

C++ Constructor Insanity

CSE 333 Autumn 2021

Instructor: Chris Thachuk

Teaching Assistants:

Arpad (John) Depaszthory

Ian Hsiao

Logan Gnanapragasam

Mengqi (Frank) Chen

Angela Xu

Khang Vinh Phan

Maruchi Kim

Cosmo Wang

Administrivia

- ❖ Exercise 8 due tonight @ 10pm
- ❖ Exercise 9 out Friday, not due until Wed (Oct. 27)
- ❖ Turn around for exercises will typically be longer going forward
- ❖ Homework 2 due next Thursday (Oct. 28)

Lecture Outline

- ❖ **Makefiles (briefly)**
- ❖ Constructors
- ❖ Copy Constructors
- ❖ -----
- ❖ Assignment
- ❖ Destructors

make

- ❖ `make` is a classic program for controlling what gets (re)compiled and how
 - Many other such programs exist (*e.g.*, `ant`, `maven`, IDE “projects”)
- ❖ `make` has tons of fancy features, but only two basic ideas:
 - 1) Scripts for executing commands
 - 2) Dependencies for avoiding unnecessary work
- ❖ To avoid “just teaching `make` features” (boring and narrow), let’s focus more on the concepts...

Building Software



- ❖ Programmers spend a lot of time “building”
 - Creating programs from source code
 - Both programs that they write and other people write
- ❖ Programmers like to automate repetitive tasks
 - Repetitive: `gcc -Wall -g -std=c17 -o widget foo.c bar.c baz.c`

Building Software




- ❖ Programmers spend a lot of time “building”
 - Creating programs from source code
 - Both programs that they write and other people write
- ❖ Programmers like to automate repetitive tasks
 - Repetitive: `gcc -Wall -g -std=c17 -o widget foo.c bar.c baz.c`
 - Retype this every time:







Building Software

- ❖ Programmers spend a lot of time “building”
 - Creating programs from source code
 - Both programs that they write and other people write
- ❖ Programmers like to automate repetitive tasks
 - Repetitive: `gcc -Wall -g -std=c17 -o widget foo.c bar.c baz.c`
 - Retype this every time: 
 - Use up-arrow or history:  (still retype after logout)

Building Software

- ❖ Programmers spend a lot of time “building”
 - Creating programs from source code
 - Both programs that they write and other people write
- ❖ Programmers like to automate repetitive tasks
 - Repetitive: `gcc -Wall -g -std=c17 -o widget foo.c bar.c baz.c`
 - Retype this every time: 
 - Use up-arrow or history:  (still retype after logout)
 - Have an alias or bash script: 

Building Software

- ❖ Programmers spend a lot of time “building”
 - Creating programs from source code
 - Both programs that they write and other people write
- ❖ Programmers like to automate repetitive tasks
 - Repetitive: `gcc -Wall -g -std=c17 -o widget foo.c bar.c baz.c`
 - Retype this every time: 
 - Use up-arrow or history:  (still retype after logout)
 - Have an alias or bash script: 
 - Have a Makefile:  (you're ahead of us)

“Real” Build Process

- ❖ On larger projects, you can't or don't want to have one big (set of) command(s) that are all run every time you change anything. To do things “smarter,” consider:

