C++

why C++?

easier to talk about data representation

"closer to the hardware"

- directly allocate memory
- more obvious translation to assembly/machine code

heavily related to Java

C++ history

K&R C (first published 1972) Dennis Ritchie, Bell Labs
- based on BCPL (1967)
- meant to be easy to make efficient compilers for

C with classes (1979) Bjarne Stoustrup, Bell Labs
- efficiecy of C with features of other languages?

early C++ (1985) Bjarne Stroustrup, Bell Labs

ANSI/ISO standard C++ (1998)
- standardization effort started in 1989 (!)
- what current compilers try to implement
- still actively being updated

why not C++?

some not great syntax choices

made in 1980s, standardized in 1990s–2010s

based on C (1970s, standardized in 1980s)

makes compromises for compatibility

incompleteness

the C++ language has a lot of features

...and is still changing

we will teach a particular subset of it

C++ hello world

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello␣World!" << endl;
    return 0;
}
```

C++ hello world

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello World!" << endl;
    return 0;
}
```

outside of any class!
called a function

main

```
int main() { ... }
```

function *outside of any class*

must have return type of int

this class: always return `0` from main

C++ hello world

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello␣World!" << endl;
    return 0;
}
```

using directive

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello␣World!" << endl;
    return 0;
}
```

```
#include <iostream>
int main() {
    std::cout << "Hello␣World!" << std::endl;
    return 0;
}
```

using directive

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello World!" << endl;
    return 0;
}
```

```
#include <iostream>
int main() {
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

using directive

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello␣World!" << endl;
    return 0;
}
```

```
#include <iostream>
int main() {
    std::cout << "Hello␣World!" << std::endl;
    return 0;
}
```

using single things

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello␣World!" << endl;
    return 0;
}
```

```
#include <iostream>
using std::cout;
using std::endl;
int main() {
    cout << "Hello␣World!" << endl;
    return 0;
}
```

C++ hello world

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello␣World!" << endl;
    return 0;
}
```

instead of `import java...`

between Java files

Foo.java

```
public class Foo {
    ...
    Bar x = new Bar();
    ...
}
```

Bar.java

```
public class Bar {
    ...
}
```

declare before use

functions, classes must be
declared before they are used

compiler processes each file in order

compiler processes files seperately

declare before use

functions, classes must be
declared before they are used

compiler processes each file in order

compiler processes files seperately

declaration versus definition (1)

```
#include <iostream>
bool even(int number);
bool odd(int number) {
    return !even(number);
}
bool even(int number) {
    if (number == 0) {
        return true;
    } else {
        return odd(number - 1);
    }
}
```

declaration versus definition (1)

```
#include <iostream>
bool even(int number);
bool odd(int
    return !even(number);
}
bool even(int number) {
    if (number == 0) {
        return true;
    } else {
        return odd(number - 1);
    }
}
```

declaration — "function prototype"

declaration versus definition (1)

```
#include <iostream>
bool even(int number);
bool odd(int number)
    return !even(number);
}
bool even(int number) {
    if (number == 0) {
        return true;
    } else {
        return odd(number - 1);
    }
}
```

definition (and declaration)

declaration versus definition (2)

```
#include <iostream>
using namepace std;

int max(int a, int b);

int main(void) {
    int x=37, y=52;
    cout << max(x, y) << endl;
    return 0;
}

int max(int a, int b) {
    return (a > b) ? a : b;
}
```

declaration versus definition (2)

```
#include <iostream>
using namespace std;

int max(int a, int b);

int main(void) {
    int x=37, y=52;
    cout << max(x, y) << endl;
    return 0;
}

int max(int a, int b) {
    return (a > b) ? a : b;
}
```

declaration — "function prototype"

declaration versus definition (2)

```
#include <iostream>
using namepace std;

int max(int a, int b);

int main(void) {
    int x=37, y=52;
    cout << max(x, y)
    return 0;
}

int max(int a, int b) {
    return (a > b) ? a : b;
}
```

definition (and (re)declaration)

functions and prototypes

functions — methods not associated with class

function prototype or *forward declaration* —

```
return_type functionName(argType name,
                         argType name,
                         argType name, ...);
```

prototype or definition must appear before function can be used

declare before use

functions, classes must be
declared before they are used

compiler processes each file in order

compiler processes files seperately

declaration versus definition (3)

main.cpp

```
#include <iostream>
extern bool even(int number);
int main() {
  if (even(42)) {
    std::cout << "42␣is␣even"
          << std::endl;
  }
  return 0;
}
```

even.cpp

```
bool even(int number) {
    return number % 2 == 0;
}
```

C++: header files (1)

main.cpp

```
#include <iostream>
#include "even.h"
int main() {
  if (even(42)) {
    std::cout << "42␣is␣even"
              << std::endl;
  }
  return 0;
}
```

C++ compiler reads from even.h

even.h

```
...
extern bool even(int number);
...
```

even.cpp

```
bool even(int number) {
    return number % 2 == 0;
```

C++: header files (2)

main.cpp

```
#include <iostream>
using namespace std;
int main() {
  cout << "Hello,␣World!"
       << endl;
}
```

iostream (comes w/ compiler)

```
...
  class ostream {
    ...
  };

  extern ostream cout;
...
```

C++ compiler reads from `iostream`

header files

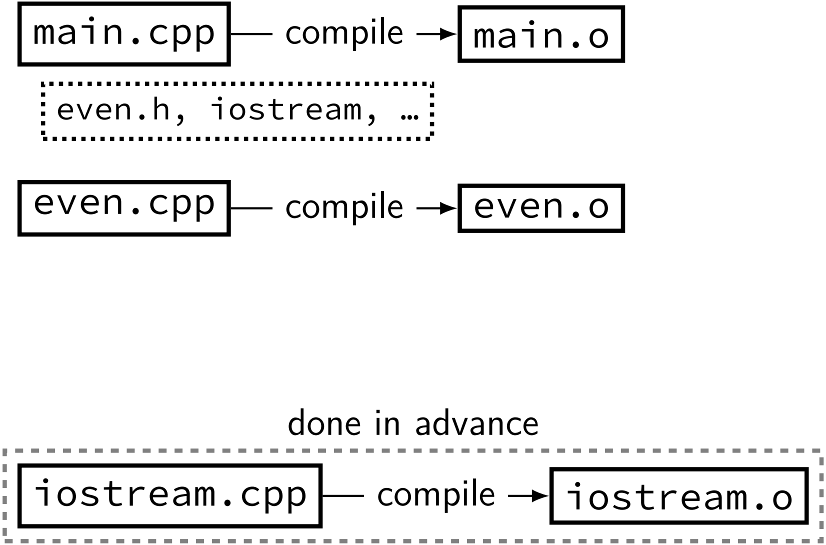
header files contain declarations

(mostly)

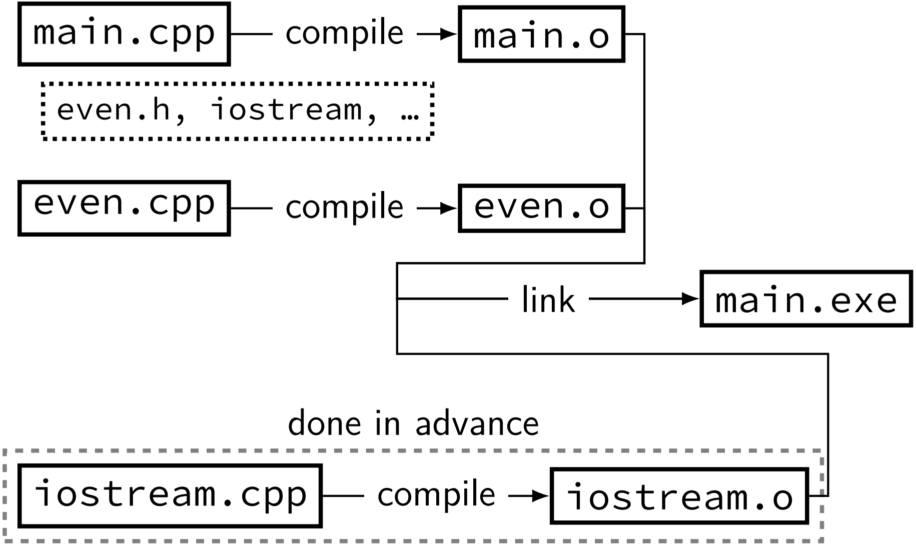
alternative to placing prototypes, etc. in every file

convention: every `.cpp` file has a `.h` file

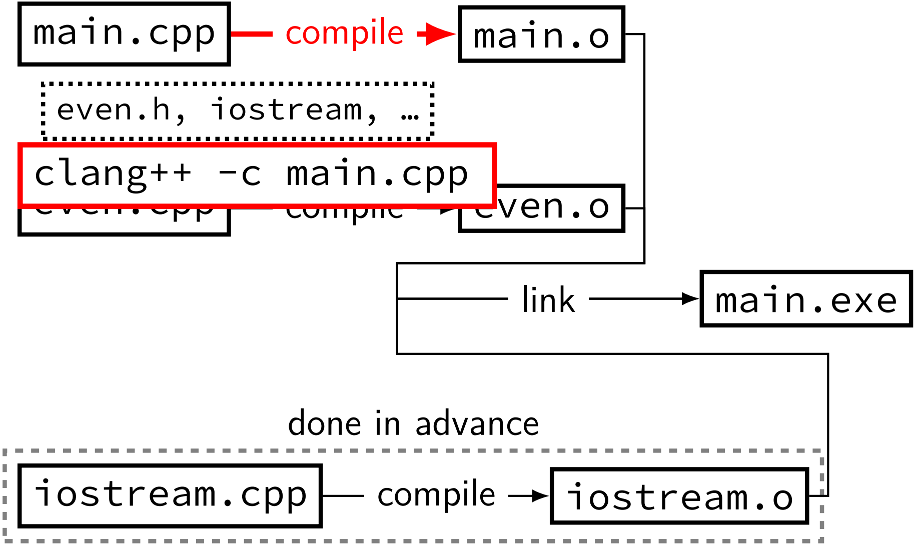
seperate compilation



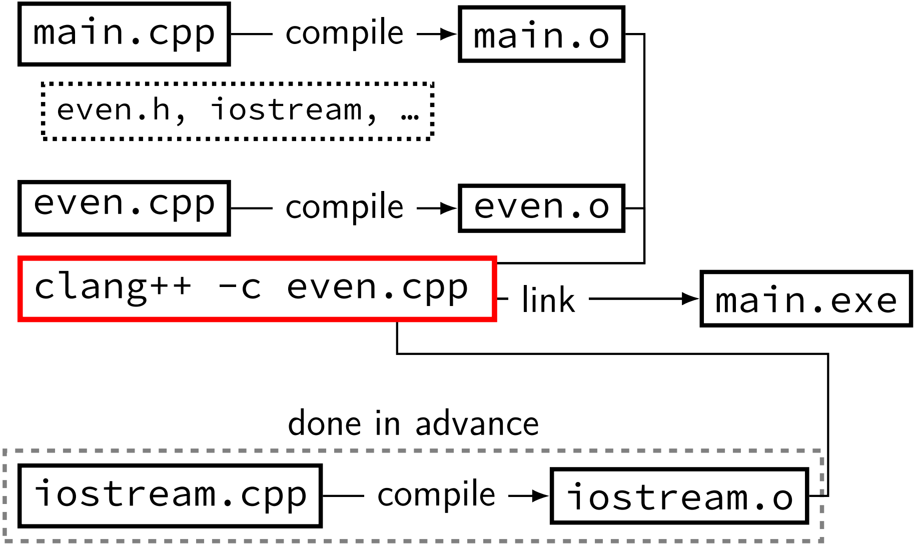
seperate compilation



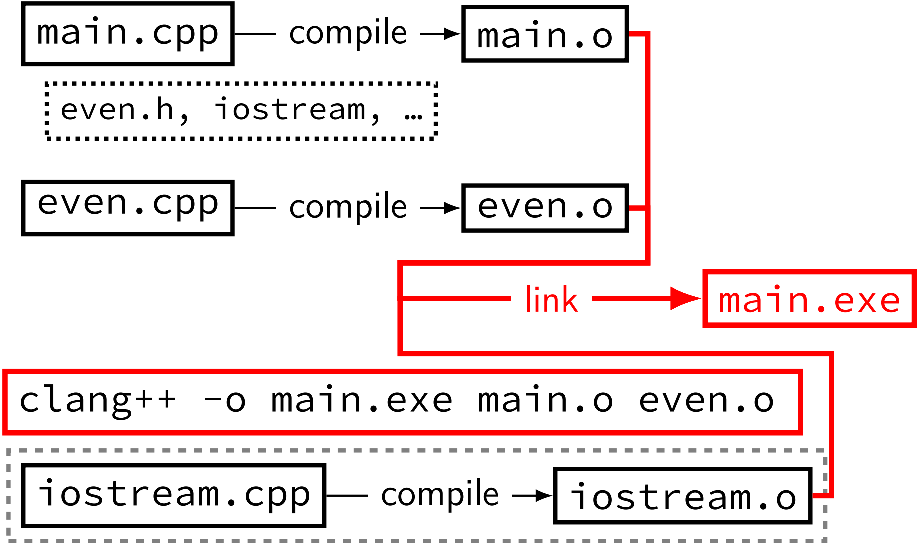
seperate compilation



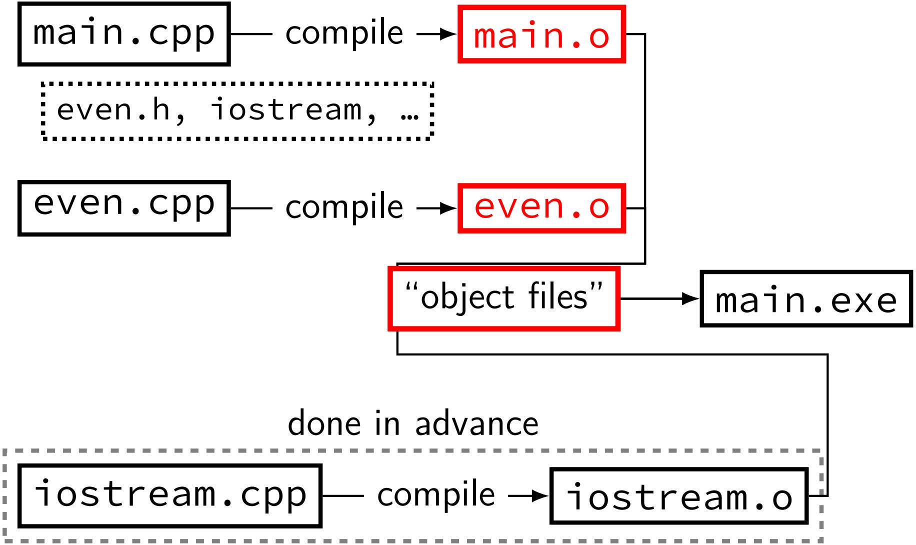
seperate compilation



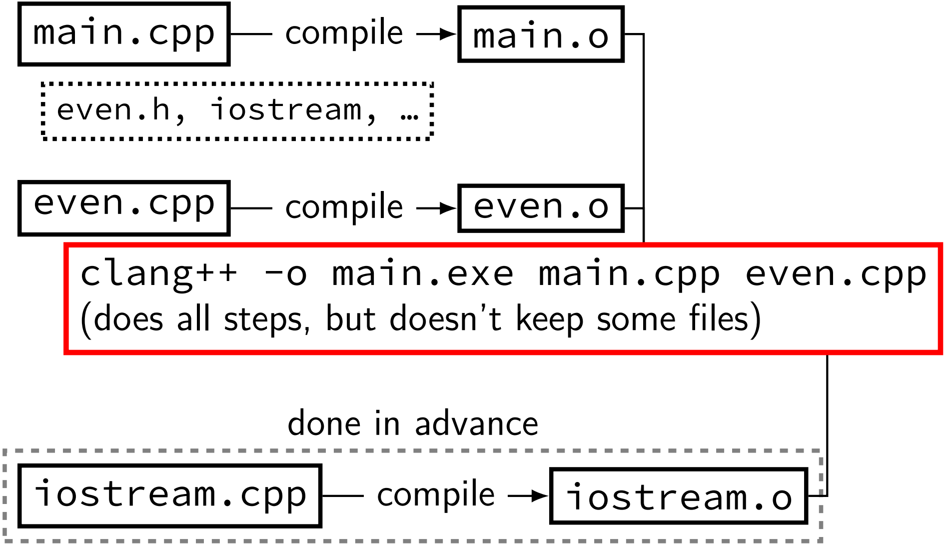
seperate compilation



seperate compilation



seperate compilation



on commands

`clang++ file1.cpp file2.cpp`

makes `a.out` or `a.exe`
`file1.h`, etc. not part of command

`clang++ -o main.exe file1.cpp file2.cpp`

makes `main.exe`

`clang++ -Wall -o main.exe file1.cpp file2.cpp`

makes `main.exe` with more compiler warnings

`clang++ -Wall -c file1.cpp`

makes `file1.o` (not executable)

Why clang++?

`clang++` our compiler of choice on lab machines

better than version of `g++` on lab machines/VM

a note on compiler warnings

