- **Lecture Outline Inheritance & virtual functions**
  - **Stroustrup Ch. 12**
- **Basic concept of inheritance**
  - **You specify inheritance by listing whether it is public, private, or protected, and the class from which this class is inheriting**
    - *To declare that a class inherits from another class, the other class declaration must be visible to the compiler - an incomplete declaration won't work.*
    - *public inheritance is the normal form of inheritance; others later*
  - **Each derived class inherits the members from its base classes**
    - *E.g. as if compiler had copy-pasted in member variables and functions from the base class*
      - member variables laid out in declaration order with the derived member variables following the base ones
      - Each class defines a subobject of the whole object.
        - ```
          class Base {
                  int i;
                  string sb;
          };

          class Derived : public Base {
                  int j;
                  int k;
          };

          Derived d;

          d object layout in memory:
          --------------------------
          ---Base subobject ----
          ||      [int i]
          ||      [string sb]
          ---Derived subobject ----
          ||      [int j]
          ||      [int k]
          --------------------------
          ```
        - rule: subobjects and member variables are laid out in declaration order
    - *But the names are also scoped in the original classes*
      - so e.g. if each class has a function of the same name, can disambiguate which you want.
        - void Base::print(): void Derived::print();
        - Derived::print() hides Base::print()
        - ```
          void Derived::foo()
          {
                  print();        // innermost scope is assumed, so Derived::print() is called
                  Base::print();        // can explicitly scope the other one and call it
          ```
        - Derived::print() can call Base::print() - a common idiom.
          - call Base::print() to print out Base class members, the Derived::print to print out derived class members
    - *Multiple inheritance object gets members from more than one base class*
      - no problem if base classes have nothing in common - later
  - **Note on terminology**

- *Class hierarchy is upside-tree, base (root) is at top, most-derived (leaves) at bottom*
  - Or,base is most general, derived are specializations of the base class
- *Note common notation is arrow from base to derived*
  - Derived classes declarations refer to their base, but not vice versa.
  - Base classes have **no** information about their derived classes, but derived classes always defined in terms of their base.
  - Not like human inheritance the parents don't know their kids, but the kids always know their parents
- **Access to members with inheritance**
  - *Public in Base or Derived means the same thing: outsiders have access.*
  - *Private in Base or Derived means the same thing: only members of the class have access.*
    - Derived members can't access Base private - not a member of the Base class.
    - Base members can't access Derived parts - not a member of Derived class.
  - *But protected members of Base class are available to Derived, but not public*
    - derived classes can access, but not outsiders
    - private to the world, public to derived
    - GUIDELINE: avoid protected member variables - better if they are all private
      - protected member functions are fine - these are services provided for derived classes
- **SUBSTITUTION PRINCIPLE**
  - *The best simple interpretation of what inheritance means.*
    - Barbara Liskov (1988)
  - *If D is derived from B, then everywhere you can use a B, you can also use a D, and the program will still be correct.*
    - D is a subtype of B
    - "is-a" relationship D "is-a" B
  - *Example*
    - Employee
      - name, pay
      - get_name(), work()
    - Manager is-a Employee and also has more:
      - e.g. list of employees that it manages
    - Employee e; Manager m.
    - everything valid for Employee is also valid for Manager - can *substitute* m for e
    - e.get_name(), m. get_name() // can use m wherever you used e
    - e.work(), m.work()
  - *Slicing:*
    - e = m;
    - e now has the Employee part of m, but not the Manager part!
    - This is called "slicing" the Manager part of m got "sliced" off.
    - Normally avoided - why would we want to discard part of an object?
    - Most common with inheritance: refer to objects via pointers.