“Real” Build Process

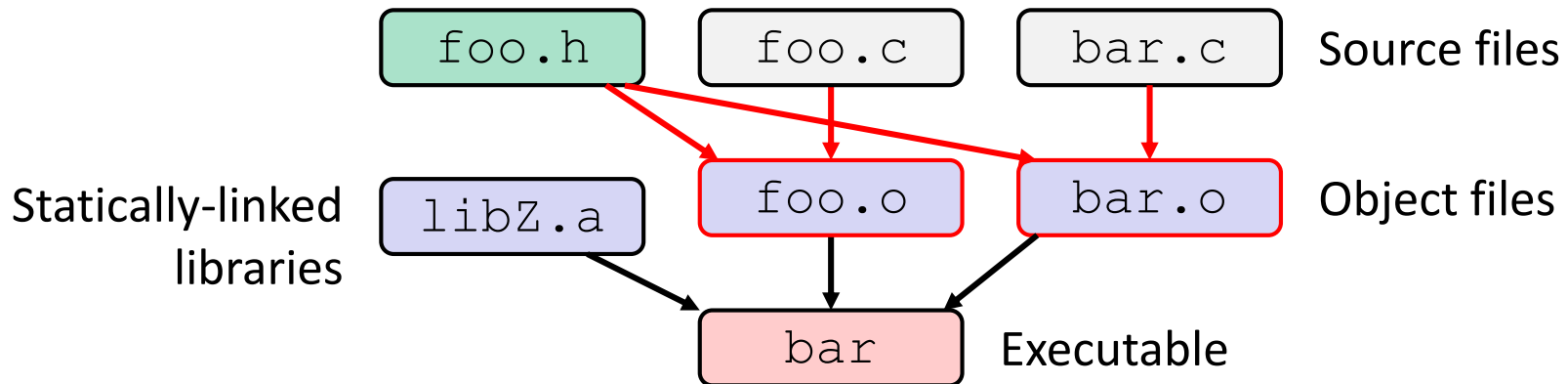
- ❖ On larger projects, you can't or don't want to have one big (set of) command(s) that are all run every time you change anything. To do things “smarter,” consider:
 - 1) It could be worse: If `gcc` didn't combine steps for you, you'd need to preprocess, compile, and link on your own (along with anything you used to generate the C files)
 - 2) Source files could have multiple outputs (*e.g.*, `javadoc`). You may have to type out the source file name(s) multiple times
 - 3) You don't want to have to document the build logic when you distribute source code; make it relatively simple for others to build
 - 4) You don't want to recompile everything every time you change something (especially if you have 10^5 - 10^7 files of source code)

“Real” Build Process

- ❖ On larger projects, you can't or don't want to have one big (set of) command(s) that are all run every time you change anything. To do things “smarter,” consider:
 - 1) It could be worse: If `gcc` didn't combine steps for you, you'd need to preprocess, compile, and link on your own (along with anything you used to generate the C files)
 - 2) Source files could have multiple outputs (*e.g.*, `javadoc`). You may have to type out the source file name(s) multiple times
 - 3) You don't want to have to document the build logic when you distribute source code; make it relatively simple for others to build
 - 4) You don't want to recompile everything every time you change something (especially if you have 10^5 - 10^7 files of source code)
- ❖ A script can handle 1-3 (use a variable for filenames for 2), but 4 is trickier

Theory Applied to C

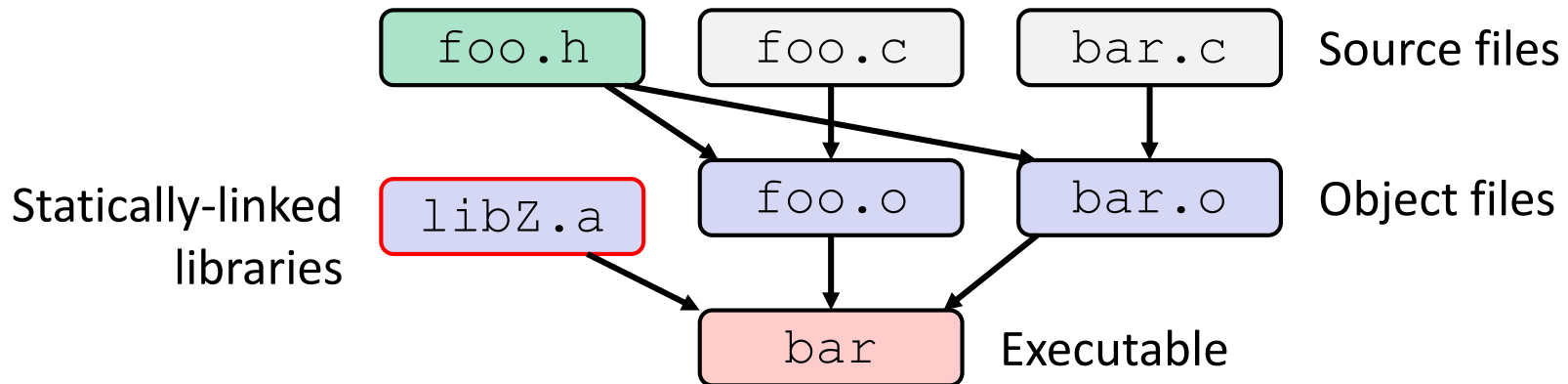
(s) source
(t) target



- ❖ Compiling a `.c` creates a `.o` – the `.o` depends on the `.c` and all included files (`.h`, recursively/transitively)

