What do you think?

How many bugs can you find?

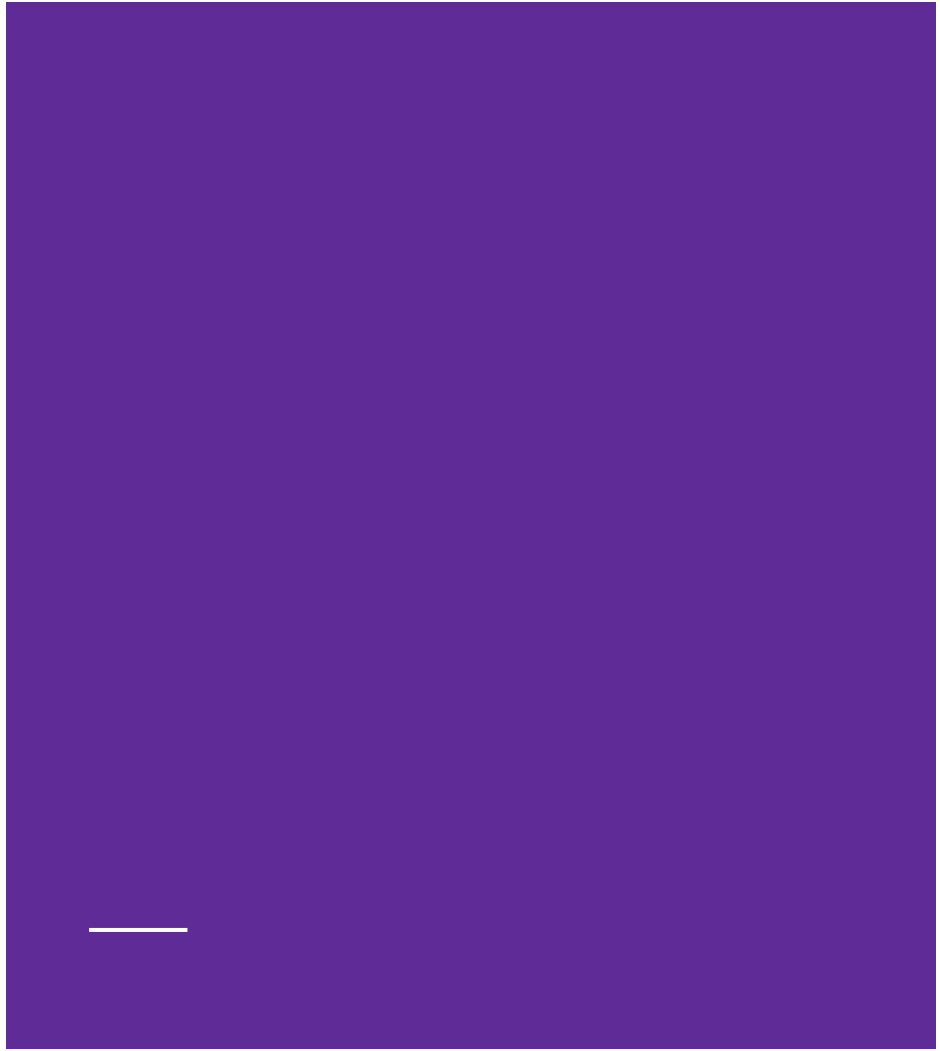
```
int* AllocateInt(int &x) {
  int* heapy_int = new int;
  *heapy_int = x;
  return heapy_int;
}
Point* AllocatePoint(int x, int y) {
  Point* heapy_point = malloc Point(x, y);
  return heapy_point;
}
int main(int argc, char** argv) {
  Point* x = AllocatePoint( 1, 2);
  int* y = AllocateInt( 3);
  cout << "x's x_ coordinate: " << x->get_x() <<
endl;
  cout << "distance between x and self: " <<
x->Distance(*x) << endl;
  cout << "y: " << y << ", *y: " << *y << endl;
  free x;
  delete y;
  return EXIT_SUCCESS;
}
```

CSE 374: Lecture 25

Inheritance

Constructors

ctor



Constructors

A constructor (ctor) initializes a newly-instantiated object

- A class can have multiple constructors that differ in parameters
  - Which one is invoked depends on *how* the object is instantiated

Written with the class name as the method name:

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

- C++ will automatically create a synthesized default constructor if you have *no* user-defined constructors
  - Takes no arguments, can be explicitly specified: `Point() = default;`

Synthesized Default Constructor

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

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

SimplePoint.h

```
#include "SimplePoint.h"
... // definitions for Distance() and SetLocation()

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

SimplePoint.cc

Synthesized Default Constructor

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

