The C++ Programming Language

Dynamic Binding

Outline

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

Motivation (cont'd)

- Therefore, it is useful to have some form of abstract "place-holder"

  – Information hiding and data abstraction provide compile-time and 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 and less error-prone

  – *i.e.*, since the compiler performs type checking at compile-time

Motivation (cont'd)

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

```
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.*,

```
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** functions

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 and object code
- This aids both *flexibility* and *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:
      1. The inverse is not necessarily valid or safe
      2. *Private* base classes have different semantics...
  - *e.g.*,

```
template <class 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[]` will be called

     Vector::**operator**[](vp, 0);
  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

     (*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 and to form "abstract data types"

Dynamic vs. Static Binding (cont'd)

- *Efficiency* vs. *flexibility* are the primary tradeoffs between static and dynamic binding
- Static binding is generally more efficient since
  1. It has less time and space overhead
  2. It also enables function 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.*,

```
struct Base {
    virtual int vf1 (void) { cout << "hello\n"; }
    int f1 (void);
};
```

- Note, virtual functions *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

Dynamic Binding in C++ (cont'd)

- *Virtual functions*:
  - These are methods with a fixed calling *interface*, where the *implementation* may change in subsequent derived classes, *e.g.*,

```
struct Derived_1 : public Base {
    virtual int vf1 (void) { cout << "world\n"; }
};
```

  - Supplying the **virtual** keyword is optional when overriding `vf1` in derived classes, *e.g.*,

```
struct Derived_2 : public Derived_1 {
    // Still a virtual...
    int vf1 (void) { cout << "hello world\n"; }
    int f1 (void); // not virtual
};
```

  - Note, you can declare a virtual function in any derived class, *e.g.*,

```
struct Derived_3 : public Derived_2 {
    virtual int vf2 (int);
    // different from vf1!
    virtual int vf1 (int); // Be careful!!!!
}
```


Dynamic Binding in C++ (cont'd)

- *Virtual functions* (cont'd):
  - The virtual function dispatch mechanism uses the "dynamic type" of an object (identified by a reference or pointer) 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 and *not* on the pointer type
  - *e.g.*,

```
void foo (Base *bp) {
    bp->vf1 (); // virtual function
}

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"
```

Dynamic Binding in C++ (cont'd)

- *Virtual functions* (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.*,

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

  - Using virtual functions adds a small amount of time and 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 or Ada 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* and *drawing*

Shape Example (cont'd)

- C or Ada solution (cont'd)
  - *e.g.*,

```
typedef struct Shape Shape;
struct Shape {
      enum {
            CIRCLE, SQUARE,
            TRIANGLE, RECTANGLE
            /* Extensions go here.... */
      } type_;
      union {
            struct Circle { /* .... */ } c_;
            struct Square { /* .... */ } s_;
            struct Triangle { /* .... */ } t_;
            struct Rectangle { /* .... */ } r_;
      } u_;
};
void rotate_shape (Shape *sp, double degrees) {
      switch (sp->type_) {
      case CIRCLE: return;
      case SQUARE: // Don't forget to break!
      // ...
      }
}
```


Shape Example (cont'd)

- Problems with the conventional approach:
  - 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 and 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 and checking type tags
      - Falling through to the next case, etc...
      - Note, Eiffel disallows **switch** statements to prevent these problems!

Shape Example (cont'd)

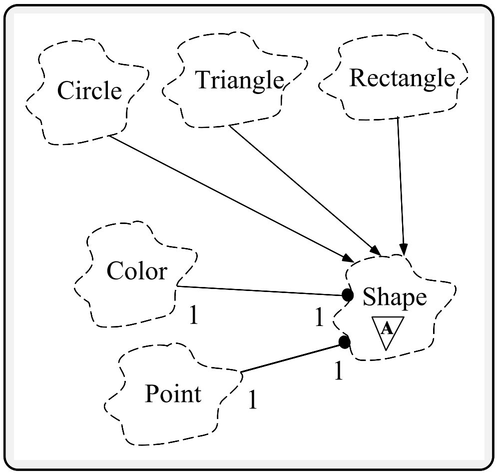
- Problems with the conventional approach (cont'd):
  - Data structures are "passive"
    - *i.e.*, functions do most of processing work on different kinds of Shapes by explicitly accessing the appropriate fields in the object
    - This lack of information hiding affects maintainability
  - Solution wastes space by making worst-case assumptions *wrt* **struct**s and **union**s
  - Must have source code to extend the system in a portable, maintainable manner

Shape Example (cont'd)

- An object-oriented solution uses inheritance and dynamic binding to derive specific shapes (*e.g.*, `Circle`, `Square`, `Rectangle`, and `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

Shape Example (cont'd)



- Note, the "OOD challenge" is to map arbitrarily complex system architectures into inheritance hierarchies

Shape Example (cont'd)

- /* Abstract Base Class and 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) { this->color_ = c; }
      Point where (void) const { return this->center_; }
      void move (Point &to) { this->center_ = to; }

private:
      Point center_;
      Color color_;
};
```