Theory Applied to C

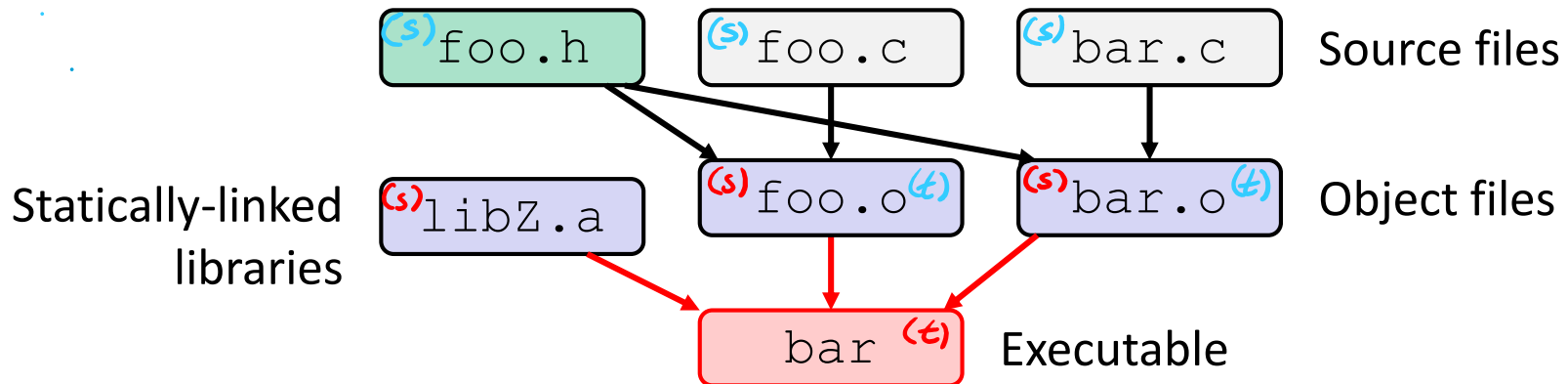
(s) source
(t) target



- ❖ Compiling a `.c` creates a `.o` – the `.o` depends on the `.c` and all included files (`.h`, recursively/transitively)
- ❖ An archive (library, `.a`) depends on included `.o` files

