Dynamic Binding C++

Douglas C. Schmidt
Professor
d.schmidt@vanderbilt.edu
Vanderbilt University
www.dre.vanderbilt.edu/~schmidt/

Motivation

When designing a system it is often the case that developers:

1. Know what class interfaces they want, without precisely knowing the most suitable representation
2. Know what algorithms they want, without knowing how particular operations should be implemented

In both cases, it is often desirable to defer certain decisions as long as possible

- Goal: reduce the effort required to change the implementation once enough information is available to make an informed decision

Copyright © 1997-2006 Vanderbilt University

Motivation (cont’d)

- Therefore, it is useful to have some form of abstract “place-holder”
  - Information hiding & data abstraction provide compile-time & link-time place-holders
    * i.e., changes to representations require recompiling and/or relinking...
  - Dynamic binding provides a dynamic place-holder
    * i.e., defer certain decisions until run-time without disrupting existing code structure
    * Note, dynamic binding is orthogonal to dynamic linking...

- Dynamic binding is less powerful than pointers-to-functions, but more comprehensible & less error-prone
  * i.e., since the compiler performs type checking at compile-time

Copyright © 1997-2006 Vanderbilt University

Motivation (cont’d)

- Dynamic binding allows applications to be written by invoking general methods via a base class pointer, e.g.,

  ```cpp
  class Base { public: virtual int vf (void); }; 
  Base *bp = /* pointer to a subclass */;
  bp->vf ();
  ```

- However, at run-time this invocation actually invokes more specialized methods implemented in a derived class, e.g.,

  ```cpp
  class Derived : public Base
  { public: virtual int vf (void); }; 
  Derived d;
  bp = &d;
  bp->vf (); // invokes Derived::vf()
  ```

- In C++, this requires that both the general and specialized methods are virtual methods
Motivation (cont’d)

- Dynamic binding facilitates more flexible and extensible software architectures, e.g.,
  - Not all design decisions need to be known during the initial stages of system development
    * i.e., they may be postponed until run-time
  - Complete source code is not required to extend the system
    * i.e., only headers & object code
- This aids both flexibility & extensibility
  - Flexibility = “easily recombine existing components into new configurations”
  - Extensibility = “easily add new components”

Dynamic vs. Static Binding

- Inheritance review
  - A pointer to a derived class can always be used as a pointer to a base class that was inherited publicly
    * Caveats:
      * The inverse is not necessarily valid or safe
      * Private base classes have different semantics...
    - e.g.,
      
      ```cpp
      template <typename T>
      class Checked_Vector : public Vector<T> { ... };
      Checked_Vector<int> cv (20);
      Vector<int> *vp = &cv;
      int elem = (*vp)[0]; // calls operator[] (int)
      ```
      - A question arises here as to which version of operator[] is called?

Dynamic vs. Static Binding (cont’d)

- The answer depends on the type of binding used...
  1. Static Binding: the compiler uses the type of the pointer to perform the binding at compile time. Therefore, `Vector::operator[] (vp, 0)` will be called
  2. Dynamic Binding: the decision is made at run-time based upon the type of the actual object. `Checked_Vector::operator[]` will be called in this case as `(*vp->vptr[1]) (vp, 0)`

- Quick quiz: how must class Vector be changed to switch from static to dynamic binding?

Dynamic vs. Static Binding (cont’d)

- When to chose use different bindings
  - Static Binding
    * Use when you are sure that any subsequent derived classes will not want to override this operation dynamically (just redefine/hide)
    * Use mostly for reuse or to form “concrete data types”
  - Dynamic Binding
    * Use when the derived classes may be able to provide a different (e.g., more functional, more efficient) implementation that should be selected at run-time
    * Used to build dynamic type hierarchies & to form “abstract data types”
Dynamic vs. Static Binding (cont’d)

- **Efficiency** vs. **flexibility** are the primary tradeoffs between static & dynamic binding

- Static binding is generally more efficient since
  1. It has less time & space overhead
  2. It also enables method inlining

- Dynamic binding is more flexible since it enables developers to extend the behavior of a system transparently
  - However, dynamically bound objects are difficult to store in shared memory

Dynamic Binding in C++

- In C++, dynamic binding is signaled by explicitly adding the keyword `virtual` in a method declaration, e.g.,
  ```cpp
  struct Base {
    virtual int vf1 (void) { cout << "hello\n"; }
    int f1 (void); 
  };
  ```
- Note, virtual methods **must** be class methods, *i.e.*, they cannot be:
  * Ordinary “stand-alone” functions
  * class data
  * Static methods

