Slides by Andrew DeOrio

EECS 280

Programming and Introductory Data Structures

Polymorphism

Review: classes and ADTs

- Recall our `Triangle` class (simplified here):

```
class Triangle {
public:
  Triangle();
  Triangle(double a_in, double b_in,double c_in);
  double area() const;
  void print() const;
private:
  double a,b,c;
};
```

Review: classes and ADTs

- The C++ `class` mechanism lets us create our own custom types
- We can use our new, custom type just like built-in types

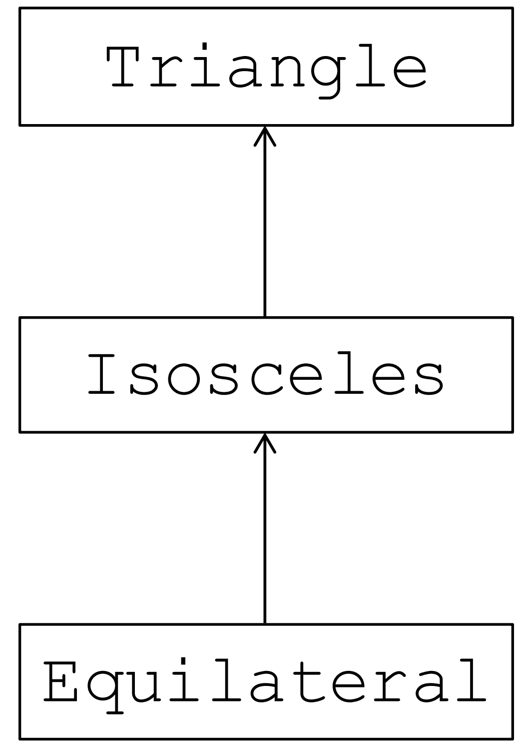
```
int main() {
  Triangle t(3,4,5);
  t.print();

  Triangle *t_ptr = &t;
  t_ptr->print();
}
```

```
class Triangle {
  //...
  void print() const {
     cout << "Triangle: "
          << "a=" << a
          << " b=" << b
          << " c=" << c
          << endl;
  }
```

```
./a.out
Triangle: a=3 b=4 c=5
Triangle: a=3 b=4 c=5
```

Review: derived types

- A derived type is used to represent an *is a* relationship
- A derived type inherits the member functions and member variables of its parent




```
class Isosceles : public Triangle {
  // ...
};
class Equilateral : public Isosceles {
  // ...
};
```

- An `Isosceles` *is a* `Triangle`
- An `Equilateral` *is a* `Isosceles`
- An `Equilateral` *is a* `Triangle`

Review: derived types

- Again, we can use our new, derived type just like built-in types

```
Isosceles i(1,12);
i.print();

Isosceles *i_ptr = &i;
i_ptr->print();
```

```
class Isosceles
  : public Triangle {
  //...
  void print() const {
    cout << "Isosceles: "
         << base=" << get_a()
         << " leg=" << get_b()
         << endl;  }
}
```

```
./a.out
Isosceles: base=1, leg=12
Isosceles: base=1, leg=12
```

Review: subtypes

- If S is a *subtype* of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of that program (correctness)

```
Isosceles i(1,12);
Isosceles Triangle *ptr = &i;
ptr->print();
```

Static type

- Often, the compiler can tell which derived type is needed
- This is called the *static type*

```
Triangle t(3,4,5);
t.print();

Triangle *t_ptr = &t;
t_ptr->print();

Isosceles i(1,12);
i.print();

Isosceles *i_ptr = &i;
i_ptr->print();
```

```
./a.out
Triangle: a=3, b=4, c=5
Triangle: a=3, b=4, c=5
Isosceles: base=1, leg=12
Isosceles: base=1, leg=12
```

Dynamic type

- Other times, the type is not known until run time
- This is called the *dynamic type*

```
//EFFECTS: asks user to select Triangle,
//         Isosceles or Equilateral
//         returns a pointer to correct object
Triangle * ask_user();

int main() {
  Triangle *t = ask_user(); //enters "Isosceles"
  t->print();
}
```

- What is the static type of `t`? What is the dynamic type?