Theory Applied to C

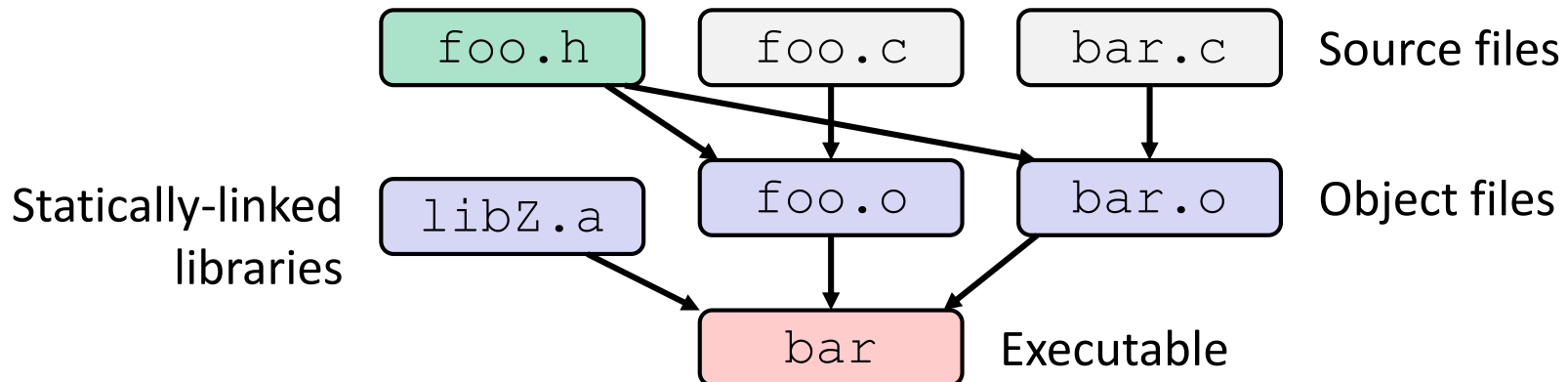
(s) = source
(t) = target



- ❖ Compiling a `.c` creates a `.o` – the `.o` depends on the `.c` and all included files (`.h`, recursively/transitively)
- ❖ An archive (library, `.a`) depends on included `.o` files
- ❖ Creating an executable (“linking”) depends on `.o` files and archives
 - Archives linked by `-L<path> -l<name>`
(*e.g.*, `-L. -lfoo` to get `libfoo.a` from current directory)

Theory Applied to C

(s) source
(t) target



- ❖ If one `.c` file changes, just need to recreate one `.o` file, maybe a library, and re-link
- ❖ If a `.h` file changes, may need to rebuild more
- ❖ Many more possibilities!

make Basics

- ❖ A makefile contains a bunch of **triples**:

```
target: sources
← Tab → command
```

- Colon after target is *required*
- Command lines must start with a **TAB**, NOT SPACES
- Multiple commands for same target are executed *in order*
 - Can split commands over multiple lines by ending lines with ‘\’

- ❖ Example:

```
foo.o: foo.c foo.h bar.h
      gcc -Wall -o foo.o -c foo.c
```

Using make

```
bash$ make -f <makefileName> target
```

❖ Defaults:

- If no `-f` specified, use a file named `Makefile` in current dir
- If no `target` specified, will use the first one in the file
- Will interpret commands in your default shell
 - Set `SHELL` variable in makefile to ensure

Using make

```
bash$ make -f <makefileName> target
```

❖ Defaults:

- If no `-f` specified, use a file named `Makefile` in current dir
- If no `target` specified, will use the first one in the file
- Will interpret commands in your default shell
 - Set `SHELL` variable in makefile to ensure

❖ Target execution:

- Check each source in the source list:
 - If the source is a target in the makefile, then process it recursively
 - If some source does not exist, then error
 - If any source is newer than the target (or target does not exist), run `command` (presumably to update the target)

“Phony” Targets

- ❖ A make target whose command does not create a file of the target’s name (*i.e.*, a “recipe”)
 - As long as target file doesn’t exist, the command(s) will be executed because the target must be “remade”

“Phony” Targets

- ❖ A make target whose command does not create a file of the target’s name (*i.e.*, a “recipe”)
 - As long as target file doesn’t exist, the command(s) will be executed because the target must be “remade”
- ❖ *e.g.*, target `clean` is a convention to remove generated files to “start over” from just the source

```
clean:  
    rm foo.o bar.o baz.o widget *~
```

“Phony” Targets

- ❖ A make target whose command does not create a file of the target’s name (*i.e.*, a “recipe”)
 - As long as target file doesn’t exist, the command(s) will be executed because the target must be “remade”
- ❖ *e.g.*, target `clean` is a convention to remove generated files to “start over” from just the source

```
clean:
```

```
    rm foo.o bar.o baz.o widget *~
```

- ❖ *e.g.*, target `all` is a convention to build all “final products” in the makefile
 - Lists all of the “final products” as sources

“all” Example

```
all: prog B.class someLib.a
      # notice no commands this time

prog: foo.o bar.o main.o
      gcc -o prog foo.o bar.o main.o

B.class: B.java
          javac B.java

someLib.a: foo.o baz.o
            ar r foo.o baz.o

foo.o: foo.c foo.h header1.h header2.h
          gcc -c -Wall foo.c

# similar targets for bar.o, main.o, baz.o, etc...
```

“a11” Example

```
all: prog B.class someLib.a
      # notice no commands this time

prog: foo.o bar.o main.o
      gcc -o prog foo.o bar.o main.o

B.class: B.java
          javac B.java

someLib.a: foo.o baz.o
            ar r foo.o baz.o

foo.o: foo.c foo.h header1.h header2.h
          gcc -c -Wall foo.c

# similar targets for bar.o, main.o, baz.o, etc...
```

