Abstract classes, casts, overloading, static methods and members in C++

Abstract classes

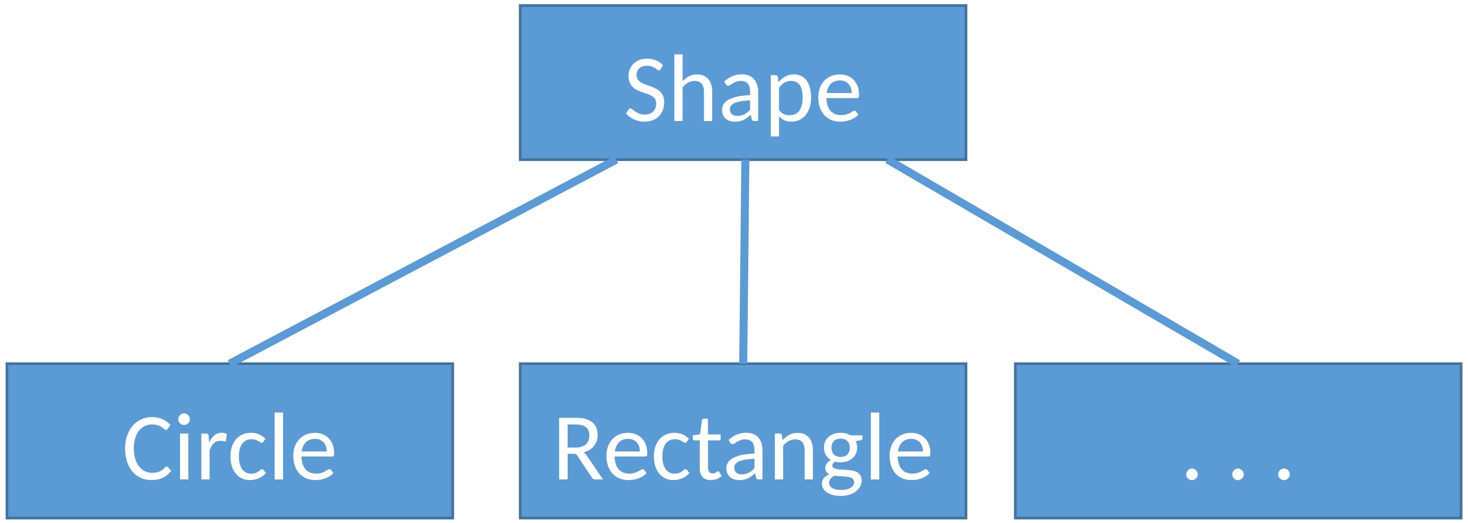
- Abstract classes are classes for which objects cannot be constructed
- They can be inherited from, however
- What are they good for?

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, Rectangle, etc., but not an amorphous shape
- However, it may be useful to have arrays of shapes to hold different kinds of shapes
- And it may be useful to *know* that every shape has a perimeter, area, color, etc., property implemented
- Abstract classes allow this

The Shape abstract class (see Abstract)

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

- The Shape abstract class requires any class that inherits from it to implement area and circumference.
- This lets us call these functions on any class that extends Shape.
- C++ abstract classes can also declare variables and non-abstract functions

The Square class (Circle is similar, see Abstract.zip)

```
class Square : public Shape {
public:
   Square(float);
   ~Square( );
   double area( );
   double circumference();
private:
   float side;
};
```

```
Square::Square(float s) : side(s) { };
Square::~Square( ) { }
double Square::area( ) {
   return side*side;
}

double Square::circumference() {
   return 4.0*side;
}
```

What happens if we don't declare/define area?

```
class Circle : public Shape {
public:
   Circle(float);
   ~Circle( );
   // double area( );
   double circumference();
   static const double
     PI=3.141592653589;
private:
   float radius;
};
```

```
Circle::Circle(float r) : radius(r) { }
Circle::~Circle( ) { }
// double Circle::area( ) {
//   return Circle::PI*radius*radius;
//}

double Circle::circumference( ) {
   return 2*3.14*radius;
}
```

G++ Circle.cpp compiles ok

The error shows up when we try and instantiate an object.

```
g++ main.cpp
main.cpp: In function ‘int main()’:
main.cpp:11:30: error: cannot allocate an object of abstract type ‘Circle’
    shapes[1] = new Circle(4.0);
                             ^
In file included from main.cpp:2:0:
Circle.h:5:7: note:   because the following virtual functions are pure within ‘Circle’:
 class Circle : public Shape {
       ^
In file included from Square.h:3:0,
                 from main.cpp:1:
Shape.h:5:19: note:     virtual double Shape::area()
    virtual double area( ) = 0;
```

