CSE 374 Programming Concepts and Tools

Lecture 18 - C++ Classes

Exam 2

Next week

Monday: C++ Class Details

Wednesday: C++ Templates, STL

Today

Intro to C++ review

const in C++

C++ Classes: Basics

- Header files, organization between files

Constructors (ctor)

Copy Constructors (cctor)

Assignment

- `op=`

Destructors (dtor)

Review: Hello World in C++

helloworld.cc

```
#include <iostream>
#include <cstdlib>

int main(int argc, char** argv) {
  std::cout << "Hello, World!" << std::endl;
  return EXIT_SUCCESS;
}
```

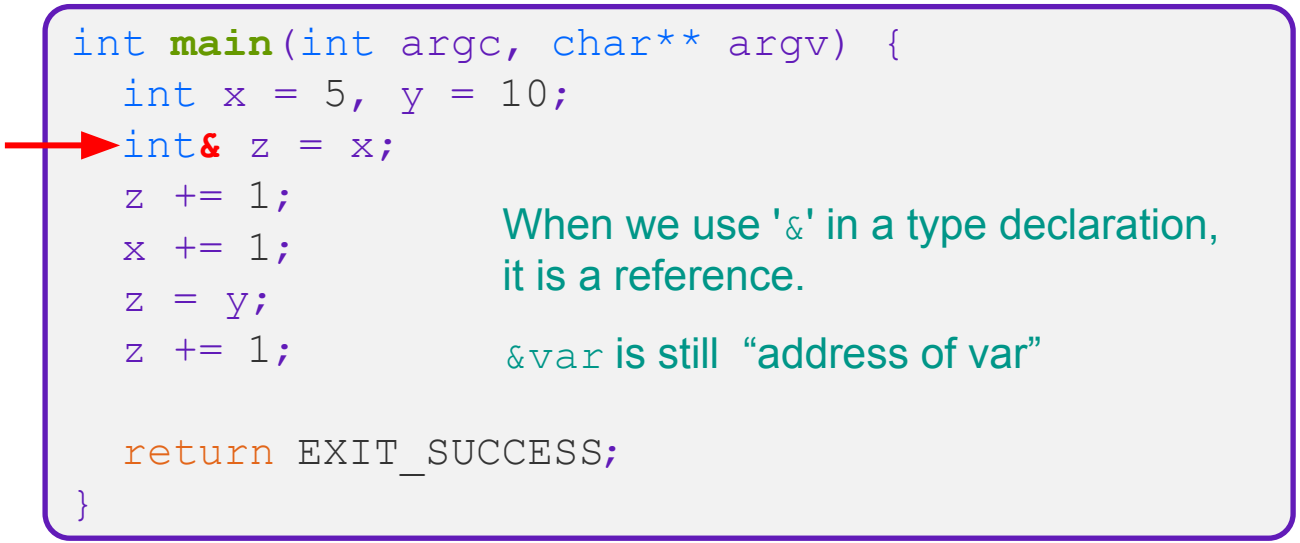
C++ makes it easy to hide a significant amount of complexity

- It's powerful, but really dangerous

Review: References

A **reference** is an alias for another variable

- *Alias*: another name that is bound to the aliased variable
  - Mutating a reference ***is*** mutating the aliased variable



Review: Pass-By-Reference

Real pass-by-*reference*: client passes in an argument with normal syntax, function uses reference parameters with normal syntax

- Modifying a reference parameter modifies the caller's argument!

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main(int argc, char** argv) {
  int a = 5, b = 10;
  swap(a, b);
  cout << "a: " << a << "; b: " << b << endl;
  return EXIT_SUCCESS;
}
```

Parameters are attached to variables provided by caller

Pointer, Reference

A pointer holds the address of another object. Because references are not objects, they don't have addresses. Hence, we **may not define a pointer to a reference**.

However, because a pointer is an object, we **can define a reference to a pointer**:

```
int i = 42;

int* p;          // p is a pointer to int

int*& r = p;  // r is a reference to the pointer p
```

The easiest way to understand the type of `r` is to read the definition right to left.

- i.e., `int*& = (int*)&`

`const` in C++

const

`const`: this cannot be changed/mutated

- Used *much* more in C++ than in C
- Signal of intent to compiler (compile-time errors); meaningless at hardware level