- *Consequence of substitutability: A Base * pointer can point to a Derived object*
  - Any derived object is also a base object, so a base-type pointer can point to any derived-type object.
  - e.g. Base * bp;
  - Derived * dp = new Derived;
  - bp = dp; // no problem at all, is always valid
  - sometimes called an ***upcast*** - casting upwards in the inheritance tree
  - Always valid, always correct, not a special situation at all! By definition it is correct!
  - But notice: When using a Base * pointer, can only access members declared in the Base class!
- **Why use inheritance?**
  - *Reuse implementation code.*
    - Put member variables and functions in the base class, derived classes then include it. To modify, debug, fix it in the base class, it is then fixed everywhere.
    - Another tool for making code well-organized and easy to work with.
  - *Reuse interface.*
    - The base class defines the public interface of a class; by inheriting from this, the derived classes automatically are saying they have the same interface and so can be used in the same way. Turns out to be the most powerful concept for making code well-organized and easy to work with.
    - Involves polymorphism (virtual functions) to really get the punch.
    - Polymorphism means you can talk to objects the same way (same interface), but they act differently (different implementation), depending on the type of object they are.
      - More specifically, call the same function, but each class can have a different implementation of the function.
      - Virtual functions in C++.

- **Constructors and destructors in Inheritance Hierarchies**
  - **Concept: let each class deal with its own construction and destruction.;**
  - **Derived classes can rely on base class always being properly initialized before they get initialized.**
    - *Compiler looks at your code and calls constructors in a different order from what you might expect.*
    - *Each class ctor, dtor deals with the subobject member variables for the class.*
    - *A derived class ctor can supply arguments to its immediate base class ctor, but no higher!*
      - Ensures orderly relationships no cutting out the middleman!
  - **Example:**
    - ```
      class Base {
      public:
           Base() : iB(1), jB(2), kB(3)
           {}
           Base(int x) : iB(1), jB(x)
           {kB = 3;}
      private:
           int iB;
           int jB;
           int kB;
           };

      class Derived : public Base{
      public:
           Derived() : iD(4), jD(5), kD(6)
           {}
           Derived(int x, int y) : Base(x),
                iD(5), jd(y)
           {kD = 7;}
      private:
           int iD;
           int jD;
           int kD;
           };

      class DDerived : public Derived{
      public:
           DDerived() : iDD(7), jDD(8), kDD(9)
           {}
           DDerived(int x, int y, int z) : Derived(x, y),
                iDD(9), jDD(z)
           {kDD = 11;}
      private:
           int iDD;
           int jDD;
           int kDD;
           };

      DDerived dd1;
      DDerived dd2(10, 20, 30);
      ```
      - note: DDerived can't invoke Base ctor in its constructor!
  - **Constructor: Concept: Initialize from the top down, and in order of declaration**
      ```

- *first Base initializers in declaration order, then body, then Derived initializers, then body*
- *Each class does its own initialization; derived classes can rely on base class having initialized first.*
- *Ctor body can assign to any member variables that it has access to, but happens only after the base classes have been fully initialized.*

- **Destructors: de-initialize in reverse order of construction: in reverse declaration order, and bottom up:**
  - *This allows an orderly taking apart of the object by going in reverse order, specific things can be deallocated, etc first, before more general things. Derived classes can rely on base class parts still existing.*

- **Access levels for inheritance of the base class**
  - **public inheritance**
    - *the public members of the base class become public members of the derived class*
      - everybody knows that you've inherited from base and can use base's public interface
    - *protected members of base are accessible to derived*
    - *private members of base stay private to base - inheritance does not make private available outside of a class's own members.*
    - *Only form of inheritance that obeys the substitution principle.*
  - **protected inheritance**
    - *The public members and protected members become protected members of the derived class*
    - *very rarely used - seems to be no good use for it, so nothing more to say.*
  - **private inheritance**
    - *The public and protected members of the base class become private members of the derived class*
    - *Derived class's member functions can use base class's public and protected members, but nobody else can, either clients or further derived classes.*
    - *In other words, private inheritance is nobody else's business.*
      - Used to mean "I want to use Base to help implement Derived internally, but don't want to add Base's public interface to derived. Inherit implementation, but not interface.
        - ```
          class Gizmo {
          public:
                void transmogrify();  // a useful functionality
          };