A better way to do DuckSim

```
#ifndef FLYBEHAVIOR_H_
#define FLYBEHAVIOR_H_
#include <string>

class FlyBehavior {
public:

   FlyBehavior( );
   virtual ~FlyBehavior( );
   virtual void fly( );
};
#endif /* FLYBEHAVIOR_H_ */
```

```
#include <iostream>
#include <string>
#include "FlyBehavior.h"

FlyBehavior::FlyBehavior( ) { }

FlyBehavior::~FlyBehavior( ){ }

void FlyBehavior::fly( ) { }
```

A better way to do DuckSim

```
#include <iostream>
#include <string>
#include "FlyBehavior.h"

FlyBehavior::FlyBehavior( ) { }

FlyBehavior::~FlyBehavior( ){ }

void FlyBehavior::fly( ) { }
```

- What is good about this:
  - We have a base class that we can extend to enable different behaviors
  - The base class serves as a common type for all of the derived classes, allowing us to use a pointer of type FlyBehavior to refer to lots of different objects of different derived types that specify a FlyBehavior.

- What is bad about this:
  - We can extend this class without redefining the FlyBehavior, i.e., we just use the FlyBehavior class behavior, which is nothing
  - There is not way to force the programmer to redefine the behavior

What if the fly method was not overwritten in NoFly, i.e., the code below becomes the code on the next slide

```
#ifndef NOFLY_H_
#define NOFLY_H_
#include "FlyBehavior.h"

class NoFly : public FlyBehavior {
public:

   NoFly( );
   virtual ~NoFly( );
   void fly( );

};
#endif /* NOFLY_H_ */
```

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


NoFly::NoFly( ) { }
NoFly::~NoFly( ) { }
void NoFly::fly( ) {
   std::cout << "I cannot fly!" << std::endl;
}
```

The fly method is not overridden, and this could only be caught when we are testing the program

```
#ifndef NOFLY_H_
#define NOFLY_H_
#include "FlyBehavior.h"

class NoFly : public FlyBehavior {
public:

   NoFly( );
   virtual ~NoFly( );

};
#endif /* NOFLY_H_ */
```

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


NoFly::NoFly( ) { }
NoFly::~NoFly( ) { }
```

An even better way to do DuckSim

```
#ifndef FLYBEHAVIOR_H_
#define FLYBEHAVIOR_H_
#include <string>

class FlyBehavior {
public:

   FlyBehavior( );
   virtual ~FlyBehavior( );
   virtual void fly( ) = 0;
};
#endif /* FLYBEHAVIOR_H_ */
```

```
#include <iostream>
#include <string>
#include "FlyBehavior.h"

FlyBehavior::FlyBehavior( ) { }

FlyBehavior::~FlyBehavior( ){ }
```

If the fly method is not overridden in a derived class, and we try and create an object of that derived class, we'll get a compile time error.

```
#ifndef NOFLY_H_
#define NOFLY_H_
#include "FlyBehavior.h"

class NoFly : public FlyBehavior {
public:

   NoFly( );
   virtual ~NoFly( );

};
#endif /* NOFLY_H_ */
```

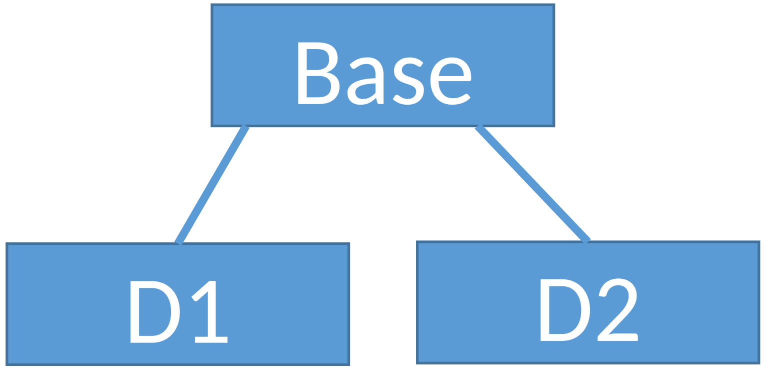
```
#include <iostream>
#include "NoFly.h"


NoFly::NoFly( ) { }
NoFly::~NoFly( ) { }
```

```
g++ main.cpp
main.cpp: In Constructor 'Rubber( )':
RubberDuck.cpp: error: cannot allocate an object of
abstract type 'NoFfly'
   setFlyBehavior(new NoFly( ));
```

Casts

- C basically has one cast, i.e. *x = (mystruct*) y;*
- C++ has a wide variety of casts
- C style casts – not recommended. No runtime checks, no conversions on pointers which depending on how objects are implemented may be necessary
- <static_cast> - casts between types that are clearly illegal cause compile time errors, but no runtime checks are used. No conversions take place
- <dynamic_cast> - for casts between objects. Casts from non-object pointers to object pointers cause a compile time error. The validity of the cast is checked with a runtime test. No conversions take place.
- Various other casts – see https://stackoverflow.com/questions/28002/regular-cast-vs-static-cast-vs-dynamic-cast