```
void BrokenPrintSquare(const int& i) {
  i = i*i;  // compiler error here!
  std::cout << i << std::endl;
}

int main(int argc, char** argv) {
  int j = 2;
  BrokenPrintSquare(j);
  return EXIT_SUCCESS;
}
```

const can be a useful tool for defensive programming

brokenpassbyrefconst.cc

`const` and Pointers

Pointers can change data in two different contexts:

1. You can change the *value* of the pointer (i.e., the address it contains)
2. You can change the *thing the pointer points to* (via dereference)

`const` can be used to prevent either/both of these behaviors!

- `const` *next to pointer name* means you *can't change the value of the pointer*
  - **`int* const ptr1;`** // cannot change the value of `ptr1`
- `const` *next to data type* pointed to means you *can't* use this pointer to change the thing being pointed to
  - **`const int* ptr2`** // cannot change the value of `*ptr2`
- Tip: read variable declaration from *right-to-left*

Example

The syntax with pointers is confusing:

```
int main(int argc, char** argv) {
  int x = 5;                  // int
  const int y = 6;            // (const int)
  y++;

  const int* z = &y;          // pointer to a (const int)
  *z += 1;
  z++;

  int* const w = &x;          // (const pointer) to a (variable int)
  *w += 1;
  w++;

  const int* const v = &x; // (const pointer) to a (const int)
  *v += 1;
  v++;

  return EXIT_SUCCESS;
}
```

Example

The syntax with pointers is confusing:

```
int main(int argc, char** argv) {
  int x = 5;                    // int
  const int y = 6;              // (const int)
  y++;                          // compiler error

  const int* z = &y;            // pointer to a (const int)
  *z += 1;                      // compiler error
  z++;                          // ok

  int* const w = &x;            // (const pointer) to a (variable int)
  *w += 1;                      // ok
  w++;                          // compiler error

  const int* const v = &x;      // (const pointer) to a (const int)
  *v += 1;                      // compiler error
  v++;                          // compiler error

  return EXIT_SUCCESS;
}
```

`const` and Parameters: What can we pass?

Suppose we have the functions `foo` and `bar`, and a variable `x` in scope

- If `x` is just an `int*`, can we pass `x` to `foo`? What about `bar`?
- If `x` is a `const int*`, can we pass `x` to `foo`? What about `bar`?

Food for thought: The compiler checks arguments to make sure they match the type of the parameters. How do we know that the function will respect x's type?

```
void foo(const int* y) {
  std::cout << *y << std::endl;
}

void bar(int* y) {
  std::cout << *y << std::endl;
}
```

`const` Parameters

A `const` parameter *cannot* be mutated inside the function

- Therefore it does not matter if the argument can be mutated or not-
  - Both const int* and int* are fine as args!

A non-`const` parameter *may* be mutated inside the function

- It would be BAD if you passed it a `const` variable
- Compiler won't let you pass in const variables as args

```
void foo(const int* y) {
  std::cout << *y << std::endl;
}

void bar(int* y) {
  std::cout << *y << std::endl;
}

int main(int argc, char** argv) {
  const int a = 10;
  int b = 20;

  foo(&a);    // OK
  foo(&b);    // OK
  bar(&a);    // not OK - error
  bar(&b);    // OK

  return EXIT_SUCCESS;
}
```

When to Use References in Parameters?

Google C++ style guide suggests (not mandated by the C++ language):

- Input parameters:
  - Either use values (for primitive types like `int` or small structs/objects)
  - Or use `const` references (for complex struct/object instances)
- Output parameters:
  - Use `const` pointers: unchangeable pointers referencing changeable data
- Ordering:
  - List input parameters first, then output parameters last

```
void CalcArea(const int& width, const int& height,
              int* const area) {
  *area = width * height;
}
```

styleguide.cc

C++ Classes

A brief look at classes in general…

Classes are a popular way to introduce new types of objects

- Found in Java, Python, Ruby, Javascript, C#, Smalltalk, Objective C, Swift, …

If an object has a particular class as its type, we know:

- what data it contains (i.e., fields), and
- what operations we can perform on it (i.e., methods)

Simula, a programming language, was developed in the 60s

- Introduced foundational concepts for OOP like classes and objects, inheritance, etc.
- Simula inspired C++!

C++ Class Example: Point

Point.h

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

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

Point.cc