          class Thing : private Gizmo {
          public:
                void defrangulate() {
                      transmogrify();  // provided by the base Gizmo
                      }
                // can call Gizmo public members, but no client of Thing can access them
          };
          ```
        - a "uses" relationship not really an isa because substitutability is violated
          - `a_Gizmo.transmogrify()` does not mean you can use a_Thing.transmogrify()
          - Substitutability only works for the public interface!
  - **Alternative to private inheritance**
    - *Instead of privately inheriting from X, it is usually preferable to just have a private member variable of type X and call its public members to do the work. Reusing the implementation, but without changes to interface, or without any complications of inheritance.*
      - ```
        class Thing {
        public:
              void defrangulate() {
                    my_gizmo.transmogrify();
                    }
        private:
              Gizmo my_gizmo;
              has a Gizmo member and can call its public functions to do work,
              but not accessible to anybody else
        };
        ```
      - "has-a" relationship
      - usually works better than private inheritance - keeps the design simpler

- **Virtual functions and polymorphism basics**
  - **The most important feature of OOP.**
  - **What not to do: switch or *branch on type:***
    - *favorite example is shapes in a graphical program - different shapes, objects in a container, and we want to draw each one.*
    - *suppose each kind of shape carried a type code*
      -
```
enum Shape_e {CIRCLE, SQUARE, etc};

class Shape {
public:
     Shape(shape_code_e code_) : code(code_) {}
     Shape_e get_shape_code() const
          {return code;}
private:
     Shape_e code;
};

class Circle : public Shape {
public:
     Circle() : Shape(CIRCLE) {}
private:
     // data
};

class Square : public Shape {
public:
     Square() : Shape(SQUARE) {}
private:
     // data
};
```
    - *decide how to draw each shape based on its type*
      -
```
Shape * theshape = get_next_shape_ptr();
switch(theshape->get_code()) {
     case CIRCLE:
          frameoval using the data
          break;
     case SQUARE:
          framerect using the data
          break;
     case TRIANGLE:
          draw three lines using the data
          break;
     // etc
     }
```
    - *could be done wiith if-else, uglier, but same concept - called **branch** on type logic.*
    - *This is evil, sinful!*
      - gets very slow if large number of classes
        - how does a switch really work?
      - a maintenance nightmare - how do you make sure the code is right?

      - to add another shape, find every time the code is used, and add the appropriate branch/case
    - ugly, ugly, error prone!
- **What virtual funcs and polymorphism are for!**
  - *LET THE COMPILER DO THE WORK OF KEEPING TRACK OF THINGS!*
  - *work in an inheritance relationship*
    - a pointer to a derived type can always be converted to a base type
    - so can refer to different kinds of shapes with a Shape * pointer
  - *shape has virtual draw function*
  - *each object knows how to draw itself:*
    - ```
      class Shape {
           virtual void drawself();

      void Circle::drawself()
           frameoval

      void Square::drawself()
           framerect

      void Triangle::drawself()
           draw three lines
      ```
  - *In client code, simply tell the object to draw itself!*
    - ```
      Shape * theshape = get_next_shape_ptr();
      theshape->drawself();
      ```
  - *`p->drawself();` // **automagically** calls the drawself for a Circle if p points to a Circle, a Square if p points to a square!*
    - if shapes is a container of Shape * pointers, then tell them all to draw themselves:
    - `for_each(shapes.begin(), shapes.end(), mem_fun(&Shape::drawself));`
  - *Shape has a virtual draw function*
    - the actual function executed is chosen at **run time**, based on the **actual type** of the pointed-to object
    - you OVERRIDE a virtual function to get class specific behavior
      - Note on terminology: you OVERRIDE virtual functions not overload them!
  - *How's it work? very efficient - Later*
- **How to declare and use virtual functions**
  - *must put "**virtual**" on the function declaration in the BASE class*
    - MUST BE IN THE CLASS DECLARATION, can't be in the .cpp file for a class
  - *it is inherited thereafter; all functions with the **same signature** in all derived classes are now virtual*
    - putting "virtual" on them is commonly done as a reminder, but it is optional
  - *call the function with a pointer or reference of BASE class type.*
    - compiler will generate code to look up and branch to the right function for the object
    - if a class hierarchy has multiple levels, then you can do virtual calls using pointers at any level, but not very common. Usually there is a single BASE class for a tree of classes that is the interface for the whole tree.