```
int foo() {
    int bad;
    return 42;
}
```

default: almost no warnings

```
$ clang++ -c foo.cpp
$
```

add `-Wall`: more warnings

```
$ clang++ -Wall -c foo.cpp
foo.cpp:2:9: warning: unused variable 'bad' [-Wunused-variable
    int bad;
        ^
1 warning generated.
```

basic I/O

```
#include <iostream>
using std::cout; using std::cin; using std::endl;
// or  using namespace std;
int main() {
    int number;
    cout << "Enter␣a␣number:␣";
    cin >> number;
    cout << "You␣entered␣" << number << endl;
}
```

basic I/O

```
#include <iostream>
using std::cout; using std::cin; using std::endl;
// or  using namespace std;
int main() {
    int number;
    cout << "Enter␣a␣number:␣";
    cin >> number;
    cout << "You␣entered␣" << number << endl;
}
```

`cin` is a global `istream` object

`cout` is a global `ostream` object

types in C++ (1)

`char`

`short`, `int`, `long`

`float`, `double`

`bool`

types in C++ (1)

`char`

- 8-bit characters (ASCII, not Unicode)
- actually integers

`short`, `int`, `long`

`float`, `double`

`bool`

types in C++ (1)

`char`

- 8-bit characters (ASCII, not Unicode)
- actually integers

`short`, `int`, `long`

- size depends on machine

`float`, `double`

`bool`

types in C++ (1)

`char`

- 8-bit characters (ASCII, not Unicode)
- actually integers

`short`, `int`, `long`

- size depends on machine

`float`, `double`

`bool`

- yes, not `boolean`

types in C++ (2)

`unsigned int`, `unsigned short`, `unsigned long`

- like int, short, long — but only positive values
- (more on this later0

classes

Java: IntCell.java (1)

```
public class IntCell {
    public IntCell() { this(0); }

    public IntCell(int initialValue) {
        storedValue = initialValue;
    }

    public int getValue() {
        return storedValue;
    }

    public void setValue(int newValue) {
        storedValue = newValue;
    }

    private int storedValue;
}
```

Java: IntCell.java (1)

```java
public class IntCell {
    public IntCell() { this(0); }

    public IntCell(int initialValue) {
        storedValue = initialValue;
    }

    public int getValue() {
        return storedValue;
    }

    public void setValue(int newValue) {
        storedValue = newValue;
    }

    private int storedValue;
}
```

Java: IntCell.java (1)

```
public class IntCell {
    public IntCell() { this(0); }

    public IntCell(int initialValue) {
        storedValue = initialValue;
    }

    public int getValue() {
        return storedValue;
    }

    public void setValue(int newValue) {
        storedValue = newValue;
    }

    private int storedValue;
}
```

C++ version: three files

`IntCell.h` — "header file" with declarations only

`#included` by both files below

`IntCell.cpp` — implementation of class

`TestIntCell.cpp` — example `main()` that uses class

IntCell.h

```
#ifndef INTCELL_H
#define INTCELL_H
class IntCell {
  public:
    IntCell( int initialValue = 0 );

    int getValue() const;
    void setValue(int val);

  private:
    int storedValue;
};
#endif
```

IntCell.h

```
#ifndef INTCELL_H
#define INTCELL_H
class IntCell {
  public:
    IntCell( int initialValue = 0 );

    int ge
    void s

  private:
    int storedValue;
};
#endif
```

"boilerplate"
used to keep preprocessor from including file twice
(more on this later)

IntCell.h

```
#ifndef INTCELL_H
#define INTCELL_H
class IntCell {
  public:
    IntCell( int initialValue = 0 );

    int getValue() c
    void setValue(i

  private:
    int storedValue;
};
#endif
```

everything after this is public until `private:` (default is private)

IntCell.h

```
#ifndef INTCELL_H
#define INTCELL_H
class IntCell {
  public:
    IntCell( int initialValue = 0 );

    int getValue() con
    void setValue(int

  private:
    int storedValue;
};
#endif
```

constructor declaration

IntCell.h

```
#ifndef INTCELL_H
#define INTCELL_H
class IntCell {
  public:
    IntCell( int initialValue = 0 );

    int getVa
    void setV

  private:
    int storedValue;
};
#endif
```

default argument
must be part of declaration (not definition)

IntCell.h

```
#ifndef INTCELL_H
#define INTCELL_H
class IntCell {
  public:
    IntCell( int initialValue = 0 );

    int getValue() const;
    void setValue(int val);

  private:
    int stored
};
#endif
```

could have two explicit constructors, too:

```
IntCell();
IntCell(int initialValue);
```

IntCell.h

```
#ifndef INTCELL_H
#define
class In
  public
    IntCell( int initialValue = 0 );

    int getValue() const;
    void setValue(int val);

  private:
    int storedValue;
};
#endif
```

method declarations
(official C++ name for methods: “member functions”)

IntCell.h

```
#ifndef INTCEL
#define INTCEL
class IntCell
  public:
    IntCell( i                     );

    int getValue() const;
    void setValue(int val);

  private:
    int storedValue;
};
#endif
```

"const" *after parenthesis* —
indicates method does not change object
(`this` is const — enforced by compiler)

IntCell.h

```
#ifndef INTCELL_H
#define INTCELL_H
class IntCell {
  public:
    IntCell( int initialValue = 0 );

    int getVal
    void setVa

  private:
    int storedValue;
};
#endif
```

instance variable
(official C++ name: “member variable”)

IntCell.h

```
#ifndef INTCELL_H
#define INTCELL_H
class IntCell {
  public:
    IntCell( int initialValue = 0 );

    int getValue() con
    void setValue(int

  private:
    int storedValue;
};
#endif
```

semicolon is required!

IntCell.cpp

```
#include "IntCell.h"

IntCell::IntCell( int initialValue ) :
        storedValue( initialValue ) {
}

int IntCell::getValue() const {
    return storedValue;
}

void IntCell::setValue( int val ) {
    storedValue = val;
}
```

IntCell.cpp

```
#include "IntCell.h"

IntCell::IntCell( int initialValue ) :
        storedValue( initialValue ) {
}

int IntCell::getValue() const {
    return storedValue;
}

void IntC
    store
}
```

all method declarations prefixed with “ClassName::”

`::` seperates class/namespace names from names within the class/namespace

IntCell.cpp

```
#include "IntCell.h"

IntCell::IntCell( int initialValue ) :
        storedValue( initialValue ) {
}

int IntCell
    return
}

void IntCell::setValue( int val ) {
    storedValue = val;
}
```

declaration had "`int initialValue = 0`" not repeated in definition (doing so is an error)

IntCell.cpp

```
#include "IntCell.h"

IntCell::IntCell( int initialValue ) :
        storedValue( initialValue ) {
}

in

}

void IntCell::setValue( int val ) {
    storedValue = val;
}
```

special syntax for initializing member variables
used to call constructors (otherwise — default constructors used!)
`: variable1(value), variable2(anotherValue), ...`

IntCell.cpp

```
#include "I

IntCell::In
        sto
}
```

const (method called on const object)
defintion and declaration
(repeated in case both const and non-const
method with same name, arguments)

```
int IntCell::getValue() const {
    return storedValue;
}

void IntCell::setValue( int val ) {
    storedValue = val;
}
```

TestIntCell.cpp

```
#include <iostream>
#include "IntCell.h"
using namespace std;

int main( ) {
    IntCell m1;
    IntCell m2( 37 );
    // output: 0 37
    cout << m1.getValue( ) << "␣"
         << m2.getValue( ) << endl;
    m1 = m2;
    m2.setValue( 40 );
    // output: 37 40
    cout << m1.getValue( ) << "␣"
         << m2.getValue( ) << endl;
    return 0;
}
```

TestIntCell.cpp

```
#include <iostream>
#include "IntCell.h"
using namespace std;

int main( ) {
    IntCell m1;
    IntCell m2( 3
    // output: 0 
    cout << m1.ge
          << m2.getValue( ) << endl;
    m1 = m2;
    m2.setValue( 40 );
    // output: 37 40
    cout << m1.getValue( ) << "␣"
          << m2.getValue( ) << endl;
    return 0;
}
```

not a reference — cannot be null
represents the object itself

TestIntCell.cpp

```
#include <iostream>
#include "IntCell.h"
using namespace std;

int main( ) {
    IntCell m1;
    IntCell m2( 37 )
    // output: 0 37
    cout << m1.getVa
         << m2.getValue( ) << endl;
    m1 = m2;
    m2.setValue( 40 );
    // output: 37 40
    cout << m1.getValue( ) << "␣"
         << m2.getValue( ) << endl;
    return 0;
}
```

calls the default constructor
`IntCell::IntCell()`

TestIntCell.cpp