- Other languages (*e.g.*, Eiffel) make dynamic binding the default...
  - This is more flexible, but may be less efficient

C++ Virtual Methods

- Virtual methods have a fixed *interface*, but derived *implementations* can change, *e.g.*,
  ```cpp
  struct Derived_1 : public Base {
    virtual int vf1 (void) { cout << "world\n"; }
  };
  ```
- Supplying `virtual` keyword is optional when overriding `vf1()` in derived classes, *e.g.*,
  ```cpp
  struct Derived_2 : public Derived_1 {
    int vf1 (void) { cout << "hello world\n"; } // Still virtual
    int f1 (void); // not virtual
  };
  ```
- You can declare a virtual method in any derived class, *e.g.*,
  ```cpp
  struct Derived_3 : public Derived_2 {
    virtual int vf2 (int); // different from vf1!
    virtual int vf1 (int); // Be careful!!!!
  };
  ```

C++ Virtual Methods (cont’d)

- Virtual method dispatching uses object’s “dynamic type” to select the appropriate method that is invoked at run-time
  - The selected method will depend on the class of the *object* being pointed at & *not* on the pointer type
  - *E.g.*,
    ```cpp
    void foo (Base *bp) { bp->vf1 (); /* virtual */ }
    Base b;
    Base *bp = &b;
    bp->vf1 (); // prints "hello"
    Derived_1 d;
    bp = &d;
    bp->vf1 (); // prints "world"
    foo (&b); // prints "hello"
    foo (&d); // prints "world"
    ```
Virtual Methods (cont’d)

- virtual methods are dynamically bound and dispatched at run-time, using an index into an array of pointers to class methods
  - Note, this requires only constant overhead, regardless of the inheritance hierarchy depth...
  - The virtual mechanism is set up by the constructor(s), which may stack several levels deep...

- e.g.,

```cpp
void foo (Base *bp) {
    bp->vf1 ();
    // Actual call
    // (*bp->vptr[1])(bp);
}
```

- Using virtual methods adds a small amount of time & space overhead to the class/object size and method invocation time.

Shape Example

- The canonical dynamic binding example:
  - Describing a hierarchy of shapes in a graphical user interface library
  - e.g., Triangle, Square, Circle, Rectangle, Ellipse, etc.

- A conventional C solution would
  1. Use a union or variant record to represent a Shape type
  2. Have a type tag in every Shape object
  3. Place special case checks in functions that operate on Shapes
     - e.g., functions that implement operations like rotation & drawing

C Shape Example Solution

```cpp
typedef struct {
    enum { CIRCLE, TRIANGLE, RECTANGLE, /* ... */
         } type_;  
    union {
        struct Circle { /* ... */ } c_;  
        struct Triangle { /* ... */ } t_;  
        struct Rectangle { /* ... */ } r_;  
          // ...
    } u_;           
} Shape;

void rotate_shape (Shape *sp, double degrees) {
    switch (sp->type_) {
        case CIRCLE: return;
        case TRIANGLE: // Don't forget to break!
        // ...
    }
}
```

Problems with C Shape Example Solution

- It is difficult to extend code designed this way:
  - e.g., changes are associated with functions and algorithms
  - Which are often “unstable” elements in a software system design & implementation
  - Therefore, modifications will occur in portions of the code that switch on the type tag

- Using a switch statement causes problems, e.g.,
  - Setting & checking type tags
    - Falling through to the next case, etc...
Problems with C Shape Example Solution (cont’d)

- Data structures are “passive”
  - *i.e.*, functions do most of processing work on different kinds of Shapes by explicitly accessing the appropriate fields in the object
  - This lack of information hiding affects maintainability
- Solution wastes space by making worst-case assumptions *wrt* structs & unions
- Must have source code to extend the system in a portable, maintainable manner

Object-Oriented Shape Example

- An object-oriented solution uses inheritance & dynamic binding to derive specific shapes (*e.g.*, `Circle`, `Square`, `Rectangle`, & `Triangle`) from a general Abstract Base class (ABC) called `Shape`
- This approach facilities a number of software quality factors:
  1. Reuse
  2. Transparent extensibility
  3. Delaying decisions until run-time
  4. Architectural simplicity

