Object Composition

In the real world, complex objects are built from smaller, simpler objects
- A car is built from a frame, transmission, tires, steering wheel, etc.
- A computer is built from a CPU, a hard drive, a power supply, RAM, etc.

Object composition is the process of creating more complex objects from simple ones
- Use classes as member variables in other classes
- Becomes a "Has-A" relationship
  - e.g. A car "Has-A" steering wheel
Composition

Why should we use composition?
- It keeps your objects simple...an individual class should focus on doing *ONE* thing well
- Making objects self-contained and single-purpose promotes reusability
- More complex objects don’t need to know the details of what other objects are doing

Composition usually implies ownership
- Destroying the complex object means destroying all the objects it contains

Composition without ownership (a.k.a. Aggregation)
- Destroying the complex object does not destroy all the objects it contains
Composition Example

A creature in a videogame
  ➔ Has a name and hit points
  ➔ Has a physical location on a grid

We’ve already created a class to handle physical locations
(remember "Point"?)

```java
class Creature {
  string name;
  int hp;
  Point location;
};
```
Object Inheritance

In the real world, complex objects share many attributes with similar objects

- A manager and a programmer both have a name, a salary, a job title, etc. They are both types of employees.

Object inheritance is the process of creating more complex objects by acquiring the attributes and behaviors of existing ones

- Create new classes from existing classes
  - A "Base" class (a.k.a. Superclass)
  - A "Derived" class (a.k.a. Subclass)
- Becomes an "Is-A" relationship
  - e.g. A programmer "Is-An" employee
Inheritance

Object Inheritance can always be viewed as a tree structure
- Derived classes inherit all data/functions from their parent

Both Managers & Programmers have a salary, phone #, etc.
- Managers may also have a list of people they manage
- Programmers may also have a list of languages they know

This is referred to as an Inheritance Tree
- In Computer Science, tree grows downward
- Shape is the root of the tree
- Triangle and Square are the leaves of the tree
Inheritance

Object Inheritance can always be viewed as a tree structure

- Derived classes inherit all data/functions from their parent

<table>
<thead>
<tr>
<th>Employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>- salary: int</td>
</tr>
<tr>
<td>- phone: string</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>- managed: list&lt;Employee&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Programmer</th>
</tr>
</thead>
<tbody>
<tr>
<td>- languages: list&lt;string&gt;</td>
</tr>
</tbody>
</table>

- Both Managers & Programmers have a salary, phone #, etc.
  - Managers may also have a list of people they manage
  - Programmers may also have a list of languages they know

- This is referred to as an Inheritance Tree
  - In Computer Science, tree grows downward
  - Shape is the root of the tree
  - Triangle and Square are the leaves of the tree
Inheritance

Object Inheritance can be many levels deep

- Shape
  - Triangle
  - Rectangle
    - Square
Inheritance

Object Inheritance can be many levels deep

Shape
- centerX: int
- centerY: int

Triangle
- base: int
- height: int

Rectangle
- width: int
- length: int

Square
Inheritance

Object Inheritance can be many levels deep

- Shape
  - centerX: int
  - centerY: int

- Triangle
  - base: int
  - height: int

- Rectangle
  - width: int
  - length: int

- Square

This is referred to as an *Inheritance Tree*
- In Computer Science, *tree* grows *downward*
- Shape is the *root* of the tree
- Triangle and Square are the *leaves* of the tree
Inheritance

Why should we use inheritance?
  → Code Reuse (Don’t repeat code!)
      ♦ Our Manager and Programmer classes would probably have *a lot* in common
      ♦ Directly reuse existing code and just tack on the features you want (or hide those you don’t)
  → Specialization
      ♦ Take existing objects and create related objects with a more finely-honed purpose
  → Behavior Overidding
      ♦ Derived classes can redefine behaviors from the base class implementation
  → Polymorphism
Inheritance Example

Define a base class "Shape"

```cpp
class Shape {
    private:
        Point center;
    public:
        Shape(Point c);
        Shape(int x, int y);
        Point getCenter() const;
        void setCenter(Point c);
        void print() const;
};
```
Inheritance Example

Define a derived class "Triangle"

```cpp
class Triangle : public Shape {
private:
    int base;
    int height;
public:
    Triangle(int b, int h);
    int getBase() const;
    void setBase(int b);
    int getHeight() const;
    void setHeight(int h);
};
```
Inheritance: Superclass Constructor

When we instantiate a Triangle, how do we set the "Point center" in Shape?