Examples



D1 ISA Base

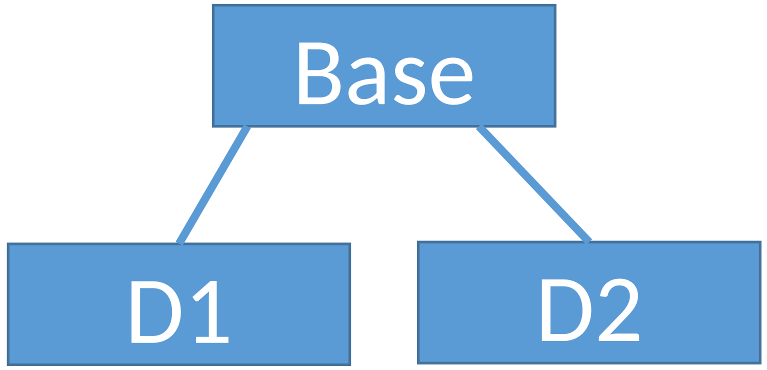
D2 ISA Base

D1 ¬ISA D2

D2 ¬ISA D1

Because of the ISA relations, the following are true

1. A pointer to a D1 object can be assigned to a Base pointer
2. A pointer to a D1 object can be assigned to a D1 pointer
3. A pointer to a D2 object can be assigned to a Base pointer
4. A pointer to a D2 object can be assigned to a D2 pointer
5. A pointer to a D1 object *should never* be assigned to a D2 pointer, and *vice versa*
6. A pointer to a Base object should never be assigned to a D1 or D2 pointer *unless we know that it points to a D1 or D2 object, or an object of a class that inherits from D1 or D2.*

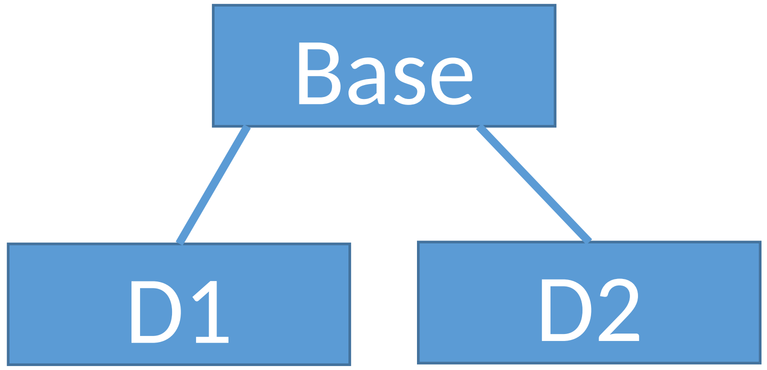


C style using the declarations:

Casts.zip

```
Base* b;
D1* d1 = new D1();
D2* d2 = new D2();
void* v;
```

```
b = (Base*) d1;
b = (Base*) d2;
d2 = (D2*) b; // compiles, but can cause problems at runtime
d1 = (D1*) b; // compiles, but can cause problems at runtime
d1 = (D1*) d2; // compiles, but will cause problems at runtime
d2 = (D2*) d1; // compiles, but will cause problems at runtime
v = (void*) b;
d1 = (D1*) v;  // compiles, but will cause problems at runtime (d1 points to a
Base object)
```

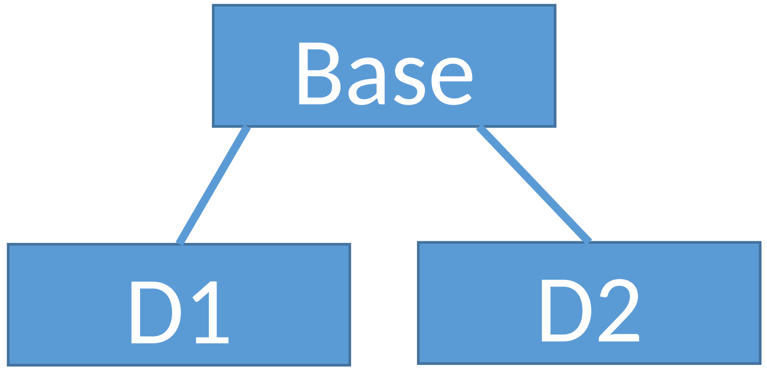


static_cast using the declarations:

```
Base* b;
D1* d1 = new D1();
D2* d2 = new D2();
void* v;
```

```
b = static_cast<Base*>(d1);
b = static_cast<Base*>(d2);
d2 = static_cast<D2*>(b);
d1 = static_cast<D1*>(b);
// compile time ERROR d1 = static_cast<D1*>(d2);
// compile time ERROR d2 = static_cast<D2*>(d1);
v = static_cast<void*>(b);
d1 = static_cast<D1*>(v);  // compiles, but invalid
```

