CSE 165/ENGR 140 Intro to Object Orient Program

Lecture 13 – Polymorphism (2)

Announcement

- Reading assignment
  - Ch. 15

The Virtual Table

```
//: C15:Instrument4.cpp
enum note { middleC, Csharp, Eflat }; // Etc.
class Instrument {
public:
  virtual void play(note) const { cout << "Instrument::play" << endl; }
  virtual char* what() const { return "Instrument"; }
  // Assume this will modify the object:
  virtual void adjust(int) {}
};

class Wind : public Instrument {
public:
  void play(note) const { cout << "Wind::play" << endl; }
  char* what() const { return "Wind"; }
  void adjust(int) {}
};

class Percussion : public Instrument {
public:
  void play(note) const { cout << "Percussion::play" << endl; }
  char* what() const { return "Percussion"; }
  void adjust(int) {}
};

class Stringed : public Instrument {
public:
  void play(note) const { cout << "Stringed::play" << endl; }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};
```

The Virtual Table

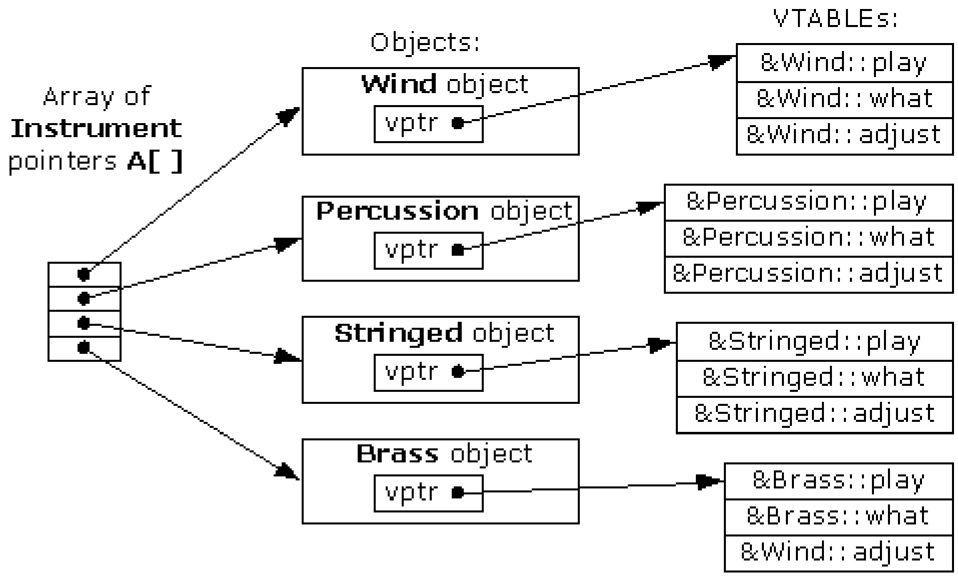
```cpp
///: C15:Instrument4.cpp (continue…)
class Brass : public Wind {
public:
  void play(note) const { cout << "Brass::play" << endl; }
  char* what() const { return "Brass"; }
};
void tune(Instrument& i) {i.play(middleC);}
// New function:
void f(Instrument& i) { i.adjust(1); }

// Upcasting during array initialization:
Instrument* A[] = {
  new Wind,
  new Percussion,
  new Stringed,
  new Brass,
};

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  f(flugelhorn);
} ///:~
```

Output:

```
Wind::play

Percussion::play

Stringed::play

Brass::play
```

The Virtual Table

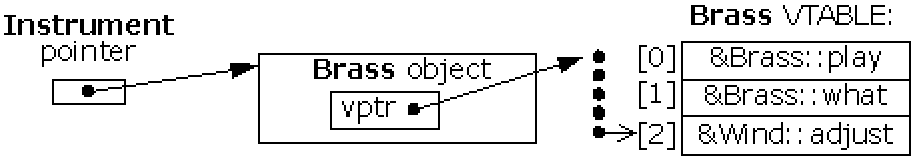
- Here are the vptrs and vtables created:



  - Each class has 1 vptr point to its vtable.
    - Objects of the same class can share vtables.
  - Each vtable keeps pointers to all virtual methods of an object.

The Virtual Table

- Example:
  - when a call to Brass::adjust is made, the compiler will say "call vptr+2" :



  - the correct pointers are stored at object creation
  - the correct methods to call can then be found at run-time even after upcasting (late binding).

