Dynamic Binding: Class-Appropriate behavior

Steven Zeil

September 19, 2013

Contents

To be considered “object-oriented”, a language must support inheritance, subtyping, and dynamic binding. We have seen the first two. Now it’s time to look at the third.

If you want to claim that a program is written in an object-oriented style, the code must be designed to take advantage of these three features. Otherwise, it’s just a traditionally styled program written in an object-oriented programming language (OOPL).

1 Dynamic Binding

Dynamic Binding

Dynamic binding is a key factor in allowing different classes to respond to the same message with different methods.

- Arguably, no language without dynamic binding is an OOPL

. . . .

1.1 What is dynamic binding?

What is binding?

Binding is a term used in programming languages to denote the association of information with a symbol. It’s a very general term with many applications.

- `a = 2;` is a binding of a value to a variable.
- `String s;` is a binding of a type (*String*) to the variable name (*s*).

. . . .

Binding Function Calls to Bodies

In OOP, we are particularly interested in the binding of a function body (method) to a function call.

Given the following:

```
a = foo(b);
```

When is the decision made as to what code will be executed for this call to `foo`?

. . . .

Compile-Time Binding

In traditionally compiled languages (FORTRAN, PASCAL, C, etc), the decision is made at compile-time.

- Decision is immutable
  - If this statement is inside a loop, the same code will be invoked for `foo` each time.
- Compile-time binding is cheap – there's very little execution-time overhead.

. . . .

Run-Time Binding

In traditionally interpreted languages (LISP, BASIC, etc), the decision is made at run-time.

- Decision is often mutable
  - If this statement is inside a loop, different code may be invoked for foo each time.
- Run-time binding can be expensive (high execution-time overhead) because it suggests that some sort of decision or lookup is done at each call.

. . . .

Dynamic Binding = the Happy Medium?

OOPLs typically feature dynamic binding, an "intermediate" choice in which

- the choice of method is made from a relatively small list of options
  - that list is determined at compile time
  - the final choice made at run-time
- the options that make up that list are organized according to the inheritance hierarchy

. . . .

Dynamic Binding and Programming Languages

- In Java, *all* function calls are resolved by dynamic binding.
- In C++, we can choose between compile-time and dynamic binding. [1]

. . . .

Actually, both the class designer and the application programmer have a say in this. Dynamic binding happens only if the class designer says it's OK for a given function, and then only if the application programmer makes calls to that function in a way that permits dynamic binding.

1.2 Dynamic Binding in C++

Dynamic Binding in C++: virtual functions

- A non-inherited function member is subject to dynamic binding if its declaration is preceded by the word `virtual`.
- An inherited function member is subject to dynamic binding if that member in the base class is subject to dynamic binding.
  - Using the word `virtual` in subclasses is optional (but recommended).

. . . .

Dynamic Binding in C++: virtual functions

- Declaring a function as virtual gives programmers permission to call it via dynamic binding.
  - But not *all* calls will be resolved that way.
- Let `foo` be a virtual function member.

[1] This is one of the reasons that C++ is often called a "hybrid" OOPL, as opposed to "pure" OOPLs like SmallTalk.

- `x.foo()`, where *x* is an object, is bound at compile time
- `x.foo()`, where *x* is a reference, is bound at run-time (dynamic).
- `x->foo()`, where *x* is a pointer, is bound at run-time (dynamic).

. . . .

2 An Example of Dynamic Binding

An Animal Inheritance Hierarchy

For this example, we will introduce a simple hierarchy.

```
class Animal {
public:
  virtual String eats() {return "???";}
  String name() {return "Animal";}
};
```

We begin with the base class, *Animal*, which has two functions.

- One of those functions has been marked `virtual`, and so is eligible for dynamic binding.

. . . .

Plant Eaters

Now we introduce a subclass of *Animal* that overrides both those functions.

```
class Herbivore: public Animal {
public:
   virtual String eats() {return "plants";}
   String name() {return "Herbivore";}
};
```

- The "`virtual`" on function `eats` is optional.

  - That function is already virtual because it was declared that way in the base class. But listing the `virtual` in the subclass is a nice reminder to the reader.

. . . .

Cud-Chewers

Now we introduce a subclass of *that* class.

```
class Ruminants: public Herbivore {
public:
   virtual String eats() {return "grass";}
   String name() {return "Ruminant";}
};
```

. . . .