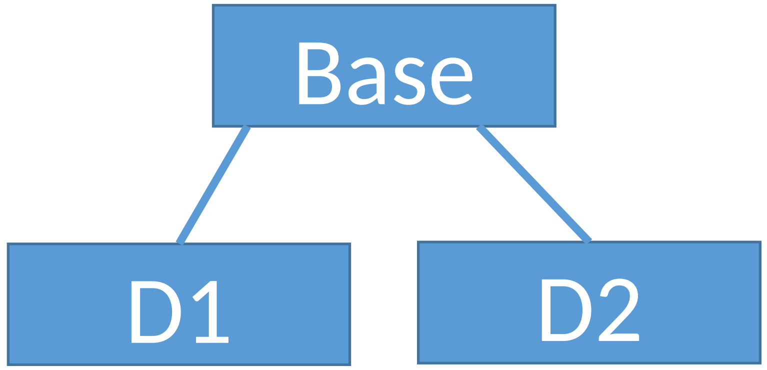
If it must be wrong given the declared types, gives a compile time error.
If it could be right, compiles.



dynamic_cast using the declarations:

```
Base* b;
D1* d1 = new D1();
D2* d2 = new D2();
void* v;
```

```
b = dynamic_cast<Base*>(d1);
b = dynamic_cast<Base*>(d2);
d2 = dynamic_cast<D2*>(b);
d1 = dynamic_cast<D1*>(b);
d1 = dynamic_cast<D1*>(d2); // compiles, but will give an seg fault at runtime
d2 = dynamic_cast<D2*>(d1); // compiles, but will cause seg fault at runtime
v = dynamic_cast<void*>(b);
// compile time ERROR d1 = dynamic_cast<D1*>(v); // must have an object ptr
```



Test the result of the static cast after doing the cast

```
Base* b;
D1* d1 = new D1();
D2* d2 = new D2();
void* v;
```

```
 d1 = dynamic_cast<D1*>(d2);
If (d1 == NULL) {std::cout << “bad cast to d1” << std::flush << std::endl;
d2 = dynamic_cast<D2*>(d1); // compiles, but will cause seg fault at runtime
If (d1 == NULL) {std::cout << “bad cast to d1” << std::flush << std::endl;
```

Overloading in C++ (vs Overriding)

- Overriding occurs when a derived class has a function with the same name and signature as an inherited base class function
  - Overriding allows us to change the implementation in the base class, i.e., it allows selective reuse of the base class code
  - Overriding can be thought of as occurring vertically down the inheritance chain
- Overloading in when multiple methods have the same name, but different signatures
  - Overloading allows us to perform similar functionality with different input arguments
  - Overloading occurs horizontally across the function names visible in a class

Overloading example

```
class User {
public:
   User(std::string nm, std::string addr, int id, int age);
   User( );
   virtual ~User( );
   void print( );
   std::string getUser(std::string nm, int age);
   std::string getUser(std::string nm, std::string addr);
   std::string getUser(int id, int age);
private: // "private" to make explicit
   std::string intToString(int i);
   std::string name;
   std::string address;
   int age;
   int idNum;
};
```

```
User::User(std::string nm, std::string addr, int id,
   int age) : name(nm), address(addr), idNum(id) { }

User::User( ) { }
```

Overloading example

```
class User {
. . .
    std::string getUser(std::string nm, int age);
   std::string getUser(std::string nm, std::string addr);
   std::string getUser(int id, int age);. . .
};
```

```
std::string User::getUser(std::string nm,
    std::string addr) {
    std::cout << "User::getUser(str, str)";
    std::cout << std::endl;
    return nm + " " + addr;
}
```

```
std::string User::getUser(std::string nm, int age) {
    std::cout << "User::getUser(str, int)";
    std::cout << std::endl;
    return nm+" "+intToString(age);
}
```

```
std::string User::getUser(int id, int age) {
    std::cout << "User::getUser(int, int)";
    std::cout << std::endl;
    return intToString(id) + " " + intToString(age);
}
```

Invoking overloaded functions

```
int main(int argc, char * argv[ ]) {
   std::string s1 = "aa";
   std::string s2 = "bb";
   User u = User( );
   std::cout << u.getUser("Laura", 22) << std::endl;
   std::cout << u.getUser("Sam", "EE 310") << std::endl;
   std::cout << u.getUser(70, 45) << std::endl;
}
```

```
User::getUser(str, int)
Laura 22
User::getUser(str, str)
Sam EE 310
User::getUser(int, int)
70 45
```

Invoking overloaded functions

```
int main(int argc, char * argv[ ]) {
  std::string s1 = "aa";
  std::string s2 = "bb";
  User u = User( );
  std::cout << u.getUser("Laura", 22) << std::endl;
  std::cout << u.getUser("Sam", "EE 310") << std::endl;
  std::cout << u.getUser(70, 45) << std::endl;
}
```

Deciding which function to call is easy when the arguments match exactly. Dealing with not exact matches is a topic for another day.

Overloading is not a new concept

- The new part is that it applies to functions
- In the expression a + b, the “=“ can mean an integer addition, or a single, double or quad precision addition.
- And, as we will see, C++ will let us overload most of its operators.

