Simpler polymorphism demo
(~mikec/cs32/demos/figures)

- Base: `Figure` has `virtual void print()`
  - `print()` is used in `printAt(lines)`
- Derived: `Rectangle` *just* overrides `print()`
- Which `print()` is used in the following code?

```
Figure *ptr = new Rectangle,
       &ref = *new Rectangle('Q', 5, 10, 4);
ptr->printAt(1); ref.printAt(1);
```

- What if `print()` was not declared `virtual`?
- What if line 2 above just had `ref`, not `&ref`?
  - To know why, see "slicing" … a few slides from now

"Pure virtual" and abstract classes

- Actually class Figure's print() function is useless
  - It should have been a pure virtual function:

```
virtual void draw() const = 0;
```

  - Says not defined in this class – means any derived class must define its own version, or be abstract itself
- A class with one or more pure virtual functions is an abstract class – so *it can only be a base class*
  - An actual instance would be an incomplete object
  - So *any instance must be a derived class instance*

Types when inheritance is involved

- Consider: `void func (Sale &x) {…}` or similarly: `void func (Sale *xp) {…}`
  - What type of object is x (or *xp), really? Is it a Sale?
  - Or is it a DiscountSale, or even a CrazyDiscountSale?
- Just Sale members are available
  - But might be virtual, and Sale might even be abstract
  - `&` and `*` variables allow polymorphism to occur
- Contrast: `void func (Sale y) {…}`
  - What type of object is y? It's a Sale. Period.
  - Derived parts are "sliced" off by Sale's copy ctor
  - Also in this case, Sale cannot be an abstract class

Type compatibility example

```
class Pet {
public: // pls excuse bad info hiding
   string name;
   virtual void print();
};

class Dog : public Pet {
public:
   string breed;
   virtual void print();
};
```

- Consider:

```
Dog d; Pet p;
d.name = "Tiny";
d.breed = "Mutt";
p = d; // "slicing" here
```

  - All okay – a Dog "is a" Pet
- Reverse is not okay
  - A Pet might be a Bird, or …
- And `p.breed`? Nonsense!
- Also see `slicing.cpp` at `~mikec/cs32/demos/`

Destructors should be virtual

- Especially if class has virtual functions
- Derived classes might allocate resources via a base class reference or pointer:

```
Base *ptrBase = new Derived;
... // a redefined function allocates resources
delete ptrBase;
```

- If dtor not virtual, derived dtor is not run!
- If dtor is virtual – okay: run derived dtor, immediately followed by base dtor

Casting and inherited types

- Consider again: `Dog d; Pet p;`
- "Upcasting" (descendent to ancestor) is legal:

```
p = d; // implicitly casting "up"
p = static_cast<Pet>(d); // like (Pet)d
```

  - But objects sliced if not pointer or reference
- Other way ("downcasting") is a different story:

```
d = static_cast<Dog>(p); // ILLEGAL
```

  - Can only do by pointer and *dynamic cast* :

```
Pet *pptr = new Dog; // we know it's a Dog
Dog *dptr = dynamic_cast<Dog*>(pptr)
```

  - But can be dangerous, and is rarely done

Multiple inheritance and virtual

- Idea: a `ClockRadio` is a `Radio` *and* an `AlarmClock`
  - But what if class `Radio` and class `AlarmClock` are both derived from another class, say `Appliance`?
  - Doesn't each derived object contain an `Appliance` portion?
  - So wouldn't a `Clockradio` have two copies of that portion, and how can such a scheme possibly work properly?
- Answer: it can work, but only by using *virtual* inheritance!

```
class Radio : virtual public Appliance;
class AlarmClock : virtual public Appliance;
class ClockRadio : public Radio, public AlarmClock;
```

  - Now a `Clockradio` has just one `Appliance` portion, not two
- See demo code in `~mikec/cs32/demos/multi-inherit`
- But note: hierarchy is still messed up, and still lots of chances for ambiguity – best to avoid multi-inheritance!

How do virtual functions work?

- Not exactly magic, but safe to consider it so
- `virtual` tells compiler to “wait for instructions” until the function is used in a program
- So the compiler creates a virtual function table for the class, with pointers to all virtual functions
- In turn, every *object* of such a class will be made to store a pointer to its own class's virtual function table
- At runtime: follow the pointers to find the code!