  - *a derived class can provide its own definition of a virtual function OVERRIDES any inherited one.*
  - *if doesn't provide its own definition, it gets the most derived, "closest" inherited one.*
- **If objects are deleted via a Base class pointer, you must declare Base destructor virtual.**
  - *Compiler will then make a virtual call to the derived class destructor when you delete it.*
  - *Subobjects won't get deleted otherwise!*
  - *A common pattern: create objects and point to each one with a base class pointer; then delete them with that pointer.*

- **Basic techniques with virtual functions**
  - **The fundamental Object-Oriented Programming Technique: Use a polymorphic class hierarchy - inheritance with virtual functions.**
    - *Arrange a set of classes in an inheritance hierarchy. The base class defines the interface to all of the derived classes. The virtual functions show where the behavior of derived classes might need to be different. Derived classes override the virtual functions to produce their own behavior.*
    - *By doing virtual calls through a base class pointer, the client code can access the capabilities regardless of the specific type of the object.*
    - *By a good choice of base class and virtual functions, we can arrange it so that the client code for a family of classes does not know about the specific classes or objects; talks to them only through the base class interface!*
      - Each class type knows what to do when called; client is not responsible for keeping track of who should do what!
    - *This means that derived classes and objects can be added, modified, etc without the client code being modified.*
      - If a feature must be changed, change it only in the affected classes, rest remains untouched.
    - *Makes it possible to add features without forcing change everywhere:*
      - "Add features by adding code, not by changing code!"
      - "Hide what changes under a base class interface."
  - **Technique: "default" virtual functions**
    - *choose a "default" virtual function definition that applies to every derived class unless overridden*
    - *captures idea of the common, default behavior for all objects in the hierarchy*
  - **Let Derived class supply additional specialized work to the work done by the Base class.**
    - *Technique: a virtual function can have a definition at each level of the hierarchy; the derived version can call its base version before or after doing its own work.*
      - Example: Each class has a print() member function that outputs information about that classes member variables. Each class is responsible for outputting its own information.
        - Derived::print() calls Base::print() to output the Base member variables, then Derived::print() adds the information about its own member variables.
        - Repeat at each level of the hierarchy.
        - Contrast to the most Derived::print() accessing the data members of each Base class and printing it out. Code is highly repetitious, difficult to maintain. E.g. suppose we add another variable to the Base class ....
  - **A non-virtual member function can call a virtual member function.**
    - *Use to provide a class-specific piece of functionality in the context of an ordinary member function.*
      - Example: non-virtual of(), virtual vf():
        ```
        Base::of()
        {
            vf();  // does virtual call if Class1 has Derived classes
        }
        ```
      - Why this works: if a member function calls a member function, the call is through the this pointer:
        - `this->vf();`
      - if this object's class is a base class, the `this` pointer has type `Base *`, so a virtual call will be made.
      - Can define a base class non-virtual function to do "setup" work common to all derived classes, and then it can call a virtual function that does the specific work for each derived class.
        - Called "non virtual interface pattern"
        - Separates interface in base class from details of implementation provided in derived classes