A useful piece of code

- strStream is an object.
- In the ostringstream class, the “<<“ operator is overloaded to a function that takes an ostringstream an int as arguments.
- It converts the int into characters and appends them to the (empty, in this case) ostringstream
- The str( ) function then returns an string representation of the ostringstream
- The effect is to convert the int to a string.

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

The *this* pointer

- Every time an object’s method is called, a pointer to the object is passed in as an argument.
- This is done automatically, and we don’t have to do anything.
- Let’s see how this works

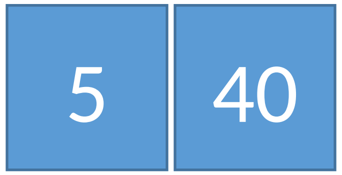
```
class Item {
public:
   Item(int, float);
   Item( );
   virtual ~Item( );
   virtual void updateItemNum(int);
   virtual void updatePrice(float);
   virtual int getItemNum( );
   virtual float getPrice( );
   virtual void print( );
private: // "private" to make explicit
   int itemNum;
   float price;
};
```

```
Item::Item(int i, float p) : itemNum(i), price(p) { }
. . .
void Item::updatePrice(float price) {
   price = price;
}

void Item::print( ) {
   std::cout << "num: " << itemNum << ", price: ";
   std::cout  << price << std::endl;
}
```

Item.h and part of BadItem.cpp

Let’s see how this works (poorly...)

```
#include "Item.h"

int main(int argc, char * argv[ ]) {
   Item *iP = new Item(5, 40.0);
   iP->updatePrice(50.0);
   iP->print( );
}
```



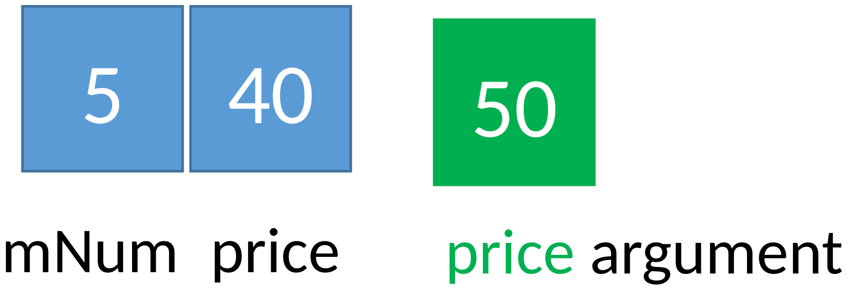
itemNum price

```
Item::Item(int i, float p) : itemNum(i), price(p) { }
. . .
void Item::updatePrice(float price) {
   price = price;
}

void Item::print( ) {
   std::cout << "num: " << itemNum << ", price: “;
   std::cout  << price << std::endl;
}
```

Let’s see how this works (poorly...)

```
#include "Item.h"

int main(int argc, char * argv[ ]) {
   Item *iP = new Item(5, 40.0);
   iP->updatePrice(50.0);
   iP->print( );
}
```

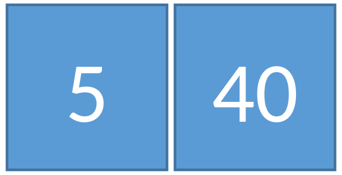
```
Item::Item(int i, float p) : itemNum(i), price(p) { }
. . .
void Item::updatePrice(float price) {
   price = price;
}

void Item::print( ) {
   std::cout << "num: " << itemNum << ", price: “;
   std::cout  << price << std::endl;
}
```



price = price

Let’s see how this works (poorly…)

```
#include "Item.h"

int main(int argc, char * argv[ ]) {
   Item *iP = new Item(5, 40.0);
   iP->updatePrice(50.0);
   iP->print( );
}
```



itemNum price

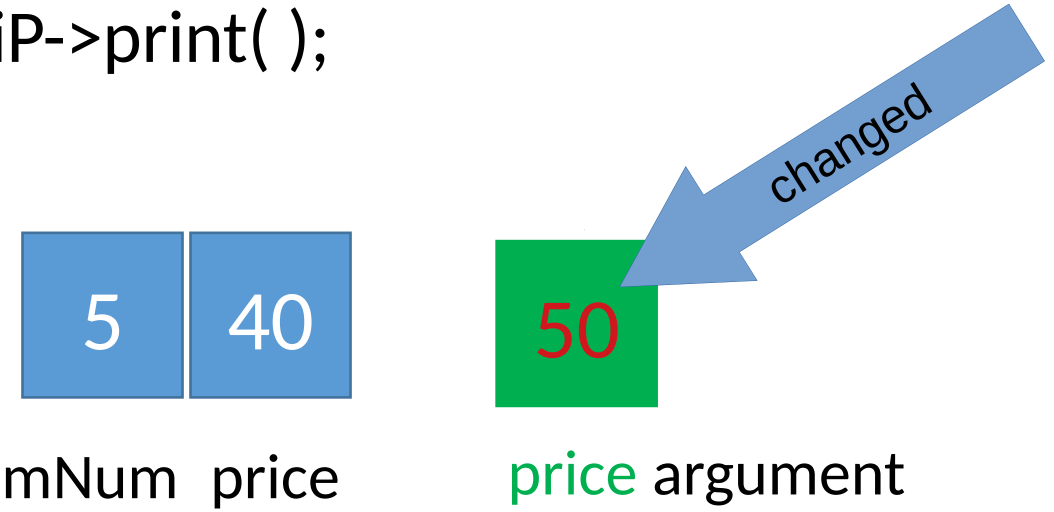
```
Item::Item(int i, float p) : itemNum(i), price(p) { }
. . .
void Item::updatePrice(float price) {
   price = price;
}

void Item::print( ) {
   std::cout << "num: " << itemNum << ", price: “;
   std::cout  << price << std::endl;
}
```

```
num: 5, price: 40
```

