C++ Encapsulation, Heap

CSE 333 Spring 2018

Instructor: Justin Hsia

Teaching Assistants:

Danny Allen
Kevin Bi
Michael Poulain
Waylon Huang
Dennis Shao
Jack Xu
Renshu Gu
Wei Lin
Eddie Huang
Matthew Neldam
Robby Marver

Administrivia

- Exercise 10 released today, due Monday
  - Write a substantive class in C++!
  - Refer to `Complex.h`/`Complex.cc`
- Homework 2 due next Thursday (4/26)
  - File system crawler, indexer, and search engine

Lecture Outline

- **Class Encapsulation**
- Using the Heap
  - `new` / `delete` / `delete[]`

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 the member functions of the class and any *derived* classes

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

Nonmember Functions

- "Nonmember functions" are just normal functions that happen to use our 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
  - Declaration goes in header file, but *outside* of class definition

`friend` Nonmember Functions

- A class can give a nonmember function (or class) access to its non`public` members by declaring it as a `friend` within its definition
  - Access modifiers do not apply; function is not a member
  - `friend` functions are unnecessary if your class includes "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) {
  ...
}
```

Complex.cc

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 namespaces can discontiguous
  - Definitions can appear outside of the namespace definition
    ```

Classes vs. Namespaces

- They look very similar, but classes are *not* namespaces:
  - There are no instances/objects of a namespace; a namespace is just a group of logically-related members
  - 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

Complex Example Walkthrough

See:

`Complex.h`

`Complex.cc`

`testcomplex.cc`

Lecture Outline

- Class Encapsulation
- **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`)
  - 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!
    - *Never* `free()` something allocated with `new`
    - *Never* `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 y;
  return 0;
}
```

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 tell if it was allocated with `new type[size];` or `new type;`
    - Results in 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]();
  int* heap_init_error = new int[10](12);

  ...

  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* error_point_arr = new Point[10];

  Point* error2_point_arr = new Point[10](1, 2);

  ...

  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 (*doesn't need a cast*)
When out of memory	returns `NULL`	throws an exception
Deallocating	`free()`	`delete` or `delete[]`

Dynamically Allocated Class Members

- What will happen when we invoke **bar()**?
  - Vote at http://PollEv.com/justinh
  - If there is an error, how would you fix it?

A. Bad dereference

B. Bad delete

C. Memory leak

D. "Works" fine

E. We're lost…

```
Foo::Foo(int val) { Init(val); }
Foo::~Foo() { delete foo_ptr_; }

void Foo::Init(int val) {
   foo_ptr_ = new int;
  *foo_ptr_ = val;
}

Foo& Foo::operator=(const Foo& rhs) {
  delete foo_ptr_;
  Init(*(rhs.foo_ptr_));
  return *this;
}

void bar() {
  Foo a(10);
  Foo b(20);
  a = a;
}
```

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:
  Str();                  // default ctor
  Str(const char* s);     // c-string ctor
  Str(const Str& s);      // copy ctor
  ~Str();                 // dtor

  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);

 private:
  char* st_;  // c-string on heap (terminated by '\0')
};  // class Str
```

Str::append

- Complete the append() member function:
  - `char* strcpy(char* dst, const char* src);`
  - `char* strcat(char* dst, const char* src);`

```
#include <cstring>
#include "Str.h"
// append contents of s to the end of this string
void Str::append(const Str& s) {




}
```

Extra Exercise #1

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