```
#include "SimplePoint.h"

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

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

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

Multiple Constructors (overloading)

```
#include "SimplePoint.h"

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

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

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

Initialization Lists

C++ lets you *optionally* declare an initialization list as part of a constructor definition

- Initializes fields according to parameters in the list
- The following two are (nearly) identical:

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

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

Body can be empty

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
```

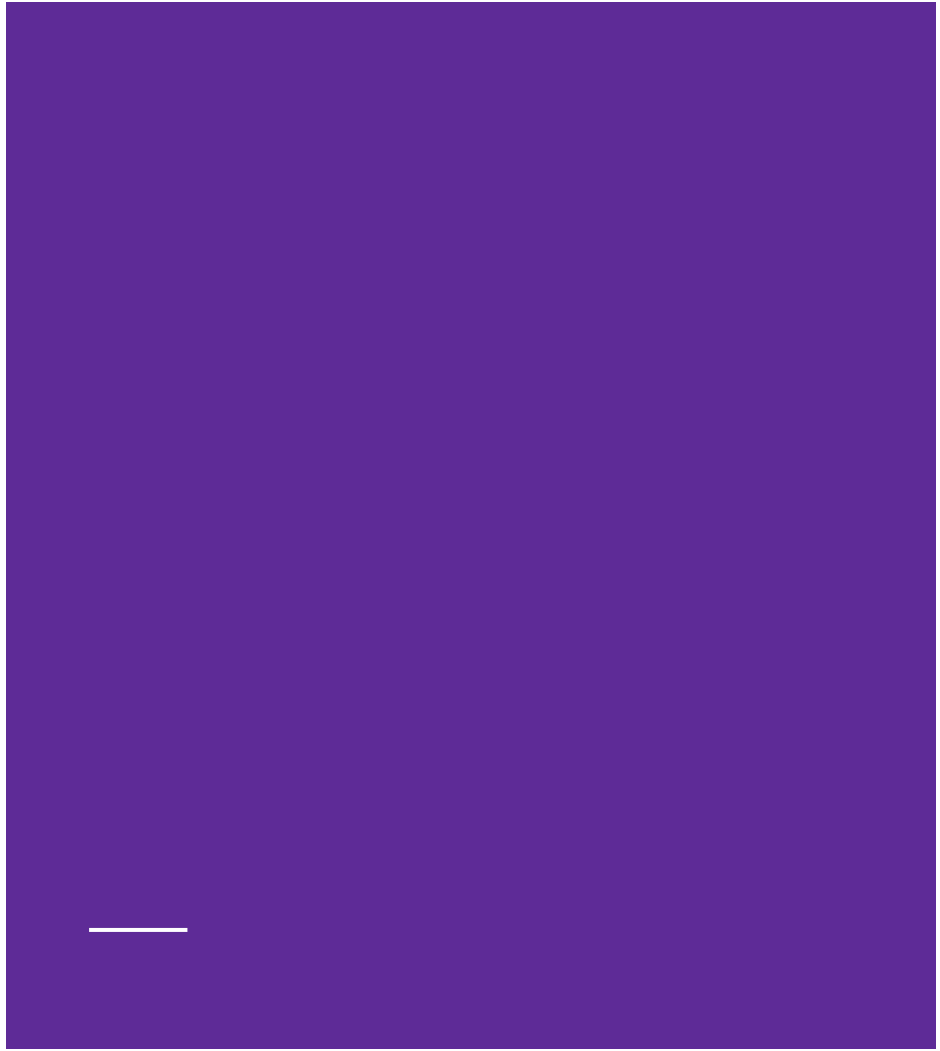
First, initialization list is applied.

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

Copy Constructors

cctor



Copy Constructors

C++ has the notion of a copy constructor (cctor)

- Used to create a new object as a copy of an existing object

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

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

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

Use a cctor since we are constructing based on x

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

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

Copy Constructors (w/ initialization list)

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

// copy constructor w/ initialization list
Point::Point(const Point& copyme): x_(copyme.x_), y_(copyme.y_) { }

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

Synthesized Copy Constructor

If you don't define your own copy constructor, C++ will synthesize one for you

- It will do a *shallow* copy of all of the fields (*i.e.* member variables) of your class
  - Does assignment for primitives; could be problematic with pointers
- Sometimes the right thing; sometimes the wrong thing

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

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

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

When Do Copies Happen?

