CSE 165/ENGR 140
Intro to Object Orient Program
Lecture 12 – Polymorphism
Announcement

- Lab #4
  - Due date: 3/5 (Thursday) at 11:59PM.

- Mid-term exam
  - Tuesday 3/31

- 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 A
{
    public:
    void method1 () { cout<<"A::method1()\n"; }
    virtual void method2 () { cout<<"A::method2()\n"; }
};

class B : public A
{
    public:
    void method1 () { cout<<"B::method1()\n"; } // 1) method redefined
    virtual void method2 () { cout<<"B::method2()\n"; } // 2) overridden!
};

void main ()
{
    B b;
    A* a = (A*) &b; // 3) upcast b to a pointer to A
    a->method1(); // 4) will print: "A::method1()"
a->method2(); // 5) will print: "B::method2()"!
}
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 object 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...
}
virtual methods will cause “late biding”
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 to point 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 {}  // Void function
    int i() const { return 1; }  // Non-virtual function
};

class OneVirtual {
    int a;
public:
    virtual void x() const {}  // Virtual function
    int i() const { return 1; }  // Non-virtual function
};

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

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;
    return 0;
}

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
gen 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) {};
};
```
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
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”