CSE 165/ENGR 140 Intro to Object Orient Program

Lecture 12 – Polymorphism

Announcement

- Reading assignment
  - Ch. 15

Redefining versus overriding methods

- Redefinition of Methods
  - Methods with same name in a base and derived classes are disambiguated by the type of the object.
- Overriding Methods
  - The virtual keyword allows to call a descendant method even if the object being used is of the base class type.
  - Makes sense only when ***upcasting*** is used.
- Polymorphism
  - The use of virtual methods is the key concept behind polymorphism.

Redefining versus overriding methods

```
class Animal
 { public:
    void eat () { cout<<“I eat generic food\n"; }
    virtual void fur () { cout<<“I have fur\n"; }
 };

class Cat : public Animal
 { public:
    void eat () { cout<<“I eat cat food\n"; } // 1) method redefined
    void fur () { cout<<“I have fluffy cat fur\n"; } // 2) overrided!
  };

void main ()
 {
   Cat cat;
   cout<<cat.eat();
   Animal* animal = (Animal*) &cat; // 3) upcast cat to a pointer to Animal
   cout<< animal->eat();   // 4) will print: “I eat generic food”
   cout<< animal->fur();   // 5) will print: “I have fluffy cat fur”
 }
```

Polymorphism

- **Inheritance** lets us inherit attributes and methods from another class
- **Polymorphism** uses those methods to perform different tasks
- Polymorphism in C++ is achieved with **virtual** functions.
  - allows an object to have its behavior extended, without the need to know about derived types, or if it was derived or not.

- It can be seen as the third essential feature in objected oriented programming.
  - the other two are:
    - data abstraction and
    - inheritance

Example Without Virtual Methods

```cpp
///: C15:Instrument2.cpp - Inheritance & upcasting
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.

class Instrument {
public:
  void play(note) const { cout << "Instrument::play" << endl; }
};

// Wind objects are Instruments because they have the same interface:
class Wind : public Instrument {
public:
  // Redefine interface function:
  void play(note) const { cout << "Wind::play" << endl; }
};

void tune(Instrument& i) { // Takes in an Instrument type
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
}
```

method Instrument::play()
will be called here...

Example With Virtual Methods

```
///: C15:Instrument3.cpp - Inheritance & upcasting
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.

class Instrument {
public:
  virtual void play(note) const { cout << "Instrument::play" << endl; }
};

// Wind objects are Instruments because they have the same interface:
class Wind : public Instrument {
public:
  // Override interface function:
  void play(note) const { cout << "Wind::play" << endl; }
};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
}
```

virtual methods will cause “late biding”

method **Wind::play()** will now be called!

Wake up!

- https://youtu.be/lSYEnvl3LeE

The Virtual Table

- How C++ knows which method to call?
  - for each class with a virtual method a hidden VTABLE is created.
  - each class with a virtual method will have a hidden pointer VPTR pointing to its VTABLE.

- The extra hidden code achieves the polymorphism
  - compilers may implement their virtual tables in different ways, there is no standard for how the “hidden code” has to be.

The Virtual Table – example 1

```
// C15:Sizes.cpp - Object sizes with/without virtual functions
class NoVirtual {
  int a;
public:
  void x() const {}
  int i() const { return 1; }
};

class OneVirtual {
  int a;
public:
  virtual void x() const {}
  int i() const { return 1; }
};

class TwoVirtuals {
  int a;
public:
  virtual void x() const {}
  virtual int i() const { return 1; }
};

int main() {
  cout << "int: " << sizeof(int) << endl;
  cout << "void* : " << sizeof(void*) << endl;
  cout << "NoVirtual: " << sizeof(NoVirtual) << endl;
  cout << "OneVirtual: " << sizeof(OneVirtual) << endl;
  cout << "TwoVirtuals: " << sizeof(TwoVirtuals) << endl;
}
```

Output:

```
int: 4
void* : 4
NoVirtual: 4
OneVirtual: 8
TwoVirtuals: 8
```

Only 1 VPTR is
added even when a
class has two virtual methods:

let’s check what is printed.