The copy constructor is invoked if:

- You *initialize* an object from another object of the same type:

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

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

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

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

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

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

Assignment

Opt=

Assignment != Construction

"=" is the assignment operator

- Assigns values to an *existing, already constructed* object

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

Overloading the “=” Operator

You can choose to define the “=” operator

- But there are some rules you should follow:

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

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

More important when data members are dynamic memory

Should be a reference to *this to allow chaining

Bad style, just for demo

Synthesized Assignment Operator

If you don't define the assignment operator, C++ will synthesize one for you

- It will do a *shallow* copy of all of the fields (*i.e.* member variables) of your class
- Sometimes the right thing; sometimes the wrong thing
  - Usually wrong whenever a class has dynamically allocated data

```
#include "SimplePoint.h"

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

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

Class Constructors (4 types)

- A *default constructor* takes zero arguments. If you don't define any constructors for your class, the compiler will generate one of these constructors for you.
- A *copy constructor* takes a single parameter which is a *const reference* (const T&) to another object of the same type, and initializes the fields of the new object with a COPY of the fields in the referenced object.
- *User-defined constructors* initialize fields and take whatever arguments you like.
- *Conversion constructors* are constructors that take a single argument. For our string example this is like:

```
String(const char* raw);
String s = "foo";
```

Implicit constructors & destructors

Conversion constructors are implicit: automatically applied when a constructor is called with one argument.

If you want a single argument constructor that is not implicit, must use

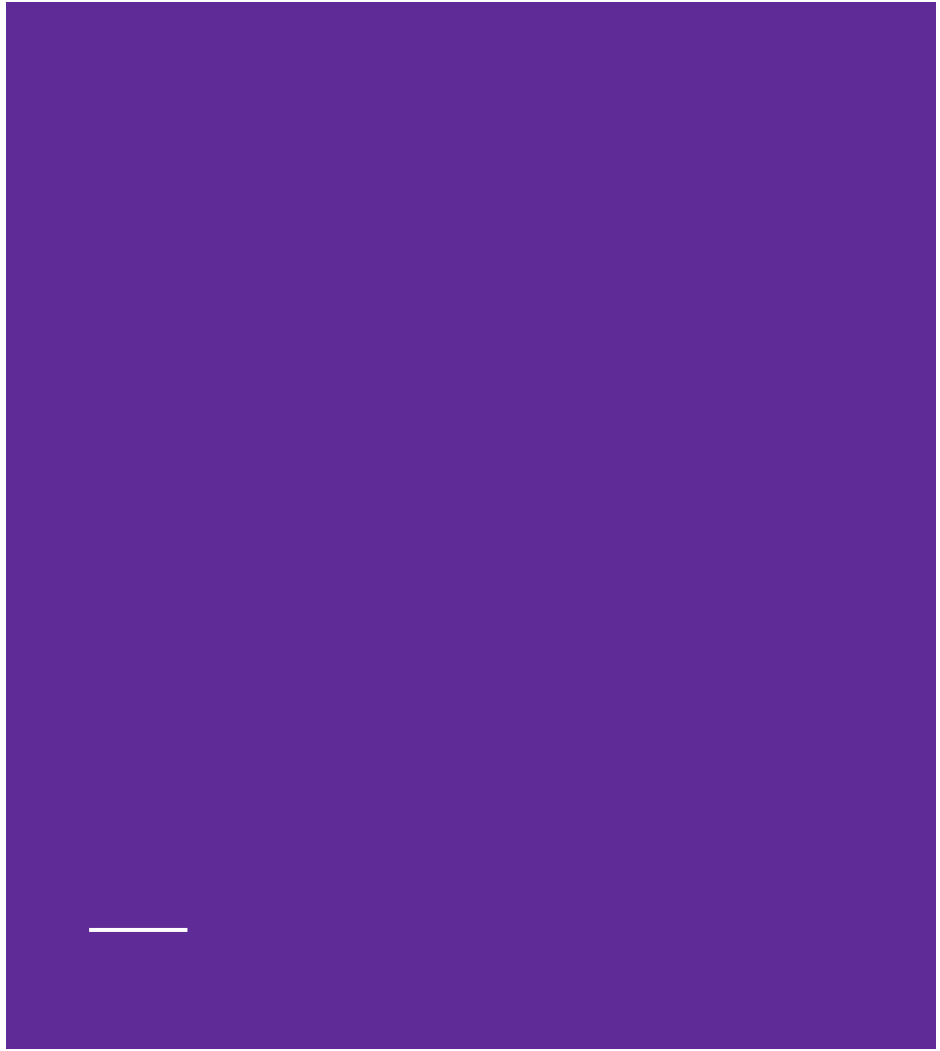
```
explicit String(const
char* raw);
```

Destructors are used by ‘delete’ to clean up when freeing memory.