Dynamic type

Returns pointer to global variable

```
static Triangle g_triangle(3,4,5);
static Isosceles g_isosceles(1,12);
static Equilateral g_equilateral(5);

Triangle * ask_user() {
  cout << "Triangle, Isosceles or Equilateral? ";
  string s;
  cin >> s;
  if (s == "Triangle")    return &g_triangle;
  if (s == "Isosceles")   return &g_isosceles;
  if (s == "Equilateral") return &g_equilateral;
  cout << "Unrecognized shape `" << s << "'\n";
  exit(1); //crash
}
```

I wish we could create as many objects as the user asked for ...

Forshadowing

- I wish we could create as many objects as the user asked for ...
- We will soon know how!

•Dynamic memory

- For now, we will use this global variable work-around

```
static Triangle g_triangle(3,4,5);
//...
Triangle * ask_user() {
  //...
  if (s == "Triangle") return &g_triangle;
  //...
}
```

Dynamic type

```
//EFFECTS: asks user to select Triangle,
//         Isosceles or Equilateral
//         returns a pointer to correct object
Triangle * ask_user();

int main() {
  Triangle *t = ask_user(); //enters "Isosceles"
  t->print();
}
```

- What is the output of this program?

Dynamic type

```
//...
Triangle *t = ask_user(); //"Isosceles"
t->print();
```

```
./a.out
Triangle, Isosceles or Equilateral?  Isosceles
Triangle: a=1 b=12 c=12
```

- Problem: the `Triangle::print()` function was called
  - Because the *static type* of `t` is `Triangle`
- But we wanted the `Isosceles::print()` function instead
  - Because the *dynamic type* of `t` is `Isosceles`

Polymorphism

```
//...
Triangle *t = ask_user(); //"Isosceles"
t->print();
```

- `t` can change types at runtime, in other words it is *polymorphic*
- We can use the *virtual function* mechanism in C++ to check the *dynamic type* of `t` at runtime and call the correct version of `print()`

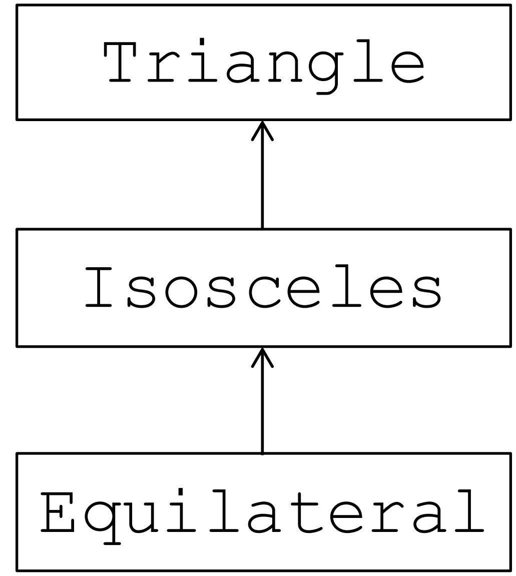
Polymorphism

- Polymorphism is the ability to associate many behaviors with one function name dynamically (at runtime)

- A polymorphic type is any type with a virtual function

- Virtual functions are the C++ mechanism used to implement polymorphism

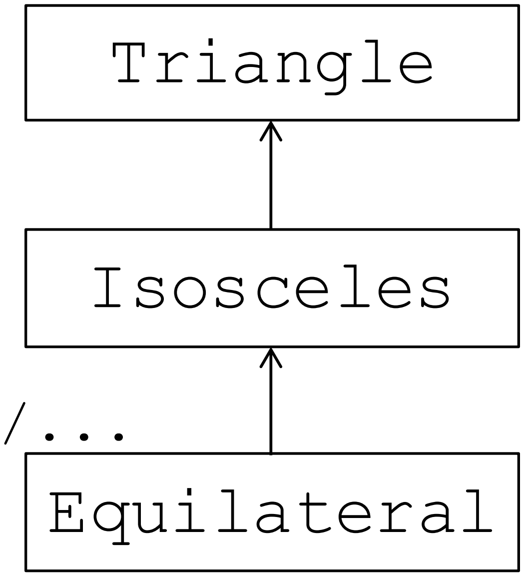
Polymorphism example




```
class Triangle {
  virtual void print() const {/*...*/}
  //...
};
```