```
#include <cmath>
#include "Point.h"

Point::Point(const int x, const int
y) {
  x_ = x;
  this->y_ = y;
}

double Point::Distance(const Point&
p) const {
  // implementation of distance…
}

void Point::SetLocation(const int x,
const int y) {
  x_ = x;
  y_ = y;
}
```

C++ Class Example: Point

Point.h

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

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

Point.cc

```
#include <cmath>
#include "Point.h"

Point::Point(const int x, const int
y) {
  x_ = x;
  this->y_ = y;
}

double Point::Distance(const Point&
p) const {
  // implementation of distance…
}

void Point::SetLocation(const int x,
const int y) {
  x_ = x;
  y_ = y;
}
```

Class Organization: Header Files

Class definition is part of *interface* and should go in `.h` file

- **Private** member declarations **still must be included in class definition** (**!**)
  - C++ compiler needs to know how much space to allocate

Since the header is an interface, implementation details should generally be kept out

BUT, you *can* put implementations in the header file

- This changes how member functions will be compiled
- If defined in the header file, those member functions are *inlined*
  - any calls will be replaced by the body of the function

C++ Class Example: Point

Point.h

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

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

Point.cc

```
#include <cmath>
#include "Point.h"

Point::Point(const int x, const int
y) {
  x_ = x;
  this->y_ = y;
}

double Point::Distance(const Point&
p) const {
  // implementation of distance…
}

void Point::SetLocation(const int x,
const int y) {
  x_ = x;
  y_ = y;
}
```

Class Organization: Companion .cc Files

Usually put member function definitions into companion `.cc` file with implementation details

- Common exception: setter and getter methods
  - These often go into the header files. Why?
- These files can also include non-member functions that use the class

Unlike Java, you can name files anything you want

- Good practice though to name the files sensibly
- Typically `ClassName.cc` and `ClassName.h` for `class ClassName`

Classes

Class definition syntax (in a `.h` file):

```
class ClassName {
 public:
  // public member declarations/definitions go here

 private:
  // private member declarations/definitions go here
};  // class ClassName
```

- Members can be functions (methods) or data (variables)

Class member function definition syntax (in a `.cc` file):

```
retType ClassName::MethodName(type1 param1, …, typeN
paramN) {
  // body statements
}
```

Class Definition (`.h` file)

Point.h

```
#ifndef POINT_H_
#define POINT_H_

class Point {
 public:
  Point(const int x, const 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(const int x, const int y); // member function

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

#endif  // POINT_H_
```

const means the “this” object we are calling on, can’t be changed

Inline definition ok for simple getters/setters

Google C++ naming conventions for data members

Class Member Definitions (. cc file)

Point.cc

```
#include <cmath>
#include "Point.h"

Point::Point(const int x, const int y) {
  x_ = x;
  this->y_ = y;  // "this->" is optional unless name conflicts
}

double Point::Distance(const Point& p) const {
  // We can access p's x_ and y_ variables either through the get_x(),
  // get_y() accessor functions or the x_, y_ private member variables
  // directly, since we're in a member function of the same class.
  double distance = (x_ - p.get_x()) * (x_ - p.get_x());
  distance += (y_ - p.y_) * (y_ - p.y_);
  return sqrt(distance);
}

void Point::SetLocation(const int x, const int y) {
  x_ = x;
  y_ = y;
}
```

This code uses bad style for demonstration purposes

Can't modify the "this" object inside the function

const won't affect caller, but good style

Can't be const. We have to mutate the Point.

Class Usage (. cc file)

usepoint.cc

```
#include <iostream>
#include "Point.h"

using namespace std;

int main(int argc, char** argv) {
  Point p1(1, 2);  // allocate a new Point on the Stack
  Point p2(4, 6);  // allocate a new Point on the Stack

  cout << "p1 is: (" << p1.get_x() << ", ";
  cout << p1.get_y() << ")" << endl;

  cout << "p2 is: (" << p2.get_x() << ", ";
  cout << p2.get_y() << ")" << endl;