```
#include <iostream>
#include "IntCell.h"
using namespace std;

int main( ) {
    IntCell m1;
    IntCell m2( 37 );
    // output: 0 37
    cout << m1.getValue( ) << "␣"
         << m2.getValue( ) << endl;
    m1 = m2;
    m2.setValue( 40 );
    // output: 37 40
    cout << m1.getValue( ) << "␣"
         << m2.getValue( ) << endl;
    return 0;
}
```

calls `IntCell(37)` constructor

TestIntCell.cpp

```
#include <iostream>
#include "IntCell.h"
using namespace std;

int main( ) {
    IntCell m1
    IntCell m2
    // output:
    cout << m1
         << m2.getValue( ) << endl;
    m1 = m2;
    m2.setValue( 40 );
    // output: 37 40
    cout << m1.getValue( ) << "␣"
         << m2.getValue( ) << endl;
    return 0;
}
```

copies m2 into m1

like assigning each member variable

C++ objects are values (not references)

C++: Rational.h

```cpp
#ifndef RATIONAL_H
#define RATIONAL_H

class Rational {
  public:
    Rational();
    Rational(int numerator, int denominator);
    ~Rational();
    void print() const;
    Rational times(Rational b) const;
    Rational plus(Rational b) const;
    Rational reciprocal() const;
    Rational divides(Rational b) const;
  private:
    int num, den; // the numerator and denominator
    static int gcd(int m, int n); // helper function
};

#endif
```

C++: Rational.h

```
#ifndef RATIONAL_H
#define RATIONAL_H

class Rat
  public:
    Ratio
    Rational(int numerator, int denominator);
    ~Rational();
    void print() const;
    Rational times(Rational b) const;
    Rational plus(Rational b) const;
    Rational reciprocal() const;
    Rational divides(Rational b) const;
  private:
    int num, den; // the numerator and denominator
    static int gcd(int m, int n); // helper function
};

#endif
```

marked `const`

since they don't change the object they're called on

C++: Rational.h

```
#ifndef RATIONAL_H
#define RATIONAL_H

class Rational {
  public:
    Rational();
    Rational(int numerator, int denominator);
    ~Rational();
    void print() const;
    Rational times(Rational b) const;
    Rational plus(Rational b) const;
    Rational reciprocal() const;
    Rational divides(Rational b) const;
  private:
    int num, den; // the numerator and denominator
    static int gcd(int m, int n); // helper function
};

#endif
```

default constructor

C++: Rational.h

```
#ifndef RATIONAL_H
#define RATIONAL_H

class Rational {
  public:
    Rational();
    Rational(int numerator, int denominator);
    ~Rational();
    void print() const;
    Rational times(Rational b) const;
    Rational plus(Rational b) const;
    Rational reciprocal() const;
    Rational divides(Rational b) const;
  private:
    int num, den; // the numerator and denominator
    static int gcd(int m, int n); // helper function
};

#endif
```

another constructor

C++: Rational.h

```
#ifndef RATIONAL_H
#define RATIONAL_H

class Rational {
  public:
    Rational();
    Rational(int numerator, int denominator);
    ~Rational();
    void print() const;
    Rational times(Rational b) const;
    Rational plus(Rational b) const;
    Rational reciprocal() const;
    Rational divides(Rational b) const;
  private:
    int num, den; // the numerator and denominator
    static int gcd(int m, int n); // helper function
};

#endif
```

destructor — not actually useful yet

C++: Rational.h

```
#ifndef RATIONAL_H
#define RATIONAL_H

class Rational {
  public:
    Rational();
    Ration
    ~Ratio
    void p
    Rational times(Rational b) const;
    Rational plus(Rational b) const;
    Rational reciprocal() const;
    Rational divides(Rational b) const;
  private:
    int num, den; // the numerator and denominator
    static int gcd(int m, int n); // helper function
};

#endif
```

`static` — like Java, method doesn't take object
only appears on declaration

C++: Rational.cpp — constructors

```
...
// default constructor: initialize to 0/1
Rational::Rational() : num(0), den(1) {
}

Rational::Rational(int numerator, int denominator) {
    if (denominator == 0) {
        cout << "Denominator␣is␣zero" << endl;
    }
    int g = gcd(numerator, denominator);
    num = numerator   / g;
    den = denominator / g;
}
```

C++: Rational.cpp — constructors

```
...
// default constructor: initialize to 0/1
Rational::Rational() : num(0), den(1) {
}

Rational::Rational(int numerator, int denominator) {
    if (denomi
        cout 
    }
    int g = gcd(numerator, denominator);
    num = numerator   / g;
    den = denominator / g;
}
```

probably should throw exception instead?

C++: Rational.cpp — constructors

```
...
// default constructor: initialize to 0/1
Rational::Rational() : num(0), den(1) {
}

Rational::Rational(int numerator, int denominator) {
    if (denominator ==
        cout << "Denomi                    dl;
    }
    int g = gcd(numerator, denominator);
    num = numerator   / g;
    den = denominator / g;
}
```

call to utility method

C++: Rational.cpp — constructors

```
...
// default constructor: initialize to 0/1
Rational::Rational() : num(0), den(1) {
}

Rational::Rational(int numerator, int denominator) {
    if (denomina
        cout <<
    }
    int g = gcd(numerator, denominator);
    num = numerator   / g;
    den = denominator / g;
}
```

member variables initialized in body instead of : LIST syntax

C++: Rational.cpp — times

```
...
Rational Rational::times(Rational b) const {
    return Rational(num * b.num, den * b.den);
}
```

C++: Rational.cpp — times

```
...
Rational Rational::times(Rational b) const {
    return Rational(num * b.num, den * b.den);
}
```

syntax to create new Rational object

C++: Rational.cpp — times

```
...
Rational Rational::times(Rational b) const {
    return Rational(num * b.num, den * b.den);
}
```

need to mark definition `const`
because it's possible to have `const` and
non-`const` function with same name

IntCell.h

```
#ifndef INTCELL_H
#define INTCELL_H
class IntCell {
  public:
    IntCell( int initialValue = 0 );

    int ge
    void s

  private:
    int storedValue;
};
#endif
```

“boilerplate”
used to keep preprocessor from including file twice
(more on this later)

preprocessor

two steps to compilation

preprocessing

- `#include`, `#define`, `#ifdef`, etc
- can run alone: `clang++ -E file.cpp`

compilation

the preprocessor is dumb

Foo.h

```
class Foo { /* ... */ };
```

Bar.h

```
#include "Foo.h"
class Bar { /* ... uses Foo ... */ };
```

main.cpp

```
#include "Foo.h"
#include "Bar.h"
```

the preprocessor is dumb

Foo.h

```
class Foo { /* ... */ };
```

Bar.h

```
#include "Foo.h"
class Bar { /* ... uses Foo ... */ };
```

main.cpp

```
#include "Foo.h"
#include "Bar.h"
```

```
In file included from main.cpp:2:
In file included from ./Bar.h:1:
./Foo.h:1:7: error: redefinition of 'Foo'
class Foo {};
      ^
./Foo.h:1:7: note: previous definition is here
class Foo {};
```

running the preprocessor alone

(some lines omitted)

```
prompt$  clang++ -E main.cpp
# 1 "main.cpp"
# 1 "./Foo.h" 1
class Foo {};
# 2 "main.cpp" 2
# 1 "./Bar.h" 1
# 1 "./Foo.h" 1
class Foo {};
# 2 "./Bar.h" 2
class Bar {};
```

compiler generates this first
(as a temporary file)

running the preprocessor alone

(some lines omitted)

```
prompt$  clang++ -E main.cpp
# 1 "main.cpp"
# 1 "./Foo.h" 1
class Foo {};
# 2 "main.cpp" 2
# 1 "./Bar.h" 1
# 1 "./Foo.h" 1
class Foo {};
# 2 "./Bar.h" 2
class Bar {};
```

line numbers/file names for error messages

#define

```
/* make 'FOO' equivalent to 'something' */
#define FOO something

/* make 'BAR' equivalent to '' */
#define BAR

foo is FOO.
bar is BAR.
```

```
prompt$ clang++ -E define-example1.cpp
...

foo is something.
bar is .
```

#ifndef

```
#ifndef FOO
if shown after preprocessing:
foo not defined first time
#endif
#define FOO
#ifndef FOO
if shown after preprocessing:
foo not defined second time
#endif
```

```
prompt$ clang++ -E define-example2.cpp
...

if shown after preprocessing:
foo not defiend first time
```

#ifndef

```
#ifndef FOO
if shown after preprocessing:
foo not defined first time
#endif
#define FOO
#ifndef FOO
if shown after preprocessing:
foo not defined second time
#endif
```

omitted since after #define of FOO

```
prompt$ clang++ -E define-example2.cpp
...

if shown after preprocessing:
foo not defiend first time
```

the boilerplate

```
#ifndef FOO_H
#define FOO_H
  (contents here)
#endif
```

first time included — FOO_H not defined yet

sceond time included — FOO_H defined

preprocessor commands (subset)

`#define NAME replacement`

`#undef NAME`

`#ifndef NAME`, `#ifdef NAME`

`#if` *`expression`*

e.g. `#if defined(X) && defined(Y)`

`#define NAME(`*`X`*`, `*`Y`*`) thing w/ X and Y`

`NAME(foo, bar)` → `thing w/ foo and bar`

...

pointers

store memory addresses

- the location of values

memory?

memory (as 64-bit values)

address	value (64-bit)
0	123999
8	323232
16	434093
…	…
10000	1
10008	5
10016	7
…	…

memory?

memory (as 64-bit values)

address	value (64-bit)
0	123999
8	323232
16	434093
…	…
10000	1
10008	5
10016	7
…	…

(as 8-bit values)

address	value (8-bit)
0	95
1	228
2	1
3	0
4	0
5	0
6	0
7	0
8	160
9	238
10	4
11	…
…	…

values in memory

```
long aLong = 42;
int anInt = 43;
int anotherInt = 44;
```

memory (as 64-bit values)

address	value
…	…
10000	42
10008	43 | 44
10016	…
…	…

values in memory

```
long aLong = 42;
int anInt = 43;
int anotherInt = 44;
```

memory (as 64-bit values)

address	value	
…	…	
10000	42	aLong
10008	43 | 44	anInt, anotherInt
10016	…	
…	…	

values in memory

```
long aLong = 42;
int anInt = 43;
int anotherInt = 44;
```

memory (as 64-bit values)

address	value	
…	…	
10000	42	aLong
10008	43 | 44	anInt, anotherInt
10016	…	
…	…	

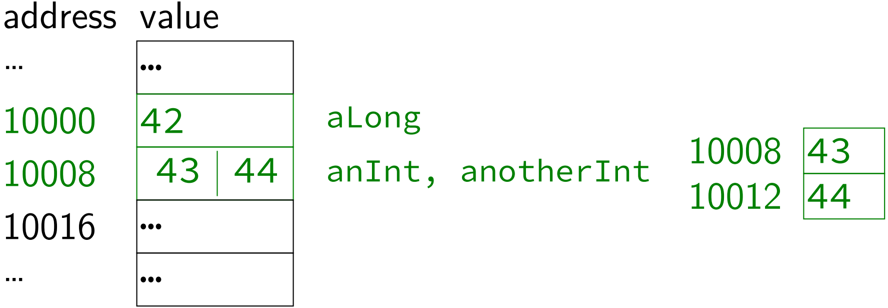
10008	43
10012	44

values in memory

```
long aLong = 42;
int anInt = 43;
int anotherInt = 44;
```

all variables kept in memory (array of bytes where 'everything' is stored)

memory (as 64-bit values)



pointers

```
long anInteger;
long *pointerToAnInteger;
anInteger = 42;
pointerToAnInteger = &anInteger;
*pointerToAnInteger = 43;
cout << pointerToInteger;
     // output: address (10000)
        // lab machines: in hexadecimal
cout << *pointerToInteger;
     // output: 43
```

memory (as 64-bit values)

address	value	
…	…	
10000	42	anInteger
10008	?	pointerToAnInteger
10016	…	
…	…	

pointers

```
long anInteger;
long *pointerToAnInteger;
anInteger = 42;
pointerToAnInteger = &anInteger;
*pointerToAnInteger = 43;
cout << pointerToInteger;
     // output: address (10000)
        // lab machines: in hexadecimal
cout << *pointerToInteger;
     // output: 43
```

&: "address of"

memory (as 64-bit values)

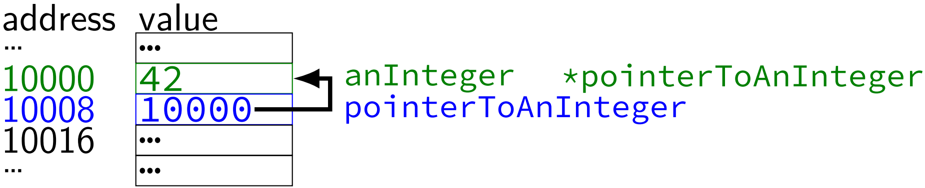
address	value	
…	…	
10000	42	anInteger
10008	?	pointerToAnInteger
10016	…	
…	…	

pointers

