Some details about polymorphism, function overriding, simple function overloading and encapsulation
C++ support for encapsulation (1)

- **public** names are accessible from outside of the class
- typically **public** names are:
  - methods needed by others to perform computations
  - methods to put values into a private field (**putters or accessors**)
  - methods to extract values from a private field (**getters or accessors**)
- **Nothing** should be public without a good reason

- protected names are accessible within the class, or objects of the class, or by classes (and their objects) that are derived from this it
  - Derivation can be immediate or indirect
- typically protected names are
  - helper functions whose results are needed not only to support the interface
  - data or state maintained by the object
  - included for efficiency
C++ support for encapsulation (1)

- *private* names are only accessible from inside the class or objects of the class or by friends
- typically *private* names are
  - helper functions whose results are not directly needed to support the interface
  - data or state maintained by the object
- C++ fields default to private, but it is a good idea to make this explicit for documentation purposes

- Access modifiers on inherited classes allows the inheriting class to reduce the visibility of fields and functions in the inherited class
Friend allows selective breaking of encapsulation in C++

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

class Y { 
private:
    X* x;
    int t;
public:
    Y(X* xobj) {x = xobj; t = x->m + x->n; }
    int get_t( ) { return t; }
};

void print(X* ptr) {cout << ptr->m << " " << ptr->n << endl;}

int main( ) {
    X* ptr = new X(100, 200);
    Y y(ptr);
    cout << y.get_t( ) << endl;
    print(ptr);
    return 0;
}
```

YHL/SPM ©2010 - 2014
#include <iostream>
using namespace std;

class Y;

class X {
private:
    int m;
    int n;
public:
    X(int mm, int nn) {m = mm; n = nn;}
friend class Y;
friend void print(X*);
};

class Y {
private:
    X* x;
    int t;
public:
    Y(X* xobj) {x = xobj; t = x->m + x->n; }
    int get_t() { return t; }
};

Allows functions in class Y (and objects of type Y) to access X’s private variables, functions, etc.

From cpp/friend/
```cpp
#include <iostream>
using namespace std;

class Y;

class X {
  private:
    int m;
    int n;
  public:
    X(int mm, int nn) {m = mm; n = nn;}

    friend class Y;
    friend void print(X*);
};

void print(X* ptr) {cout << ptr->m << " " << ptr->n << endl;}  
```
Creating Objects
```cpp
#include <iostream>
using namespace std;

class Y;

class X {
    int m;
    int n;
public:
    X(int mm, int nn) {m = mm; n = nn;}
    friend class Y;
    friend void print(X*);
};

