Dynamic Binding Implementation

Object-Oriented Programming

236703

Winter 2015-6
Dynamic Binding

- Reminder: dynamic binding is required when the *dynamic type* can be different from the *static type*
  - I.e., *polymorphism* is involved
- We focus on statically-typed languages
  - Given: static type *protocol*, inclusion polymorphism
  - Required: dynamic type *behavior*
  - Can we check the receiver’s type, go to the class object, and invoke the right method?
    - Maybe. But we can do much better.
- We will also discuss dynamically-typed languages a bit
Languages usually define **semantics** and not **implementation**  
- E.g., C++ requires dynamic binding of virtual functions, but does not care how that binding is achieved  
  - No ABI (Application Binary Interface) – good luck linking GCC and VS object files  

The following 3 lectures present common, not mandatory, **implementations**  
- Enough for the final exam, possibly not for professional programming
Anatomy of C++ Polymorphic* Object

vpointer (vpTR)

Vtable (vtbl)

Object fields

Vtable (vtbl)

Virtual function implementation

Virtual f1

Virtual f2

Virtual f3

Q1: Why don’t all C++ objects look like this?

Q2: How is this anatomy defined in the C++ Standard?

Note: while we discuss C++, other languages use similar techniques

* In C++ terminology, a class is *polymorphic* if it has a virtual function
A* x = new A();
x->f1();

#1: Read vpctr at the head of the object,
    ≈
    *reinterpret_cast<vtable_t*>(x)

#2: Read entry of f1
    Index known at compile time!

#3: Call address read on step 2
    Like invoking via function pointer

A* x = new A();
x->f1();
Virtual Tables & Inheritance

Vtable of Derived begins with Base virtual functions (in the same order)

- And so does every vtable of every derived class
- Recursive relation, holds for descendants of Derived too
Virtual Functions & Inheritance

Index of f1 is always the same

• Guaranteed by inclusion polymorphism – can’t assign pointer to object of unrelated type to x
• Pointed function might differ if it was overridden, though

```cpp
A* x = getAorBorC();
x->f1();
```

A defines f1  B overrides f1  C overrides f1
Vtable Static Limitations

- The compiler knows what the static type’s vtable looks like (functions and order)
- Can we call virtual functions that are in the dynamic type’s vtable, but not in the static type’s?
  - Index unknown: different classes can define different f4s
  - Really want to call f4? Downcast!
    - What will happen to the vpointer and vtable upon downcast?

```
A* x = new B();
x->f4(); // ok??
```
To allow run-time binding, each object must have some link to some type info
  • So, each object has a vpointer

Behavior is determined by dynamic type
  • So, all objects of same type can share a vtable

Class data also includes RTTI
  • Usually stored in special vtable entries
  • So each class usually has a different vtable regardless of its content (which can be identical to that of the base class)

Class and Object Data

![Diagram of Class and Object Data]

- f1
- f2
- f3

- f1'
- f2'
- f3'
Borland Style Virtual Pointer

```cpp
struct Base {
    int x, y, z;
    virtual void f();
};
```

- Virtual pointer is always located at the beginning of the object
  - Given, of course, the class is polymorphic
- Easy access to vptr – always at the same offset (0)
  - Dynamic binding = exactly 2 pointer dereferences
If Base isn’t polymorphic and Derived is, this adjustment is required upon cast

- sizeof(vptr) must be added or subtracted
- nullptr check must sometimes be done as well (why?)
Gnu Style Virtual Pointer

```
struct Base {
    int x, y, z;
};
struct Derived : Base {
    int d;
    virtual void f();
};
```

- VPTR is located at the beginning of the first sub-object that has virtual functions
  - Must add `sizeof(Base)` to reach `vptr` – on every virtual function call!
    - Note: the offset is calculated at compile time; the addition is done at run time
- But now, casting is free
  - Well, not `dynamic_cast`, which must do type checking…
Borland vs. Gnu

- Optimization decision: what should work faster?
  - Borland – virtual functions invocation
  - Gnu – casting

- Can’t mix binaries using different styles
  - But that’s the case with every aspect of virtual functions, RTTI, multiple inheritance etc. – C++ has no standard ABI 😞
  - A compiler can use both styles as long as each class is treated consistently

- In practice, most compilers use Borland style (yes, even GCC – the Gnu Compiler Collection...)
Run-time Type Information (RTTI)

- Conceptually and practically related to virtual functions and virtual tables:
  - No RTTI if class not polymorphic
  - RTTI usually reached via virtual table
    - So each class must have a unique vtable even if neither overriding nor adding new virtual functions
- Use: dynamic_cast, typeid
- Content: implementation specific
Binding within Constructors (and Destructors)

• Given an object of class B, which inherits class A; how is it initialized?