```
long anInteger;
long *pointerToAnInteger;
anInteger = 42;
pointerToAnInteger = &anInteger;
*pointerToAnInteger = 43;
cout << pointerToInteger;
     // output: address (10000)
        // lab machines: in hexadecimal
cout << *pointerToInteger;
     // output: 43
```

*: "dereference"
use value
at address

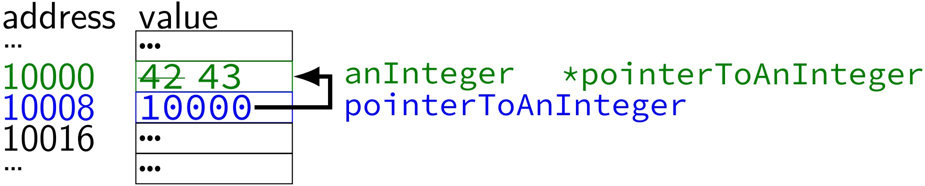
memory (as 64-bit values)



pointers

```
long anInteger;
long *pointerToAnInteger;
anInteger = 42;
pointerToAnInteger = &anInteger;
*pointerToAnInteger = 43;
cout << pointerToInteger;
     // output: address (10000)
        // lab machines: in hexadecimal
cout << *pointerToInteger;
     // output: 43
```

memory (as 64-bit values)



declaring pointers

```
float *X; // X is a pointer to float
float* X; // X is a pointer to float
float * X; // X is a pointer to float

Rational *Y; // Y is a pointer to Rational
Rational* Y; // Y is a pointer to Rational

Rational **Z; // Z is a pointer to pointer to Rational
```

declaring multiple pointers

```
float *X, *Y; // X and Y are pointers to float
float *Z, ThisIsProbablyAMistake;
    // Z is a pointer to float
    // ThisIsProbablyAMistake is a float
```

pointers to other types

```
Rational aFraction(2, 3);
Rational *pointerToFraction;
pointerToFraction = &aFraction;
*pointerToFraction =
    (*pointerToFraction).times(*pointerToFraction);
```

memory

address	value
...	...
10000	2 | 3
10008	?
10016	...
...	...

pointers to other types

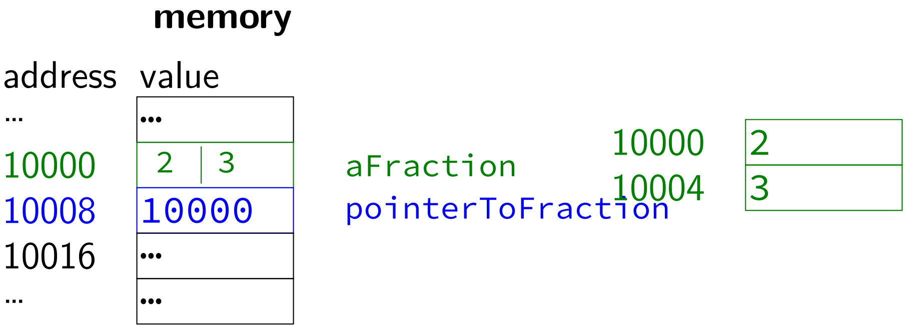
```
Rational aFraction(2, 3);
Rational *pointerToFraction;
pointerToFraction = &aFraction;
*pointerToFraction =
    (*pointerToFraction).times(*pointerToFraction);
```

memory

address	value	
...	...	
10000	2 | 3	aFraction
10008	?	pointerToFraction
10016	...	
...	...	

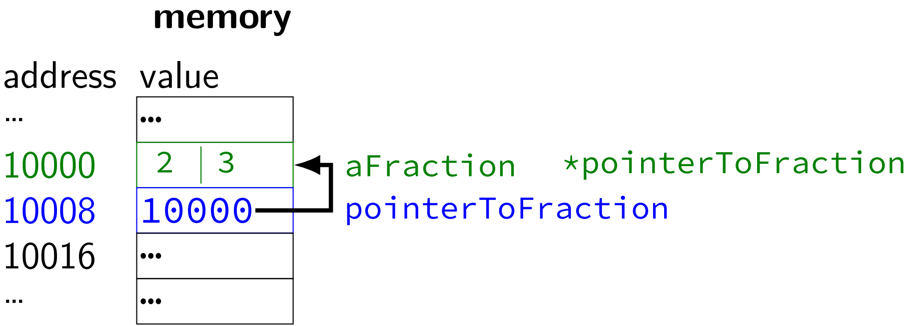
pointers to other types

```
Rational aFraction(2, 3);
Rational *pointerToFraction;
pointerToFraction = &aFraction;
*pointerToFraction =
    (*pointerToFraction).times(*pointerToFraction);
```



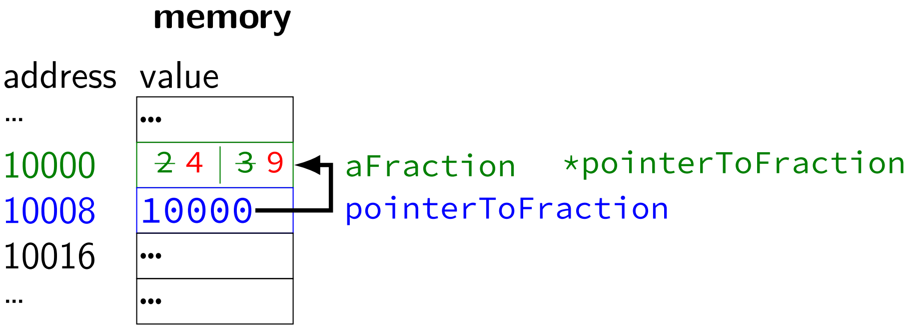
pointers to other types

```
Rational aFraction(2, 3);
Rational *pointerToFraction;
pointerToFraction = &aFraction;
*pointerToFraction =
    (*pointerToFraction).times(*pointerToFraction);
```



pointers to other types

```
Rational aFraction(2, 3);
Rational *pointerToFraction;
pointerToFraction = &aFraction;
*pointerToFraction =
    (*pointerToFraction).times(*pointerToFraction);
```



dereference operator

expression: `*foo` is “value pointed to by `foo`”

(different than declaration: `Type *foo` means “foo is a pointer to Type”)

`cout << *foo;` — output value foo points to

`*foo = 42;` — set value foo points to to 42

dereference v declare

```
int *pointer = &foo;
// same as:
int *pointer;
pointer = &foo;
```

---

dereference v declare

```
int *pointer = &foo;
// same as:
int *pointer;
pointer = &foo;
```

```
int *pointer = &foo;
*pointer = bar;  // sets foo to bar
pointer = &bar;  // changes where pointer points
```

address-of operator

in an expression: `&foo` is "address of `foo`"

(different than declaration: `int &foo = 42;` means 'foo is a *reference*" — more on that later)

returns address of variable/value

`&variable`, `&array[42]`, `&obj.instVar`

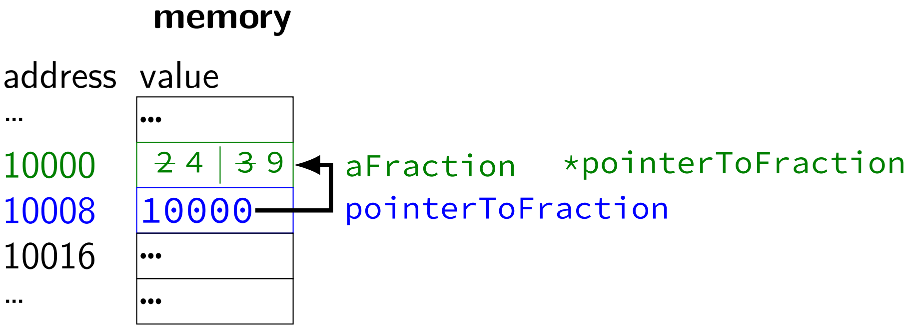
error if applied to temporary values (e.g. ~~`&(2+2)`~~)

`cout << &foo;` — output address of foo

`foo = &bar;` — set `foo` to be a pointer to `bar`

pointers to other types

```
Rational aFraction(2, 3);
Rational *pointerToFraction;
pointerToFraction = &aFraction;
*pointerToFraction =
    (*pointerToFraction).times(*pointerToFraction);
```



-> operator

`(*foo).bar` same as `foo->bar`

```
Rational *pointerToFraction = &aFraction;

aValue = pointerToFraction->times(
               *pointerToFraction);
// same as:
aValue = (*pointerToFraction).times(
               *pointerToFraction);
```

NULL

NULL or `0` — explicitly invalid pointer

for NULL: `#include <cstddef>`, etc.

```
int anInt = 42;
int *pointer = NULL;
int *pointer = 0; // same as above
// NOT same as: int *pointer;

*pointer = anInt;    // ERROR: crash (hopefully)
anInt = *pointer;    // ERROR: crash (hopefully)
pointer = anInt;     // ERROR: need cast

if (pointer == NULL) { ... }
if (!pointer) { ... } // same as above

if (pointer != NULL) { ... }
if (pointer) { ... } // same as above
```

NULL

NULL or 0 — explicitly invalid pointer

for NULL: `#include <cstddef>`, etc.

```
int anInt = 42;
int *pointer = NULL;
int *pointer = 0; // same as above
// NOT same as: int *pointer;

*pointer = anInt;    // ERROR: crash (hopefully)
anInt = *pointer;    // ERROR: crash (hopefully)
pointer = anInt;     // ERROR: need cast

if (pointer == NULL) { ... }
if (!pointer) { ... } // same as above

if (pointer != NULL) { ... }
if (pointer) { ... } // same as above
```

crash (hopefully)

Java — using a null pointer triggers `NullPointerException`

C++ — using a null pointer usually crashes

but not always — not required

uninitialized values

uninitialized pointers are not always null

whatever was stored in that part of memory before

might crash or
might silently point to something important

pointer-to-pointers

```
int valueOne = 42, valueTwo = 100;
int *pointer = &valueOne;
int **ptrToPtr = &pointer;
**ptrToPtr -= 10;
*ptrToPtr = &valueTwo;
**ptrToPtr += 10;
// output: 32 110 110
cout << valueOne << " " << valueTwo << " "
     << *pointer << endl;
```

address	value
...	...
10000	42
10004	100
10008	10000
10016	10008
10024	...
...	...

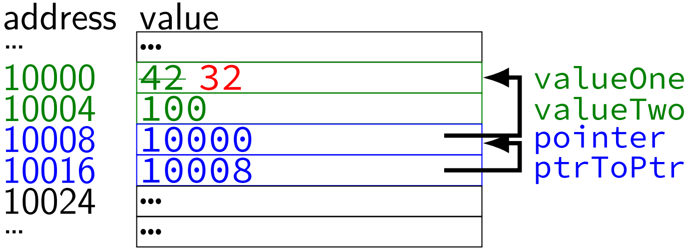
pointer-to-pointers

```
int valueOne = 42, valueTwo = 100;
int *pointer = &valueOne;
int **ptrToPtr = &pointer;
**ptrToPtr -= 10;
*ptrToPtr = &valueTwo;
**ptrToPtr += 10;
// output: 32 110 110
cout << valueOne << "␣" << valueTwo << "␣"
     << *pointer << endl;
```

address	value	
...	...	
10000	42	valueOne
10004	100	valueTwo
10008	10000	pointer
10016	10008	ptrToPtr
10024	...	
...	...	

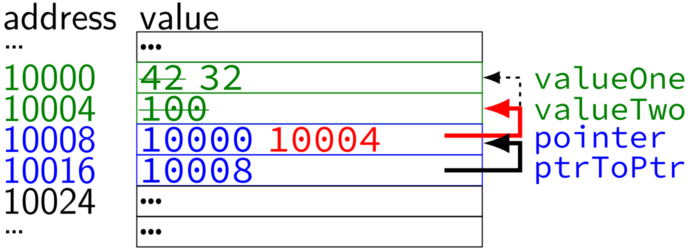
pointer-to-pointers

```
int valueOne = 42, valueTwo = 100;
int *pointer = &valueOne;
int **ptrToPtr = &pointer;
**ptrToPtr -= 10;
*ptrToPtr = &valueTwo;
**ptrToPtr += 10;
// output: 32 110 110
cout << valueOne << " " << valueTwo << " "
     << *pointer << endl;
```



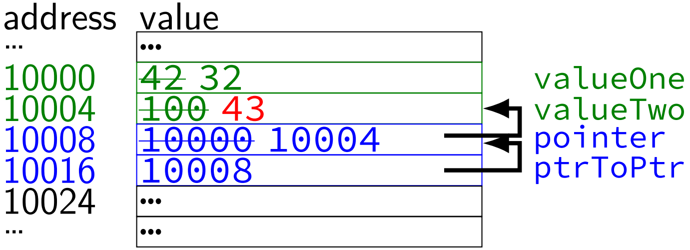
pointer-to-pointers

```
int valueOne = 42, valueTwo = 100;
int *pointer = &valueOne;
int **ptrToPtr = &pointer;
**ptrToPtr -= 10;
*ptrToPtr = &valueTwo;
**ptrToPtr += 10;
// output: 32 110 110
cout << valueOne << "␣" << valueTwo << "␣"
     << *pointer << endl;
```



pointer-to-pointers