Abstract Classes

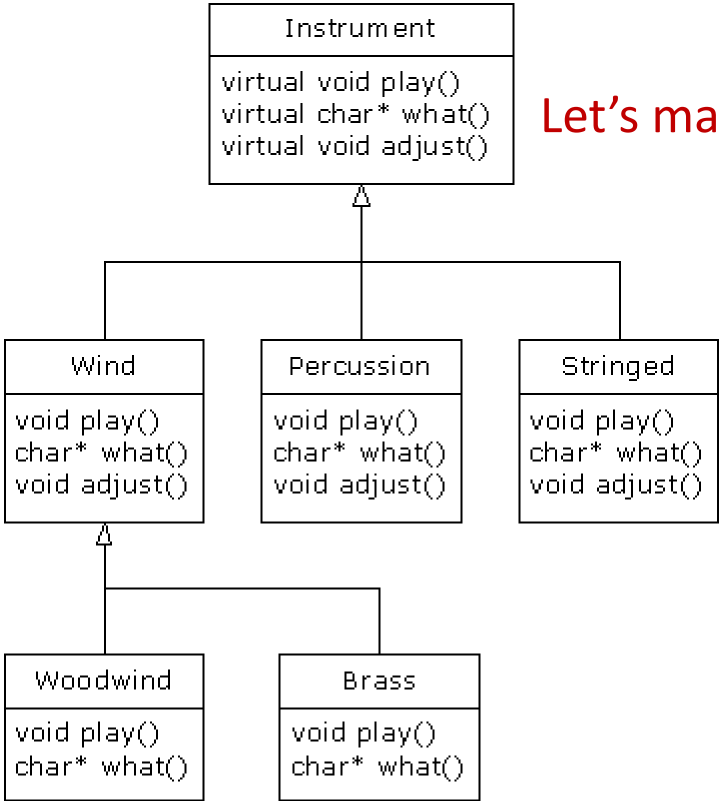
- When a class only presents an interface for derived classes
  - it cannot be instantiated
  - it sets a standard interface for extensions

- How to declare an abstract class:
  - just declare at least one "***pure virtual method***" with the "=0" syntax:

```
virtual void f()=0;
```

Abstract Classes

- Example:
  - Our “Instrument” class is a good candidate for becoming an abstract class.



Let’s make this interface abstract

Abstract Classes

```
//: C15:Instrument5.cpp - Pure abstract base classes
    class Instrument { public:
      // Pure virtual methods, all of them MUST be overridden by a derived class:
      virtual void play(note) const = 0;
      virtual char* what() const = 0;
      virtual void adjust(int) = 0;
    };

    class Wind : public Instrument { public:
      void play(note) const { cout << "Wind::play" << endl; }
      char* what() const { return "Wind"; }
      void adjust(int) {}
    };

    class Percussion : public Instrument { public:
      void play(note) const { cout << "Percussion::play" << endl; }
      char* what() const { return "Percussion"; }
      void adjust(int) {}
    };

    class Woodwind : public Wind { // Woodwind does not need to override all methods
     public:                       // since it inherits the non-abstract class Wind
      void play(note) const { cout << "Woodwind::play" << endl; }
      char* what() const { return "Woodwind"; }
    };
```

Abstract Classes

```
//: C15:Instrument5.cpp - Pure abstract base classes
(continue…)

    int main() {
      Instrument i; // not possible, will generate an error!
      Wind flute;
      Percussion drum;
      Woodwind recorder;
      ...
    }
```

Abstract Classes

- Extending Virtual Methods
  - Notice that the virtual method of the derived class (at the bottom of the hierarchy) will override the ones of the base classes

  - Sometimes we want to “add” and not really to “override”
    - For that, from your overriding method, you can always explicitly call the base class implementation using the scope operator

Abstract Classes

```
// SysWindow provides an abstract interface for windows to interact with the system
class SysWindow {
public:
  virtual void draw ()=0;    // Notice that virtual methods may have an implementation!
  virtual int handle ( const Event& e )=0;
};


// in SysWindow.cpp:
void SysWindow::draw ()
 {
   // make critical settings (but nothing to draw)
   glViewport ( ... );
   glEnable ( ... );
 }


int SysWindow::handle ( const Event& e )
 {
   // test if there is a UI attached (but SysWindow itself does not react to events)
   if ( user_interface_attached() ) return ui()->handle(e);
   return 0;
 }
```

Abstract Classes

```
class MyWindow : public SysWindow { // MyWindow implements my application
public:
  virtual void draw ();
  virtual int handle ( const Event& e );
};

// in MyWindow.cpp:
void MyWindow::draw ()
 { // first call the base class settings:
   SysWindow::draw();

   // now draw what you need to draw:
   drawWindowTitleBar();
   drawWindowDecoration();
   drawWindowContents();
 }

int MyWindow::handle ( const Event& e )
 { // first let the base class check for UI events:
   if ( SysWindow::handle ( e ) ) return 1;

   // now check events that are interesting for my window:
   if ( e.type == MouseClick )
    { moveToTop(); return 1; }
   else if ()
    { ... }

   // if event not useful:
   return 0;
 }
```