  cout << "dist : " << p1.Distance(p2) << endl;
  return 0;
}
```

Calls constructor to define an object on the stack (no “new” keyword). More on this shortly.

Dot notation to call function (like Java)

`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!

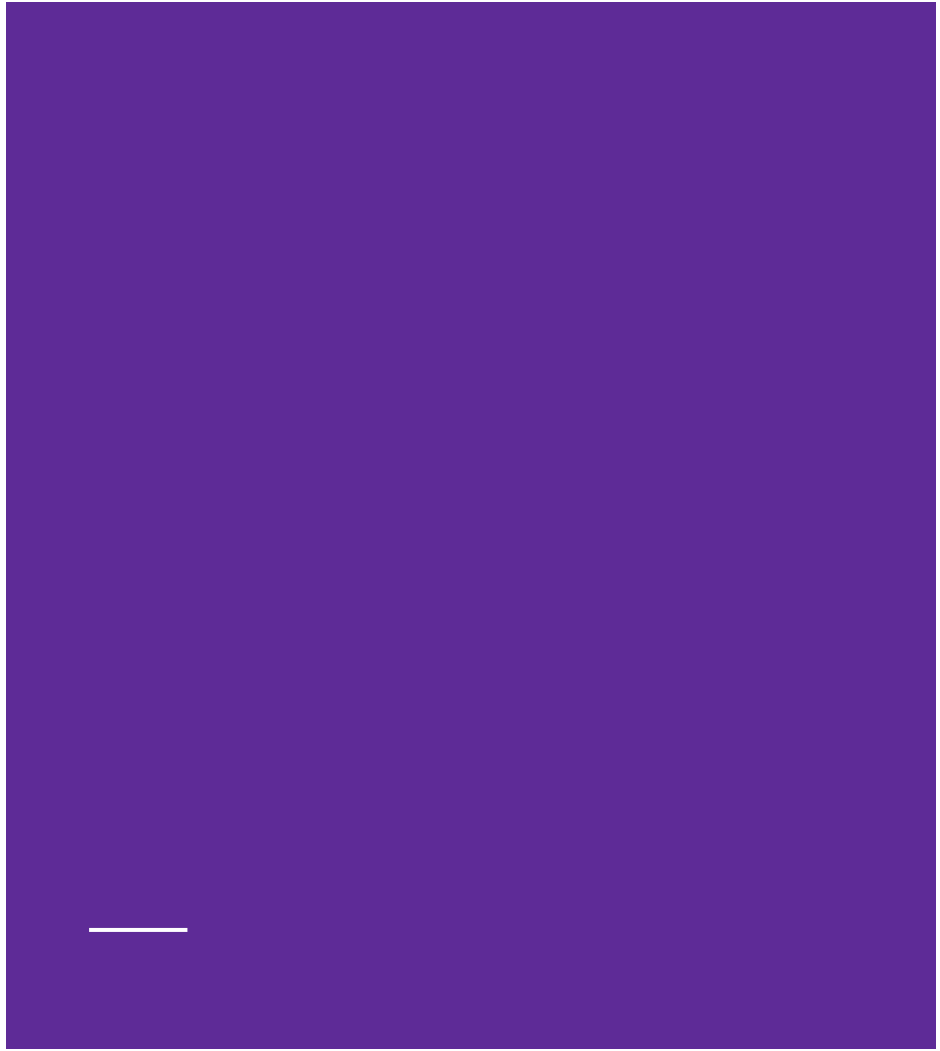
- Both define a new type (the `struct` or `class` name)
- Both can have methods and member visibility (public/private/protected)
- Only real difference: visibility
  - Struct members: default to *public*
  - Class members: default to *private*

Common style/usage convention:

- Use `struct` for simple bundles of data
- Use `class` for abstractions with data + functions

Constructors

ctor



Constructors

A constructor (ctor) initializes a newly-instantiated object

- A class can have multiple constructors that differ in parameters
  - Which one is invoked 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
  - You can explicitly specify to use the synthesized default constructor:

    ```
    Point() = default;
    ```

Synthesized Default Constructor

```
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++ assumes you have defined all the ones you intend to be available and will *not* add any others

```
#include "SimplePoint.h" // need to change to add ctor with args…

// 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" // need to change to add constructors…

// 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
-

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

Body can be empty

```
// 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

```
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
```