```
int valueOne = 42, valueTwo = 100;
int *pointer = &valueOne;
int **ptrToPtr = &pointer;
**ptrToPtr -= 10;
*ptrToPtr = &valueTwo;
**ptrToPtr += 10;
// output: 32 110 110
cout << valueOne << "␣" << valueTwo << "␣"
     << *pointer << endl;
```



swap

```
void swap(Rational *a, Rational *b) {
    Rational temp = *a;
    *a = *b;
    *b = temp;
}

...
Rational first(4, 3);
Rational second(2, 7);
swap(&first, &second);
first.print();  // output: 2/7
```

pointer question

```
int a = 10, b = 20;
int *p; int *q;
p = &a;
q = p;
p = &b;
*p += 1;
*q = b;
```

What are the values of a, b?

A. a=10, b=21
B. a=11, b=21
C. a=20, b=21
D. a=21, b=21
E. something else
F. possible crash

inline methods (1)

```
class Foo {
public:
    Foo();
    int getValue() const {
        return value;
    }

    void setValue(int newValue) {
        value = newValue;
    }
    ...
private:
    int value;
    ...
};
```

inline methods (1)

```
class Foo {
public:
    Foo();
    int getValue() const {
        return value;
    }

    void setValue(int newValue) {
        value = newValue;
    }

pri
    ...
};
```

member function implemented in class declaration

this is allowed — even though implementation in many .cpp files

inline methods (1)

```
class Foo {
public:
    Foo();
    int getValue() const {
        return value;
    }

    void setValue(int newValue) {
        value = newValue;
    }
    ...
private:
    int
    ...
};
```

only advisible for very short methods

one copy of method *for each C++ file that uses class*

inline methods (2)

```
class Foo {
public:
    Foo();
    int getValue() const;
    ...
private:
    int value;
    ...
};
inline int Foo::getValue() const {
    return value;
}
```

inline methods (2)

```
class Foo {
public:
    Foo();
    int getValue() const;
    ...
private:
    int value;
    ...
};
inline int Foo::getValue() const {
    return
}
```

`inline` keyword — same as putting in class itself

still only advisible for short methods

must be included by every `.cpp` that uses class

inline methods (2)

```
class Foo {
public:
    Foo();
    int getValue() const;
    ...
private:
    int value;
    ...
};
inline int Foo::getValue() const {
    return value;
}
```

C++ local variables (1)

```
Rational getTwoThirds() {
    Rational twoThirds(2, 3);
    return twoThirds;
}
```

two thirds is copied when function returns

C++ local variables (2)

```
HugeValue computeHugeInteger() {
    HugeValue theHugeNumber = ...;
    return theHugeNumber;
}
```

copy huge number — very inefficiect?

C++: pointer to local variables?

```
Rational *brokenGetTwoThirds() {
    Rational twoThirds(2, 3);
    return &twoThirds;  // ERROR
}
```

twoThirds no longer exists when function returns

address likely to be reused for something else

new in C++

```
Rational *getTwoThirds() {
    Rational *twoThirdsPointer = new Rational(2, 3);
    return twoThirdsPointer;
}
HugeValue *computeHugeNumber() {
    HugeValue *theHugeNumber = new HugeValue;
    ... /* set *theHugeNumber */ ...
    return theHugeNumber;
}
```

does not copy — returns a pointer

new allocates space somewhere

need for delete (1)

```
Rational *getTwoThirds() {
    Rational *twoThirdsPointer;
    twoThirdsPointer = new Rational(2, 3);
    return twoThirdsPointer;
}

void showTwoThirds() {
    Rational *twoThirdsPointer = getTwoThirds();
    twoThirdsPointer->print();
}
```

what happens to where `twoThirdsPointer` points?

need for delete (1)

```
Rational *getTwoThirds() {
    Rational *twoThirdsPointer;
    twoThirdsPointer = new Rational(2, 3);
    return twoThirdsPointer;
}

void showTwoThirds() {
    Rational *twoThirdsPointer = getTwoThirds();
    twoThirdsPointer->print();
}
```

what happens to where `twoThirdsPointer` points?

memory remains used and allocated

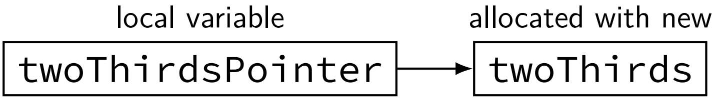
"memory leak"

need for delete (2)

```
Rational *getTwoThirds() {
    Rational *twoThirdsPointer = new Rational(2, 3);
    return twoThirdsPointer;
}

void showTwoThirds() {
    Rational *twoThirdsPointer = getTwoThirds();
    twoThirdsPointer->print();
}

int main() { showTwoThirds(); aThing(); return 0; }
```



need for delete (2)

```
Rational *getTwoThirds() {
    Rational *twoThirdsPointer = new Rational(2, 3);
    return twoThirdsPointer;
}

void showTwoThirds() {
    Rational *twoThirdsPointer = getTwoThirds();
    twoThirdsPointer->print();
}

int main() { showTwoThirds(); aThing(); return 0; }
```

need for delete (2)

```
Rational *getTwoThirds() {
    Rational *twoThirdsPointer = new Rational(2, 3);
    return twoThirdsPointer;
}

void showTwoThirds() {
    Rational *twoThirdsPointer = getTwoThirds();
    twoThirdsPointer->print();
}

int main() { showTwoThirds(); aThing(); return 0; }
```

fixed example

```
Rational *getTwoThirds() {
    Rational *twoThirdsPointer = new Rational(2, 3);
    return twoThirdsPointer;
}

void showTwoThirds() {
    Rational *twoThirdsPointer = getTwoThirds();
    twoThirdsPointer->print();
    delete twoThirdsPointer;
    // accessing twoThirdsPointer is now an ERROR
}
```

fixed example

```
Rational *getTwoThirds() {
    Rational *twoThirdsPointer = new Rational(2, 3);
    return twoThirdsPointer;
}

void showTwoThirds() {
    Rational *twoThirdsPointer = getTwoThirds();
    twoThirdsPointer->print();
    delete twoThirdsPointer;
    // accessing twoThirdsPointer is now an ERROR
}
```

an error — but may or may not crash (!)

whatever ends up at same address

C++: fixed-sized arrays

```
int arrayOfTenValues[10];
...
int fourthValue = arrayOfTenValues[3];
arrayOfTenValues[5] = newSixthValue;
```

C++: variable sized arrays?

```
int n;
cout << "Enter␣size:␣";
cin >> n;
...
int brokenArrayOfNValues[n];
...
```

not part of C++

(but some compilers allow an extension)

```
$ clang++ -Wall -pedantic -c test.cpp
test.cpp:3:29: warning: variable length arrays are a C99 featu
    int brokenArrayOfNValues[n];
```

C++: dynamic arrays (1)

```
int n;
cout << "Enter␣size:␣";
cin >> n;

// use the user's input to create an array of int
int * ages = new int [n];
```

address	value	
10000	90000	ages
…	…	
90000	?	ages[0]
90004	?	ages[1]
90008	?	ages[2]
…	…	
$90000+(n\text{-}1)\times 4$	?	ages[n-1]

C++: dynamic arrays (1)

```
int n;
cout << "Enter␣size:␣";
cin >> n;

// use the user's input to create an array of int
int * ages = new int [n];
```

address	value	
10000	90000	ages
…	…	
90000	?	ages[0]
90004	?	ages[1]
90008	?	ages[2]
…	…	
$90000+(n\text{-}1)\times 4$	?	ages[n-1]

C++: dynamic arrays (2)

```
int * ages = new int [n];
... /* use ages[i] */ ...
delete[] ages;
```

---

must explicitly free memory …

…otherwise, remains allocated (until program exits)

“memory leak”

C++: dynamic arrays (2)

```
int * ages = new int [n];
... /* use ages[i] */ ...
delete[] ages;
```

---

must explicitly free memory …

…otherwise, remains allocated (until program exits)

“memory leak”

C++: dynamic arrays (3)

```
int * ages = new int [n];
for (int i = 0; i < n; i++) {
    cout << "Value␣for␣ages[" << i << "]:␣";
    cin >> ages[i];
}
for (int i = 0; i < n; i++)
    cout << "ages[" << i << "]␣=␣" << ages[i]
         << endl;
delete[] ages;
```

C++: dynamic arrays (3)

```
int * ages = new int [n];
for (int i = 0; i < n; i++) {
    cout << "Value␣for␣ages[" << i << "]:␣";
    cin >> ages[i];
}
for (int i = 0; i < n; i++)
    cout << "ages[" << i << "]␣=␣" << ages[i]
         << endl;
delete[] ages;
```

new/delete

```
// single integer
int *p;             p = new int;            delete p;
int *p;             p = new int(3);         delete p;

// array of integers
int *p;             p = new int[100];       delete[] p;

Rational *p;        p = new Rational;       delete p;
Rational *p;        p = new Rational(3,4);  delete p;
```

new/delete

```
// single integer
int *p;          p = new int;            delete p;
int *p;          p = new int(3);         delete p;

// array of integers
int *p;          p = new int[100];       delete[] p;

Rational *p;     p = new Rational;       delete p;
Rational *p;     p = new Rational(3,4);  delete p;
```

`delete[]` form needed for new with arrays
idea: size information must be stored for arrays,
but single values

new/delete

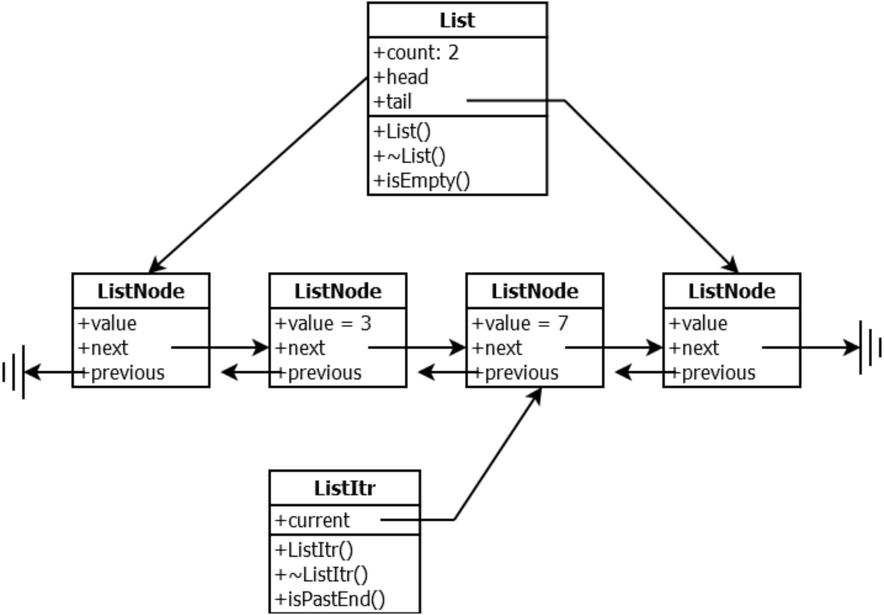
```
// single integer
int *p;          p = new int;            delete p;
int *p;          p = new int(3);         delete p;

// array of integers
int *p;          p = new int[100];       delete[] p;

Rational *p;     p = new Rational;       delete p;
Rational *p;     p = new Rational(3,4);  delete p;
```

`new` *`TYPE`*`(arg1, arg2)` — calls constructor
built-in constructors for primitive types takes value to copy

next lab: doubly-linked list



the lab's list declaration

```
class ListNode {

public:
    ListNode();                    // Constructor
    ...
private:
    int value;
    ListNode *next, *previous;

    friend class List;
    friend class ListItr;
};
```

the lab's list declaration

```
class ListNode {

public:
    ListNode();                         // Constructor
    ...
private:
    int value;
    ListNode *next, *previous;

    friend cl
    friend cl
};
```

∗ binds to name — declares two pointers;
(why I write ∗ next to names)

the lab's list declaration

```
class ListNode {

public:
    ListNode();                    // Constructor
    ...
private:
    int value;
    ListNode *next, *previous;

    friend class List;
    friend class ListItr;
};
```

the class `List` can access private members of `ListNode`

the lab's list declaration

```
class ListNode {

public:
    ListNode();                   // Constructor
    ...
private:
    int value;
    ListNode *next, *previous;

    friend class List;
    friend class ListItr;
};
```

the class `ListItr` can access private members of `ListNode`

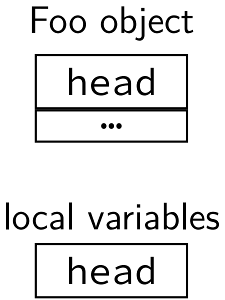
a common mistake (1)

```
class Foo {
public:
  Foo();
private:
  ListNode *head;
  ...
};
Foo::Foo() {
  ListNode *head = new ListNode; // BROKEN!
}
```

what's wrong with this?

a common mistake (1)

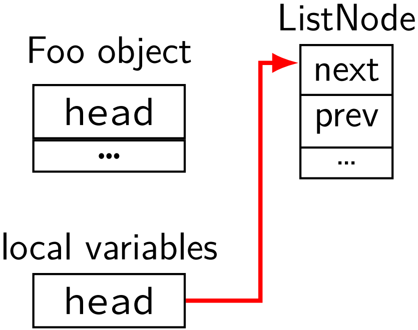
```
class Foo {
public:
  Foo();
private:
  ListNode *head;
  ...
};
Foo::Foo() {
  ListNode *head = new ListNode; // BROKEN!
}
```



what's wrong with this?

a common mistake (1)