GoodItem has a different updatePrice( )

```
#include "Item.h"

int main(int argc, char * argv[ ]) {
   Item *iP = new Item(5, 40.0);
   iP->updatePrice(50.0);
   iP->print( );
}
```

```
Item::Item(int i, float p) : itemNum(i), price(p) { }
. . .
void Item::updatePrice(float price) {
   this->price = price;
}
void Item::print( ) {
   std::cout << "num: " << itemNum << ", price: “;
   std::cout  << price << std::endl;
}
```

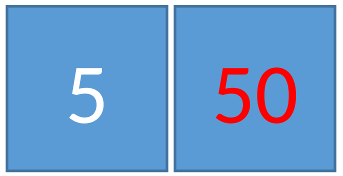


this->price = price

GoodItem has a different updatePrice( )

```
#include "Item.h"

int main(int argc, char * argv[ ]) {
   Item *iP = new Item(5, 40.0);
   iP->updatePrice(50.0);
   iP->print( );
}
```

```
Item::Item(int i, float p) : itemNum(i), price(p) { }
. . .
void Item::updatePrice(float price) {
   this->price = price;
}
void Item::print( ) {
   std::cout << "num: " << itemNum << ", price: “;
   std::cout  << price << std::endl;
}
```

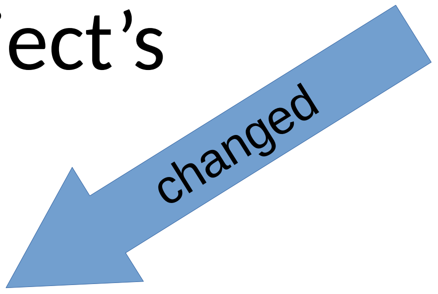


itemNum price

```
num: 5, price: 50
```

This example was sort of contrived

- The main uses of the *this* pointer
  - If a method in an object needs to pass the object's address as an argument to a function
  - To enable fields to be addressed
    - There is only one copy of the function code for a class
    - There are many objects
    - The *this* pointer allows that one code copy to access the fields of the object the function is called on
    - The *this* ponter is passed as an *implicit* argument



Object fields and methods

- So far, the variables and methods we’ve seen in classes are associated with an object
- When an object is allocated, a separate copy of variables are made
- There is only one copy of method code, but when called it will be passed the *this* pointer to the object and its variables.
- And variables and methods associated with an object require we have access to an object, require we allocate an object, and are hard to share among all objects of a class.

Static fields and methods

- A *static field* is shared among all of the objects
- It is associated with a class, and not with an object of that class
- They become like global variables and are easy to share across all of the objects of a class (or objects of different classes)
- A static function is associated with a class
  - Static functions are not passed a *this* pointer since they are not associated with an object
  - Of course, like any code, they can allocate objects and call functions on that allocated object, and access public fields of that object.

This code is in the Static directory

```
class Item {
public:
   Item(int, float);
   Item( );
   virtual ~Item( );
   virtual void print( );
   static int getNumberOfItems( );
private: // "private" to make explicit
   static int numberOfItems;
   int itemNum;
   float price;
};
```

```
int Item::getNumberOfItems( ) {
   return numberOfItems;
}

Item::Item(int i, float p) : itemNum(i), price(p) {
   numberOfItems++;
}

Item::Item( ) { }

Item::~Item( ) { }

void Item::print( ) {
   std::cout << "number of items: " << numberOfItems;
   std::cout << ", item number: " << itemNum;
   std::cout << ", price: " << price << std::endl;
}
```

Important!