The Virtual Table

```cpp
///: 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

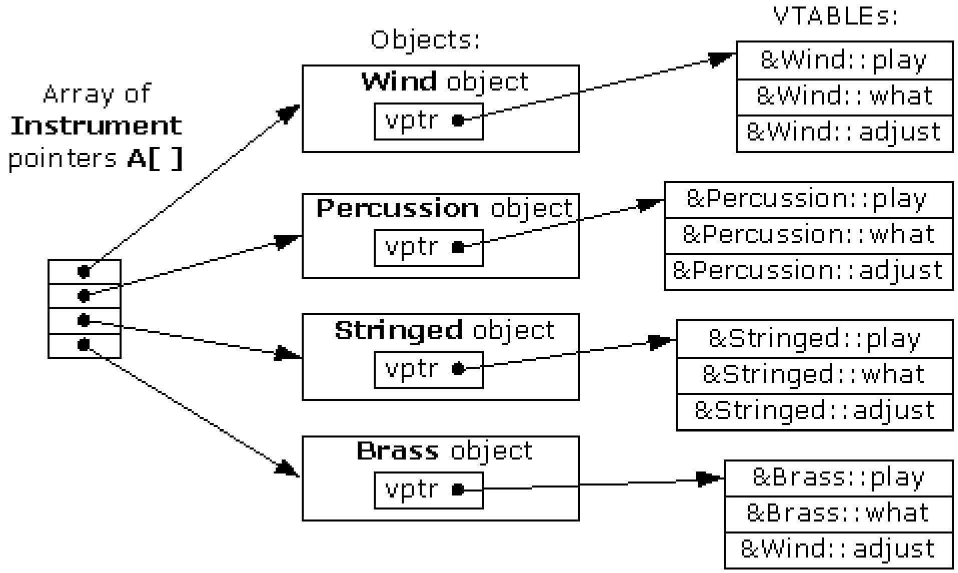
```
///: C15:Instrument4.cpp (continue…)
class Brass : public Wind {
public:
  void play(note) const { cout << "Brass::play" << endl; }
  char* what() const { return "Brass"; }
};
void tune(Instrument& i) {i.play(middleC);}
// New function:
void f(Instrument& i) { i.adjust(1); }

// 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

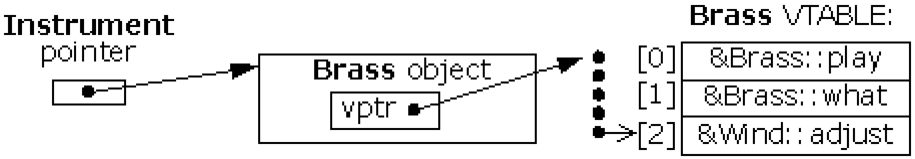
- Here are the vptrs and vtables created:



  - 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).

The Virtual Table

- If methods are not declared virtual:
  - then they are simple methods, no vtable overhead is used
  - polymorphism is limited.
- Why virtual methods are not always employed in C++?
  - Idea is: “If you don't use it, you don't pay for it”

Abstract Classes

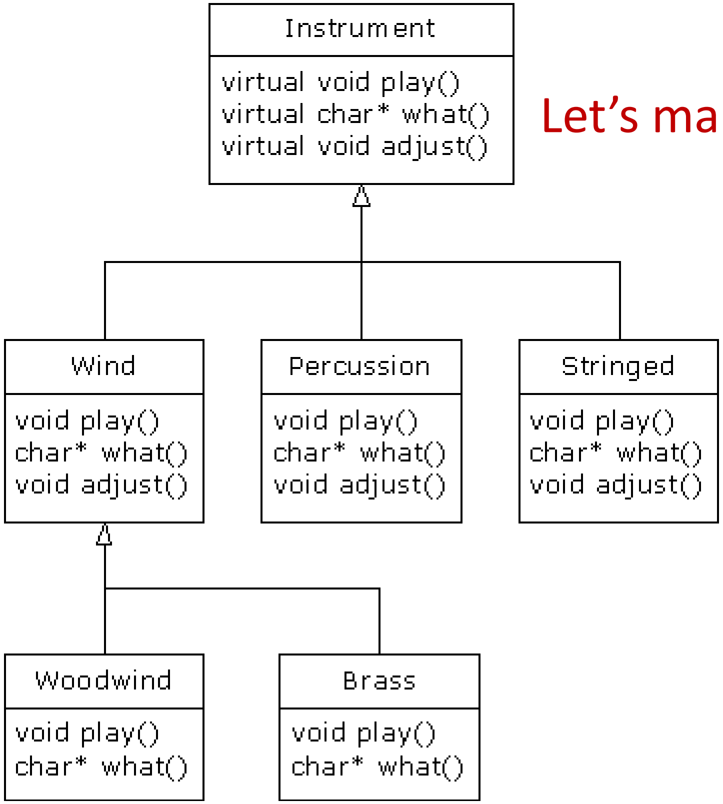
- When a class only presents an interface for derived classes
  - it cannot be instantiated
  - it sets a standard interface for extensions

- 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;
      ...
    }
```