```
class Foo {
public:
  Foo();
private:
  ListNode *head;
  ...
};
Foo::Foo() {
  ListNode *head = new ListNode; // BROKEN!
}
```



what's wrong with this?

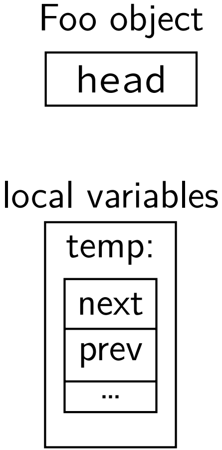
a common mistake (2)

```
class Foo {
public:
  Foo();
private:
  ListNode *head;
  ...
};
Foo::Foo() {
  ListNode temp;
  head = &temp;
}
```

what's wrong with this?

a common mistake (2)

```
class Foo {
public:
  Foo();
private:
  ListNode *head;
  ...
};
Foo::Foo() {
  ListNode temp;
  head = &temp;
}
```

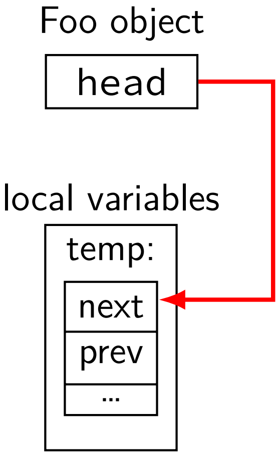


what's wrong with this?

a common mistake (2)

```
class Foo {
public:
  Foo();
private:
  ListNode *head;
  ...
};
Foo::Foo() {
  ListNode temp;
  head = &temp;
}
```

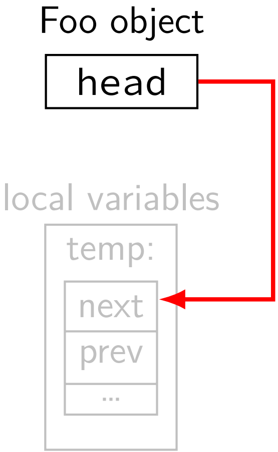
what's wrong with this?



a common mistake (2)

```
class Foo {
public:
  Foo();
private:
  ListNode *head;
  ...
};
Foo::Foo() {
  ListNode temp;
  head = &temp;
}
```

what's wrong with this?



memory.cpp

```
class Foo { long x, y; };
int main() {
    cout << "sizeof(long):␣" << sizeof(long) << endl;
    cout << "sizeof(Foo):␣" << sizeof(Foo) << endl;
    Foo *quux = new Foo;
    Foo *bar = new Foo;
    long diff = ((long)bar)-((long)quux);
    cout << "First␣foo:␣" << bar << endl;
    cout << "Second␣foo:␣" << quux << endl;
    cout << "Difference:␣" << diff << endl;
    delete quux; delete bar;
    return 0;
}
```

memory.cpp

```
class Foo { long x, y; };
int main() {
    cout << "sizeof(long): " << sizeof(long) << endl;
    cout << "sizeof(Foo): " << sizeof(Foo) << endl;
    Foo *quux = new Foo;
    Foo *bar = new Foo;
    long diff = ((long)bar)-((long)quux);
    cout << "First foo: " << bar << endl;
    cout << "Second foo: " << quux << endl;
    cout << "Difference: " << diff << endl;
    delete quux; delete bar;
    return 0;
}
```

`sizeof` operator — how many bytes is X?

memory.cpp

```
class Foo { long x, y; };
int main() {
    cout << "sizeof(long): " << sizeof(long) << endl;
    cout << "sizeof(Foo): " << sizeof(Foo) << endl;
    Foo *quux = new Foo;
    Foo *bar = new Foo;
    long diff = ((long)bar)-((long)quux);
    cout << "First foo: " << bar << endl;
    cout << "Second foo: " << quux << endl;
    cout << "Difference: " << diff << endl;
    delete quux; delete bar;
    return 0;
}
```

convert pointers to integers, subtract
= distance in memory

memory.cpp

```
class Foo { long x, y; };
int main() {
    cout << "sizeof(long): " << sizeof(long) << endl;
    cout << "sizeof(Foo): " << sizeof(Foo) << endl;
    Foo *quux = new Foo;
    Foo *bar = new Foo;
    long diff = ((long)bar)-((long)quux);
    cout << "First foo: " << bar << endl;
    cout << "Second foo: " << quux << endl;
    cout << "Difference: " << diff << endl;
    delete quux; delete bar;
    return 0;
}
```

prints out address

memory.cpp output

One (of many) possible output:

```
sizeof(long): 8
sizeof(Foo): 16
1st Foo: 0x1ec4030
2nd Foo: 0x1ec4050
Difference: 32
```

32 bytes apart? — 16 extra bytes?

implementation of `new` storing metadata

- need extra space *somewhere* to track size, etc.

C++ references

```
int x, y;
int &referenceToX = x;
x = 42; y = 100;
cout << referenceToX << "␣";  // output: 42
referenceToX = y;  // sets x
cout << referenceToX << "␣";  // output: 100
y = 99;
cout << x << "␣" << y;        // output: 100 99
```

references

'alternate name' for a value

like pointers that are automatically dereferenced

- stored like pointers!
- same issues with values that stop existing

can only bind references at initialization

swap with references

```
void swapWithPointers(int *x, int *y) {
    int temp = *y;
    *y = *x;
    *x = temp;
}

void swapWithReferences(int &x, int &y) {
    int temp = y;
    y = x;
    x = temp;
}
```

using swap

```
int main(void) {
    int x = 42, y = 100;
    swapWithPointers(&x, &y);
    cout << x << "␣" << y << endl;
        // output: 100 42

    x = 42; y = 100;
    swapWithReferences(x, y);
    cout << x << "␣" << y << endl;
        // output: 100 42
    return 0;
}
```

references to classes

```
class Square {
    ...
public:
    int sideLength;
};
...
Square *ptr = ...;
doSomethingWith(ptr->sideLength);
doSomethingWith((*ptr).sideLength);
Square &ref = ...;
doSomwthingWith(ref.sideLength);
```

* and &

`int *p = q` — p is a pointer to int
initially contains *address* q

`&y` — pointer to y

`int *p = &y; cout << *p` — outputs y's value

`int *p; p = &y; cout << *p` — outputs y's value

`int &r = y` — r is a reference to int
bound to y

`int &r = y; cout << r` — outputs y's value

reminder: re arrow syntax

```
Rational r; r.num = 4;
Rational *p = new Rational;
(*p).num = 4;
(*p).print();
p->num = 4;  // "follow the pointer"
p->print();
Rational &ref = r;
ref.num = 4;
ref.print();
```

pass-by-value (1)

```
class IntWrapper { public: int value; };
void foo(IntWrapper arg) {
    arg.value = 42;
}
int main(void) {
    IntWrapper iw;
    iw.value = 100;
    foo(iw);
    cout << iw.value;
}
```

what is the output?

A: 42
B: 100
C: crashes/doesn't compile
D: none of the above

pass-by-value (1)

```
class IntWrapper { public: int value; };
void foo(IntWrapper arg) {
    arg.value = 42;
}
int main(void) {
    IntWrapper iw;
    iw.value = 100;
    foo(iw);
    cout << iw.value;
}
```

what is the output?

A: 42
B: 100
C: crashes/doesn't compile
D: none of the above

pass-by-value (2)

```
class IntWrapper { public: int value; };
void foo(IntWrapper &arg) {
    arg.value = 42;
}
int main(void) {
    IntWrapper iw;
    iw.value = 100;
    foo(iw);
    cout << iw.value;
}
```

arg bound to iw

what is the output?

A: 42
B: 100
C: crashes/doesn't compile
D: none of the above

pass-by-value (2)

```
class IntWrapper { public: int value; };
void foo(IntWrapper &arg) {
    arg.value = 42;
}
int main(void) {
    IntWrapper iw;
    iw.value = 100;
    foo(iw);
    cout << iw.value;
}
```

arg bound to iw

what is the output?

A: 42 C: crashes/doesn't compile
B: 100 D: none of the above

pass-by-value (3)

```
class IntWrapper { public: int value; };
void foo(IntWrapper *arg) {
    arg.value = 42;
}
int main(void) {
    IntWrapper iw;
    iw.value = 100;
    foo(&iw);
    cout << iw.value;
}
```

what is the output?

A: 42 : C: crashes/doesn't compile
B: 100 D: none of the above

pass-by-value (3)

```
class IntWrapper { public: int value; };
void foo(IntWrapper *arg) {
    arg.value = 42;
}
int main(void) {
    IntWrapper iw;
    iw.value = 100;
    foo(&iw);
    cout << iw.value;
}
```

what is the output?

A: 42 : **C: crashes/doesn't compile**

B: 100 D: none of the above

pass-by-value (3)

```
class IntWrapper { public: int value; };
void foo(IntWrapper *arg) {
    arg.value = 42;
}
int main(void) {
    IntWrapper iw;
    iw.value = 100;
    foo(&iw);
    cout << iw.value;
}
```

what is the output?

A: 42 : **C: crashes/doesn't compile**
B: 100 D: none of the above

pointers don't have member variables

pass-by-value (4)

```
class IntWrapper { public: int value; };
void foo(IntWrapper *arg) {
    arg->value = 42;  // same as:  (*arg).value = 42;
}
int main(void) {
    IntWrapper iw;
    iw.value = 100;
    foo(&iw);
    cout << iw.value;
}
```

what is the output?

A: 42 C: crashes/doesn't compile

B: 100 D: none of the above

pass-by-value (4)

```
class IntWrapper { public: int value; };
void foo(IntWrapper *arg) {
    arg->value = 42;  // same as:  (*arg).value = 42;
}
int main(void) {
    IntWrapper iw;
    iw.value = 100;
    foo(&iw);
    cout << iw.value;
}
```

what is the output?

A: 42 C: crashes/doesn't compile

B: 100 D: none of the above

pass-by-value (4)

```
class IntWrapper { public: int value; };
void foo(IntWrapper *arg) {
    arg->value = 42;  // same as:  (*arg).value = 42;
}
int main(void) {
    IntWrapper iw;
    iw.value = 100;
    foo(&iw);
    cout << iw.value;
}
```

what is the output?

A: 42 C: crashes/doesn't compile
B: 100 D: none of the above

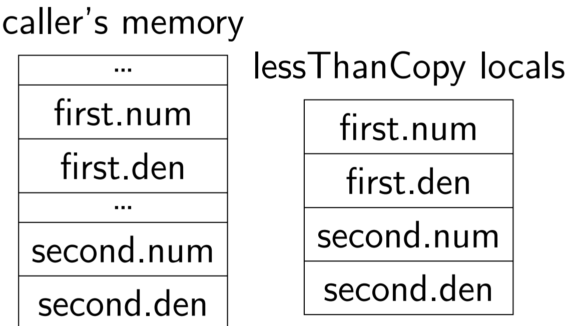
pointer's value (address) is copied

avoiding copying

```
bool lessThanCopy(Rational first, Rational second) {
    return first.num * second.den < second.num * first.den;
}
bool lessThanNoCopy(const Rational &first,
                    const Rational &second) {
    return first.num * second.den < second.num * first.den;
}
```

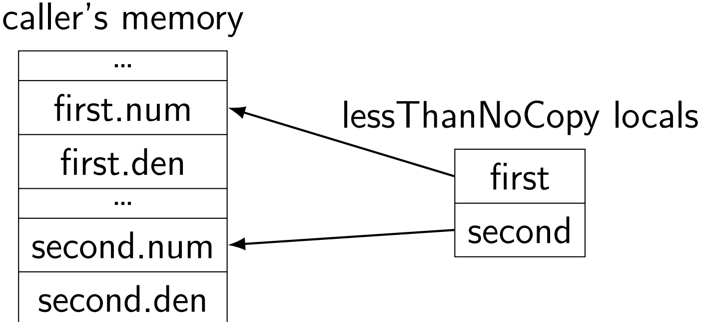
avoiding copying

```
bool lessThanCopy(Rational first, Rational second) {
    return first.num * second.den < second.num * first.den;
}
bool lessThanNoCopy(const Rational &first,
                    const Rational &second) {
    return first.num * second.den < second.num * first.den;
}
```



avoiding copying

```
bool lessThanCopy(Rational first, Rational second) {
    return first.num * second.den < second.num * first.den;
}
bool lessThanNoCopy(const Rational &first,
                    const Rational &second) {
    return first.num * second.den < second.num * first.den;
}
```



const and args

```
// no copy, modifies original
void foo(Rational& value) {
    value = Rational(4, 3);
}

// makes copy, modifies copy
void fooBroken1(Rational value) {
    value = Rational(4, 3);  // BROKEN
}

// makes const(ant) copy, error modifying
void fooBroken2(const Rational value) {
    value = Rational(4, 3);  // ERROR
}