• In C++ and Java, the constructor of A is invoked before the constructor of B
  • Why?
    • So B’s constructor never sees uninitialized attributes

• What happens if A’s constructor invokes a virtual function?
  • And that virtual function is overridden by B?
Binding within Constructors – C++

• The binding of function calls within constructors is static – must be *as if* it is static. Why?
  • B’s memory has not been initialized yet

```cpp
struct A {
    int x;
    virtual void f() { cout << "x=" << x; }
    A() : x(1) { f(); }
};

struct B : A {
public:
    int y;
    void f() override { cout << "y=" << y; }
    B() : y(2) {}
};
```

• The output of `new B()` should be ”x=1”
Statically Binding In Constructors

- If binding must be *as if* it is static, why not just use static binding?
  - \( \text{A}() \{ \text{f}(); \} \rightarrow \text{A}() \{ \text{A}::\text{f}(); \} \) will work!

- Now, say we have some global function:
  ```
  \text{void \ g(A* \ a) \{ \ a->\text{f}(); \ \}}
  ```

- What should the compiler do if A’s constructor is modified as follows?
  ```
  \text{A}() \{ \ \text{g(this);} \}
  ```

- Static binding can’t handle indirect invocations!
Bounding Dynamic Binding

• Instead of statically binding within constructors, dynamic binding can be used but limited

• The compiler generates code as follows when B’s constructor is invoked:
  1. Have vptr point on A’s vtable
  2. Execute A’s constructor
  3. Have vptr point on B’s vtable
  4. Execute B’s constructor

• Now, the B::A is really an A during construction
  • Including indirect calls and RTTI

• This is why abstract classes must have vtables!
  • Once constructed, vtable of derived class is used
Pitfall of Bounded Dynamic Binding

```cpp
struct A {
    virtual void f() = 0;
    A() { f(); }
};

struct B : A {
    void f() override { cout << "B’s f"; }
};
```

- What happens in `new B()`?
- Some compilers do not allow calling a pure virtual function directly from constructors
  - But indirect invocations can’t always be detected
- Invoking a pure virtual function is Undefined Behavior
  - In practice, will probably yield an error message and abort
Binding within Constructors – Java

• Function binding within constructors is fully dynamic
  • An initialization phase precedes the constructor invocation, setting fields to default values

```java
class A {
    private int x = 1;
    public void f() { System.out.print("x=\"+x); }
    public A() { f(); }
}

class B extends A {
    private int y = 2;
    public void f() { System.out.print("y=\"+y); }
    public B() {}
}
```

• The output of `new B()` is: ”y=0”

Why can’t C++ have a similar initialization phase?
Pitfall of Full Dynamic Binding

```java
class A {
    public A() {
        System.out.print( toString() );
    }
}
class B extends A {
    private String s = "Class B"
    public String toString() {
        return s.toLowerCase();
    }
}
```

- What happens in `new B();`?
  - `s` is initialized to `null` when A’s constructor is invoked
  - B’s `toString()` is invoked from A’s constructor
  - The result: `NullPointerException`
Dynamic Binding & Dynamic Typing

• Dynamic Typing: no constraints on the values stored in a variable
  • Usually implies reference semantics

• Run-time type information: dynamic type is associated with the value
  • There is no notion of static type to be associated with a variable

• No type safety: run-time error if an object doesn't recognize a message
Dynamic Typing & Virtual Tables

• Vtables are arrays, matching index to function pointer
  • Efficient direct access, lookup impossible
• Method index unknown at compile time
  • No static types
    • ⇒Virtual tables inappropriate
• Needed: data structure allowing run-time
  • So method names can be bound to method implementation
Dispatch Tables

• Used in dynamic type systems
• Support:
  • Runtime introduction of new types
  • Runtime changes to type hierarchy
  • “Method not found” error messages

◆ Space Efficiency: “optimal”
◆ Time Efficiency: lousy; mitigated by a cache of triples:
  ● Class where search started
  ● Selector searched
  ● Address of method found
Statically typed languages use virtual tables, while dynamically typed languages use dispatch tables (AKA method dictionaries)

Virtual tables are much faster – direct access instead of lookup

- Access is determined on compile type based on static type, hence N/A for dynamic languages
- Still, even statically typed languages must sometimes do a lookup
  - E.g., Java interfaces – more on that in 2 weeks