“a11” Example

```
1all: prog B.class someLib.a
      2 # notice no commands this time

prog: foo.o bar.o main.o
        gcc -o prog foo.o bar.o main.o

B.class: B.java
           javac B.java

someLib.a: foo.o baz.o
             ar r foo.o baz.o

foo.o: foo.c foo.h header1.h header2.h
          gcc -c -Wall foo.c

# similar targets for bar.o, main.o, baz.o, etc...
```

“a11” Example

```
1all: prog B.class someLib.a
      2 # notice no commands this time
prog: foo.o bar.o main.o
      3 gcc -o prog foo.o bar.o main.o
B.class: B.java
           javac B.java
someLib.a: foo.o baz.o
            ar r foo.o baz.o
foo.o: foo.c foo.h header1.h header2.h
          gcc -c -Wall foo.c

# similar targets for bar.o, main.o, baz.o, etc...
```

“a11” Example

```
1 all: prog B.class someLib.a
  2 # notice no commands this time
prog: foo.o bar.o main.o
  3 gcc -o prog foo.o bar.o main.o
B.class: B.java
  javac B.java
someLib.a: foo.o baz.o
  ar r foo.o baz.o
foo.o: foo.c foo.h header1.h header2.h
  gcc -c -Wall foo.c

# similar targets for bar.o, main.o, baz.o, etc...
```

The diagram illustrates the dependencies between targets in the Makefile. Red arrows indicate the following relationships:

- all** depends on **prog**, **B.class**, and **someLib.a**.
- prog** depends on **foo.o**, **bar.o**, and **main.o**.
- foo.o** depends on **foo.c**, **foo.h**, **header1.h**, and **header2.h**.

“a11” Example

```
1 all: prog B.class someLib.a
2 # notice no commands this time
prog: foo.o bar.o main.o
3 gcc -o prog foo.o bar.o main.o
B.class: B.java
javac B.java
someLib.a: foo.o baz.o
ar r foo.o baz.o
foo.o: foo.c foo.h header1.h header2.h
gcc -c -Wall foo.c
# similar targets for bar.o, main.o, baz.o, etc...
```

“a11” Example

```
1 all: prog B.class someLib.a
2 # notice no commands this time
prog: foo.o bar.o main.o
3 gcc -o prog foo.o bar.o main.o
B.class: B.java
javac B.java
someLib.a: foo.o baz.o
ar r foo.o baz.o
foo.o: foo.c foo.h header1.h header2.h
gcc -c -Wall foo.c

# similar targets for bar.o, main.o, baz.o, etc...
```

“a11” Example

```
1 all: prog B.class someLib.a
2 # notice no commands this time
prog: foo.o bar.o main.o
3 gcc -o prog foo.o bar.o main.o
B.class: B.java
javac B.java
someLib.a: foo.o baz.o
4 ar r foo.o baz.o
foo.o: foo.c foo.h header1.h header2.h
5 gcc -c -Wall foo.c
6
7 # similar targets for bar.o, main.o, baz.o, etc...
```

“a11” Example

```
1 all: prog B.class someLib.a
2 # notice no commands this time
prog: foo.o bar.o main.o
3 gcc -o prog foo.o bar.o main.o
B.class: B.java
javac B.java
someLib.a: foo.o baz.o
4 ar r foo.o baz.o
5
6
foo.o: foo.c foo.h header1.h header2.h
7 gcc -c -Wall foo.c
8

# similar targets for bar.o, main.o, baz.o, etc...
```

Lecture Outline

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

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

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
```

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

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

```
#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"

// 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:

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;  
}
```

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

```
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
```



Initialization vs. Construction

First, initialization list is applied.

```
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
```



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.



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 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 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 (ctor)**
 - 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;"  
}
```



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  
                // 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:

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
```

When Do Copies Happen?

❖ The copy constructor is invoked if:

- You *initialize* an object from another object of the same type:
- You pass a non-reference object as a value parameter to a function:

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

```
void foo(Point x) { ... }  
  
Point y;           // default ctor  
foo(y);           // copy ctor
```

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?  
}
```