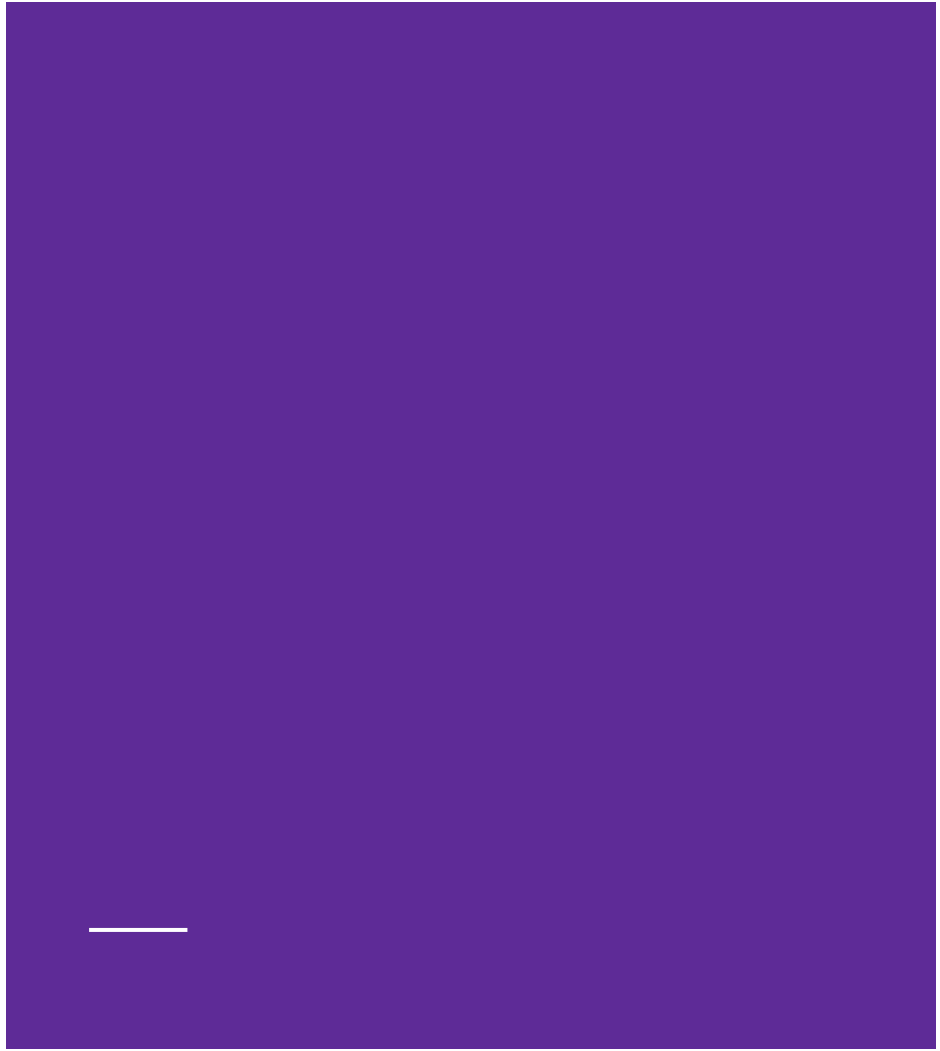
First, initialization list is applied.

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

Copy Constructors

cctor



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 x(1, 2);  // invokes the 2-int-arguments constructor
  Point y(x);     // invokes the copy constructor
  Point z = y;    // also invokes the copy constructor
}
```

Use a cctor since we are constructing based on x

Point z didn't exist before, a ctor must be called

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

Copy Constructors (w/ initialization list)

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

// copy constructor w/ initialization list
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
  Point z = y;    // also invokes the copy constructor
}
```

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
  - Does assignment for *primitives*; could be problematic with pointers
-

```
#include "SimplePoint.h"                    // In this example, synthesized cctor is fine

... // 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?
}
```

Compiler Optimization

```
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?
}
```

Compiler Optimization

`main` stack frame

x	{1, 2}

```
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?
}
```

Compiler Optimization

main	stack frame
x	{1, 2}
y	{1, 2}

```
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?
}
```

Compiler Optimization

`main` stack frame

x	{1, 2}
y	{1, 2}

`foo` stack frame

```
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?
}
```