- `virtual` means "check the dynamic type at runtime, then select the correct `print()` member function"
- `virtual` is inherited, so the overridden `print()` in `Isosceles` and `Equilateral` will automatically become `virtual`

Polymorphism example




```
class Triangle {
  virtual void print() const {/*...*/} //...
};
class Isosceles {
  virtual void print() const {/*...*/} //...
};
class Equilateral {//...
  virtual void print() const {/*...*/} //...
};
```

- Optionally add `virtual` keyword to derived types
- This is more clear ☺

Dynamic type

```
//...
Triangle *t = ask_user(); //"Isosceles"
t->print();
```

```
./a.out
Triangle, Isosceles or Equilateral?  Isosceles
Isosceles: base=1 leg=12
```

- Now our program works correctly ☺

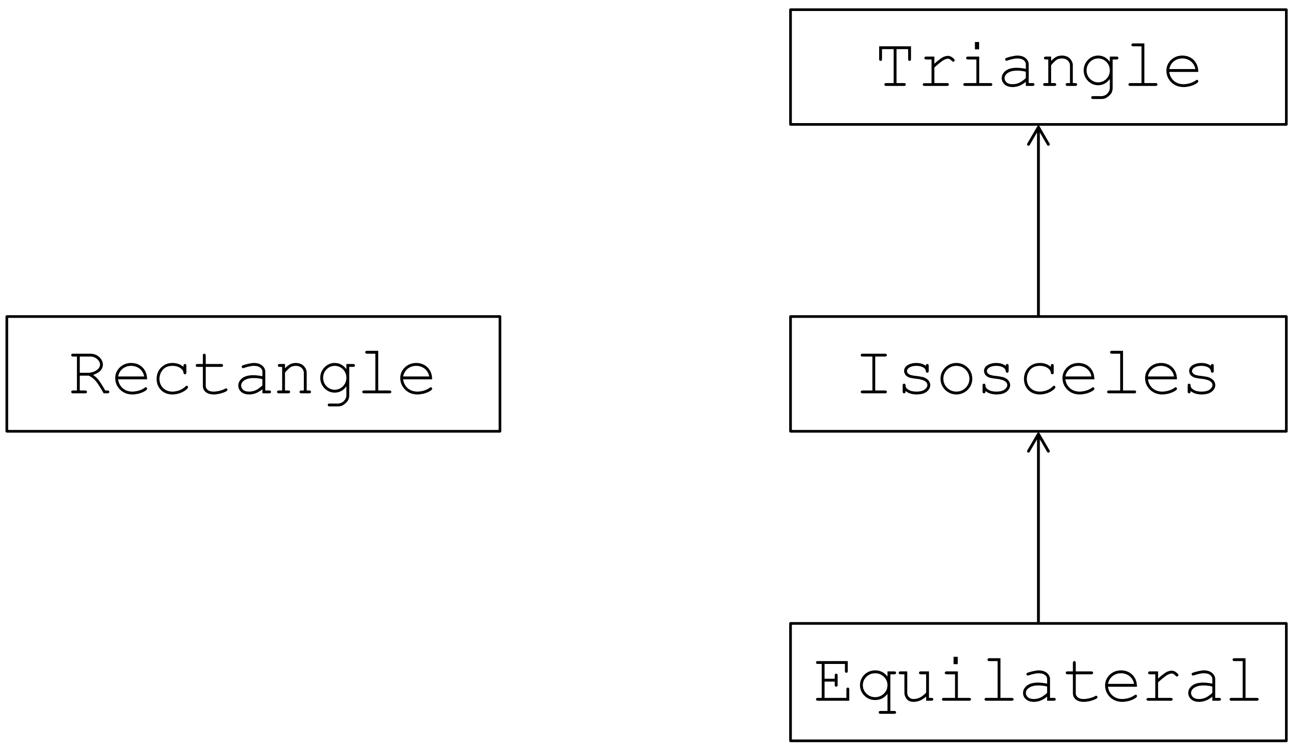
Extending the class hierarchy

- Recall our `Rectangle` class (simplified here):

```
class Rectangle {
public:
  Rectangle();
  Rectangle(double a_in, double b_in);
  double area() const;
  void print() const;
private:
  double a,b;
};
```