Object-Oriented Shape Example (cont’d)

```
// Abstract Base class & Derived classes for Shape.
class Shape {
public:
  Shape (double x, double y, Color &c)
    : center_ (Point (x, y)), color_ (c) {}  
  Shape (Point &p, Color &c): center_ (p), color_ (c) {}  
  virtual int rotate (double degrees) = 0;
  virtual int draw (Screen &) = 0;
  virtual ~Shape (void) = 0;
  void change_color (Color &c) { color_ = c; }
  Point where (void) const { return center_; }
  void move (Point &to) { center_ = to; }
private:
  Point center_;  
  Color color_;}
```

Note, the “OOD challenge” is to map arbitrarily complex system architectures into inheritance hierarchies
**C++ Shape Class (cont’d)**

- Note, certain methods only make sense on subclasses of class Shape
  - *e.g.*, Shape::rotate() & Shape::draw()
- The Shape class is therefore defined as an abstract base class
  - Essentially defines only the class interface
  - Derived (i.e., concrete) classes may provide multiple, different implementations

**Abstract Base Classes (ABCs)**

- ABCs support the notion of a general concept (e.g., Shape) of which only more concrete object variants (e.g., Circle & Square) are actually used
- ABCs are only used as a base class for subsequent derivations
  - Therefore, it is illegal to create objects of ABCs
  - However, it is legal to declare pointers or references to such objects...
  - ABCs force definitions in subsequent derived classes for undefined methods
- In C++, an ABC is created by defining a class with at least one “pure virtual method”

---

**Pure Virtual Methods**

- Pure virtual methods must be methods
- They are defined in the base class of the inheritance hierarchy, & are often never intended to be invoked directly
  - *i.e.*, they are simply there to tie the inheritance hierarchy together by reserving a slot in the virtual table...
- Therefore, C++ allows users to specify ‘pure virtual methods’
  - Using the pure virtual specifier = 0 indicates methods that are not meant to be defined in that class
  - Note, pure virtual methods are automatically inherited...

---

**Pure Virtual Destructors**

- The only effect of declaring a pure virtual destructor is to cause the class being defined to be an ABC
- Destructors are not inherited, therefore:
  - A pure virtual destructor in a base class will not force derived classes to be ABCs
  - Nor will any derived class be forced to declare a destructor
- Moreover, you will have to provide a definition (i.e., write the code for a method) for the pure virtual destructor in the base class
  - Otherwise you will get run-time errors!
C++ Shape Example

- In C++, special case code is associated with derived classes, e.g.,

```cpp
class Circle : public Shape {
public:
    Circle (Point &p, double rad);
    virtual void rotate (double degrees) {} // ...
private:
    double radius_;}
class Rectangle : public Shape {
public:
    Rectangle (Point &p, double l, double w);
    virtual void rotate (double degrees); // ...
private:
    double length_, width_;}
```

C++ Shape Example (cont'd)

- C++ solution (cont’d)
  - Using the special relationship between base classes & derived subclasses, any Shape * can now be “rotated” without worrying about what kind of Shape it points to
  - The syntax for doing this is:
    ```cpp
    void rotate_shape (Shape *sp, double degrees) {
        (*sp->vptr[1]) (sp, degrees);
    }
    ```
  - Note, we are still “interface compatible” with original C version!

C++ Shape Example (cont'd)

- The C++ solution associates specializations with derived classes, rather than with function rotate_shape() 
- It’s easier to add new types without breaking existing code since most changes occur in only one place, e.g.:

```cpp
class Square : public Rectangle {
    // Inherits length & width from Rectangle
public:
    Square (Point &p, double base);
    virtual void rotate (double degree) {
        if (degree % 90.0 != 0)
            // Reuse existing code
            Rectangle::rotate (degree);
    } /* .... */
};
```
### C++ Shape Example (cont’d)

- We can still rotate any `Shape` object by using the original function, *i.e.*, 
  
  ```c++
  void rotate_shape (Shape *sp, double degrees) {
    sp->rotate (degrees);
  }
  
  Square s;
  Circle c;
  Rectangle r;
  
  rotate_shape (&s, 100.0);
  rotate_shape (&r, 250.0);
  rotate_shape (&c, 17.0);
  ```

### Comparing the Two Approaches

- If support for `Square` was added in the C solution, then every place where the type tag was accessed would have to be modified
  - *i.e.*, modifications are spread out all over the place
  - Including both header files and functions