Compiler Optimization

`main` stack frame

x	{1, 2}
y	{1, 2}

`foo` stack frame

y	{0, 0}

```
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?
}
```

Compiler Optimization

`main` stack frame

x	{1, 2}
y	{1, 2}

`foo` stack frame

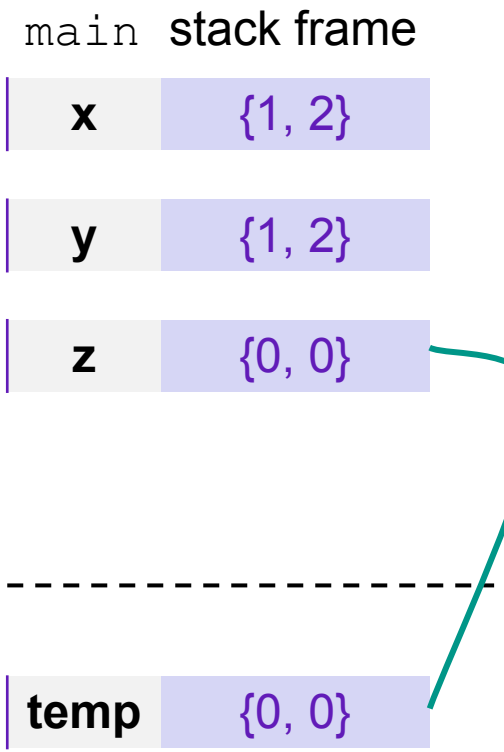
y	{0, 0}

?? Temp object??

temp	{0, 0}

```
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?
}
```

Compiler Optimization

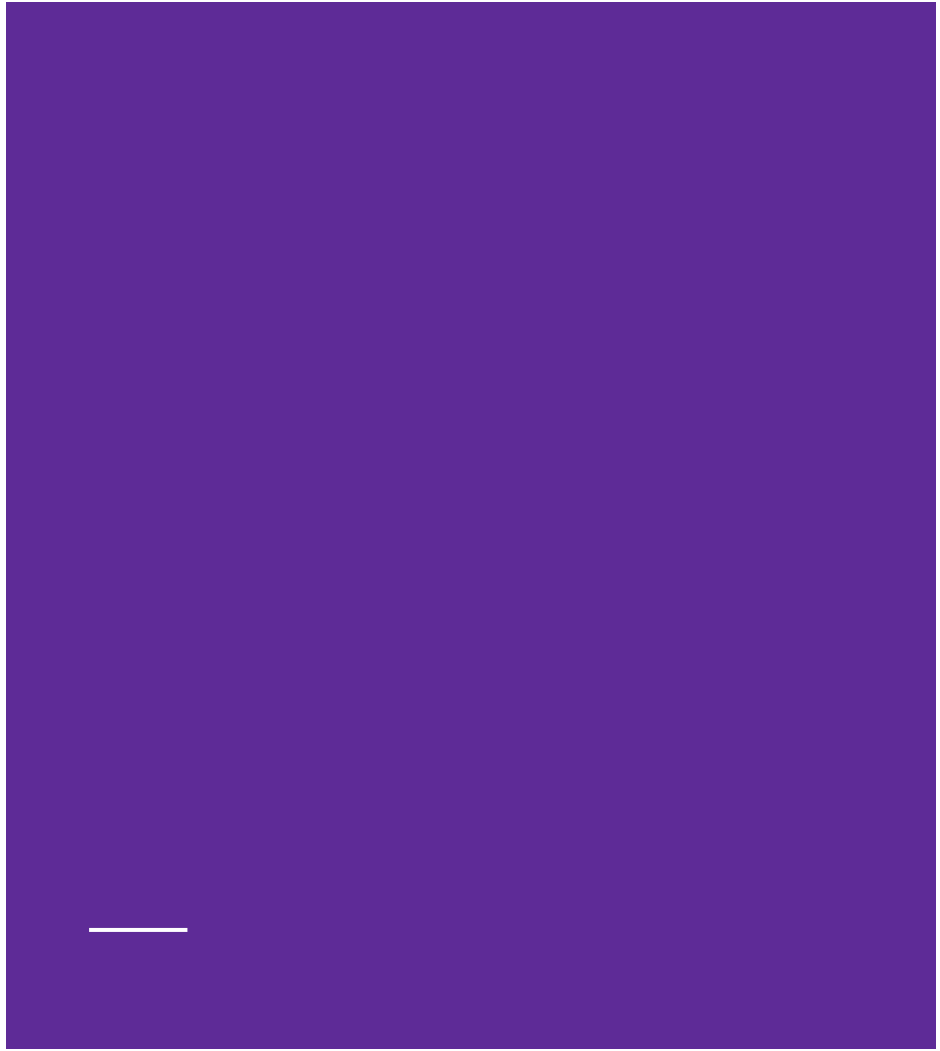



```
Point foo() {
  Point y;          // default ctor
  return y;         // copy ctor? optimized?
}

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?
}
```