- **Technique: label a class as abstract**
  - *A way to convey the idea that there is no such object, only subtypes of this object*
    - You can have an actual concrete real dog or a cat, but you can't have a "mammal" or "animal" or "thing" by itself. These are abstract categories for the concrete objects.
  - *Label a class as an **abstract base class** (ABC) by giving it at least one "**pure virtual function**"*
    - e.g. in ABC declaration: virtual void foo () = 0; // stupidest syntax in C++
      - should be able to say "class is abstract" and "virtual function is pure" but no, can't do!
  - *Declaring a pure virtual function does two things:*
    - Each leaf class **must** supply or inherit an overriding definition of the pure virtual function.
      - So a "pure virtual" declaration means "this must get overridden"
      - If a class doesn't get an overriding declaration, then it can't be instantiated because the compiler and linker won't know what function to call on it. So it is "abstract".
    - This class (ABC) is now an abstract class - you can't create an object of this class.
      - Compiler will not let you declare an object from an abstract class
        - A a; // error "illegal use of abstract class"
      - Because lacks a usable definition of the virtual function.
      - Any derived class that doesn't have an override definition is itself also an abstract class.
  - *Now that this is perfectly clear, let's muddle it up.*
    - Note that normally any definition of the pure virtual function won't get called. All derived classes have their own version of the function, and you can't create an object of the abstract class which would normally result in the function being called.
      - If all leaf classes have (or inherit) an overriding definition of the virtual function foo, then a virtual call like base_ptr->foo(); will never reach the Base::foo!
    - Therefore, normally no definition of the pure virtual function is provided in the ABC class; doesn't have to be one because it will normally never be called.
    - But you can provide a definition if you want, and you can call it explicitly: Base::foo();
      - explicit class qualification turns off the virtual function call - you say explicitly which version of the function you want!
      - base_ptr->Base::foo();
      - Rarely, you need to do this.
  - *Suppose you want a class to be abstract, but there is no obvious choice of which virtual function to make a pure virtual function. What do you do?*
    - Convention: make the **destructor function** a pure virtual function (it should be virtual anyway, in this context), but you must provide a definition for it, even if it is an empty definition.
      - If you don't declare a destructor function, the compiler will synthesize one for you. But if you declare it, the compiler won't synthesize it. But the base class destructor will need to get called, so if declared, it must be defined.
      - Example:
        - In Base.h
          ```
          class Base {
          Base (/* whatever */);   // constructor
          virtual ~Base() = 0;     // pure virtual destructor
          /* etc */
          };
          ```
        - in Base.cpp
          ```
          Base::~Base()
          { /* must have definition, even if empty */}
          ```

    - You can often get part of the effect of an abstract base class by making it impossible for the client to instantiate objects of the class without declaring a pure virtual function.
        - Make Base ctor protected - this way, only Derived class objects can be instantiated!
            - Needs to be protected, not private, so that Derived class ctor can invoke Base class ctor.
- **Technique: a base class that specifies only an "interface" to a set of classes that are going to be used polymorphically - called an "interface class".**
    - *All functions might be pure virtual*
        - all must be overridden
    - *The class just specifies how the subclasses are accessed by a common interface*
    - *A cool idea in Java, special construct called an "interface," not a class*
    - *A true interface class will have no member variables, but often handy to have a bit of functionality in such a class.*
    - *Stroustrup example of interface widgets*
        - insulating the main body of an application from the details of which GUI toolkit is being used
        - the abstract base class for the widget provides the interface for the whole family of widgets.
            - main body of code just needs to know about this class to use the widget.
        - to create the specific widget, new BB_slider not insulated from details of BB toolkit, etc.
            - so use a factory instead
            - somewhere create a BB_maker, as it to create a Slider, get the pointer, use it through the interface.
        - main body of code doesn't need to know anything about the BB versus the CW toolkit, etc.
        - "figure out what varies and encapsulate it"
        - example of a design pattern will return to.

- **Specifics on Inheritance and Overriding relationships**
  - **Illustrate rules for which functions are inherited and overridden.**
    - *Work through a specific example in detail.*
  - **Class Hierarchy, objects, and pointers for examples**
    - *A*
      - B inherits from A
        - C inherits from B
        - D inherits from B
      - E inherits from A
    - *Object declarations*
      - A a; B b; C c; D d; E e;
    - *Pointer declarations and initializations (could also do with new)*
      - A * pa = &a;
B * pb = &b;
C * pc = &c;
D * pd = &d;
E * pe = &e;
  - **Facts about ordinary functions with different names**
    - *A ofa()*
      - B ofb()
        - C ofc()
        - D ofd()
      - E ofe()
    - *Inherited functions*
      - A has ofa
      - B has ofa, ofb
        - C has ofa, ofb, ofc
        - D has ofa, ofb, ofd
      - E has ofa, ofe
    - *good and bad calls with an object*
      - a.ofa(); // ok
      - a.ofb(); // error doesn't have
      - c.ofa(); // ok
      - c.ofb(); // ok
      - c.ofe()// error
    - *good and bad calls with a pointer*
      - pa ->ofa(); // ok type of pointer gives class it is suposed to be
      - pa ->ofb(); // error doesn't have
      - pc-> ofa(); // ok
      - pc-> ofb(); // ok
      - pc-> ofe()// error