Shape Example (cont'd)

- Note, certain methods only make sense on subclasses of class Shape
  - *e.g.*, `Shape::rotate` and `Shape::draw`
- Therefore, **class** `Shape` is defined as an *Abstract Base Class*
  - Essentially defines only the class interface
  - Derived (*i.e.*, *concrete*) classes may provide multiple, different implementations

Shape Example (cont'd)

- Abstract Base Classes (ABCs)
  1. ABCs support the notion of a general concept (*e.g.*, `Shape`) of which only more concrete object variants (*e.g.*, `Circle` and `Square`) are actually used
  2. 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 function"
  - Compare with deferred classes in Eiffel...

Shape Example (cont'd)

- *Pure virtual functions*
  - Pure virtual functions must be methods
  - They are defined in the base class of the inheritance hierarchy, and 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 functions"
    * Using the pure virtual specifier = 0 indicates methods that are not meant to be *defined* in that class
    * Note, pure virtual functions are automatically inherited...

Shape Example (cont'd)

- Side note regarding *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
  - Furthermore, 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!

Shape Example (cont'd)

- The C++ solution to the Shapes example uses inheritance and dynamic binding
  - In C++, the special case code is associated with the derived class data structures
  - *e.g.*,

```
class Circle : public Shape {
public:
      Circle (Point &p, double rad);
      virtual void draw (Screen &);
      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);
      virtual void draw (Screen &);
      // ...
private:
      double length_, width_;
};
```

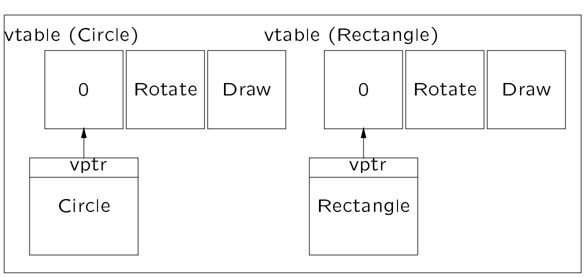

Shape Example (cont'd)

- C++ solution (cont'd)
  - Using the special relationship between base classes and derived subclasses, any `Shape *` can now be "rotated" without worrying about what kind of `Shape` it points to
  - The syntax for doing this is:

```
void rotate_shape (Shape *sp, double degrees) {
      sp->rotate (degrees);
      // (*sp->vptr[1]) (sp, degrees);
}
```

  - Note, we are still "interface compatible" with original C version!

Shape Example (cont'd)



- This code will continue to work regardless of what derived class of `Shape` that `sp` actually points to, *e.g.*,

```
Circle c;
Rectangle r;

rotate_shape (&c, 100.0);
rotate_shape (&r, 250.0);
```


Shape Example (cont'd)

- Characteristics of the C++ dynamic binding solution:
  - Associate all specializations with the derived class
    - ∗ Rather than with function `rotate_shape`
  - This makes it possible to add new types (derived from base **class** `Shape`) without breaking existing code
    - ∗ *i.e.*, most extensions/changes occur in only one place
  - *e.g.*, add a new **class** Square derived from **class** Rectangle:

```
class Square : public Rectangle {
// Inherits length and width from Rectangle
public:
      Square (Point &p, double base);
      virtual void draw (Screen &);
      virtual void rotate (double degree) {
            if (degree % 90.0 != 0)
                  // Reuse existing code
                  Rectangle::rotate (degree);
      }
      /* .... */
};
```

Shape Example (cont'd)

- C++ solution with dynamic binding (cont'd)
  - We can still rotate any `Shape` object by using the original function, *i.e.*,

```
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);
```


Shape Example (cont'd)