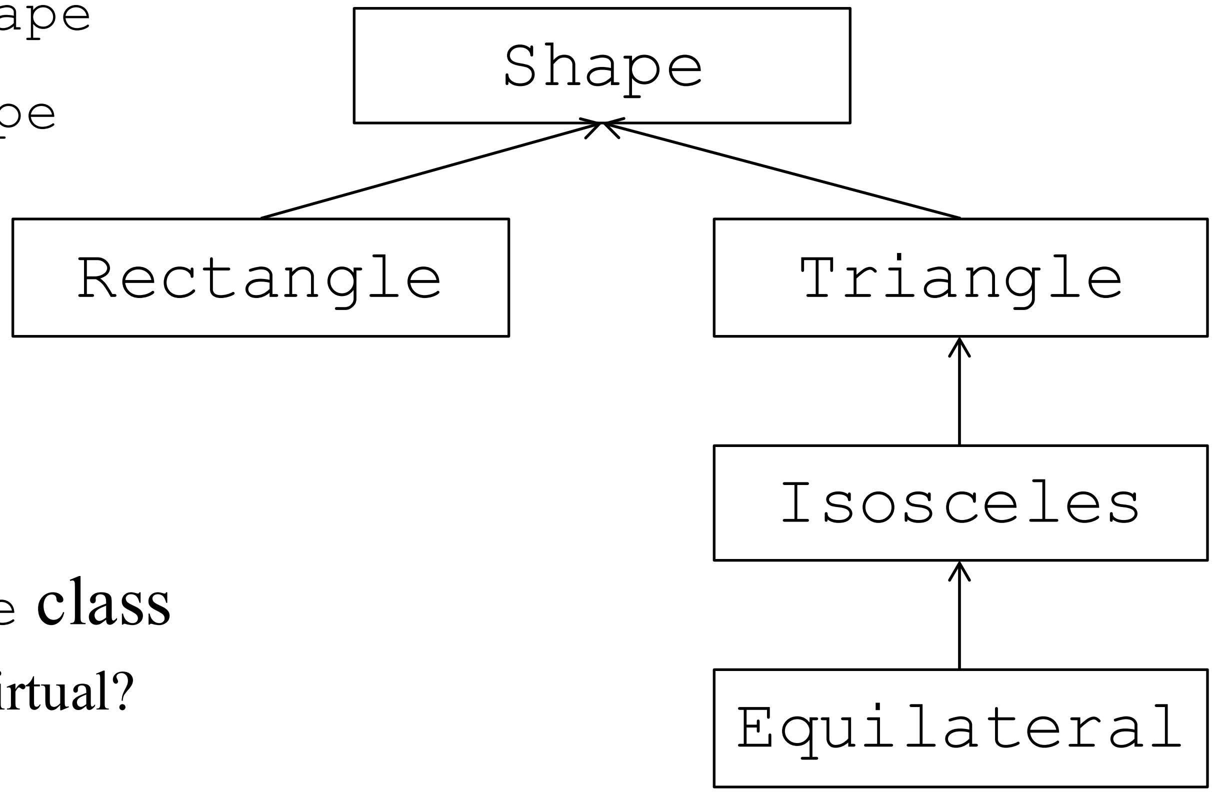
Exercise

- Create a class hierarchy for `Triangle`, `Isosceles`, `Equilateral` and `Rectangle`. Draw it.



Exercise

- A `Rectangle` *is a* `Shape`
- A `Triangle` *is a* `Shape`
- *etc.*



- Now: write the `Shape` class
  - Member functions? Virtual?
  - Member variables?
    - Think: what if we wanted to add a Circle class?