Basic initialization lists do *NOT* work
- The initialization list can only set the values of variables in the current class!

```
// This won’t compile
Triangle::Triangle(int b, int h, Point c) : base(b), height(h), center(c) { }
```

Initializing Shape’s variables in Triangle is a *BAD* idea and will *NOT* always work (don’t want to skip calling the Shape constructor)

```
// This won’t compile
Triangle::Triangle(int b, int h, Point c) { base = b; height = h; center = c; }
```
Inheritance: Superclass Constructor

We can use an initialization list to call a Shape (base class) constructor from a Triangle (derived class) constructor

Triangle::Triangle(int b, int h, Point c) :
    Shape(c), base(b), height(h) { }

Triangle::Triangle(int b, int h, int x, int y) :
    Shape(x,y), base(b), height(h) { }
Inheritance: Order of Creation/Destruction

- The superclass constructor will *ALWAYS* be called before the subclass constructor
  - The constructor of "Shape: is called before the constructor of "triangle"
  - The subclass can be dependent on data in the superclass
  - Create the inheritance tree from the root downward

- The subclass destructor will *ALWAYS* be called before the superclass destructor
  - Destroy the inheritance tree from the leaves upward
Inheritance: Access Specifiers

- **public**
  - Member data can be directly accessed by everyone

- **private**
  - Member data can only be accessed within the class that contains it. Subclasses cannot access superclass private data!
  - Triangle can’t modify "Point center" by default!

- **protected**
  - Member data can only be accessed within the class that contains it *OR* within its subclasses
Inheritance Example

Redefine a base class "Shape" with a protected member

class Shape {
    protected:
        //Shape, Triangle, Rectangle, Square can all access!
        Point center;
    public:
        Shape(Point center);
        Shape(int x, int y);
        Point getCenter() const;
        void setCenter(Point c);
        void print() const;
};
Making Derived Class Methods

You can add methods to your derived class just like you would any other class

- Derived class methods are *NOT* visible in the base class

```cpp
class Triangle : public Shape {
    ...
    double getArea() const;
};

Shape s;
// does not compile
double a = s.getArea();
```
Redefining Superclass Methods

Subclass methods have the ability to overload and redefine superclass methods

Compiler starts at the bottom of the inheritance tree and walks up it looking for a method

class Shape {
    ...
    void print() const;
};

class Triangle : public Shape {
    ...
    //must be identical to Shape’s to override!
    void print() const;
};
Redefining Superclass Methods

Subclass methods have the ability to overload and redefine superclass methods, but can also call the original method

```cpp
void Shape::print() const {
    cout << "I am a shape!" << endl;
}

void Triangle::print() const {
    //call superclass' print
    Shape::print();
    cout << "I am a triangle!" << endl;
}
```

- Actually, anyone can call Shape::print() since it’s a public member function of Shape
- It’s just like a regular function
Show/Hide Superclass Methods

Subclasses can redefine some superclass visibility

class Shape {
    public:
        Point getCenter() const;
    protected:
        void hiddenPrint() const;
};

class Triangle : public Shape {
    public:
        //make this visible
        Shape::hiddenPrint;
    private:
        //hide this from everyone
        Shape::getCenter;
};

Probably not a good idea since it breaks information hiding
Multiple Inheritance

A single class is actually allowed to derive from multiple classes at once
- Not used often. It gets confusing/complicated fast.

- In practice, the only time you can do multiple inheritance is when you inherit from Interfaces (will talk about in a couple of lectures)
Polymorphism

What is polymorphism?
- The ability of an object of one type to appear and be used like an object of a different type

Remember our Shape and Triangle classes?
- Based on inheritance, Triangle Is-A Shape so...
- We can actually pass Triangles around as Shapes!

```
Triangle t(5,5,15,20);
Shape rval = t;
Shape &tref = t;
Shape *tptr = &t;
Shape *tptr2 = new Triangle(1,2,55,62);
```

You can write very advanced code with polymorphism
- Write algorithms & frameworks that manipulate base class
- Can even write code for subclasses that hasn’t been realized or designed!
Shape \( s(5,10) \);
cout << \&s << endl;
cout << \&s.centerX << endl;
cout << \&s.centerY << endl;

