Virtual functions – concepts

- **Virtual**: exists in essence though not in fact
- Idea is that a virtual function can be “used” before it is defined
  - And it might be defined many, many ways!
- Relates to OOP concept of **polymorphism**
  - Associate many meanings to one function
- Implemented by **dynamic binding**
  - A.k.a. late binding – happens at run-time
Polymorphism example: figures

- Imagine classes for several kinds of figures
  - Rectangles, circles, and ovals (to start)
  - All derive from one base class: Figure
- All “Figure” objects inherit: void draw()
  - Of course, each one implements it differently!
    
    Rectangle r;
    Circle c;
    r.draw(); // Calls Rectangle class’s draw()
    c.draw(); // Calls Circle class’s draw
- Nothing new here yet …
Figures example cont. – center()

- Consider that base class `Figure` has functions that apply to “all” figures
  - e.g., `center()`: moves figure to screen center
    - Erases existing drawing, then re-draws the figure
    - So `Figure::center()` uses `draw()` to re-draw

- But which `draw()` function will be used?
  - We’re implementing base class `center()` function, so we have to use the base class `draw()` function. Right?

- Actually, it turns out the answer depends on how `draw()` is handled in the base class
Poor solution: base works hard

- Figure class tries to implement draw to work for all (known) figures
  - First devise a way to identify a figure’s “type”
  - Then `Figure::draw()` uses conditional logic:
    ```cpp
    if ( /* the Figure is a Rectangle */ )
        Rectangle::draw();
    else if ( /* the Figure is a Circle */ )
        Circle::draw();
    ...
    ```

- But what if a new kind of figure comes along?
  - e.g., how to handle a derived class `Triangle`?
Better solution: virtual function

- Base class declares that the function is virtual:
  ```
  virtual void draw() const;
  ```
- Remember it means `draw()` exists in essence
- Such a declaration tells compiler “I don’t know how this function is implemented, so wait until it is used in a program, and then get its implementation from the object *instance*.”
- The instance will exist in fact (eventually)
  - Therefore, so will the implementation at that time!
- Function “binding” happens late – dynamically
Another virtual function example
(Sale, DiscountSale, Display 15.11)

- Record-keeping system for auto parts store
  - Track sales, compute daily gross, other stats
  - All based on data from individual bills of sale
- Problem: lots of different types of bills
- Idea – start with a very general Sale class that has a virtual bill() function:
  virtual double bill() const;
- Rest of idea – many different types of sales will be added later, and each type will have its own version of the bill() function
double Sale::savings(const Sale &other) const
{
    return (bill() - other.bill());
}

bool operator < (const Sale &first, const Sale &second)
{
    return (first.bill() < second.bill());
}

- Notice both functions use member function bill()!
A class derived from Sale

class DiscountSale : public Sale {
public:
    DiscountSale();
    DiscountSale(double price,
                 double discount);
    double getDiscount() const;
    void setDiscount(double newDiscount);
    double bill() const;  // implicitly virtual
private:
    double discount;      // inherits price
};
DiscountSale’s bill() function

- First note – it is automatically virtual
  - Inherited trait, applies to any descendants
  - Also note – rude not to declare it explicitly
- Of course, definition never says virtual:
  ```cpp
double DiscountSale::bill() const {
    double fraction = discount/100;
    return (1 - fraction)*getPrice();
}
```
  - Must use access method as price is private
The power of virtual is actual!

- e.g., base class `Sale` written long before derived class `DiscountSale`
- `Sale` had members `savings` and `<` before there was any idea of class `DiscountSale`
- Yet consider what the following code does
  ```cpp
  DiscountSale d1, d2;
  d1.savings(d2); // calls `Sale`'s savings function
  ```
- In turn, class `Sale`'s savings function uses class `DiscountSale`'s bill function.
  Wow!
Clarifying some terminology

- Recall that overloading ≠ redefining
- Now a new term – **overriding** means *redefining a virtual function*
- Polymorphism is an OOP concept
  - Overriding gives many meanings to one name
- Dynamic binding is what makes it all work
- “Thus,” as Savitch puts it, “polymorphism, late binding, and virtual functions are really all the same topic.”
Why not all virtual functions?

● Philosophy issue: pure OOP vs. efficiency
  – All functions are virtual by default in another popular programming language (Java) – there must take steps to make functions non-virtual
  – C++ default is non-virtual – programmer must explicitly declare (except when inherited trait)

● Virtual functions have more “overhead”
  – More storage – for class virtual function table
  – Slower – a look-up step; less optimization
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?**
  ```cpp
  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:
    ```cpp
    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
A sorting hierarchy

See .../demos/sorting
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

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:
  ```cpp
  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!

```c++
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 – try .../demos/`sizeofvirtual.cpp`
- At runtime: follow the pointers to find the code!