Exercise

```
class Shape {
public:
  virtual double area() const {/*...*/}
  virtual void print() const {/*...*/}
};
class Triangle : public Shape {/*...*/};
class Rectangle : public Shape {/*...*/};
```

- All shapes have `area()` and a `print()` member functions
- Use `virtual`, so pointers to derived types will call the right version of `area()` and `print()`

Exercise

```
class Shape {
public:
  virtual double area() const {/*...*/}
  virtual void print() const {/*...*/}
};
```

- No member variables
- Not all shapes have common attributes, e.g., circle and triangle

Exercise

- This code is perfectly legal C++, but it makes no sense!

```
Shape s;
s.print();
```

- A shape is an abstract idea
- Our shape should only be an interface to ensure that all shapes behave the same way

Abstract class

- Problem: an instance of an abstract idea makes no sense
  `Shape s; //what kind of shape???`

- Solution: abstract classes, AKA an *interface only* class

- You cannot create an instance of an abstract class, which is exactly what we want for a `Shape`

- An abstract class will force derived types to all behave the same way

Pure virtual functions

- Because there will be no implementation, we need to declare member functions in a special way:
  - Declare each method as a `virtual` function
  - “Assign" a `0` to each of these virtual functions.
- These are called *pure virtual functions*
- You can think of them as a set of function pointers, all of which point to `NULL` (AKA `0`)

```
class Shape {
public:
  virtual double area() const = 0;
  virtual void print() const = 0;
};
```

Pure virtual functions

- Shape is now an abstract class
- You cannot create an instance of an abstract class

```
Shape s; //compiler error
```

- You *can* create a pointer to an abstract class, and then assign the pointer to a *concrete class* derived from the base class
  - This is subtyping at work

```
Rectangle r(2,4); //concrete derived type
Shape *s = &r;    //OK, Rectangle is a Shape
s->print();       //virtual, so correct version
                  //of print() is called
```

```
./a.out
Rectangle: a=2 b=4
```

Pure virtual functions

```
//EFFECTS: asks user to select a shape
//         returns a pointer to correct object
Shape * ask_user();

int main() {
  Shape *s = ask_user(); //"Rectangle"
  s->print();
}
```

```
./a.out
Rectangle, Triangle, Isosceles or Equilateral?  Rectangle
Rectangle: a=2 b=4
```

- Now we can expand the `ask_user()` function to work with any `Shape`

Factory functions

```
//EFFECTS: asks user to select a shape
//          returns a pointer to correct object
Shape * ask_user();
```

- `ask_user()` is an example of a *factory function*
- A factory function creates objects for another programmer who doesn't need to know their actual types

Upcast

```
Shape *s = ask_user(); //"Isosceles"
```

```
Shape * ask_user() {
   cout << "Rectangle, Triangle, Isosceles or Equilateral?
   string s;
   cin >> s;
   if (s == "Triangle")     return &g_triangle;
   if (s == "Isosceles")    return &g_isosceles;
   //...
}
```