int main() {
    X* ptr = new X(100, 200);
    Y y(ptr);
    cout << y.get_t() << endl;
    print(ptr);
    return 0;
}
```

Creates a variable \( y \) on the runtime stack that is a \( Y \) object.

Creates an \( X \) object on the heap pointed to by \( \text{ptr} \).
A forward reference to reference objects

• There are also objects called *reference objects* in C++. We will discuss these later.
Polymorphism

• Polymorphism is one of the most important concepts in OO programming, and provides much of the power of OO programming

• Polymorphism is the ability of objects to behave as more (poly) than one type (morphism - denoting something having a specified form or character)

• Java has a much cleaner implementation of polymorphism than C++
  – Polymorphism only works on virtual functions.
  – In Java, all non-static functions are virtual, i.e., all methods are functions
  – In C++ functions are not virtual, by default
Virtual Functions in C++

• In C++ a function can support polymorphism only if it is
  – Declared **virtual in the least derived class** and
  – The **object** is accessed via a pointer or a reference

    ClassNameDerived **someObject**(*arguments*);

    ClassName * obj = new ClassNameDerived(*arguments*);

    obj->someVirtualFunction( );

    ClassName * obj = &**someObject**;

    obj->someVirtualFunction( );

• This does not give polymorphism when **someObject** is an object variable.
  **someObject**.someVirtualFunction( );
Virtual in C++

- A virtual method in a derived class must have the same signature (i.e. name and argument types) as the base class method to override it (this is a little simplistic, as we will see in slide 25).
  - If *signatures* are the same *and* the return types are both different and primitives (e.g. float, int), it is an error
  - If *signatures* are the same and the return type is pointer or reference, the return type of the overriding method can be a derived class of the original type
Overriding and virtual/non-virtual functions

• virtual: the derived class is not required to override, but can
• not virtual: derived class should not override

– Having a non-virtual function in a base class is the base class programmer's way of saying "please don't override this"

– compiler will allow this, but don’t ask for trouble - you are violating wishes of the base class implementer and the interface.

– Because it is non-virtual, calls will not be made through the VFT. What is called will depend on how the call is made. For example, calling through different pointer types can lead to different methods being called.
Why non-virtual by default in C++

• why non-virtual default in C++?
  – slightly better performance for non-virtual but . . .
    • Makes programming more complicated
    • The overhead of virtual function invocation can sometimes be optimized away by the compiler
  – Goals of C++ were performance, compatibility with existing C code base, ability to write OO programs
  – Compatibility with C code and its non-virtual functions
• Then why virtual by default in Java?
  – Java had as a primary goal safety and easy of programming and a clean slate design
  – C++ had as primary goals performance, compatibility with existing C code base, ability (but not requirement) to write OO programs
Some caveats

• In general, **functions in C++ should be virtual** unless you have strong reasons for making them non-virtual (and know/understand what you are doing)
• **Constructors** are not virtual and virtual constructors don't really make sense
• **Destructors** should always be virtual (more about this later)
  – Not doing this can lead to memory leaks
Using polymorphism in C++

• Polymorphism is enabled (i.e. the derived class functions are used even if referenced as a base type) **if and only if:**

1. The objects are manipulated through pointers or *references* [1]
2. the function is declared as *virtual* in the class that is the class of the *reference type* or *pointer type*. Note that an inherited virtual function remains virtual.

• You must do both of these to get polymorphic behavior

[1] C++ references are *not the same* as Java references, and we will discuss them later.
```cpp
#ifndef FOO_H_
#define FOO_H_
#include <string>
using namespace std;
class Foo
{
public:
    Foo(string ln);
    virtual ~Foo();
    virtual void print();

protected:
    const string fooString;
};
#endif /*FOO_H_*/

#ifndef DFOO_H_
#define DFOO_H_
#include "Foo.h"
#include <string>
using namespace std;
class DFoo : public Foo
{
public:
    DFoo(string ln);
    virtual ~DFoo();
    virtual void print(); // this is virtual

protected:
    const string fooString;
};
#endif /*DFOO_H_*/
```
Implementations of Foo and DFoo classes (.cpp files)

No need to mention virtual

```cpp
#include "Foo.h"
#include <iostream>

Foo::Foo(string ln) : fooString(ln) { }
Foo::~Foo() { }
void Foo::print() {
    cout << "Foo: " << fooString << endl;
    cout << endl;
}

DFoo::DFoo(string ln) :
    fooString(ln) { }
DFoo::~DFoo() { }
void DFoo::print() {
    cout << "DFoo: " << fooString << endl;
    cout << endl;
}
```
void test(DFoo f, Foo *p) {
    cout << "in test, f.print( )" << endl;
    ((Foo) f).print( ); // call from a variable of type Foo
    cout << "in test, p->print( )" << endl;
    p->print( ); // call from a pointer to a Foo
}

int main(int argc, char * argv[]) {
    Foo *f = new Foo("Foo object");
    DFoo *d = new DFoo("DFoo object");
    test(*d, d);
}

• test receives:
  • f: a DFoo object
  • p: a pointer to the same DFoo object

From cpp/goodPoly/
C++ polymorphism example

using namespace std;

void test(DFoo f, Foo *p) {
    cout << "in test, f.print( )" << endl;
    ((Foo) f).print();
    cout << "in test, p->print( )" << endl;
    p->print();
}

int main(int argc, char * argv[]) {
    Foo *f = new Foo("Foo object");
    DFoo *d = new DFoo("DFoo object");
    test(*d, d);
}

From cpp/goodPoly/

in test, f.print( )

Foo: DFoo object

The DFoo object is not being accessed through a pointer, therefore the print called is based on the type the call is made from (Foo via a cast). Polymorphism does not hold. Thus Foo's print is invoked.
C++ polymorphism example

using namespace std;

void test(DFoo f, Foo *p) {
    cout << "in test, f.print( )" << endl;
    ((Foo) f).print( );
    cout << "in test, p->print( )" << endl;
    p->print( );
}

int main(int argc, char * argv[]) {
    Foo *f = new Foo("Foo object");
    DFoo *d = new DFoo("DFoo object");
    test(*d, d);
}

in test, p->print( )
DFoo: DFoo object

The DFoo object is accessed through a pointer, therefore polymorphism holds.

DFoo's print called.

In both cases the object being accessed is a DFoo object or part of a DFoo object.
int main(int argc, char * argv[]) {

Foo *f = new Foo("Foo object");
DFoo *d = new DFoo("DFoo object");

. . .
((Foo) *d).print();
((Foo *) d)->print();

Foo* fooArray[2] = {f, d}; // array of ptrs
for (int i = 0; i < 2; i++) {
    fooArray[i]->print();
}

return 0;
}

C++ polymorphism example

Foo: DFoo object
Not through a pointer, no polymorphism

DFoo: DFoo object
Through a pointer, polymorphism

A Foo
Base object, base object

A DFoo
DFoo: DFoo object
Through a pointer, polymorphism

From cpp/goodPoly/
Some Gotcha's
Some Gotcha's

What if non-virtual functions are overridden?

• Base declares