- Comparison between 2 approaches
  - If support for `Square` was added in the C or Ada 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 or Ada 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 and object code is required to allow extensibility in C++

Shape Example (cont'd)

- Comparison between 2 approaches (cont'd)

```
/* 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;
    }
}
```


Shape Example (cont'd)

- Example function that rotates `size` shapes by `angle` degrees:

```
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 function call
  - It is resolved at run-time according to the actual type of object pointed to by `vec[i]`
  - *i.e.*,

```
vec[i]->rotate (angle) becomes
(*vec[i]->vptr[1]) (vec[i], angle);
```

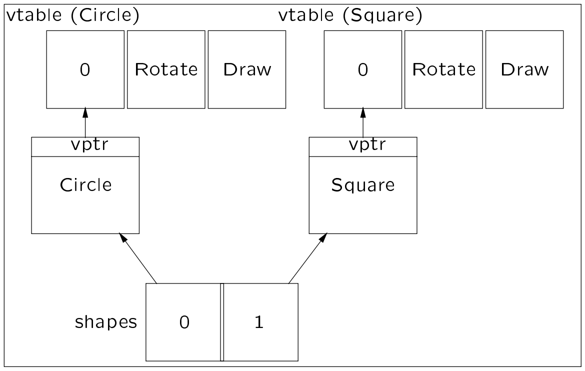
Shape Example (cont'd)

- Sample usage of function `rotate_all` is

```
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

Shape Example (cont'd)



- Here's what the memory layout looks like

Shape Example (cont'd)

- Note that both the inheritance/dynamic binding and **union**/**switch** statement approaches provide mechanisms for handling the design and 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 function to an inheritance hierarchy is difficult, but relatively easier using **union**/**switch** (since the code for the function is localized)

Calling Mechanisms

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

Calling Mechanisms (cont'd)

- *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 function expansion
  - Main disadvantage: the *least* flexible scheme

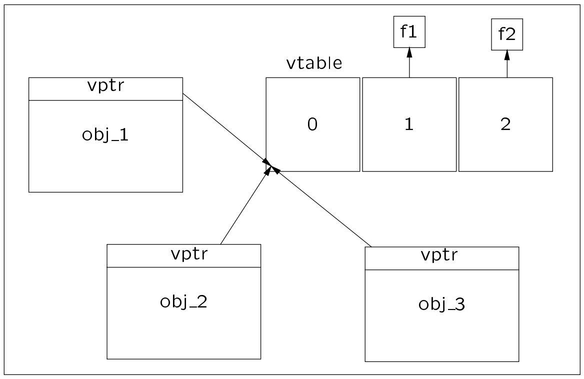
Calling Mechanisms (cont'd)

- *Virtual Function Tables*
  - Method `f` is converted into an index into a table of pointers to functions (*i.e.*, the *virtual function 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 function table (`vtable`)
  - Virtual functions 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

Calling Mechanisms (cont'd)

- *Virtual Function Tables* (cont'd)
  - Main advantages
    1. More flexible than static binding
    2. There only a constant amount of overhead (compared with method dispatching)
       - *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 function calls...

Calling Mechanisms (cont'd)



- *e.g.*,

```
class Foo {
public:
      virtual int f1 (void);
      virtual int f2 (void);
      int f3 (void);
private:
      // data ...
};
Foo obj_1, obj_2, obj_3;
```

Calling Mechanisms (cont'd)

- *Method Dispatch Tables*
  - Method `f` is looked up in a table that is created and 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*
    - and allows users to invoke *any* method for *any* object
  - Main disadvantage: generally inefficient and 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

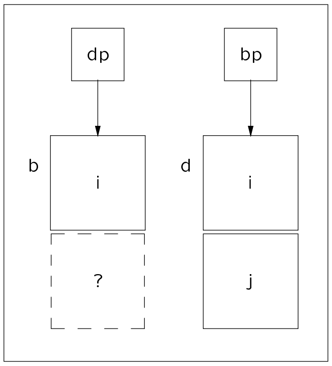
- Downcasting is defined as:
  - *Either manually or automatically casting a pointer or reference of a base class type to a type of a pointer or reference to a derived class.*
  - *i.e.*, going the opposite direction from usual "base-class/derived-class" inheritance relationships...
- Downcasting is 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 and restoring its original type*
     - Also hard to do, unless the only access is via the interface of the base class

Downcasting (cont'd)

- Contravariance
  - Downcasting can lead to trouble due to contravariance
    - It is consequence of inheritance that works against programmers in a symmetrically opposing fashion to the way inheritance works for them
  - Consider the following derivation hierarchy:

```
struct Base {
      int i_;
      virtual int foo (void) { return this->i_; }
};
struct Derived : public Base {
      int j_;
      virtual int foo (void) { return this->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
}
```


Downcasting (cont'd)



- Problem: what happens if dp->j_ is referenced or set?

