Poll Everywhere

pollev.com/naomila

Vibe Check: How are you feeling about C++ right now?

Keep this poll open and change your answer as the lecture goes on!

CSE333 Systems Programming

C++ Classes, Constructors, and Copies (oh my!!)

Instructors:

Naomi Alterman

Teaching Assistants:

Ann Baturytski	Barbora Buzkova	Mendel Carroll
Camden Harris	Hannah Hempstead	Rishabh Jain
Irene Lau	Jonathan Nister	Advay Patil
Aviel Wood	Angela Wu	Jennifer Xu

Administriva ⚡ ⍰

- Exercise 6 out today, due next Wednesday (4/29)
  - Playing with C++ classes
- Homework 2 due next Thursday (4/30)
  - File system crawler, indexer, and search engine
  - Start now!!
- Midterm exam in just over a week (5/4)
  - Located in this room during class time
  - Midterm review session next week, time to be announced
  - Practice midterms and solutions on the course website

`struct` vs. `class`

- 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`

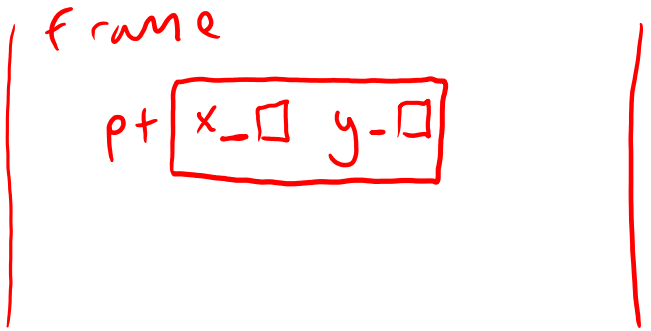
- Common style convention:
  - Use `struct` for simple bundles of data
  - Use `class` for abstractions with data + functions

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 in block-and-arrow diagrams as variables within variables:

```
class Point {
 ...

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



Lecture Outline

- **Constructors**
- Copy Constructors
- Assignment
- Destructors

Constructors

Point() {...} ← Default ctor

- A constructor (ctor) is a method initializes a newly-instantiated object
  - Written as a method with the same name as the class type:

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

2 int ctor

  - 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

- 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

```
#include "SimplePoint.h"

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

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

SimplePoint.cc

Synthesized Default Constructor

- If you define *any* constructors, C++ will *not* add a synthesized default constructor

```
#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)

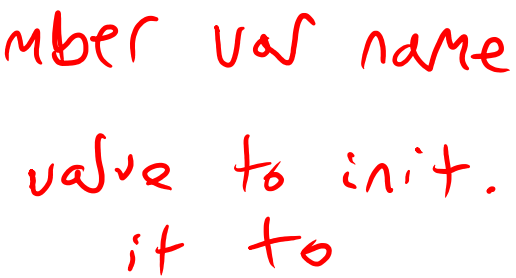
```
#include "SimplePoint.h"

// 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
}
```

5 SimplePoints on stack

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;
}
```

member var name

value to init. it to

Initialization vs. Construction

technically bad style ☹

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.

implied ③ z_()

order within list

- 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

Copy Constructors

- 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 a(1, 2);  // invokes the 2-int-arguments constructor

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

  - 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
  - Note this is separate from non-copy constructors
  - 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

x_ = other.x_
y_ = other.y_

```
#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: "Copy Elision"

- The compiler may eliminate unnecessary copies
  - You might not see a constructor get invoked when you expect it
- Most common is when an object is returned by value (*i.e.*, copied) and passed into *another* copy constructor
  - Since C++17, this kind of copy elision is **guaranteed**

```
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?
}
```

Unnamed Return value optimization

URVO

Lecture Outline

- Constructors
- Copy Constructors
- **Assignment**
- Destructors

Assignment isn't 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
                  // equiv. y.operator=(x);
```

Overloading the "=" Operator

const Point &rhs = us ;
Point *this = us ;

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

ref. to a const point

```
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**

Destructors

- C++ has the notion of a destructor (dtor)
  - A method whose name is the class name prefixed with a tilde
  - Invoked automatically when a class instance is deleted or goes out of scope (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**)
  - After destructor body finishes, destruct members in reverse order of declaration (*i.e.*, reverse of initialization list)

```
Point::~Point() {   // destructor
  // Do any cleanup needed when a Point object goes away.
  // Nothing to do here, but what if we had 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;
}
```

Class Definition (from last lecture)

Point.h

```
#ifndef POINT_H_
#define POINT_H_

class Point {
 public:
  Point(int x, int y);                    // constructor
  int get_x() const { return x_; }        // inline member function
  int get_y() const { return y_; }        // inline member function
  double Distance(const Point& p) const;      // member function
  void SetLocation(int x, int y);             // member function

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

#endif  // POINT_H_
```

Poll Everywhere

pollev.com/naomila

How many times does Point's *destructor* run?

- Assume `Point` with everything defined (ctor, cctor, =, dtor)
- Assume no compiler optimizations

A. 1
B. 2
C. 3
D. 4
E. We're lost…

test.cc

```
Point PrintRad(Point& pt) {
  Point origin(0, 0);
  double r = origin.Distance(pt);
  double theta = atan2(pt.get_y(), pt.get_x());
  cout << "r = " << r << endl;
  cout << "theta = " << theta << " rad" << endl;
  return pt;
}

int main(int argc, char** argv) {
  Point pt(3, 4);
  PrintRad(pt);
  return EXIT_SUCCESS;
}
```

Wait…what?

test.cc

```
Point PrintRad(Point& pt) {
  Point origin(0, 0);
  double r = origin.Distance(pt);
  double theta = atan2(pt.get_y(), pt.get_x());
  cout << "r = " << r << endl;
  cout << "theta = " << theta << " rad" << endl;
  return pt;
}

int main(int argc, char** argv) {
  Point pt(3, 4);
  PrintRad(pt);
  return EXIT_SUCCESS;
}
```

ctor	cctor	op=	dtor

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