```cpp
void foo() {cout << "in base foo" << endl;}
```

• Derived declared

```cpp
void foo() {cout << "in derived foo" << endl;}
```

• code looks like:

```cpp
int main (...) {
    B* b;
    D* d = new Derived();
    b = d;
    b->foo(); // prints in base foo
    d->foo(); // prints in derived foo
    // if virtual foo, both invocations would print in derived foo, but in overridden
    // non-virtual functions, the pointer type says what class’s function is used to
    // determine if function is virtual
```
Some Gotcha's (new slide)
What if non-virtual functions are overridden as virtual?

- Base declares `void foo( ) {cout << "in base foo" << endl;}`
- Derived1 declared `virtual void foo( ) {cout << "in derived foo" << endl;}`
- Derived2 declared `void foo( ) {cout << "in derived foo" << endl;}`
- code looks like:

```cpp
int main (...) {
    B* b;
    D1* d1;
    D2* d2 = new Derived2( );
    b = d2; d1 = d2;
    b->foo( ); // prints in base foo
    d1->foo( ); // prints in Derived2 foo
    d2->foo( ); // prints in Derived2 foo
```

class Base {
public:
    virtual void f(double x);  // doesn't matter whether or not this is virtual
};

class Derived : public Base {
public:
    virtual void f(char c);  // doesn't matter whether or not this is virtual
};

int main() {
    Derived* d = new Derived();
    Base* b = d;
    b->f(65.3);  // okay: passes 65.3 to f(double x)
    d->f(65.3);  // converts 65.3 to a char ('A' if ASCII) and passes it to f(char c);
    // does NOT call f(double x)!!
}

This is what I was referring to in slide 12 when I said it was a little simplistic that arg types had to be the same.
Why C++ allows this problem to exist

• There are two possible targets for the call to function $f$ - the base function and the derived function

• **Base function case** (i.e., function only declared in base function):
  – If through a pointer and not virtual, call the base function (normal)
  – If a base function pointer, and $f$ is virtual, nothing overrides the base virtual function, so call it

• **Derived function case** (i.e., function may be declared in base and derived):
  – If through a pointer and $f$ is not virtual, call the derived function, as always
  – If a derived function pointer, and $f$ is virtual look for the closest match in the *derived class*, and go for it, since presumably the interface extension was meant to be used. (Java philosophy assumes augmenting interface, C++ replacing, if same function/method name used.)
How to prevent hiding

• If you declare a function in a derived class with
  – the same name as some base virtual function type
  – but with different signature

• *then override all forms of the virtual function.*

• You have extended the interface, and should provide valid
  implementations for the new interface.

• In the previous example, Derived should define both
  1. virtual void f(char c); // already defined
  2. virtual void f(double x); // not defined in Derived class, but should be
In C++ you can still call the base function

- The new implementation can call `Base::f(float)`, e.g. `Base::f(65.3);`
- Use the `using` keyword if supported by your compiler

```cpp
class Derived : public Base {
public:
    using Base::f;  // this un-hides `Base::f(double x)`. Now `f(65.3)` on
                      // Derived object will call the Base class
                      // `f(double x);`
    void f(char c);
};
```
Downcasts or specializing casts

- The downcast shown is illegal
- Java will give a compile time error if it can, and will perform a runtime check if it cannot do a compile time check
- C++ allows you to use (may need to set options)
  - `static_casts` if you know that the cast is correct
  - `dynamic_casts` if you want C++ to do a runtime check. If the runtime check fails a null pointer is returned from the cast.

