and class members default to private

Nonmember Functions

- "Nonmember functions" are just normal functions that happen to use some class
 - Called like a regular function instead of as a member of a class object instance
 - This gets a little weird when we talk about operators...
 - These do not have access to the class' private members
- Useful nonmember functions often included as part of interface to a class
 - Declaration goes in header file, but *outside* of class definition
 - Super useful for class-related things like stream I/O (operator<<, etc.), overloaded operators (operator+, etc.), ...

Review: Operator Overloading

- Can overload operators using member functions
 - Restriction: left-hand side argument must be a class you are implementing

Complex& operator+=(const Complex &a) { ... }

- Can overload operators using nonmember functions
 - No restriction on arguments (can specify any two)
 - Our only option when the left-hand side is a class you do not have control over, like ostream or istream.
 - But no access to private data members

Complex operator+(const Complex &a, const Complex &b) { ... }

friend Nonmember Functions

A class can give a nonmember function (or class) access to its nonpublic members by declaring it as a friend within its definition

L13: C++ Heap

- friend function is not a class member, but has access privileges as if it were
- friend functions are usually unnecessary if your class includes appropriate "getter" public functions

Complex.h

```
class Complex {
    ...
    friend std::istream& operator>>(std::istream& in, Complex& a);
```

```
}; // class Complex
```

```
std::istream& operator>>(std::istream& in, Complex& a) {
```

```
• •
```

UNIVERSITY of WASHINGTON

Complex.cc ₁₃

Namespaces

- Each namespace is a separate scope
 - Useful for avoiding symbol collisions
- Namespace definition:

```
namespace name {
   // declarations go here
}
```

- Creates a new namespace name if it did not exist, otherwise adds to the existing namespace (!)
 - This means that components (classes, functions, etc.) of a namespace can be defined in multiple source files

Classes vs. Namespaces

- They seems somewhat similar, but classes are not namespaces:
 - There are no instances/objects of a namespace; a namespace is just a group of logically-related things (classes, functions, etc.)
 - To access a member of a namespace, you must use the fully qualified name (i.e. nsp_name::member)
 - Unless you are using that namespace
 - You only used the fully qualified name of a class member when you are defining it outside of the scope of the class definition

Lecture Outline

- Class Details
 - Filling in some gaps from last time
- **Solution** Using the Heap
 - new/delete/delete[]

C++11 nullptr

- C and C++ have long used NULL as a pointer value that references nothing
- C++11 introduced a new literal for this: nullptr
 - New reserved word
 - Interchangeable with NULL for all practical purposes, but it has type T* for any/every T, and is not an integer value
 - Avoids funny edge cases (see C++ references for details)
 - Still can convert to/from integer 0 for tests, assignment, etc.
 - Advice: prefer nullptr in C++11 code
 - Though NULL will also be around for a long, long time

new/delete

- To allocate on the heap using C++, you use the new keyword instead of malloc() from stdlib.h
 - You can use new to allocate an object (e.g. new Point)
 - Will execute appropriate constructor as part of object allocate/create
 - You can use new to allocate a primitive type (e.g. new int)
- * To deallocate a heap-allocated object or primitive, use the delete keyword instead of free() from stdlib.h
 - Don't mix and match!
 - <u>Never</u> free() something allocated with new
 - <u>Never</u> delete something allocated with malloc()
 - Careful if you're using a legacy C code library or module in C++

new/delete Example

```
int* AllocateInt(int x) {
    int* heapy_int = new int;
    *heapy_int = x;
    return heapy int;
```

```
Point* AllocatePoint(int x, int y) {
   Point* heapy_pt = new Point(x,y);
   return heapy_pt;
```

heappoint.cc

```
#include "Point.h"
using namespace std;
... // definitions of AllocateInt() and AllocatePoint()
int main() {
    Point* x = AllocatePoint(1, 2);
    int* y = AllocateInt(3);
    cout << "x's x_ coord: " << x->get_x() << endl;
    cout << "y: " << y << ", *y: " << *y << endl;
    delete x;
    delete x;
    delete y;
    return 0;
}</pre>
```

Dynamically Allocated Arrays

- To dynamically allocate an array:
 - Default initialize:

type* name = new type[size];

- To dynamically deallocate an array:
 - Use delete[] name;
 - It is an incorrect to use "delete name;" on an array
 - The compiler probably won't catch this, though (!) because it can't always tell if name* was allocated with new type[size]; or new type;
 - Especially inside a function where a pointer parameter could point to a single item or an array and there's no way to tell which!
 - Result of wrong delete is undefined behavior

Arrays Example (primitive)

arrays.cc

```
#include "Point.h"
using namespace std;
int main() {
 int stack int;
 int* heap int = new int;
 int* heap init int = new int(12);
 int stack arr[10];
 int* heap arr = new int[10];
  int* heap init arr = new int[10](); // uncommon usage
  int* heap init error = new int[10](12); // bad syntax
  . . .
  delete heap int; //
  delete heap init int; //
                        //
 delete heap arr;
  delete[] heap_init_arr; //
 return 0;
```

Arrays Example (class objects)

arrays.cc

```
#include "Point.h"
using namespace std;
int main() {
  . . .
  Point stack point(1, 2);
  Point* heap point = new Point(1, 2);
 Point* err pt arr = new Point[10]; // no Point() ctor
  Point* err2 pt arr = new Point[10](1,2); // bad syntax
  . . .
  delete heap point;
  . . .
  return 0;
```

malloc vs. new

	malloc()	new
What is it?	a function	an operator or keyword
How often used (in C)?	often	never
How often used (in C++)?	rarely	often
Allocated memory for	anything	arrays, structs, objects, primitives
Returns	a void* (should be cast)	appropriate pointer type (<i>doesn't need a cast</i>)
When out of memory	returns NULL	throws an exception
Deallocating	free()	delete or delete[]

Heap Member Example

- Let's build a class to simulate some of the functionality of the C++ string
 - Internal representation: c-string to hold characters
- What might we want to implement in the class?

Str Class Walkthrough

Str.h

```
#include <iostream>
using namespace std;
class Str {
public:
          // default ctor
 Str();
  Str(const char* s); // c-string ctor
  Str(const Str& s); // copy ctor
          // dtor
 ~Str();
 int length() const; // return length of string
  char* c str() const; // return a copy of st
 void append(const Str& s);
  Str& operator=(const Str& s); // string assignment
  friend std::ostream& operator<<(std::ostream& out, const Str& s);</pre>
private:
  char* st ; // c-string on heap (terminated by '\0')
}; // class Str
```

Str Example Walkthrough

See: Str.h Str.cc strtest.cc

- * Look carefully at assignment operator=
 - self-assignment test is especially important here

Extra Exercise #1

- Write a C++ function that:
 - Uses new to dynamically allocate an array of strings and uses delete[] to free it

L13: C++ Heap

- Uses new to dynamically allocate an array of pointers to strings
 - Assign each entry of the array to a string allocated using new
- Cleans up before exiting
 - Use delete to delete each allocated string
 - Uses delete[] to delete the string pointer array
 - (whew!)