By calling the base class methods first we are able to ADD functionality to the base class implementation

Abstract Classes

```
class MyWindow : public SysWindow { // MyWindow implements my application
public:
  virtual void draw ();
  virtual int handle ( const Event& e );
};

// in MyWindow.cpp:
void MyWindow::draw ()
 { // first call the base class settings:
   SysWindow::draw();
   // now draw what you need to draw:
   ...
 }

int MyWindow::handle ( const Event& e )
 { // here we check a high-priority event that we do not
   // want the base class to handle:
   if ( e.type == UIClick ) { doSomethingElse(); return 1; }

   // ok now let the base class do its work:
   if ( notInFocus() )
    if ( SysWindow::handle ( e ) ) return 1;

   // finally check my events:
   if ( e.type == MouseClick )
    { moveToTop(); return 1; }
   ...
   // if event not useful:
   return 0;
 }
```

We can also selectively override some of the behavior of the base class in different ways

Wakeup

- https://youtu.be/6Z-y5-7Ywko

Abstract Classes

```
// Example with more complex derivation hierarchies:

class AppRect // Generic Graphical Object
public:
  virtual void draw ()=0;
  virtual int handle ( const Event& e )=0;
};

// Here is one specific graphical object:
class RectButton : public AppRect {
public:
  void draw (); // draw a button-like object
  int handle ( const Event& e ); // respond to mouse clicks
};
```

Abstract Classes

```
// Example with more complex derivation hierarchies: (continue…)

// Here is a new specialized abstract class:
class RectData : public AppRect {
public:
  bool load ( const char* file );
  bool save ( const char* file );
  Vec computeAverage ();
  ...
  virtual void draw ()=0; // the implementation here shows the data points
  virtual int handle ( const Event& e ); // interact (pan,zoom,etc) with the data
  virtual void swapmem ( void* pt, unsigned bytes )=0; // new pure method for
                                                       // custom memory management
};

class RectLineGraph : public RectData {
public:
  virtual void draw (); // draw a line graph from the data
  virtual void swapmem ( void* pt, unsigned bytes )=0; // may use disk swap
};

class RectBarGraph : public RectData {
public:
  ...
};
```

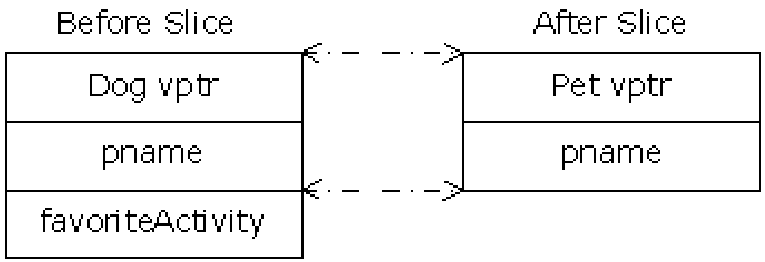
New Abstract Classes
may be created at any point

Object Slicing – passing by values

```
////: C15:ObjectSlicing.cpp

class Pet {
  string pname;
public:
  Pet(const string& name) : pname(name) {}
  virtual string name() const { return pname; }
  virtual string description() const {
    return "This is " + pname;
  }
};

class Dog : public Pet {
  string favoriteActivity;
public:
  Dog(const string& name, const string& activity)
    : Pet(name), favoriteActivity(activity) {}
  string description() const {
    return Pet::name() + " likes to " +
      favoriteActivity;
  }
};

void describe(Pet p) { // Slices the object
  cout << p.description() << endl;
}
```

Output:

```
This is Alfred
This is Fluffy
```



Before Slice

Dog vptr

pname

favoriteActivity

After Slice

Pet vptr

pname

```
int main() {
  Pet p("Alfred");
  Dog d("Fluffy", "sleep");
  describe(p);
  describe(d);
}
```

Avoid Object Slicing With Pointers

```
void describe(Pet *p) {
  cout << p->description() << endl;
}


int main() {
  Pet* p = new Pet("Alfred");
  Dog* d = new Dog("Fluffy“, “sleep”);
  describe(p);
  describe(d);
}
```

Output:

```
This is Alfred
Fluffy likes to sleep
```

Overload vs Override

- Sound similar, but they are very different things
- What’s the difference?