Poll Everywhere

pollev.com/cse333

About how long did Exercise 5 take you?

A. [0, 2) hours
B. [2, 4) hours
C. [4, 6) hours
D. [6, 8) hours
E. 8+ Hours
F. I didn’t submit / I prefer not to say

C++ Constructor Insanity

CSE 333 Summer 2023

Instructor: Timmy Yang

Teaching Assistants:

Jennifer Xu Leanna Nguyen Pedro Amarante

Sara Deutscher Tanmay Shah

Relevant Course Information

- Exercise 6 released today, next Monday (7/17)
  - Write a substantive class in C++ (uses a lot of what we will talk about in lecture today)

- Homework 2 due next Thursday (7/20)
  - File system crawler, indexer, and search engine
  - Note: `libhw1.a` (yours or ours) and the `.h` files from hw1 need to be in right directory (`~yourgit/hw1/`)
  - Note: use Ctrl-D to exit `searchshell`
  - Tip: test on directory of small self-made files

- Quiz 1 closes at 11:59 pm tonight (7/12)

`struct` vs. `class`

STYLE TIP

- In C, a `struct` can only contain data fields
  - No methods and all fields are always accessible

- In C++, `struct` and `class` are (nearly) the same!
  - Both can have methods and member visibility (public/private/protected)
  - Minor difference: members are default *public* in a `struct` and default *private* in a `class`

- Common style convention:
  - Use `struct` for simple bundles of data
  - Use `class` for abstractions with data + functions

Memory Diagrams for Objects

- An **object** is an instance of a class that maintains its *state* independent from other objects
  - This state is the collection of its data members
  - Conceptually, an object acts like a collection of data fields (plus class metadata)
    - Layout is *not* specified or guaranteed, unlike structs in C

- Drawn out as variables within variables:

```
class Point {
 ...

 private:
  int x_;   // data member
  int y_;   // data member
};  // class Point
```

Lecture Outline

- **Constructors**
- Copy Constructors
- Assignment
- Destructors
- Extra Details

Constructors

- A constructor (ctor) initializes a newly-instantiated object
  - A class can have multiple constructors that differ in parameters
  - A constructor *must* be invoked when creating a new instance of an object – which one depends on *how* the object is instantiated

- Written with the class name as the method name:

```
Point(const int x, const int y);
```

  - C++ will automatically create a synthesized default constructor if you have *no* user-defined constructors
    - Takes no arguments and calls the default ctor on all non-"plain old data" (non-POD) member variables
    - Synthesized default ctor will fail if you have non-initialized const or reference data members

Synthesized Default Constructor Example

```
class SimplePoint {
 public:
  // no constructors declared!
  int get_x() const { return x_; }     // inline member function
  int get_y() const { return y_; }     // inline member function
  double Distance(const SimplePoint& p) const;
  void SetLocation(int x, int y);

 private:
  int x_;  // data member
  int y_;  // data member
};  // class SimplePoint
```

SimplePoint.h

SimplePoint.cc

```
#include "SimplePoint.h"

... // definitions for Distance() and SetLocation()

int main(int argc, char** argv) {
  SimplePoint x;  // invokes synthesized default constructor
  return EXIT_SUCCESS;
}
```

Synthesized Default Constructor

- If you define *any* constructors, C++ assumes you have defined all the ones you intend to be available and will *not* add any others

```
#include "SimplePoint.h"

// defining a constructor with two arguments
SimplePoint::SimplePoint(const int x, const int y) {
  x_ = x;
  y_ = y;
}

void Foo() {
  SimplePoint x;        // compiler error:  if you define any
                        // ctors, C++ will NOT synthesize a
                        // default constructor for you.

  SimplePoint y(1, 2);  // works:  invokes the 2-int-arguments
                        // constructor
}
```