```java
class Test {
    public static void main(String args[]) {
        Derived d1 = new Derived();
        Derived2 d2 = new Derived2();
        Base b2 = d1;
        ...
        ((Derived2) b2).print2();
    }
}
```

```cpp
MyClass *m = (MyClass *)ptr;
MyClass *m = static_cast<MyClass *>(ptr);
MyClass *m = dynamic_cast<MyClass *>(ptr);
```
Overloading in C++

class User {
public:
    User(string nm, int a);
    User();
    void print();

    "string foo(string s, int i);
    "string foo(string s1, string s2);
    "string foo(int, int);

    virtual ~User();

private:
    string intToString(int i);
    string name; // defaults to private, say
    int age;     // "private:" to make this explicit
};

string User::intToString(int i) {
    ostringstream strStream;
    strStream << i;
    return strStream.str();
}

string User::foo(string s, int i) {
    cout << "User::foo called with string and int arguments" << endl;
    return s + intToString(i);
}

string User::foo(string s1, string s2) {
    cout << "User::foo called with string and string arguments" << endl;
    return s1 + s2;
}

string User::foo(int i, int j) {
    cout << "User::foo called with int and int arguments" << endl;
    return intToString(i) + intToString(j);
}
class User
{
public:
  User(string nm, int a);
  User();
  void print();
  string foo(string s, int i);
  string foo(string s1, string s2);
  string foo(int, int);
  virtual ~User();
private:
  string intToString(int i);
  string name; // defaults to private, say
  int age;    // "private:" to make this explicit
};

string User::intToString(int i) {
  ostringstream strStream;
  strStream << i;
  return strStream.str();
}

Not overloading, but this can be handy
Overloading in C++

```cpp
string User::foo(string s, int i) {
    cout << "User::foo called with string and int arguments" << endl;
    return s+intToString(i);
}

string User::foo(string s1, string s2) {
    cout << "User::foo called with string and string arguments" << endl;
    return s1+s2;
}

string User::foo(int i, int j) {
    cout << "User::foo called with int and int arguments" << endl;
    return intToString(i) + intToString(j);
}
```
Overloading in C++

```cpp
string User::foo(string s, int i) {
    cout << "User::foo called with string and int arguments" " endl;
    return s + intToString(i);
}

string User::foo(string s1, string s2) {
    cout << "User::foo called with string and string arguments" " endl;
    return s1 + s2;
}

string User::foo(int i, int j) {
    cout << "User::foo called with int and int arguments" " endl;
    return intToString(i) + intToString(j);
}
```

For future reference, + is also overload within String as part of the String implementation.
Invoking overloaded functions

```cpp
int main(int argc, char * argv[]) {
    string s1 = "aa";
    string s2 = "bb";
    User u = User();
    cout << u.foo("aa", -3) << endl;
    cout << u.foo("aa", "bb") << endl;
    cout << u.foo(50, -11) << endl;
}
```

If no exact match rules to pick are more complicated than Java’s. We will discuss these in detail later.

User::foo called with string and int arguments
aa-3
User::foo called with string and string arguments
aabb
User::foo called with int and int arguments
50-11
The \textit{this} pointer

- It really has to do with invoking non-static functions

- When the method invocation $o->funcy(arg)$; happens, what is really called is \textit{funcy(arg, o)}

- This is all done automatically -- you don’t need to declare this as an argument, and you don’t need to pass it. \textit{this} is used to access object fields.

Obj o = new Obj(x);
o->funcy(arg);
public class C {
    private k;
    void setK(int k) {
        k = k;
    }
}

Green refers to the parameter $k$, blue is the class member variable $k$
Static members and methods
Static members (fields)

class StudentUser : public User
{
public:
    StudentUser(string nam, int y, string school);
    virtual ~StudentUser();
    int f0;
    int f1
private:
    int f2;
    int f3
};

• In the code to the left, each created object will contain storage for f0, f1, f2 and f3
• What if we want to have one one copy shared for every object in the class?
• static members allow that (same as Java)
Static members (fields)

In the code to the left, each created object contains space for f0, f1, f2 and f3

What if we want to have one one copy shared for every object in the class?

static members allow that

f1 and f2 are now static members

The syntax is the same for C++ and Java

Can reference through the class (Student.f1, Student::f1 or through objects (s1.f2 = 5; sPtr->f2 = 5;)

class StudentUser : public User
{
 public:
   StudentUser(string nam, int y, string school);
   virtual ~StudentUser();
   int f0;
   static int f1

 private:
   static int f2;
   int f3;
};
C++ statics and initializers

• Cannot put statics into constructor initializer lists
• Does anyone know why?
C++ statics and initializers

• Cannot put statics into constructor initializer lists
• Does anyone know why?

Static variables can be initialized in a single translation unit outside the class declaration* and they can be assigned to in the constructor. That is, they need to be initialized during the class not object, definition, but can be changed during constructor execution.
Static methods

• Static methods are associated with the class and *not* an object
• Static methods do not have access to a *this* pointer since they are not associated with any object
• Since static methods do not have access to a *this* pointer, they cannot access object members (since they are associated with no object, it is not clear whose object's members they would access.
• In C++, static methods within a class can be declared like by:

  ```c++
  static int GetValue();
  ```
Simple constructors and using constructors and *final* to control inheritance
Constructors

• Constructors are the functions that create an object, i.e. an instance of the class
• Storage is allocated for an object on the stack frame (object variables in C++) or on the heap (new in C++ and Java).
• Constructors initialize the storage for an object storage.
• Initialization can happen by default, and thus constructors perform both system and programmer specified actions
• Let’s look at some examples in C++ and then Java
C++ constructor example

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

class User
{
    public:
        User(string nm, int a);
        void print( );
        virtual ~User();
    private:
        string name;
        int age;
};

User::User(string nm, int a) {
    name = nm; age=a;
}

void User::print( ) {
    cout << "Name: " << name <<", Age: " << age;
}

User::~User( ) { }
```

In codew2b/cpp/constructor & .../ref
C++ constructor example (2)

```cpp
#include "User.h"
#include "StudentUser.h"
#include <iostream>
using namespace std;

class StudentUser : public User
{
public:
    StudentUser(string name, int y, string school) : User(name, y)
    {
        schoolEnrolled = sch;
    }
    void print( )
    {
        User::print( );
        cout << " School enrolled: " << schoolEnrolled << endl;
    }
    virtual ~StudentUser();
private:
    string schoolEnrolled;
};

#include "User.h"
#include "StudentUser.h"
#include <iostream>
using namespace std;

StudentUser::StudentUser(string name, int y, string sch) : User(name, y) 
{
    schoolEnrolled = sch;
}

void StudentUser::print( )
{
    User::print( );
    cout << " School enrolled: " << schoolEnrolled << endl;
}

StudentUser::~StudentUser( )
{
}
```

What if the base class constructor is not called in the initializer list?

• A call to a zero-arg constructor `base::base()` is automatically supplied
  – This happens even if a base class constructor is called in the derived class constructor `body`
  – In this case, both the zero-arg constructor *and* the constructor in the constructor body are called!

• If the zero-arg constructor is not declared, a *declaration* is made for the constructor. The declaration looks like `base();`
• If no other constructors for base are declared, a default definition `base() {};` is provided for the zero-arg constructor
• if other constructors are declared
  – No default definition is provided
  – You will get an error message saying undefined constructor
What if the base class constructor is not called in the initializer list?

- A call to a zero-arg constructor \texttt{base::\texttt{base}()} is automatically supplied
  - This happens even if a base class constructor is called in the derived class constructor body
  - In this case, both the zero-arg constructor \texttt{and} the constructor in the constructor body are called!