- *good and bad calls within a member function*
  - void C::ofc()
    ```
    {
        ofa(); // ok
        ofb(); // ok
        ofc(); // ok, recursive
        ofd() // error!
    }
    ```
- **Facts about ordinary functions same names, same signature**
  - *A A::of()*
    - B of() shadows A::of()
      - C of() shadows B::of()
      - D of() shadows B::of()
    - E of() shadows A::of()
  - *Inherited functions*
    - A has of
    - B has A::of(), B::of()
      - C has A::of(), B::of(), C::of()
      - D has A::of(), B::of(), D::of()
    - E has A::of(), E::of()
  - *good and bad calls with an object*
    - a.of(); // ok must mean A::of()
    - a.A::of(); // ok same thing
    - a.B::of(); // error doesn't have
    - c.of(); // ok must mean C::of()
    - c.B::of(); // ok
    - c.A::of(); // ok
    - c.E::of(); // error doesn't have
  - *good and bad calls with a pointer*
    - pa ->of(); // ok must mean A::of()
    - pa ->B::of(); // error doesn't have
    - pb ->of(); // ok must mean B::of()
    - pc-> A::of(); // ok
    - pc-> B::of(); // ok
    - pc-> C::of(); // ok
    - pc-> of(); // ok must mean C::of()
    - pc-> E::of()// error
  - *good and bad calls within a member function*
    - void C::of()
      ```
      {
          B::of(); // ok
          A::of(); // ok
      ```

C::of(); // ok, recursive
of(); // ok, recursive same thing
E::of() // error!
}

- **Simple situation with virtual functions always have same name, same signature, a function declared in every class**
  - *A virtual vf() {...}*
    - B virtual vf() {...}
      - C virtual vf() {...}
      - D virtual vf() {...}
    - E virtual virtual vf() {...}
  - *Inherited functions*
    - A has A::vf
    - B has A::vf(), B::vf()
      - C has A::vf(), B::vf(), C::vf()
      - D has A::vf(), B::vf(), D::vf()
    - E has A::vf(), E::vf()
  - *good and bad calls with an object same as with ordinary functions*
    - a.vf(); // ok must mean A::vf()
    - a.A::vf(); // ok same thing
    - a.B::vf(); // error doesn't have
    - c.vf(); // ok must mean C::vf()
    - c.B::vf(); // ok
    - c.A::vf(); // ok
    - c.E::vf(); // error doesn't have
  - *good and bad calls with a pointer CAN BE THE same as with ordinary functions*
    - if use scope qualifier to say which function, works just the same!
      - pa ->A::vf(); // get this one
      - pa ->B::vf(); // error doesn't have
      - pc-> A::vf(); // ok
      - pc-> B::vf(); // ok
      - pc-> C::vf(); // ok
      - pc-> E::vf()// error
    - if pointer to most derived class, and plain name, works just the same
      - pc-> vf(); // ok must mean C::vf()
      - pd ->vf() // ok means D::vf()
  - *POLYMORPHISM MAGIC*
    - happens with a pointer to a base class that points to an object whose class is in the hierarchy
      - REMEMBER can convert upwards
        - assign a pointer to derived to a pointer to base
      - A * p;

- call with the base-class pointer to unqualified name of a function that is virtual in that base class
  - p = &a;
    - p ->vf(); // get's A::vf()
  - p = &b;
    - p ->vf(); // get's B::vf() OVERRIDES A::vf
  - p = &c;
    - p ->vf(); // get's C::vf() OVERRIDES A::vf and B::vf
  - p = &d;
    - p ->vf(); // get's D::vf() OVERRIDES A::vf and B::vf
  - p = &e;
    - p ->vf(); // get's E::vf() OVERRIDES A::vf
- can have more than one base class in a single inheritance hierarchy B is a base for C and D
  - B * p;
  - p = &b;
    - p -> vf(); // get's B::vf()
  - p = &c;
    - p -> vf(); // get's C::vf()
  - p = &d;
    - p -> vf(); // get's D::vf()
- RULE: unqualified call of a virtual function through a pointer of the base class type calls the function for the object's actual class
  - note: calling with arrow-operator equivalent is also a virtual call
    - (*p).vf();
  - *was the object identified with a pointer, as opposed to the actual object?*