Multiple Constructors (overloading)

```
#include "SimplePoint.h"

// default constructor
SimplePoint::SimplePoint() {
  x_ = 0;
  y_ = 0;
}

// constructor with two arguments
SimplePoint::SimplePoint(const int x, const int y) {
  x_ = x;
  y_ = y;
}

void Foo() {
  SimplePoint x;        // invokes the default constructor
  SimplePoint y(1, 2);  // invokes the 2-int-arguments ctor
  SimplePoint a[3];     // invokes the default ctor 3 times
}
```

Initialization Lists

- C++ lets you *optionally* declare an initialization list as part of a constructor definition
  - Initializes fields according to parameters in the list
  - The following two are (nearly) identical:

```
Point::Point(const int x, const int y) {
  x_ = x;
  y_ = y;
  std::cout << "Point constructed: (" << x_ << ",";
  std::cout << y_<< ")" << std::endl;
}
```

```
// constructor with an initialization list
Point::Point(const int x, const int y) : x_(x), y_(y) {
  std::cout << "Point constructed: (" << x_ << ",";
  std::cout << y_<< ")" << std::endl;
}
```

Initialization vs. Construction

STYLE TIP

```
class Point3D {
 public:
  // constructor with 3 int arguments
  Point3D(const int x, const int y, const int z) : y_(y), x_(x) {
    z_ = z;
  }

 private:
  int x_, y_, z_;   // data members
};  // class Point3D
```

First, initialization list is applied.

Next, constructor body is executed.

- Data members in initializer list are initialized in the order they are defined in the class, not by the initialization list ordering (!)
  - Data members that don't appear in the initialization list are *default initialized/constructed* before body is executed

- Initialization preferred to assignment to avoid extra steps
  - Real code should never mix the two styles

Lecture Outline

- Constructors
- **Copy Constructors**
- Assignment
- Destructors
- Extra Details

Copy Constructors

STYLE TIP

- C++ has the notion of a copy constructor (cctor)
  - Used to create a new object as a copy of an existing object

```
Point::Point(const int x, const int y) : x_(x), y_(y) { }

// copy constructor
Point::Point(const Point& copyme) {
  x_ = copyme.x_;
  y_ = copyme.y_;
}

void Foo() {
  Point x(1, 2);  // invokes the 2-int-arguments constructor

  Point y(x);     // invokes the copy constructor
                  // could also be written as "Point y = x;"
}
```

  - Initializer lists can also be used in copy constructors (preferred)

Synthesized Copy Constructor

- If you don’t define your own copy constructor, C++ will synthesize one for you
  - It will do a *shallow* copy of all of the fields (*i.e.*, member variables) of your class
  - Sometimes the right thing; sometimes the wrong thing

```
#include "SimplePoint.h"

... // definitions for Distance() and SetLocation()

int main(int argc, char** argv) {
  SimplePoint x;
  SimplePoint y(x);  // invokes synthesized copy constructor
  ...
  return EXIT_SUCCESS;
}
```

When Do Copies Happen?

- The copy constructor is invoked if:
  - You *initialize* an object from another object of the same type:

```
Point x;          // default ctor
Point y(x);       // copy ctor
Point z = y;      // copy ctor
```

  - You pass a non-reference object as a value parameter to a function:

```
void Foo(Point x) { ... }

Point y;          // default ctor
Foo(y);           // copy ctor
```

  - You return a non-reference object value from a function:

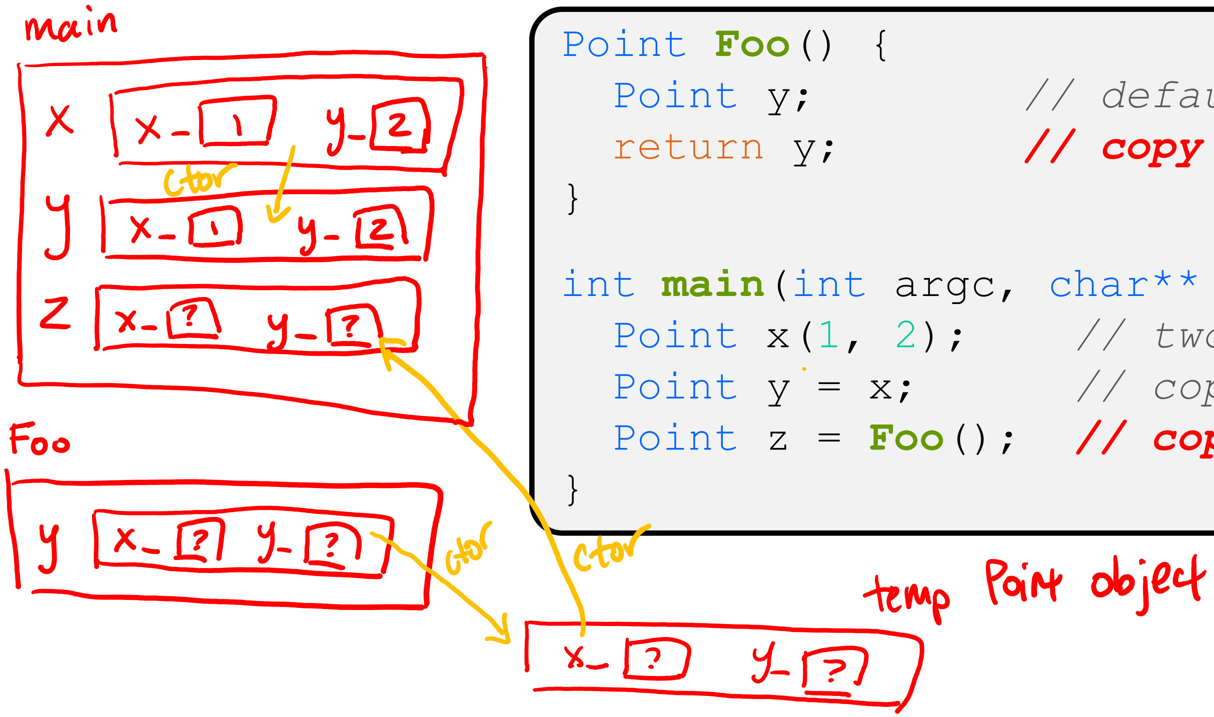
```
Point Foo() {
  Point y;        // default ctor
  return y;       // copy ctor
}
```

Compiler Optimization

- The compiler sometimes uses a "return by value optimization" or "move semantics" to eliminate unnecessary copies
  - Sometimes you might not see a constructor get invoked when you might expect it

```
Point Foo() {
  Point y;              // default ctor
  return y;             // copy ctor? optimized?
}

int main(int argc, char** argv) {
  Point x(1, 2);        // two-ints-argument ctor
  Point y = x;          // copy ctor
  Point z = Foo();      // copy ctor? optimized?
}
```



Lecture Outline

- Constructors
- Copy Constructors
- **Assignment**
- Destructors
- Extra Details

Assignment != Construction

- "=" is the assignment operator
  - Assigns values to an *existing, already constructed* object

```
Point w;            // default ctor
Point x(1, 2);      // two-ints-argument ctor
Point y(x);         // copy ctor
Point z = w;        // copy ctor
y = x;              // assignment operator
```

Overloading the "=" Operator

STYLE TIP

- You can choose to define the "=" operator
  - But there are some rules you should follow:

```
Point& Point::operator=(const Point& rhs) {
  if (this != &rhs) {  // (1) always check against this
    x_ = rhs.x_;
    y_ = rhs.y_;
  }
  return *this;        // (2) always return *this from op=
}

Point a;       // default constructor
a = b = c;     // works because = return *this
a = (b = c);   // equiv. to above (= is right-associative)
(a = b) = c;   // "works" because = returns a non-const
```