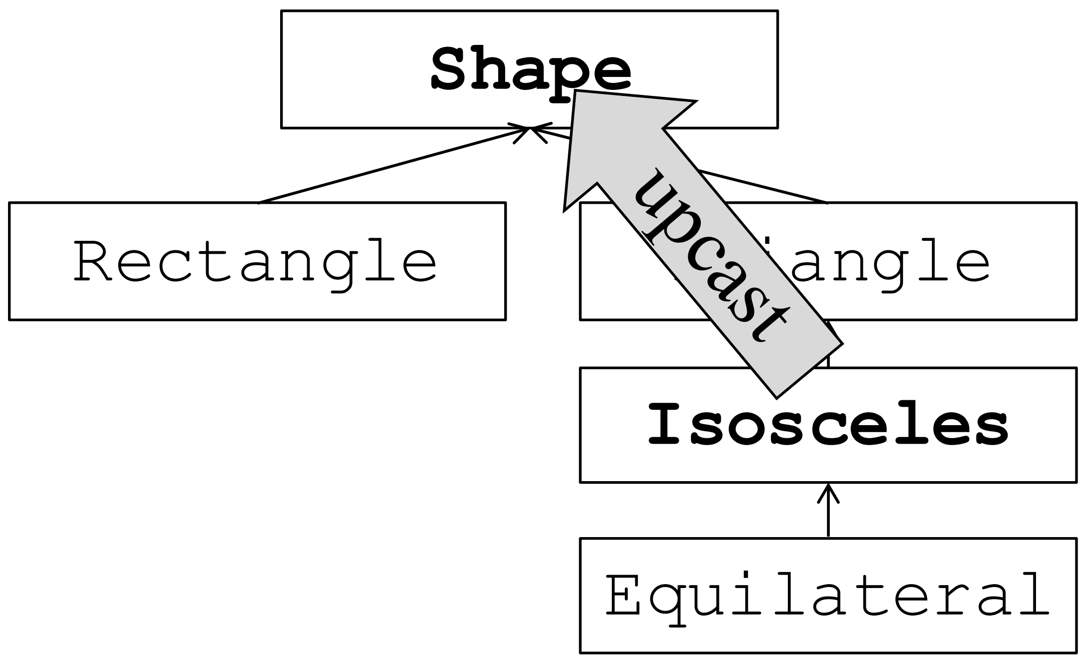
upcast

- Substitute subtype (`Isosceles`) for supertype (`Shape`)
  - This is called an *upcast*
- Type conversion is automatic, an *implicit cast*

Upcast

```
Shape *s = ask_user(); //"Isosceles"
```

- Think of *upcast* as a cast from one type in the class hierarchy to a higher one.
- Since a `Isosceles` *is a* `Shape`, this cast can happen automatically.



```
Shape * ask_user();
```

Downcast

```
Isosceles *t= ask_user();//enter "Isosceles"
```

```
error: invalid conversion from 'Shape*' to 'Isosceles*'
```

- Can't convert supertype (`Shape`) to subtype (`Isosceles`)
- Type conversion is not automatic
- This is called a *downcast*

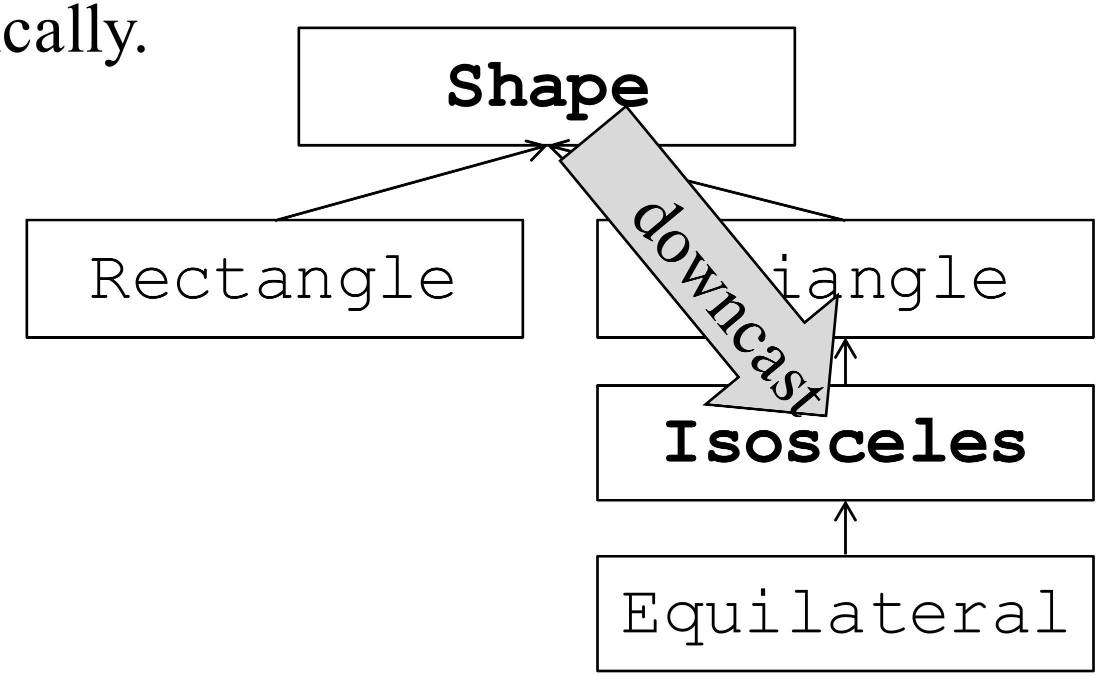
```
Shape * ask_user()
```

Downcast

```
Isosceles *t= ask_user();//enter "Isosceles"
```

```
error: invalid conversion from ‘Shape*’ to ‘Isosceles*’
```