Constructs the Point (**temp**) directly into the memory location of **z**, avoiding the cctor altogether!

Assignment

`operator=`



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

You can choose to define the “=” operator

But there are some rules you should follow:

1. Always check that the RHS ***is not*** **`this`**
2. Always return **`*this`** (which is returned as a reference) to allow chaining

Extra credit question (try answering this in today’s exercise):

What’s the reason behind rule #1? What’s so bad about the following?

```
Point& Point::operator=(const Point& rhs) {
  x_ = rhs.x_;
  y_ = rhs.y_;
  return *this;
}
```

Overloading the "=" Operator

A correct(ish) example:

```
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
                  // reference to *this
```

More important when data members are dynamic memory

Should be a reference to *this to allow chaining

Bad style, just for demo

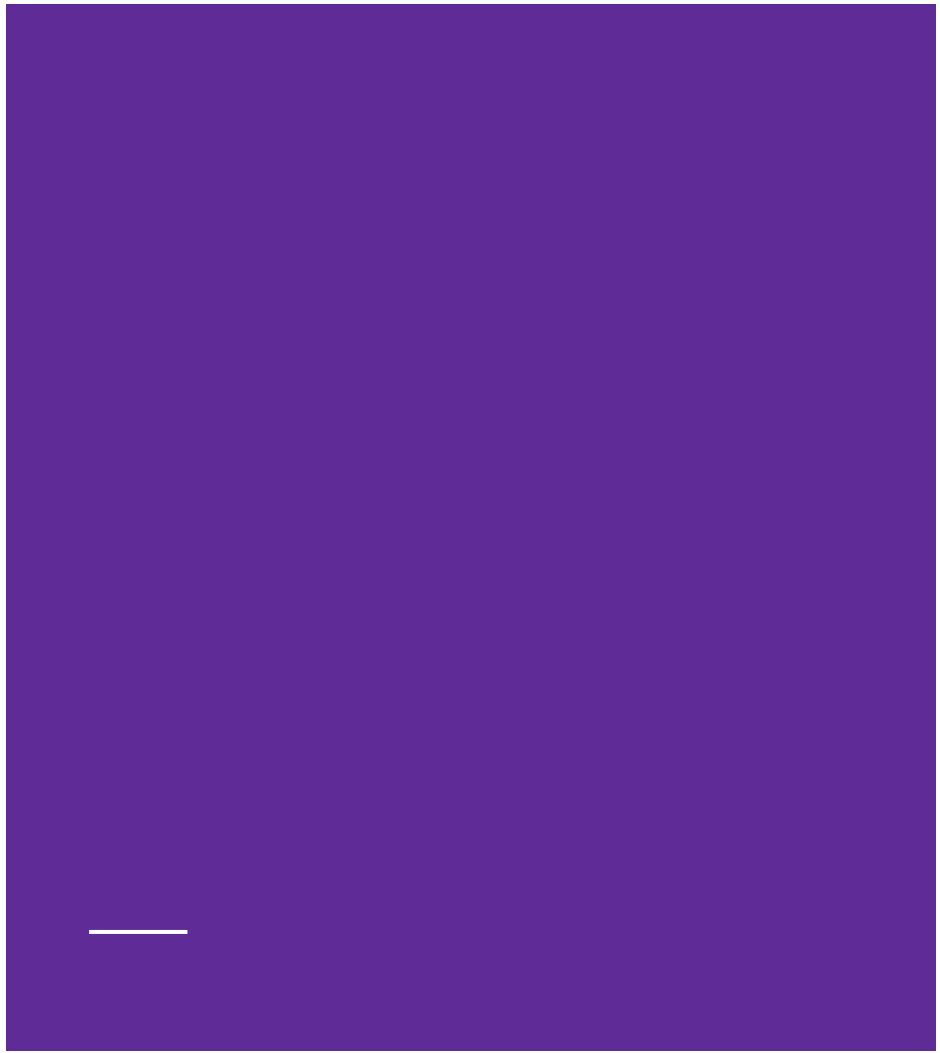
Synthesized Assignment Operator

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;
}
```

Destructors



Destructors

C++ has the notion of a destructor (dtor)

- Invoked automatically when a class instance is deleted (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
```

