C++ Class Details, Heap

CSE 333 Fall 2023

Instructor: Chris Thachuk

Teaching Assistants:

Ann Baturytski
Yuquan Deng
Noa Ferman
James Froelich
Hannah Jiang
Yegor Kuznetsov
Humza Lala
Alan Li
Leanna Mi Nguyen
Chanh Truong
Jennifer Xu

Relevant Course Information

- Exercise 6 due Wednesday
- Exercise 7 out tomorrow (not due this week)
  - Will build on Exercise 6 and use what a lot of is discussed today
- Homework 2 due next Monday (10/30)
  - **Hw2 partner declaration due this Thursday (10/26)**
  - File system crawler, indexer, and search engine
  - Don't forget to clone your repo to double-/triple-/quadruple-check compilation!
  - Don't modify the header files!
- Midterm this Friday in class (10/27)
  - A single 3"x5" index card with handwritten notes is allowed.

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):

```
class Point {
 public:
  Point() = default;                                  // the default ctor
  ~Point() = default;                                 // the default dtor
  Point(const Point& copyme) = default;               // the default cctor
  Point& operator=(const Point& rhs) = default; // the default "="
  ...
```

Dealing with the Insanity (C++11)

- C++ style guide tip:
  - Disabling the copy constructor and assignment operator can avoid confusion from implicit invocation and excessive copying

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 "="
  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)
```

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 (maybe through getters)

- Useful nonmember functions often included as part of interface to a class
  - Declaration goes in header file, but *outside* of class definition

Member

named function:
```
double Point::Distance(Point&);
pt1.Distance(pt2);
```

operator:
```
float Vector::operator*(Vector&);
vec1 * vec2;
```

can't tell!

Non-member

named function:
```
double Distance(Point&, Point&);
Distance(pt1, pt2);
```

operator:
```
float operator*(Vector&, Vector&);
vec1 * vec2;
```

`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
  - 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
```

declaration only

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

definition outside of class

Complex.cc

When to use Nonmember and `friend`

There is more to C++ object design that we don't have time to get to; these are good rules of thumb, but be sure to think about your class carefully!

- Member functions:
  - Operators that modify the object being called on
    - Assignment operator (`operator=`)
  - "Core" non-operator functionality that is part of the class interface
- Nonmember functions:
  - Used for commutative operators
    - *e.g.*, so `v1 + v2` is invoked as `operator+(v1, v2)` instead of `v1.operator+(v2)`
  - If operating on two types and the class is on the right-hand side
    - *e.g.*, `cin >> complex;`
  - Returning a "new" object, not modifying an existing one
  - Only grant `friend` permission if you NEED to

Poll Everywhere

pollev.com/cse333

doesn't modify objects, commutative

If we wanted to overload `operator==` to compare two `Point` objects, what type of function should it be?

no need for friend

- Reminder that `Point` has getters and a setter

A. non-friend + member

B. ~~friend + member~~ this is not a thing, as member functions can always access non-public data members

C. non-friend + non-member

D. friend + non-member

E. I'm lost...

Namespaces

Same name, but different namespace

ll::Iterator
ht::Iterator

- Each namespace is a separate scope
  - Useful for avoiding symbol collisions!

lowercase

- Namespace definition:
  - ```
    namespace name {
    // declarations go here
    } // namespace name
    ```

Namespace doesn't add indentation to contents

Comment to remind that this is end of namespace

no ;

  - Doesn't end with a semi-colon and doesn't add to the indentation of its contents
  - Creates a new namespace name if it did not exist, otherwise *adds to the existing namespace* (**!**)
    - This means that components (*e.g.*, classes, functions) 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

Complex Example Walkthrough

See:

`Complex.h`

`Complex.cc`

`testcomplex.cc`