\textbf{Output:}
0xbff5a2e8
0xbff5a2e8
0xbff5a2ec
Shape s(5,10);
cout << &s << endl;
cout << &s.centerX << endl;
cout << &s.centerY << endl;

Output:
0xbff5a2e8
0xbff5a2e8
0xbff5a2ec

<table>
<thead>
<tr>
<th>centerX</th>
<th>0x00</th>
<th>0x00</th>
<th>0x00</th>
<th>0x05</th>
</tr>
</thead>
<tbody>
<tr>
<td>centerY</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x0a</td>
</tr>
</tbody>
</table>
Triangle t(20,40,5,10);
cout << &t << endl;
cout << &t.base << endl;
cout << &t.height << endl;

- Output:
  0xbff5a2e8
  0xbff5a2f0
  0xbff5a2f4

0xbff5a2e8

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x14</td>
</tr>
<tr>
<td>height</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x28</td>
</tr>
</tbody>
</table>
Triangle \( t(20,40,5,10); \)
cout << &t << endl;
cout << &t.base << endl;
cout << &t.height << endl;

- Output:
  0xbff5a2e8
  0xbff5a2f0
  0xbff5a2f4

```
Shape
- centerX: int
- centerY: int

Triangle
- base: int
  - height: int

Rectangle
- width: int
  - length: int

Square
```

<table>
<thead>
<tr>
<th>centerX</th>
<th>0x00</th>
<th>0x00</th>
<th>0x00</th>
<th>0x05</th>
</tr>
</thead>
<tbody>
<tr>
<td>centerY</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x0a</td>
</tr>
<tr>
<td>base</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x14</td>
</tr>
<tr>
<td>height</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x28</td>
</tr>
</tbody>
</table>

Copyright © Sheila Tejada
Triangle t(20,40,5,10);
Shape *sptr = &t;
cout << sptr << endl;
cout << &sptr->base << endl;
cout << &sptr->height << endl; - height: int

Output:
0xbff5a2e8
0xbff5a2e8
0xbff5a2ec

0xbff5a2e8

centerX 0x00 0x00 0x00 0x05
centerY 0x00 0x00 0x00 0x0a
dwidth 0x00 0x00 0x00 0x14
height 0x00 0x00 0x00 0x28

Shape
Triangle
- centerX: int
- centerY: int
- base: int
- height: int
Rectangle
- width: int
- length: int
Square
Triangle \( t(20,40,5,10) \); Shape *sptr = &t;
cout << sptr << endl;
cout << &sptr->base << endl;
cout << &sptr->height << endl; - height: int

Output:
0xbff5a2e8
0xbff5a2e8
0xbff5a2ec

Example of polymorphism

- What object lives at memory location 0xbff5a2e8?
- Triangle and Shape are both correct answers

Shape
- centerX: int
- centerY: int

Triangle
- base: int

Rectangle
- width: int
- length: int

Shape

Triangle

Rectangle

Output:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbff5a2e8</td>
<td>0x00</td>
</tr>
<tr>
<td>0xbff5a2e8</td>
<td>0x00</td>
</tr>
<tr>
<td>0xbff5a2ec</td>
<td>0x00</td>
</tr>
</tbody>
</table>

Shape

Triangle

Rectangle

Output:
CS101
Extra Slides
Prof Tejada
/*
 * 1) Create a class Shape.
 * 2) *Protected* member Point center. This is so that a subclass can access the center. Use "point.h" and "point.cpp" from previous lecture.
 * 3) In the overloaded constructor use constructor initialization list.
 * 4) Add accessor and mutator.
 * 5) Add print() member function.
 */
#ifndef SHAPE_H
#define SHAPE_H_

#include "point.h"

class Shape
{
    protected:
        Point center;

    public:
        Shape(Point pt)
            : center(pt.getX(), pt.getY()) // call Point class constructor
        {
        }
        Shape(int x, int y)
            : center(x, y) // call Point class overloaded constructor
        {
        }

        Point getCenter() const
            { return center; }
        void setCenter(Point p)
            { center = p; }

        void print() const
        {
            std::cout << "Shape located at " << center << std::endl;
        }
};
#endif
triangle.h