Downcasting (cont'd)

- Contravariance (cont'd)
  - Since a Derived object always contains a Base part certain operations are well defined:

```
bp = &d;
bp->i_ = 10;
bp->foo (); // calls Derived::foo ();
```

  - However, since base objects do not contain the data portions of any of their derived classes, other operations are not defined
    * *e.g.*, this assignment accesses information beyond the end of object b:

```
dp = (Derived *) &b;
dp->j_ = 20; // big trouble!
```

  - Note, C++ permits contravariance if the programmer explicitly provides a *downcast*, *e.g.*,

```
dp = (Derived *) &b; // unchecked cast
```

  - It is the programmer's responsibility to make sure that operations upon dp don't access non-existent fields or methods

Downcasting (cont'd)

- Traditionally, downcasting was necessary due to the fact that C++ originally did not support overloading on function "return" type
  - *e.g.*, in C++ the following is currently not allowed in most compilers:

```
struct Base {
    virtual Base *clone (void);
};
struct Derived : public Base {
    virtual Derived *clone (void); // Error!
};
```

  - However, assuming we make the appropriate **virtual** Base *clone (**void**) change in **class** Derived...

```
Base *ob1 = new Derived;
Derived *ob2 = new Derived;
```

  - The following are syntax "errors" (though they are actually type-secure):

```
Derived *ob3 = ob1->clone (); // error
Derived *ob4 = ob2->clone (); // error
```

  - To perform the intended operation, we must use a cast to "trick" the type system, *e.g.*,

```
Derived *ob5 = (Derived *) ob1->clone ();
```


Downcasting (cont'd)

- The *right* way to handle this is to use the C++ *Run-Time Type Identification* (RTTI) feature
- However, since most C++ compilers do not support type-safe downcasting, some workarounds include:
  1. Don't do it, since it is potentially non-type-safe
  2. Use an explicit cast (*e.g.*, ob5) and cross your fingers
  3. Encode type tag and write massive switch statements
     - Which defeats the purpose of dynamic binding
  4. Manually encode the return type into the method name:

```
Derived *ob6 = ob2->cloneDerived ();
```


Run-Time Type Identification

- 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 and storage layout compatibility problems

Run-Time Type Identification (cont'd)

- RTTI could be used in our original example involving `ob1`

```
Base *ob1 = new Derived;
if (Derived *ob2 = dynamic_cast<Derived *>(ob1->clone ()))
     /* use ob2 */;
else
     /* error! */
```

- For a dynamic cast to succeed, the "actual type" of `ob1` 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)

- **dynamic_cast** used with references
  - A reference **dynamic_cast** that fails throws a bad_cast exception
- *e.g.*,

```
void clone (Base &ob1)
{
     try
     {
          Derived &ob2 =
               dynamic_cast<Derived &>(ob1);
          /* ...*/
     }
     catch (bad_cast)
     {
          /* ...*/
     }
}
```


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.*,

typeid (*type_name*) → **const** Type_info &
typeid (*expression*) → **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 (cont'd)

- Here are some short examples

```
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 (cont'd)

- A common gripe is RTTI will encourage the dreaded "switch statement of death," *e.g.*,

```
void foo (Object *op) {
      op->do_something ();
      if (Foobar *fbp = dynamic_cast<Foobar *> (op))
            fbp->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 ();
}
```

- Implementing this style of *type tagging* by hand (rather than by the compiler) leads to an alternative, slower method of dispatching methods
  - *i.e.*, duplicating the work of *vtables* in an unsafe manner that a compiler cannot double check
  - However, even an automated approach can be hard to make efficient!

Summary

- Dynamic binding enables applications and 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 and extensibility
  - *e.g.*, it is possible to modify functionality without modifying existing code
- There is some additional run-time overhead from using dynamic binding...
  - However, alternative solutions also incur overhead
    - *e.g.*, the **union**/**switch** approach