int Item::numberOfItems = 0; // initialize statics like this

This code is in Static.

```
class Item {
public:
   Item(int, float);
   Item( );
   virtual ~Item( );
   virtual void print( );
   static int getNumberOfItems( );
private:
   static int numberOfItems;
   int itemNum;
   float price;
};
```

```
int Item::getNumberOfItems( ) {
  return numberOfItems;
}

Item::Item(int i, float p) : itemNum(i), price(p) {
   numberOfItems++;
}

Item::Item( ) {numberOfItems++;}

Item::~Item( ) { }

void Item::print( ) {
  std::cout << "number of items: " << numberOfItems;
  std::cout << ", item number: " << itemNum;
  std::cout << ", price: " << price << std::endl;
}

int Item::numberOfItems = 0; // initialize statics like this
```

```
class Item {
public:
   Item(int, float);
   Item( );
   virtual ~Item( );
   virtual void print( );
   static int getNumberOfItems( );
private: // "private" to make explicit
   static int numberOfItems;
   int itemNum;
   float price;
};
```

```
int Item::getNumberOfItems( ) {
   return numberOfItems;
}

Item::Item(int i, float p) : itemNum(i), price(p) {
   numberOfItems++;
}

Item::Item( ) { }

Item::~Item( ) { }

void Item::print( ) {
   std::cout << "number of items: " << numberOfItems;
   std::cout << ", item number: " << itemNum;
   std::cout << ", price: " << price << std::endl;
}

int Item::numberOfItems = 0;
```

Why is numberOfItems initialized in the .cpp file?

1. C++ requires variables (including statics) be initialized exactly once

- We cannot initialize it in the .h file
    - This would imply that every time the .h file is included by a class, initialization code would be created.
    - The compiler would then have to track which .h's have executed initialization code.
        - What if the initialization is a function of a macro that is expanded at compile time and changes each time it is expanded?
        - What should the value be initialized to? The value the first time a .h is included? The last time?
        - Confusing behavior would result.

Simple constructors

- Constructors are the functions that initialize an object
  - The object is created by the new operator or when it is added to the runtime stack
- Constructors initialize an object’s storage.
- Initialization can happen by default, and thus constructors perform both system and programmer specified actions

Constructor.zip

```
class Person {
public:
  Person(std::string, std::string);
  virtual ~Person( );

  void updateName(std::string);
  string getName( );

  virtual void updateAddr(string);
  virtual std::string getAddr( );
private:
  std::string name;
  std::string addr;
};
```

```
Person::Person(std::string nm, std::string ad) {
  name = nm;
  addr = ad;
  std::cout << "Person Constructor" << std::endl;
}

Person::~Person( ){ }

void Person::updateName(string nm) {
  name = nm;
}

string Person::getName( ) {
  return name;
};
```

```
class Student : public Person {

public:
   Student(std::string, std::string,
                  std::string);
   virtual ~Student( );

   virtual void codo(std::string);
   virtual std::string getMajor( );

private:
   std::string major;


};
```

```
Student::Student(string nm, string address,
                                  string degOption) :
   Person(nm, address), major(degOption) {
   std::cout << "Student Constructor" << std::endl;
}

Student::~Student( ) { }

void Student::codo(std::string newDegree) {
   major = newDegree;
}

string Student::getMajor( ) {
   return major;
}
```

```
Student::Student(string nm, string address,
                              string degOption) :
   Person(nm, address), major(degOption) {
   std::cout << "Student Constructor" << std::endl;
}
```

The base class constructor must be called in the initializer list, whose actions are performed before the constructor body executes.
If the base class constructor is not called the compiler will insert a call to the *zero argument constructor.*

What if there is no zero arg constructor?

- If no *declaration* for a zero arg constructor is provided by the programmer, C++ will provide a declaration of the form

  Base( );

- If no other constructors are *defined* a default declaration Base::Base( ) { } is defined.
- If other constructors have been *defined*
  - No default definition will be provided by the compiler
  - You will get an error message about *undefined constructor*

What does the default constructor do?

- It initializes all fields, *in a system dependent way*, to the zero for that field.
  - 0 for integer, 0.0 for floating point, null string for string
  - I read *system dependent* as *not portable*
- In general I don't like defaults, I like knowing what the value actually is
  - Probably only matters if you are doing bit-wise comparisons on default values

What happens if a base constructor is called in the body of the constructor? (see ConstCall)

```
Base::Base( ) {
   a = 40;
   b = 50;
}

Base::Base(int ar) {
   a = 4;
}
```

```
Derived::Derived( ) {
   Base(5);
}
```

This is bad!

```
int main (int argc, char *argv[]) {

   Derived *d = new Derived( );
   d->print( );
}
```

```
a: 40, b: 50
```

Base(5) creates a new temporary Base object and initializes it using the Base(int) constructor. In short, it doesn't do what we want.