- Note, the C approach prevents extensibility if the provider of `Square` does not have access to the source code of function `rotate_shape()`!
  - *i.e.*, only the header files & object code is required to allow extensibility in C++

```c
/* C solution */
void rotate_shape (Shape *sp, double degree) {
    switch (sp->type_) {
    case CIRCLE: return;
    case SQUARE:
        if (degree % 90 == 0)
            return;
        else /* FALLTHROUGH */
        case RECTANGLE:
            // ...
            break;
    }
}
```

- Example function that rotates `size` shapes by `angle` degrees:
  ```c
  void rotate_all (Shape *vec[], int size, double angle) {
    for (int i = 0; i < size; i++)
        vec[i]->rotate (angle);
  }
  
  vec[i]->rotate (angle) is a virtual method call
  - It is resolved at run-time according to the actual type of object pointed to by `vec[i]`
  - *i.e.*,
  
  ```c
  (*vec[i]->vptr[1]) (vec[i], angle);
  ```
Comparing the Two Approaches (cont’d)

- Sample usage of function `rotate_all()` is

```c
Shape *shapes[] = {
    new Circle /* .... */,
    new Square /* .... */
};
int size = sizeof shapes / sizeof *shapes;
rotate_all (shapes, size, 98.6);
```

- Note, it is not generally possible to know the exact type of elements in variable `shapes` until run-time
  - However, at compile-time we know they are all derived subtypes of base class `Shape`
  - This is why C++ is not fully polymorphic, but is strongly typed

Here’s what the memory layout looks like

Comparing the Two Approaches (cont’d)

- Note that both the inheritance/dynamic binding & union/switch statement approaches provide mechanisms for handling the design & implementation of variants
- The appropriate choice of techniques often depends on whether the class interface is stable or not
  - Adding a new subclass is easy via inheritance, but difficult using union/switch (since code is spread out everywhere)
  - On the other hand, adding a new method to an inheritance hierarchy is difficult, but relatively easier using union/switch (since the code for the method is localized)

Calling Mechanisms

- Given a pointer to a class object (e.g., `Foo *ptr`) how is the method call `ptr->f (arg)` resolved?
- There are three basic approaches:
  1. Static Binding
  2. Virtual Method Tables
  3. Method Dispatch Tables
- C++ & Java use both static binding & virtual method tables, whereas Smalltalk & Objective C use method dispatch tables
- Note, type checking is orthogonal to binding time...
Static Binding

- Method f's address is determined at compile/link time
- Provides for strong type checking, completely checkable/resolvable at compile time
- Main advantage: the most efficient scheme
  - e.g., it permits inline method expansion
- Main disadvantage: the least flexible scheme

Virtual Method Tables

- Method f() is converted into an index into a table of pointers to functions (i.e., the “virtual method table”) that permit run-time resolution of the calling address
  - The *ptr object keeps track of its type via a hidden pointer (vptr) to its associated virtual method table (vtable)
- Virtual methods provide an exact specification of the type signature
  - The user is guaranteed that only operations specified in class declarations will be accepted by the compiler

Virtual Method Tables (cont’d)

- Main advantages
  1. More flexible than static binding
  2. There only a constant amount of overhead (compared with method dispatching)
  3. e.g., in C++, pointers to functions are stored in a separate table, not in the object!
- Main disadvantages
  - Less efficient, e.g., often not possible to inline the virtual method calls...
Method Dispatch Tables

- Method \( f \) is looked up in a table that is created & managed dynamically at run-time
  - \( i.e., \) add/delete/change methods dynamically
- Main advantage: the most flexible scheme
  - \( i.e., \) new methods can be added or deleted on-the-fly
  - \& allows users to invoke any method for any object
- Main disadvantage: generally inefficient & not always type-secure
  - May require searching multiple tables at run-time
  - Some form of caching is often used
  - Performing run-time type checking along with run-time method invocation further decreases run-time efficiency
  - Type errors may not manifest themselves until run-time

Downcasting

- Downcasting is defined as casting a pointer or reference of a base class type to a type of a pointer or reference to a derived class
  - \( i.e., \) going the opposite direction from usual “base-class/derived-class” inheritance relationships...
- Downcasting is useful for
  1. Cloning an object
     - \( e.g., \) required for “deep copies”
  2. Restoring an object from disk
     - This is hard to do transparently...
  3. ‘Taking an object out of a heterogeneous collection of objects & restoring its original type’
     - Also hard to do, unless the only access is via the interface of the base class

Contravariance

- Downcasting can lead to trouble due to contravariance, \( e.g., \):
  ```
  struct Base {
    int i_; 
    virtual int foo (void) { return i_; } 
  };
  struct Derived : public Base {
    int j_; 
    virtual int foo (void) { return j_; } 
  };
  void foo (void) {
    Base b;
    Derived d;
    Base *bp = &d; // "OK", a Derived is a Base
    Derived *dp = &b; // Error, a Base is not necessarily a Derived
  }
  ```

Contravariance (cont’d)

- Problem: what happens if \( dp->j_ \) is referenced or set?
**Contravariance (cont’d)**

- Since a Derived object always has a Base part certain operations are ok:
  
  ```
  bp = &d;
  bp->i_ = 10;
  bp->foo (); // calls Derived::foo ()
  ```

- Since base objects don’t have subclass data some operations aren’t ok
  - e.g., accesses information beyond end of b:
    ```
    dp = (Derived *) &b;
    dp->j_ = 20; // big trouble!
    ```

- C++ permits contravariance if the programmer explicitly casts, e.g.,
  ```
  dp = (Derived *) &b; // unchecked cast
  ```

- Programmers must ensure correct operations, however...

---

**Run-Time Type Identification (RTTI)**

- RTTI is a technique that allows applications to use the C++ run-time system to query the type of an object at run-time
  - Only supports very simple queries regarding the interface supported by a type

- RTTI is only fully supported for dynamically-bound classes
  - Alternative approaches would incur unacceptable run-time costs & storage layout compatibility problems

---

**Run-Time Type Identification (cont’d)**

- RTTI could be used in our earlier example
  ```
  if (Derived *dp = dynamic_cast<Derived *>(&b))
    /* use dp */;
  else
    /* error! */
  ```

- For a dynamic cast to succeed, the “actual type” of b would have to either be a Derived object or some subclass of Derived

- if the types do not match the operation fails at run-time

- if failure occurs, there are several ways to dynamically indicate this to the application:
  - To return a NULL pointer for failure
  - To throw an exception
  - e.g., in the case of reference casts...

---
Run-Time Type Identification (cont’d)

- Along with the `dynamic_cast` extension, the C++ language now contains a `typeid` operator that allows queries of a limited amount of type information at run-time
  - Includes both dynamically-bound and non-dynamically-bound types...
- e.g.,
  ```cpp
typeid(type_name) yields const Type_info &
typeid(expression) yields const Type_info &
```
- Note that the `expression` form returns the run-time type of the expression if the class is dynamically bound...

Run-Time Type Identification Examples