// no copy, error modifying
void fooBroken3(const Rational& value) {
    value = Rational(4, 3);  // ERROR
}
```

const and args

```
// no copy, modifies original
void foo(Rational& value) {
    value = Rational(4, 3);
}

// makes copy, modifies copy
void fooBroken1(Rational value) {
    value = Rational(4, 3);  // BROKEN
}

// makes const(ant) copy, error modifying
void fooBroken2(const Rational value) {
    value = Rational(4, 3);  // ERROR
}

// no copy, error modifying
void fooBroken3(const Rational& value) {
    value = Rational(4, 3);  // ERROR
}
```

const and args

```
// no copy, modifies original
void foo(Rational& value) {
    value = Rational(4, 3);
}

// makes copy, modifies copy
void fooBroken1(Rational value) {
    value = Rational(4, 3);  // BROKEN
}

// makes const(ant) copy, error modifying
void fooBroken2(const Rational value) {
    value = Rational(4, 3);  // ERROR
}

// no copy, error modifying
void fooBroken3(const Rational& value) {
    value = Rational(4, 3);  // ERROR
}
```

const and args

```
// no copy, modifies original
void foo(Rational& value) {
    value = Rational(4, 3);
}

// makes copy, modifies copy
void fooBroken1(Rational value) {
    value = Rational(4, 3);  // BROKEN
}

// makes const(ant) copy, error modifying
void fooBroken2(const Rational value) {
    value = Rational(4, 3);  // ERROR
}

// no copy, error modifying
void fooBroken3(const Rational& value) {
    value = Rational(4, 3);  // ERROR
}
```

const and args

```
// no copy, modifies original
void foo(Rational& value) {
    value = Rational(4, 3);
}

// makes copy, modifies copy
void fooBroken1(Rational value) {
    value = Rational(4, 3);  // BROKEN
}

// makes const(ant) copy, error modifying
void fooBroken2(const Rational value) {
    value = Rational(4, 3);  // ERROR
}

// no copy, error modifying
void fooBroken3(const Rational& value) {
    value = Rational(4, 3);  // ERROR
}
```

return-by-reference

```
int counter;  // global variable
int &get_counter_reference() {
    return counter;
}
...
get_counter_reference() = 42;
cout << get_counter_reference() << endl;  // output: 42
```

return-by-reference — caution

```
int &get_counter_reference() {
    int counter = 0;
    return counter;    // ERROR
}
...
get_counter_reference() = 42;
    // ERROR -- writing unallocated object
```

return-by-reference — caution

```
int &get_counter_reference() {
    int counter = 0;
    return counter;    // ERROR
}
...
get_counter_reference() = 42;
    // ERROR -- writing unallocated object
```

same problem as:

```
int &get_counter_pointer() {
    int counter = 0;
    return &counter;    // ERROR
}
...
*get_counter_pointer() = 42;
    // ERROR -- writing unallocated object
```

reference member variables

```
class Foo {
public:
    Foo(int &x) : refToX(x) {}
    int &refToX;
};
...
  int value = 42;
  Foo foo(value);
  foo.refToX = 100;
  cout << value << endl; // output: 100
...
```

reference member variables

```
class Foo {
public:
    Foo(int &x) : refToX(x) {}
    int &refToX;
};
...
  int value = 42;
  Foo foo(value);
  foo.refToX = 100;
  cout << value << endl; // output: 100
...
```

Should you ever do this? Almost certainly not.

implicit methods

```
class Foo {};
```

Foo has the following methods:

- `Foo()` — default constructor
- `Foo(const Foo&)` — copy constructor
- `~Foo()` — destructor
- `Foo &operator=(const Foo&)` — assignment operator

created by compiler, but you can override

default constructor/destructor (1)

```
class Foo { public: Foo(); ~Foo(); };
Foo::Foo() { cout << "Foo::Foo()" << endl; }
Foo::~Foo() { cout << "Foo::~Foo()" << endl; }
int main() {
    Foo local;
    cout << "(1)\n";
    Foo *ptr = new Foo;
    cout << "(2)\n";
    delete ptr;
    cout << "(3)\n";
    return 0;
};
```

output:

```
Foo::Foo()
(1)
Foo::Foo()
(2)
Foo::~Foo()
(3)
Foo::~Foo()
```

default constructor/destructor (2)

```
class Foo { public: Foo(); ~Foo(); };
Foo::Foo() { cout << "Foo::Foo()" << endl; }
Foo::~Foo() { cout << "Foo::~Foo()" << endl; }
int main() {
    Foo *foos = new Foo[3];
    cout << "(1)\n";
    delete[] foo;
};
```

output:

```
Foo::Foo()
Foo::Foo()
Foo::Foo()
(1)
Foo::~Foo()
Foo::~Foo()
Foo::~Foo()
```

why destructors (1)

```
class DynamicArray {
    ...
    ~DynamicArray();
private:
    int *pointer;  // allocated with new int[...]
};
...
DynamicArray::~DynamicArray() {
    delete[] pointer;
}
```

why destructors (2)

close files, network connections, ...

```
#include <fstream>
void writeSomeText() {
    std::ofstream out("output.txt");
    out << "This␣is␣some␣text\n";
    // ofstream::~ofstream() called here
    // no explicit close needed!
}
```

copy constructors, operator= (1)

```
Foo a, b; // invokes Foo::Foo() twice

// invokes Foo::Foo(const Foo&)
Foo copy1(a);

// invokes Foo::Foo(const Foo&)
Foo copy2 = a;

// invokes Foo::operator=(const Foo&);
b = a;

// invokes Foo::operator=(const Foo&);
b.operator=(a);
```

default implementations (1)

```
// equivalent to default implementation:
Rational::Rational(const Rational &other)
    : den(other.den), num(other.num) {
}

// equivalent to default implementation:
Rational &Rational::operator=(
        const Rational &other) {
    // copy all members
    den = other.den;
    num = other.num;
    // return reference to this so
    //    foo = bar = baz
    // works
    return *this;
}
```

default implementations (2)

```
class Foo { public: Foo(); ~Foo(); };
Foo::Foo() { cout << "Foo::Foo()" << endl; }
Foo::~Foo() { cout << "Foo::~Foo()" << endl; }
class Bar { public: Foo x; };
int main() {
    Bar local;
    cout << "(1)\n";
    Bar *ptr = new Bar;
    cout << "(2)\n";
    delete ptr;
    cout << "(3)\n";
    return 0;
};
```

output:

```
Foo::Foo()
(1)
Foo::Foo()
(2)
Foo::~Foo()
(3)
Foo::~Foo()
```

default implementations (3)

```
class Foo { public: Foo(); ~Foo(); };
Foo::Foo() { cout << "Foo::Foo()" << endl; }
Foo::~Foo() { cout << "Foo::~Foo()" << endl; }
class Bar { public: Bar(); ~Bar(); Foo x; };
Bar::Bar() {} Bar::~Bar() {}
int main() {
    Bar local;
    cout << "(1)\n";
    Bar *ptr = new Bar;
    cout << "(2)\n";
    delete ptr;
    cout << "(3)\n";
    return 0;
};
```

output:

```
Foo::Foo()
(1)
Foo::Foo()
(2)
Foo::~Foo()
(3)
Foo::~Foo()
```

default implementations (3)

```
class Foo { public: Foo(); ~Foo(); };
Foo::Foo() { cout << "Foo::Foo()" << endl; }
Foo::~Foo() { cout << "Foo::~Foo()" << endl; }
class Bar { public: Bar(); ~Bar(); Foo x; };
Bar::Bar() {} Bar::~Bar() {}
int main() {
    Bar local;
    cout << "(1)\n";
    Bar *ptr = new Bar;
    cout << "(2)\n";
    delete ptr;
    cout << "(3)\n";
    return 0;
};
```

output:

```
Foo::Foo()
(1)
Foo::Foo()
(2)
Foo::~Foo()
(3)
Foo::~Foo()
```

missing defaults?

```
#include <iostream>
using namespace std;

class Foo {
public:
    Foo(int x) { cout << "x␣=␣" << x << endl; }
};

int main(void) {
    Foo x;
}
```

```
example.cpp:10:9: error: no matching constructor for
                          initialization of 'Foo'
    Foo x;
....
```

rule: no implicit default constructor if there are others

can still explicitly write `Foo() {}`

operator=

```
class Foo { public:
    Foo& operator=(const Foo& other);
}
Foo & Foo::operator=(const Foo& other) {
    cout << "called␣Foo::operator=";
    return *this;
}
int main() {
    Foo x, y;
    x = y;  // output:  called Foo::operator=
}
```

overrided operators can do whatever

```
class Bar {};
class Foo { public:
    Foo();
    Foo& operator=(const Bar& other);
    int count;
}
Foo::Foo() : count(0) {}
Foo & Foo::operator=(const Bar& other) {
    cout << "count=" << ++count;
    return *this;
}
int main() {
    Foo x;
    x = Bar();
        // output count=1
    x = Bar();
        // output count=2
}
```

more operator overriding (1)

```
    // ostream &ostream::operator<<(const char*)
cout << "Foo" << endl;
    // istream &istream::operator>>(int&)
cin >> number;
```

```
#include <string>
using std::string;
...
    // string::string(const char*)
string x = "This␣is␣the␣first␣part.";
string y = "␣And␣this␣is␣the␣second␣part.";

    // string string::operator+(const string&) const
string y = x + z;

    // string &string::operator+=(const char*)
y += "␣And␣this␣is␣the␣third␣part.";
```

more operator overloading (2)

```
class Rational {
    ...
    Rational operator*(const Rational& other)
        const;
    Rational operator+(const Rational& other)
        const;
    ...
};
Rational x(2, 4), y(4, 5);
Rational z = x * y + y;
```

operator overloading with methods

```
int x = 42;
cout << "The␣value␣is:␣" << x << endl;
// same as:
cout.operator<<("The␣value␣is:␣").operator<<(x).operator<<(endl);
```

```
/* approximate code ... */
class ostream {
    ...
    ostream &operator<(int value);
};
...
ostream &ostream::operator<<(int value) {
    ...
    return *this;
}
```

operator overloading with functions

```
#include <string>
using std::cout; using std::string; using std::endl;
...
string x = ...;  // like Java String class
cout << x << x << endl;
// same as:
operator<<(operator<<(cout, x), x).operator<<(endl);
```

```
ostream& operator<<(ostream& out, const string &s) {
    ...
    return out;
}
```

C++ combined example

test class to demo constructors, operator=, etc.

single file (slides page in git) with all examples for test class:
`cpptest.cpp`

this lecture: in independent pieces

C++ combined example (.h)

```
// test.h:
class test {
    static int idcount;
    const int id;
    int value;
  public:
    test();
    test(int v);
    test(const test& x);
    ~test();
    test& operator=(const test& other);
    friend ostream& operator<<(ostream& out,
                               const test& f);
};
```

C++ combined example (.h)

```
// test.h:
class test {
    static int idcount;
    const int id;
    int value;
  public:
    test();
    test(int v);
    test(const test& x);
    ~test();
    test& operator=(const test& other);
    friend ostream& operator<<(ostream& out,
                               const test& f);
};
```

const — must be set in constructor

C++ combined example (.h)

```
// test.h:
class test {
    static int idcount;
    const int id;
    int value;
  public:
    test();
    test(int v);
    test(const test& x);
    ~test();
    test& operator=(const test& other);
    friend ostream& operator<<(ostream& out,
                               const test& f);
};
```

friend function for outputting to an ostream (like cout)

C++ combined example (test.cpp)

```
// test.cpp:
int test::idcount = 0;

ostream &operator<<(ostream &out, const test &f) {
  out << "test[id=" << f.id << ",v="
      << f.value << "]@" << &f;
  return out;
}

test::test(const test& x) : id(x.id), value(x.value) {
  cout << "calling␣test(" << x <<");␣object␣created␣is␣" << *this
       << endl;
}

test &test::operator=(const test &other) {
  cout << "calling␣" << *this
       << ".operator=(" << other << ")" << endl;
  return *this;
}
```

C++ combined example (test.cpp)

```
// test.cpp:
int test::idcount = 0;

ostream &operator<<(ostream &out, const test &f) {
  out << "test[id=" << f.id << ",v="
      << f.value << "]@" << &f;
  return out;
}
```

```
class test { static int idcount; ... }
```

```
test::test(const test& x) : id(x.id), value(x.value) {
  cout << "calling␣test(" << x <<");␣object␣created␣is␣" << *this
       << endl;
}

test &test::operator=(const test &other) {
  cout << "calling␣" << *this
       << ".operator=(" << other << ")" << endl;
  return *this;
}
```

C++ combined example (test.cpp)

```
// test.cpp:
int test::idcount = 0;

ostream &operator<<(ostream &out, const test &f) {
  out << "test[id=" << f.id << ",v="
      << f.value << "]@" << &f;
  return out;
}

test::test(const test& x) : id(x.id), value(x.value) {
  cout << "calling␣test(" << x <<");␣object␣created␣is␣" << *this
       << endl;
}

test &test::operator=(const test &other) {
  cout << "calling␣" << *this
       << ".operator=(" << other << ")" << endl;
  return *this;
}
```

const, so must be on initialization list

C++ combined example (test.cpp)