Initializer lists Foo::Foo(Bar v) : Base(v), x(v) { . . . }

- Actions in the initializer list are performed before the body of the constructor, i.e. before what is in the { . . . }
- Four reasons this is a good thing.
  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 may be more efficient than an assignment
     1. *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)
     2. ": *x(v)*" directly copies *v* value into *x*'s memory space
  3. Easy to see where object attributes are initialized
  4. The only way to initialize **const** refs (more on **const** later) and reference members (more on references later)

What order are initializations done?

```
class Base {
public:
   Base(int,int,int);

   virtual ~Base( );

   virtual void print( );
   int a;
   int b;
private:
   int c;
   int d;
};
```

```
Base::Base(int u, int v, int w) : d(u), c(v), b(w), a(-1) { }
```

```
a: -1, b: 30, c: 20, d: 10
```

- They are not done in the order listed
- The order is:
  - Execute base class constructor first
  - Perform the initializers for non-static variables in the order *declared*
  - Variables in a less derived class are declared earlier than variables in a later declared paper
- This often doesn't make a difference, as in this case

Initialization order can make a difference
(see Initializer)

Base::Base(int u, int v, int w) : d(8), c(b), b(4), a(d) { }

Assume the variables are declared in the order a, b, c, d, as in the previous Base.h

Assume the call Base *b = new Base(90, 100, 110);

1. Storage is allocated for the object and zeroed out, i.e., a=0; b=0; c=0; d=0
2. a(d) is performed, initializing a to 0 (the current value of d)
3. b(4) is performed, initializing b to 4
4. c(b) is performed, initializing c to 4
5. d(8) is performed, initializing d to 8

```
a: 0, b: 4c: 4, d: 8
```

Preventing and controlling inheritance

- Sometimes would like to force certain functions to be inherited and not able to be overriden
- In C11++ standard we can do this as `virtual void work( ) final {`
- C++ has another, older way of doing this.
- make functions non-virtual, but this can be gotten around (see earlier examples in Poly2NonVirtualBase and PolyNonVirtualBase in L2Code
- 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.

```
class Person {
public:
   virtual ~Person( );
private:
   Person( );
};
```

```
Student::Student( ) : Person( ) { }

Student::~Student( ) { }
```

See PrivateConstructorUser for a similar program.

```
Student.h:3:31: error: expected class-name before '{' token
 class Student : public Person {
                               ^

Student.cpp: In constructor 'Student::Student()':
Student.cpp:5:23: error: class 'Student' does not have any field named 'Person'
 Student::Student( ) : Person( ) { }
                       ^
```

This can also help if we want two constructors but they have the same argument types

```
#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_;
 };
Point::Point(float x, float y) : x_(x), y_(y) { }
Point* Point::rectangular(float x, float y) { return Point(x, y); }
Point* Point::polar(float radius, float angle) {
     return Point(radius*std::cos(angle), radius*std::sin(angle));
}
```

These are called *named constructors.* Essentially, they are static functions that call the constructor and return a Point object.

And private constructors can be used to create *singleton* objects

```
class Point {
public:
  static Point* rectangular(float x, float y);
  Point* polar(float radius, float angle);
  static void printRectangular( );
  static void printPolar( );

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;
};
#endif /* POINT_H_ */
```

```
Point* Point::rectP = 0; // why not put this into the .h file?
Point* Point::polarP = 0; // why not put this into the .h file?

Point* Point::rectangular(float x, float y) {
  if (rectP == 0) {
    rectP = new Point(x, y);
    return rectP;
  } else return rectP;
}

Point* Point::polar(float radius, float angle) {
  if (polarP == 0) {
    polarP = new Point(radius*std::cos(angle), radius*std::sin(angle));
    return polarP;
  } else return polarP;
}
. . .
```

Point::rectP = 0; **// why not put this into the .h file?**

From stackoverflow:

By putting: foo::i = VALUE; into the header, foo:i will be assigned the value VALUE (whatever that is) for every .cpp file, and these assignments will happen in an indeterminate order (determined by the linker) before main() is run.

What if we #define VALUE to be a different number in one of our .cpp files? It will compile fine and we will have no way of knowing which one wins until we run the program.

Never put executed code into a header for the same reason that you never #include a .cpp file

```
Point* Point::rectP = 0; // why not put this into the .h file?
Point* Point::polarP = 0; // why not put this into the .h file?

Point* Point::rectangular(float x, float y) {
  if (rectP == 0) {
    rectP = new Point(x, y);
    return rectP;
  } else return rectP;
}

Point* Point::polar(float radius, float angle) {
  if (polarP == 0) {
    polarP = new Point(radius*std::cos(angle), radius*std::sin(angle));
    return polarP;
  } else return polarP;
}
. . .
. . .

void Point::printRectangular( ) {
   std::cout << "Rectangular, x: " << rectP-> x_;
   std::cout << ", y: " << rectP->y_ << std::endl;
}

void Point::printPolar( ) {
   std::cout << "Polar, x: " << polarP-> x_;
   std::cout << ", y: " << polarP->y_ << std::endl;
}

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

```
int main (int argc, char *argv[]) {

  Point* rP = Point::rectangular(10.0,20.0);
  Point::printRectangular( );
  rP = Point::rectangular(30.0,40.0);
  Point::printRectangular( );
}
```

```
Rectangular, x: 10, y: 20
Rectangular, x: 10, y: 20
```

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.