- Think of *downcast* as a cast from one type in the class hierarchy to a lower one.
- Since a `Shape` might not be a `Isosceles`, this cast cannot happen automatically.



```
Shape * ask_user();
```

Downcast

```
Shape *s = ask_user(); //"Isosceles"
Isosceles *i = dynamic_cast<Isosceles*>(s);
if (i != 0){ //check for NULL pointer
  //something triangular
  double c = i->get_c();
}
```

- `dynamic_cast<T*>(ptr)` downcasts `ptr` to type `T*`, if possible. Otherwise, it returns `0`.
- In this example, if the user enters "`Isoscles`", the call to `dynamic_cast` will cast from `Shape*` to `Isoscles*`

```
Shape * ask_user();
```

Downcast

```
Shape *s = ask_user(); //"Isosceles"
Isosceles *i = dynamic_cast<Isosceles*>(s);
if (i != 0){ //check for NULL pointer
  //something triangular
}
```

- Always need to check a `dynamic_cast` for success
- `dynamic_cast` only works on polymorphic types.
  - AKA classes with virtual functions

dynamic_cast vs. static_cast

- We have now seen two different kinds of cast
- `dynamic_cast`
  - Checks at runtime if it is OK to cast
  - Cast from a polymorphic base class to a derived class
  - Supertype to subtype, *downcast*
- `static_cast`
  - Does not check at runtime
  - The programmer tells the compiler "trust me"
  - Works with non-polymorphic types as well as polymorphic

Constructors and polymorphism

- Recall how constructors of derived classes work
- First, the base class constructor runs, then the derived class constructor runs, etc.
- In other words, *instances of a derived class are constructed starting from the base class*

Note: The constructor that is called automatically is the default constructor. If you want a non-default constructor, you must call it explicitly.

Constructors and polymorphism

```
class Shape {
public:
  Shape() {
    cout <<
    "Shape default ctor\n";
  }
  //...
};
```

```
class Triangle : public Shape {
public:
  Triangle() {
    cout <<
    "Triangle default ctor\n";
  }
  //...
};
```

```
int main () {
  Triangle t;
}
```

```
./a.out
Shape default ctor
Triangle default ctor
```

Constructors and polymorphism

```
class Triangle : public Shape {
  //...
  Triangle() : a(0), b(0), c(0)
  { cout << "Triangle default ctor\n"; }

  Triangle(double a_in, double b_in, double c_in)
    : a(a_in), b(b_in), c(c_in)
  { cout << "Triangle double ctor\n"; }
  //...
};
```

- Add the same messages to `Isosceles`, `Equilateral` and `Rectangle`

Exercise: What is the output?

```
int main() {
  Equilateral e;
}
```

```
int main() {
 Rectangle r;
}
```

Exercise: What is the output?

```
int main() {
  Isosceles i;
  Triangle *t_ptr = &i;
}
```

```
int main() {
  Isosceles i(1,12);
  Shape *s_ptr = &i;
}
```

Exercise: What is the output?

```
int main() {
  Rectangle r;
  Triangle *t_ptr = &r;
}
```

```
int main() {
  Isosceles i;
  Equilateral *e_ptr = &i;
}
```

Exercise: What is the output?

```
int main() {
  Equilateral e1(5);
  Equilateral e2(6);

  Shape *array[2];
  array[0] = &e1;
  array[1] = &e2;

  array[0]->print();
  array[1]->print();
}
```

Arrays and polymorphism

- Polymorphism gives us another cool feature: we can put (pointers to) objects of different types together in the same container

```
Rectangle r(2,4);
Isosceles i(1,12);
Equilateral e(5);

const int SIZE=3;
Shape * shapes[SIZE];
shapes[0] = &r;
shapes[1] = &i;
shapes[2] = &e;
```

Exercise: write a `for`-loop to call `print()` on each `Shape*` in the array. Use traversal by pointer.