/**
 * 1) Create a class Triangle which is a subclass of Shape.
 * 2) Private members base (int) and height (int).
 * 3) In the overloaded constructor, calls parent class’s
 *     constructor, set base and height.
 * 4) Add accessors and mutators.
 * 5) Add print() member function.
 */
```cpp
#ifndef TRIANGLE_H_
#define TRIANGLE_H_

#include "point.h"
#include "shape.h"

class Triangle : public Shape
{
    private:
        int base;
        int height;

    public:
        Triangle(int b, int h, Point pt)
            : Shape(pt), base(b), height(h) {}
        Triangle(int b, int h, int x, int y)
            : Shape(x, y)
            { base = b; height = h; }

        int getBase() const { return base; }
        int getHeight() const { return height; }
        void setBase(int b) { base = b; }
        void setHeight(int h) { height = h; }

        void print() const
        { std::cout << "Triangle located at "
            << center << std::endl; }
};
#endif
```
/*
1) Create a Shape with centre at (100,100), print it.
2) Create a Triangle with centre at (5,5).
3) Cannot create a Shape s2;
4) How about creating a Shape s2=t where t is a Triangle?
5) Print s2, which print() function will be called?
   Answer: Shape class’s print() will be called.
6) How about creating a Triangle t2=s?
   Will this compile?
7) Create a (Triangle*)tptr to point to (&t)
   Create a (Shape*)sptr to point to (tptr)
   Call tptr->print()
   Call sptr->print()
*/
#include <iostream>
#include <string>
#include "shape.h"
#include "triangle.h"

using namespace std;

int main()
{
    //calls Shape class "print" function
    Shape s(100,100);
s.print();

    //calls Triangle class "print" function
    Triangle t(20,40,5,10);
t.print();
    //Shape s2; //this will not compile

    //calls Shape class "print" function (why?)
    Shape tri = t;
    tri.print();

    //Triangle t2 = s; //this will not compile either
    Triangle *tptr=&t;
    Shape *sptr=tptr;
    sptr->print();
}
Let’s focus on the end of `inherit_main.cpp`:

```cpp
... int main()
{
    ...
    Triangle t(20,40,5,10);
    ...
    Triangle *tptr=&t;
    Shape *sptr=tptr;
    sptr->print();
}
```

Put everything together and compile:

```
g++ -g -Wall inherit_main.cpp point.cpp
```
Under the Cover

Run the debugger, set a breakpoint in main(), do a few next commands until you get to the last line

```
gdb a.out
(gdb) break main
(gdb) run
...
(gdb) next
(gdb) next
(gdb) ...
(gdb) next
30 sptr->print();
```
Under the Cover

What are the values of tptr and sptr?

(gdb) print tptr
$1 = (Triangle *) 0xbff5a2e8
(gdb) print sptr
$2 = (Shape *) 0xbff5a2e8

⇒ They are pointing at the same memory location

Where is sptr->center stored?

(gdb) print &sptr->center
$3 = (Point *) 0xbff5a2e8

⇒ Same place!

What about tptr->base and tptr->height?

(gdb) print &tptr->base
$4 = (int *) 0xbff5a2f0
(gdb) print &tptr->height
$5 = (int *) 0xbff5a2f4

⇒ Just below!
```
(gdb) print tptr
$1 = (Triangle *) 0xbff5a2e8
(gdb) print sptr
$2 = (Shape *) 0xbff5a2e8
(gdb) print &sptr->center
$3 = (Point *) 0xbff5a2e8
(gdb) print &tptr->base
$4 = (int *) 0xbff5a2f0
(gdb) print &tptr->height
$5 = (int *) 0xbff5a2f4
```
(gdb) print tptr
$1 = (Triangle *) 0xbff5a2e8
(gdb) print sptr
$2 = (Shape *) 0xbff5a2e8
(gdb) print &sptr->center
$3 = (Point *) 0xbff5a2e8
(gdb) print &tptr->base
$4 = (int *) 0xbff5a2f0
(gdb) print &tptr->height
$5 = (int *) 0xbff5a2f4
(gdb) print &sptr->center.x
$6 = (int *) 0xbff5a2e8
(gdb) print &sptr->center.y
$7 = (int *) 0xbff5a2ec
(gdb) print sizeof(*sptr)
$8 = 8
`tptr`

`sptr`

\[(\text{gdb}) \text{ print } \text{sizeof}(*sptr)\]

\$8 = 8

\[(\text{gdb}) \text{ print } \text{sizeof}(*tptr)\]

\$9 = 16