```
// test.cpp:
int test::idcount = 0;

ostream &operator<<(ostream &out, const test &f) {
  out << "test[id=" << f.id << ",v="
      << f.value << "]@" << &f;
  return out;
}

test::test(const test& x) : id(x.id), value(x.value) {
  cout << "calling␣test(" << x <<");␣object␣created␣is␣" << *this
       << endl;
}

test &test::operator=(const test &other) {
  cout << "calling␣" << *this
       << ".operator=(" << other << ")" << endl;
  return *this;
}
```

called like assignment doesn't actually assign!

C++ combined example (test.cpp)

```
// test.cpp:
int test::idcount = 0;

ostream &operator<<(ostream &out, const test &f) {
  out << "test[id=" << f.id << ",v="
      << f.value << "]@" << &f;
  return out;
}

test::test(const test& x) : id(x.id), value(x.value) {
  cout << "calling␣test(" << x <<");␣object␣created␣is␣" << *this
       << endl;
}

test &test::operator=(const test &other) {
  cout << "calling␣" << *this
       << ".operator=(" << other << ")" << endl;
  return *this;
}
```

return out so chains of << work

trivial test object (1)

```
int main() {
    cout << "about␣to␣create␣aa" << endl;
    test aa;
    cout << "aa␣is:␣" << aa << endl;
    return 0;
}
```

```
about to create aa
calling test(); object created is test[id=0,v=0]@0x7ffc82ba9440
aa is: test[id=0,v=0]@0x7ffc82ba9440
calling ~test() on test[id=0,v=0]@0x7ffc82ba9440
```

trivial test object (1)

```
int main() {
    cout << "about␣to␣create␣aa" << endl;
    test aa;
    cout << "aa␣is:␣" << aa << endl;
    return 0;
}
```

```
about to create aa
calling test(); object created is test[id=0,v=0]@0x7ffc82ba9440
aa is: test[id=0,v=0]@0x7ffc82ba9440
calling ~test() on test[id=0,v=0]@0x7ffc82ba9440
```

trivial test object (1)

```
int main() {
    cout << "about␣to␣create␣aa" << endl;
    test aa;
    cout << "aa␣is:␣" << aa << endl;
    return 0;
}
```

```
about to create aa
calling test(); object created is test[id=0,v=0]@0x7ffc82ba9440
aa is: test[id=0,v=0]@0x7ffc82ba9440
calling ~test() on test[id=0,v=0]@0x7ffc82ba9440
```

trivial test object (2)

```
int main() {
    cout << "about␣to␣create␣b" << endl;
    test b(1);
    cout << "b␣is:␣" << b << endl;
    return 0;
}
```

```
about to create b
calling test(1); object created is test[id=0,v=1]@0x7fffb1438590
b is: test[id=0,v=1]@0x7fffb1438590
calling ~test() on test[id=0,v=1]@0x7fffb1438590
```

trivial test object (2)

```
int main() {
    cout << "about␣to␣create␣b" << endl;
    test b(1);
    cout << "b␣is:␣" << b << endl;
    return 0;
}
```

```
about to create b
calling test(1); object created is test[id=0,v=1]@0x7fffb1438590
b is: test[id=0,v=1]@0x7fffb1438590
calling ~test() on test[id=0,v=1]@0x7fffb1438590
```

trivial test object (2)

```
int main() {
    cout << "about␣to␣create␣b" << endl;
    test b(1);
    cout << "b␣is:␣" << b << endl;
    return 0;
}
```

```
about to create b
calling test(1); object created is test[id=0,v=1]@0x7fffb1438590
b is: test[id=0,v=1]@0x7fffb1438590
calling ~test() on test[id=0,v=1]@0x7fffb1438590
```

gotcha: Object foo() makes no Object

```
int main() {
    cout << "before␣test␣a()" << endl;
    test a();
    cout << "a␣is:␣" << a << endl;
    return 0;
}
```

```
before test a()
a is: 1
```

Object foo(): warnings

```
$ clang++ -Wall -pedantic -o testgotcha \
                 testgotcha.cpp test.cpp
testgotcha.cpp:7:11: warning: empty parentheses
                    interpreted as a function
                    declaration [-Wvexing-parse]
    test a();
          ^~
testgotcha.cpp:7:11: note: remove parentheses to
                     declare a variable
    test a();
          ^~
testgotcha.cpp:8:25: warning: address of function 'a'
                     will always evaluate to 'true'
                     [-Wpointer-bool-conversion]
    cout << "a is: " << a << endl;
```

declaring function inside a function???

```
#include <iostream>
using namespace std;
// instead of declaring here...
int main() {
    // legal to declare here, but...
    // you probably should NEVER do this
    int foo(int x);
    cout << foo(21) << endl;
    // output: 42
    return 0;
}
int foo(int x) { return x * 2; }
```

new

```
int main() {
    test *c = new test(2);
    cout << "created␣*c:␣" << *c << endl;
    test *d = new test;
    cout << "created␣*d:␣" << *d << endl;
    return 0;
}
```

```
calling test(2); object created is test[id=0,v=2]@0x144dc20
created *c: test[id=0,v=2]@0x144dc20
calling test(); object created is test[id=1,v=0]@0x144e050
created *d: test[id=1,v=0]@0x144e050
```

new

```
int main() {
    test *c = new test(2);
    cout << "created␣*c:␣" << *c << endl;
    test *d = new test;
    cout << "created␣*d:␣" << *d << endl;
    return 0;
}
```

```
calling test(2); object created is test[id=0,v=2]@0x144dc20
created *c: test[id=0,v=2]@0x144dc20
calling test(); object created is test[id=1,v=0]@0x144e050
created *d: test[id=1,v=0]@0x144e050
```

new + delete

```
int main() {
    test *c = new test(2);
    test *d = new test;
    delete c;
    return 0;
}
```

```
calling test(2); object created is test[id=0,v=2]@0xe91c20
calling test(); object created is test[id=1,v=0]@0xe92050
calling ~test() on test[id=0,v=2]@0xe91c20
```

assignment

```
int main() {
    test b(1);
    test aa;
    cout << "b␣is:␣" << b << endl;
    cout << "aa␣is:␣" << aa << endl;
    aa = b;
    cout << "aa␣is:␣" << aa << endl;
    return 0;
}
```

```
calling test(1); object created is test[id=0,v=1]@0x7ffc153722a0
calling test(); object created is test[id=1,v=0]@0x7ffc15372298
b is: test[id=0,v=1]@0x7ffc153722a0
aa is: test[id=1,v=0]@0x7ffc15372298
calling operator=(test[id=0,v=1]@0x7ffc153722a0)
aa is: test[id=1,v=0]@0x7ffc15372298
calling ~test() on test[id=1,v=0]@0x7ffc15372298
calling ~test() on test[id=0,v=1]@0x7ffc153722a0
```

assignment

```
int main() {
    test b(1);
    test aa;
    cout << "b␣is:␣" << b << endl;
    cout << "aa␣is:␣" << aa << endl;
    aa = b;
    cout << "aa␣is:␣" << aa << endl;
    return 0;
}
```

```
calling test(1); object created is test[id=0,v=1]@0x7ffc153722a0
calling test(); object created is test[id=1,v=0]@0x7ffc15372298
b is: test[id=0,v=1]@0x7ffc153722a0
aa is: test[id=1,v=0]@0x7ffc15372298
calling operator=(test[id=0,v=1]@0x7ffc153722a0)
aa is: test[id=1,v=0]@0x7ffc15372298
calling ~test() on test[id=1,v=0]@0x7ffc15372298
calling ~test() on test[id=0,v=1]@0x7ffc153722a0
```

assignment

```
int main() {
    test b(1);
    test aa;
    cout << "b␣is:␣" << b << endl;
    cout << "aa␣is:␣" << aa << endl;
    aa = b;
    cout << "aa␣is:␣" << aa << endl;
    return 0;
}
```

```
calling test(1); object created is test[id=0,v=1]@0x7ffc153722a0
calling test(); object created is test[id=1,v=0]@0x7ffc15372298
b is: test[id=0,v=1]@0x7ffc153722a0
aa is: test[id=1,v=0]@0x7ffc15372298
calling operator=(test[id=0,v=1]@0x7ffc153722a0)
aa is: test[id=1,v=0]@0x7ffc15372298
calling ~test() on test[id=1,v=0]@0x7ffc15372298
calling ~test() on test[id=0,v=1]@0x7ffc153722a0
```

copy-construction

```
int main() {
    test b(1);
    test aa = b;
    cout << "b is: " << b << endl;
    cout << "aa is: " << aa << endl;
    return 0;
}
```

```
calling test(1); object created is test[id=0,v=1]@0x7ffc3f61b630
calling test(test[id=0,v=1]@0x7ffc3f61b630);
    object created is test[id=0,v=1]@0x7ffc3f61b628
b is: test[id=0,v=1]@0x7ffc3f61b630
aa is: test[id=0,v=1]@0x7ffc3f61b628
calling ~test() on test[id=0,v=1]@0x7ffc3f61b628
calling ~test() on test[id=0,v=1]@0x7ffc3f61b630
```

copy-construction

```
int main() {
    test b(1);
    test aa = b;
    cout << "b␣is:␣" << b << endl;
    cout << "aa␣is:␣" << aa << endl;
    return 0;
}
```

```
calling test(1); object created is test[id=0,v=1]@0x7ffc3f61b630
calling test(test[id=0,v=1]@0x7ffc3f61b630);
    object created is test[id=0,v=1]@0x7ffc3f61b628
b is: test[id=0,v=1]@0x7ffc3f61b630
aa is: test[id=0,v=1]@0x7ffc3f61b628
calling ~test() on test[id=0,v=1]@0x7ffc3f61b628
calling ~test() on test[id=0,v=1]@0x7ffc3f61b630
```

function call

```
test bar(test param) {
  return test(10);
}
int main() {
  test *c = new test(2);  // oops: never deleted
  cout << "about␣to␣call␣bar" << endl;
  test e = bar(*c);
  cout << "done␣calling␣bar" << endl;
}
```

```
calling test(2); object created is test[id=0,v=2]@0x17b1c20
about to call bar
calling test(test[id=0,v=2]@0x17b1c20); object created is test[id=0,
calling test(10); object created is test[id=1,v=10]@0x7ffcea937530
calling ~test() on test[id=0,v=2]@0x7ffcea937528
done calling bar
calling ~test() on test[id=1,v=10]@0x7ffcea937530
```

function call

```
test bar(test param) {
  return test(10);
}
int main() {
  test *c = new test(2);  // oops: never deleted
  cout << "about␣to␣call␣bar" << endl;
  test e = bar(*c);
  cout << "done␣calling␣bar" << endl;
}
```

```
calling test(2); object created is test[id=0,v=2]@0x17b1c20
about to call bar
calling test(test[id=0,v=2]@0x17b1c20); object created is test[id=0,
calling test(10); object created is test[id=1,v=10]@0x7ffcea937530
calling ~test() on test[id=0,v=2]@0x7ffcea937528
done calling bar
calling ~test() on test[id=1,v=10]@0x7ffcea937530
```

function call

```
test bar(test param) {
  return test(10);
}
int main() {
  test *c = new test(2);  // oops: never deleted
  cout << "about_
  test e = bar(*c
  cout << "done_c
}
```

"return value optimization" or "copy elision": compiler omitted copy constructor call for e (created return value directly inside e)

```
calling test(2); object created is test[id=0,v=2]@0x17b1c20
about to call bar
calling test(test[id=0,v=2]@0x17b1c20); object created is test[id=0,
calling test(10); object created is test[id=1,v=10]@0x7ffcea937530
calling ~test() on test[id=0,v=2]@0x7ffcea937528
done calling bar
calling ~test() on test[id=1,v=10]@0x7ffcea937530
```

function call

```
test bar(test param) {
  return test(10);
}
int main() {
  test *c = new test(2);  // oops: never deleted
  cout << "about␣to␣call bar" << endl;
  test e = bar(*c);
  cout << "done␣call
}
```

very different addresses for local vars versus new'd objects

```
calling test(2); object created is test[id=0,v=2]@0x17b1c20
about to call bar
calling test(test[id=0,v=2]@0x17b1c20); object created is test[id=0,
calling test(10); object created is test[id=1,v=10]@0x7ffcea937530
calling ~test() on test[id=0,v=2]@0x7ffcea937528
done calling bar
calling ~test() on test[id=1,v=10]@0x7ffcea937530
```

some remaining C++ topics

containers in the C++ standard library

- equivalent to `java.util.*`

templates

- equivalent to Java's *generics*

the `string` class

backup slides

argument-dependent lookup

given `foo(x, y, z)` where `x` is an `XNamespace::X`, etc.

C++ looks for `foo` in: (not necessairily in this order)

- the current namespace
- the global namespace
- `XNamespace`
- `YNamespace`
- `ZNamespace`
- anything from using directives

`x + y` is a like `operator+(x, y)` for this purpose