```cpp
Base *bp = new Derived;
Base &br = *bp;
typeid(bp) == typeid(Base *) // true
typeid(bp) == typeid(Derived *) // false
typeid(bp) == typeid(Base) // false
typeid(bp) == typeid(Derived) // false

typeid(*bp) == typeid(Derived) // true
typeid(*bp) == typeid(Base) // false

typeid(br) == typeid(Derived) // true
typeid(br) == typeid(Base) // false

typeid(&br) == typeid(Base *) // true
typeid(&br) == typeid(Derived *) // false
```

Run-Time Type Identification Problems

- RTTI encourages dreaded “if/else statements of death” e.g.,
  ```cpp
  void foo(Object *op) {
    if (Foobar *fp = dynamic_cast<Foobar *>(op))
      fp->do_foobar_things();
    else if (Foo *fp = dynamic_cast<Foo *>(op))
      fp->do_foo_things();
    else if (Bar *bp = dynamic_cast<Bar *>(op))
      bp->do_bar_things();
    else
      op->do_object_stuff();
  }
  ```
- This style programming leads to an alternative, slower method of dispatching methods
  - i.e., duplicating vtables in an unsafe manner a compiler can’t check

Summary

- Dynamic binding enables applications & developers to defer certain implementation decisions until run-time
  - i.e., which implementation is used for a particular interface
- It also facilitates a decentralized architecture that promotes flexibility & extensibility
  - e.g., it is possible to modify functionality without modifying existing code
- There may be some additional time/space overhead from using dynamic binding...
  - However, alternative solutions also incur overhead, e.g., the union/switch approach