```
Virtual ~String();
```

You do not call destructors explicitly

Destructors



Destructors

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

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

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

Rule of Three

If you define any of:

- Destructor
- Copy Constructor
- Assignment (`operator=`)

Then you should normally define all three

- Can explicitly ask for default synthesized versions (C++11):

```
class Point {
 public:
  Point() = default;                                  // the default ctor
  ~Point() = default;                                 // the default dtor
  Point(const Point& copyme) = default;               // the default cctor
  Point& operator=(const Point& rhs) = default; // the default "="
  ...
```

Other ways to control functionality

- C++ style guide tip:
  - If possible, disable the copy constructor and assignment operator if not needed – avoids implicit invocation and excessive copying. C++11 and later have direct syntax to indicate this:

```
class Point {
 public:
  Point(const int x, const int y) : x_(x), y_(y) { }   // ctor
  ...
  Point(const Point& copyme) = delete;   // declare cctor and "=" to
  Point& operator=(const Point& rhs) = delete; // be deleted (C++11)
 private:
  ...
};  // class Point

Point w;          // compiler error (no default constructor)
Point x(1, 2);    // OK!
Point y = w;      // compiler error (no copy constructor)
y = x;            // compiler error (no assignment operator)
```

Non-member Functions

"Non-member functions" are just normal functions that happen to use some class

- Called like a regular function instead of as a member of a class object instance

These do *not* have access to the class' private members

- Can access fields via getters (if they are there)

Useful non-member functions often included as part of interface of a class

- Declaration goes in header file, but **outside** of class definition
- Operators that are commutative should typically be non-members (non-commutative things can be non members too)

Example

Member function

```
double Point::distance(Point&)
pt1.distance(pt2);

float Vector::operator*(Vector&)
vec1 * vec2;
```

Non-member function

```
double distance(Point&, Point&)
distance(pt1, pt2);

float operator*(Vector&, Vector&)
vec1 * vec2;
```

Access Control

Access modifiers for members:

- `public`: accessible to *all* parts of the program
- `private`: accessible to the member functions of the class
- `protected`: accessible to member functions of the class and any *derived* classes (subclasses – more to come, later)

Reminders:

- Access modifiers apply to *all* members that follow until another access modifier is reached
- If no access modifier is specified, `struct` members default to `public` and `class` members default to `private`

Operator Overloading

Can overload operators using **member functions**

- Restriction: left-hand side argument must be a class you are implementing

```
Complex& operator+=(const Complex& a) { ... }
```

Can overload operators using **non-member functions**

- No restriction on arguments (can specify any two)
  - **Our only option** when the left-hand side is a class you do not have control over, like `ostream` or `istream`.
- But no access to private data members

```
Complex operator+(const Complex& a, const Complex& b) { ... }
```

`friend` non-member Functions

A class can give a **non-member** function (or class) access to its non-public members by declaring it as a `friend` within its definition

- Not a class member, but has access privileges as if it were
  - `friend` functions are usually unnecessary if your class includes appropriate "getter" public functions

Complex.h

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

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

Note: no Complex::

Complex.cc

When to use non-member and `friend`

Member Functions

- Operators that modify the object being called on
  - Assignment operator (`operator=`)
- "Core" non-operator functionality that is part of the class interface

Nonmember Functions

- Used for commutative operators
  - *e.g.*, so `v1 + v2` is invoked as `operator+(v1, v2)` instead of `v1.operator+(v2)`
- If operating on two types and the class is on the right-hand side
  - *e.g.*, `cin >> complex;`
- Returning a "new" object, not modifying an existing one
- Only grant `friend` permission if you NEED to, and if you are not modifying

Namespaces

Each namespace is a separate scope

- Useful for avoiding symbol collisions!

`LL:Iterator`
`HT:Iterator`
Same name, but different namespace

Namespace definition:

```
namespace name {
// declarations go here
}  // namespace name
```

- Doesn't end with a semicolon and doesn't add to the indentation of its contents
- Creates a new namespace name if it did not exist, otherwise **adds** *to the existing namespace* (**!**)
    - This means that components (*e.g.* classes, functions) of a namespace can be defined in multiple source files

Classes vs. Namespaces

They seems somewhat similar, but classes are *not* namespaces:

- There are no instances/objects of a namespace; a namespace is just a group of logically-related things (classes, functions, etc.)
- To access a member of a namespace, you must use the fully qualified name (*i.e.* `namespace_name::member`)
  - Unless you are `using` that namespace
  - You only used the fully qualified name of a class member when you are defining it outside of the scope of the class definition