Without destructor, the file wouldn't be closed

Tilde indicates destructor

FileDescriptor.h

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

Destruct the object when it falls out of scope (when we return)

Polling Time!

Please answer in the LP18 assignment on Gradescope!

How many times does the ***destructor*** get invoked?

➔ Assume `Point` with everything defined (ctor, cctor, =, dtor)

➔ Assume *no* compiler optimizations and `origin.Distance` takes a reference.

- 1
- 2
- 3
- 4

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 0;
}
```

Exercise Explained

main

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 0;
}
```

ctor	cctor	op=	dtor
0	0	0	0

Exercise Explained

main

pt	{3, 4}

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 0;
}
```

ctor	cctor	op=	dtor
1	0	0	0

Exercise Explained

main

pt (main) pt(PrintRad)	{3, 4}

PrintRa

origin	{0, 0}

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 0;
}
```

ctor	cctor	op=	dtor
2	0	0	0

Exercise Explained

main

pt (main) pt(PrintRad)	{3, 4}

PrintRa

origin	{0, 0}

Point::Distance

// Takes a const ref,
just computation

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 0;
}
```

ctor	cctor	op=	dtor
2	0	0	0

Exercise Explained

test.cc

main

pt (main) pt(PrintRad)	{3, 4}

PrintRa

origin	{0, 0}

?? Temp object ??

temp	{3, 4}

```
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 0;
}
```

ctor	cctor	op=	dtor
2	1	0	0

Exercise Explained

main

pt (main) pt(PrintRad)	{3, 4}

PrintRa

origin	{0, 0}

?? Temp object ??

temp	{3, 4}

test.cc

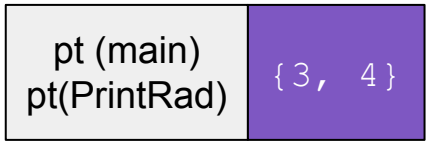
```
Point PrintRad(Point& pt) {
  Point origin(0, 0);
  double r = origin.Distance(pt);
  double theta = atan2(pt.get_y(), pt.get_x());
  cout << "r = " << r << endl;
  cout << "theta = " << theta << " rad" << endl;
  return pt;
}

int main(int argc, char** argv) {
  Point pt(3, 4);
  PrintRad(pt);
  return 0;
}
```

ctor	cctor	op=	dtor
2	1	0	1

Exercise Explained

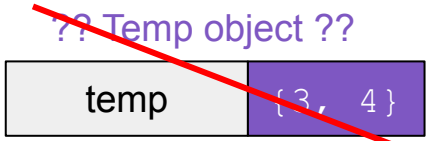
test.cc

main



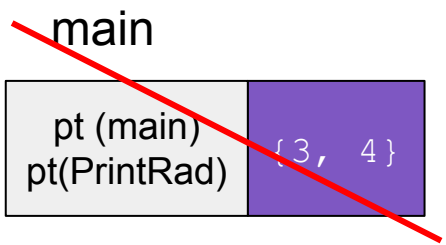

```
Point PrintRad(Point& pt) {
  Point origin(0, 0);
  double r = origin.Distance(pt);
  double theta = atan2(pt.get_y(), pt.get_x());
  cout << "r = " << r << endl;
  cout << "theta = " << theta << " rad" << endl;
  return pt;
}

int main(int argc, char** argv) {
  Point pt(3, 4);
  PrintRad(pt);
  return 0;
}
```

?? Temp object ??



ctor	cctor	op=	dtor
2	1	0	**2**

Exercise Explained



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 0;
}
```

ctor	cctor	op=	dtor
2	1	0	**3**

Assignment Reminders

EX16 due August 10

Exam 2 on Friday!

- Review session today @ 2:30
- Basement of CSE (006)

Extra Exercises 

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 #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