- If the zero-arg constructor is not declared, a \textit{declaration} is made for the constructor. The declaration looks like \texttt{base();}

- If no other constructors for base are declared, a default definition \texttt{base()} \{\} \; is provided for the zero-arg constructor

- if other constructors are declared
  - No default definition is provided
  - You will get an error message saying undefined constructor
A default zero-arg constructor is declared by the compiler, but no definition is given because other constructors are defined for User.
Eliminating User(string, int) constructor makes the compile-time error go away

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

// User::User(string nm, int a) {
//    name = nm; age=a;
// }

void User::print( ) {
    cout << "Name: " << name <<", Age: " << age;
}

User::~User( ) { };
```

g++ main.cpp User.cpp StudentUser.cpp
[ece-76-55:codew2b/cpp/zeroarg] smidkiff% ./a.out
Name: , Age: 0 School enrolled: Bug Tech
[ece-76-55:codew2b/cpp/zeroarg] smidkiff%
```
This will run “ok”

• No base constructor declared

• C++ will supply both a declaration and a definition

• C++ will, in a system dependent way, initialize each base field to the zero value for that type (e.g. generally 0 for integer, null for a string, etc.)

  – If it is written "system dependent way" read "may not be portable"

• This is a bad thing to do since you really want to know what your program will do on any machine and operating system

• And one suspects the programmer wanted the User::User(string nm, int a) {...} constructor for some reason.
Initializer lists

• Foo::Foo(Bar v) : Base(v), x(v) { . . . }
• Base(v), x(v) are called an initialization list or an initializer list.
• Actions in the initializer list are performed before the body of the constructor, i.e. before what is in the { . . . }

• Why do this?
Why use initializer lists in C++ constructors?

• Four common reasons:
  1. It allows a non-default base constructor to be called early.
  2. When initializing an object attribute (i.e. field) that is an object it is more efficient than an assignment
     • \( x = v \); may execute code that copies \( v \) into a temporary, and copies the temporary into \( x \) (compilers try to clean this up, be are not always successful)
     • \( x(v) \)" directly copies \( v \) value into \( x \)'s space
  3. Easy to see where object attributes are initialized
  4. The only way to initialize const refs (more on const variables later) and reference members (more on references later)
In what order are initializations done?

- The direct (immediate) base class constructor is called
- Non-static member initializations are performed in the order that the fields are declared, *not the order the field name appears in the initializer list*
- **What about static members?**

```c++
class StudentUser : public User
{
    public:
        StudentUser(string nam, int y, string school);
    virtual ~StudentUser();
    int f0;
    int f1
    private:
        int f2;
        int f3
};

    Student::StudentUser(int p0, int p1, int p2, int p3)
    : f2(p2), f3(p3), f1(p1), f0(p0) {}
Preventing and controlling inheritance

• Already know about protected, private and public in C++
• Also private, public and package in Java
• Also would like to force certain functions to be inherited and not overridden
• C++
  – make functions non-virtual, but this can be gotten around (see polymorphism examples)
  – can make the constructors private to force the entire class to be non-extendable!
  – This works for the following reason.
    • Consider class S with private constructors.
    • Consider class E that tries to extend it (i.e. class E will be derived from S).
    • Class E’s constructors must call S constructors (even if just zero-arg), but cannot since they are private.
    • Thus inheritance is prevented.
Preventing inheritance

```cpp
#ifndef USER_H_
#define USER_H_
#include <string>
using namespace std;

class User
{
    public:
        void print();
        virtual ~User();
    
    private:
        User(string nm, int a);
        User();
        string name;
        int age;

};
#endif /*USER_H_*/
```

User.h: In constructor `StudentUser::StudentUser(std::string, int, std::string)`: User.h:13: error: `User::User()` is private
Studentuser.cpp:7: error: within this context
[ece-76-55:codew2b/cpp/constPrivate] smidkiff

see /codew2b/cpp/constPrivate
How does one construct an object with a private constructor?

```cpp
#include <cmath>  // To get std::sin() and std::cos()

class Point {
public:
    static Point rectangular(float x, float y);  // Rectangular coord's
    static Point polar(float radius, float angle);  // Polar coordinates
    // These static methods are the so-called "named constructors"
    ...
private:
    Point(float x, float y);  // Rectangular coordinates the constructor
    float x_, y_;     
};
```
Constructing objects with private constructors

#include <cmath>     // To get std::sin() and std::cos()
class Point {
public:
    static Point rectangular(float x, float y);   // Rectangular coord's
    static Point polar(float radius, float angle);  // Polar coordinates
    // These static methods are the so-called "named constructors"
private:
    Point(float x, float y);   // Rectangular coordinates
    float x_, y_;
};
inline Point::Point(float x, float y) : x_(x), y_(y) {}

inline Point Point::rectangular(float x, float y) { return Point(x, y); }
inline Point Point::polar(float radius, float angle) { return Point(radius*std::cos(angle), radius*std::sin(angle)); }

Objects are constructed by calling a named constructor. This also allows us to "overload" constructor names/ signatures. Thus, we can have a named constructor for polar and rectangular coordinates with the same signature, but cannot have two actual constructors with the same signature.
Can also be used to create singleton objects