- **Complex situation with multiple virtual functions only some declared in each class**
  - *All must be declared in the base class*
  - *Each class and either inherit or override the virtual functions of its base class*
  - *A virtual vf1() {...}, virtual vf2() {...}*
    - B virtual vf1() {...} // overrides A::vf1, inherits A::vf2
      - C virtual vf2() {...} // inherits B::vf1, overrides A::vf2
      - D virtual vf1() {...} // overrides B::vf1, inherits A::vf2
    - E // inherits A::vf1, inherits A::vf2
  - *Polymorphic calls*
    - basically: call the most-derived-class virtual function for the object.
    - call with the base-class pointer to unqualified name of a function that is virtual in that base class
      - A * p;
      - p = &a;
        - p ->vf1(); // get's A::vf1()
        - p ->vf2(); // get's A::vf2()
      - p = &b;

- p ->vf1(); // get's B::vf1()
- p ->vf2(); // get's A::vf2()
- p = &c;
  - p ->vf1(); // get's B::vf1()
  - p ->vf2(); // get's C::vf2()
- p = &d;
  - p ->vf1(); // get's D::vf1()
  - p ->vf2(); // get's A::vf2()
- p = &e;
  - p ->vf1(); // get's A::vf1()
  - p ->vf2(); // get's A::vf2()

- **How are virtual functions implemented?**
  - **Slightly different example - add another base class function**
    - *A virtual vf1() {...} virtual vf2() {...} virtual vf3() {...}*
      - B virtual vf1() {...} // overrides A::vf1, inherits A::vf2
        - C virtual vf2() {...} // inherits B::vf1, overrides A::vf2
        - D virtual vf1() {...} // overrides B::vf1, inherits A::vf2
  - **Focus on base classes: What are their possible virtual calls through A * pointer?**
    - *list in order of function name*
    - *class C:*
      - B::vf1
      - C::vf2
      - A::vf3
    - *class D*
      - D::vf1
      - A::vf2
      - A::vf3
  - **Compiler puts these facts together in a table of virtual function addresses for each class, the vtable, tucked somewhere in memory. Each OBJECT in the class contains a pointer to this table, the vptr now e.g. 4 bytes bigger, no longer just simple struct-like collection of member variables**
    - *a C object:*
      - A member vars
      - vptr points to C's vtable
      - B member vars
      - C member vars
    - *a D object*
      - A member vars
      - vptr points to D's vtable
      - B member vars
      - C member vars
    - *C's vtable index, address's*
      - [0] &B::vf1
      - [1] &C::vf2
      - [2] &A::vf3
    - *D's vtable index, address's*
      - [0] &D::vf1
      - [1] &A::vf2
      - [2] &A::vf3
  - **Notice how the vptr is at same place from the beginning of the object, so can be found for either C or D object**
  - **Notice how each virtual function name corresponds to an index:**
    - *[0] vf1*

- *[1] vf2*
- *[2] vf3*
- **Each call through a A * pointer goes to the function at that vtable index, found using vptr**
  - *A * p = &c or &d*
  - *p -> vf1 call function whose address is in p->vptr[0]*
  - *p -> vf2 call function whose address is in p->vptr[1]*
  - *p -> vf3 call function whose address is in p->vptr[1]*
  - *or, in terms of a function pointer call:*
  - *p -> vf1(arg) would be (*p->vptr[0])(ptr, arg);*
- **Trade-offs**
  - *A tiny tad slower than a regular function call, a whole lot faster than switch-on-type logic*
  - *Could be beat by storing address of function directly in a member variable, but very difficult programming (you have to write many lines of code to initialize the function pointer member variables, hard to maintain, and saves only the subscripting operation.*