Synthesized Assignment Operator

- If you don’t define the assignment operator, C++ will synthesize one for you
  - It will do a *shallow* copy of all of the fields (*i.e.*, member variables) of your class
  - Sometimes the right thing; sometimes the wrong thing

```
#include "SimplePoint.h"

... // definitions for Distance() and SetLocation()

int main(int argc, char** argv) {
  SimplePoint x;
  SimplePoint y(x);
  y = x;             // invokes synthesized assignment operator
  return EXIT_SUCCESS;
}
```

Lecture Outline

- Constructors
- Copy Constructors
- Assignment
- **Destructors**
- Extra Details

Destructors

- C++ has the notion of a destructor (dtor)
  - Invoked automatically when a class instance is deleted, goes out of scope, etc. (even via exceptions or other causes!)
  - Place to put your cleanup code – free any dynamic storage or other resources owned by the object
  - Standard C++ idiom for managing dynamic resources
    - Slogan: "*Resource Acquisition Is Initialization*" (RAII)

```
Point::~Point() {    // destructor
  // do any cleanup needed when a Point object goes away
  // (nothing to do here since we have no dynamic resources)
}
```

Destructor Example

```
class FileDescriptor {
 public:
  FileDescriptor(char* file) {         // Constructor
    fd_ = open(file, O_RDONLY);
    // Error checking omitted
  }
  ~FileDescriptor() { close(fd_); }    // Destructor
  int get_fd() const { return fd_; }   // inline member function
 private:
  int fd_;  // data member
};  // class FileDescriptor
```

FileDescriptor.h

```
#include "FileDescriptor.h"

int main(int argc, char** argv) {
  FileDescriptor fd("foo.txt");
  return EXIT_SUCCESS;
}
```

Lecture Outline

- Constructors
- Copy Constructors
- Assignment
- Destructors
- **Extra Details**

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 "=" 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)
```

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

`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
```

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

Complex.cc

When to use Nonmember and `friend`

STYLE TIP

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

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

  - 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`

Preview for Next Lecture

```
class FileDescriptor {
 public:
  FileDescriptor(char* file) {            // Constructor
    fd_ = open(file, O_RDONLY);
    // Error checking omitted
  }
  ~FileDescriptor() { close(fd_); }       // Destructor
  int get_fd() const { return fd_; }     // inline member function
 private:
  int fd_;  // data member
};  // class FileDescriptor
```

FileDescriptor.h

```
#include "FileDescriptor.h"

int main(int argc, char** argv) {
  FileDescriptor fd1(foo.txt);
  FileDescriptor fd2(fd);      // Invokes synthesized cctor
  return EXIT_SUCCESS;
}
```

What happens when we return and destruct our objects?

(This won't crash the program, but what if we were using heap allocation instead of file descriptors?)

Extra Exercise #1

- Write a C++ program that:
  - Has a class representing a 3-dimensional point
  - Has the following methods:
    - Return the inner product of two 3D points
    - Return the distance between two 3D points
    - Accessors and mutators for the `x`, `y`, and `z` coordinates

Extra Exercise #2

- Write a C++ program that:
  - Has a class representing a 3-dimensional box
    - Use your Extra Exercise #1 class to store the coordinates of the vertices that define the box
    - Assume the box has right-angles only and its faces are parallel to the axes, so you only need 2 vertices to define it
  - Has the following methods:
    - Test if one box is inside another box
    - Return the volume of a box
    - Handles <<, =, and a copy constructor
    - Uses `const` in all the right places

Extra Exercise #3

- Modify your Point3D class from Extra Exercise #1
  - Disable the copy constructor and assignment operator
  - Attempt to use copy & assignment in code and see what error the compiler generates
  - Write a `CopyFrom()` member function and try using it instead
    - (See details about `CopyFrom()` in next lecture)

Extra Exercise #4

- Write a C++ class that:
  - Is given the name of a file as a constructor argument
  - Has a `GetNextWord()` method that returns the next whitespace- or newline-separated word from the file as a copy of a `string` object, or an empty string once you hit EOF
  - Has a destructor that cleans up anything that needs cleaning up