```cpp
#include <cmath>               // To get std::sin() and std::cos()

class Point {
public:
    static Point rectangular(float x, float y);
    static Point polar(float radius, float angle);      ...
private:
    Point(float x, float y);     // Rectangular coordinates
    float x_, y_;               
    static Point rectP; // these have been added to the class definition
    static Point polarP;
};
```
Can also be used to create singleton objects

Point::rectP = null; // why not put this into the .h file?
Point::polarP = null; // why not put this into the .h file?

**inline** Point::Point(float x, float y) : x_(x), y_(y) {
}

inline Point Point::rectangular(float x, float y) {
    if (rectP == null) {
        rectP = Point(x, y); return rectP;
    } else return rectP;
}

inline Point Point::polar(float radius, float angle) {
    if (polarP == null) {
        polarP = Point(radius*std::cos(angle), radius*std::sin(angle));
        return polarP;
    } else return polarP;
}
Some people consider singletons evil

• One reason -- create dependences across users of the singleton
  – this makes unit testing hard, since the state of the singleton object can necessarily depend on everyone that accesses it
  – acts as a global variable
  – sometimes more than one object is needed

• Singleton is an example of a pattern. See the Gang of Four (GoF) book Design Patterns: Elements of Reusable Object-Oriented Software by Gamma, Helm, Johnson and Vlissides
  – I will try and briefly cover patterns later in the course.
Abstract Classes
Abstract classes

• Abstract classes are classes for which objects cannot be constructed
• They can be derived from, however
Good for 3 things

1. Can lend organization to a class hierarchy,
2. Provides a common base class
3. Can represent a specialized behavior that when mixed with other classes gives a desired behavior

Can help build up an implementation

Let’s look at a concrete example to make these concepts clearer. In particular, let’s look at a shape class such as might be used in a drawing program
A shape class

- It makes sense to construct a Circle, a Rectangle, etc., but not necessarily a shape
- It is useful to be able to refer to all shapes with a common class
The Shape abstract class (C++)

class Shape {
public:
  virtual double area() = 0;
  virtual double circumference() = 0;
  // ... 
};

A C++ abstract class, which can also contain functions that contain code and variables

- It also makes sense for Shape to insists on certain things be computed for any shape
Can now write code like

```c++
Shape* shapes[3];
shapes[0] = new Circle( ... );
shapes[1] = new Rectangle( ... );
shapes[2] = new Rectangle( ... );

double totArea = 0;
for (int i=0; i < 3; i++) {
    totArea += shapes[i]->area( );
}
```

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

class Shape {
public:
    virtual double area() = 0;
    virtual double circumference() = 0;
};

class Circle : public Shape {
protected:
    double r;
    static double PI;
public:
    Circle() { r = 1.0; }
    Circle(double r) { this->r = r; }
    double area() { return PI*r*r; }
    double circumference() { return 2*PI*r; }
    double getRadius() { return r; }
}

double Circle::PI = 3.14159265358979323846;
```
```cpp
#include <iostream>
using namespace std;
class Shape {
public:
    virtual double double() = 0;
    virtual double circumference() = 0;
};
class Rectangle : public Shape {
    double w, h;
public:
    Rectangle() {w=0; h = 0.0;}
    Rectangle(double w, double h) {
        this->w = w; this->h = h;
    }
    double area() {return w*h;}
    double circumference() {
        return 2*(w+h);
    }
    double getWidth() {return w;}
    double getHeight() {return h;}
};
```
int main( ) {
    Shape* shapes[3];
    shapes[0] = new Circle(2.0);
    shapes[1] = new Rectangle(1.0, 3.0);
    shapes[2] = new Rectangle(4.0, 2.0);
    double totArea = 0;
    for (int i=0; i < 3; i++)
        totArea += shapes[i]->area( );
    cout << "Total area = " << totArea << end;
    return 0;
}