Meat Eaters

And another subclass of the original base class.

```
class Carnivore: public Animal {
public:
   virtual String eats() {return "meat";}
   String name() {return "Carnivore";}
};
```

. . . .

Output Function

It will also be useful in this example to have a simple utility function to print a pair of strings.

```
void show (String s1, String s2) {
         cout << s1 << "  " << s2 << endl;
}
```

. . . .

Finally, we introduce some application code that makes calls on the member functions of our inheritance hierarchy.

Let's Make Some Calls

```
Animal a, *paa, *pah, *par;
Herbivore h, *phh;
Ruminant r;
paa = &a; phh = &h; pah = &h; par = &r;

show(a.name(), a.eats());          // AHRC ?pgm
show(paa->name(), paa->eats());  // AHRC ?pgm
show(h.name(), h.eats);            // AHRC ?pgm
show(phh->name(), phh->eats());  // AHRC ?pgm
show(pah->name(), pah->eats());  // AHRC ?pgm
show(par->name(), par->eats());  //AHRC ?pgm
```

. . . .

Note the variety of variables we are using.

- We have three actual objects, *a*, *h*, and *r*, which are of type *Animal*, *Herbivore*, and *Ruminant*, respectively.
- We also have a number of pointers, all of which have names beginning with "*p*".
  - The second letter of each pointer variable name indicates its data type.
    Thus, *pa*, *pah*, and *par* are all of type *Animal**. *ph* has type *Herbivore**.
  - These pointer variables are all assigned the address of one of our three actual objects. (The unary prefix operator & in C++ is the "address-of" operator.)
    The third letter in each pointer variable's name indicates what type of object it actually points to. Thus *paa* is of type *Animal** and actually points to an *Animal* object, *a*.
    pah, on the other hand, is of type *Animal** but actually points to an *Herbivore* object, *h*. (This is possible because of subtyping - we can substitute a subtype object into a context where the supertype object is expected.)

What's the output?

Question: What will be the output of the various `show` calls?
See if you can work this out for yourself before moving on.

. . . .

Answer:

Output from the Animal Hierarchy

```
Animal a, *paa, *pah, *par;
Herbivore h, *phh;
Ruminant r;
paa = &a; phh = &h; pah = &h; par = &r;

show(a.name(), a.eats());       // Animal ??? ❶
show(paa->name(), paa->eats()); // Animal ??? ❷
show(h.name(), h.eats);         // Herbivore plants ❸
show(phh->name(), phh->eats()); // Herbivore plants ❹
show(pah->name(), pah->eats()); // Animal plants ❺
show(par->name(), par->eats()); //Animal grass ❻
```

....

....

❶ First we call the functions via the object *a*. Now, `name()` was not declared as a virtual function. So it's not eligible for dynamic binding. This call is resolved by compile-time binding which means that the `Animal::name` body is used and we print "Animal" for the name.

`eats()`, on the other hand, *is* declared as virtual, so the class designer is allowing dynamic binding. However, we are making the call via an object, *a*, not via a pointer or reference, so this call is also resolved at compile-time. The `Animal::eats` body is used and we print "???" for the food.

❷ Next we call the functions via the pointer *paa*. Now, `name()` was not declared as a virtual function. So it's not eligible for dynamic binding. This call is resolved by compile-time binding which means that the compiler looks at the data type of the pointer (*Animal**) and uses that to choose the function body. The `Animal::name` body is used and we print "Animal" for the name.

`eats()`, on the other hand, *is* declared as virtual, so the class designer is allowing dynamic binding. We are making the call via a pointer, so this call is resolved by dynamic binding. That means that the compiled code, at run-time, follows the pointer out to the heap and uses the data type of the object it actually finds there to determine the choice of function body. In this case, *paa* actually points to an *Animal*, so the `Animal::eats` body is used and we print "???" for the food.

Same output, though the reason is different.

❸ Next we call the functions via the object *h*. Now, `name()` was not declared as a virtual function. So it's not eligible for dynamic binding. This call is resolved by compile-time binding which means that the `Herbivore::name` body is used and we print "Herbivore" for the name.

`eats()`, on the other hand, *is* declared as virtual, so the class designer is allowing dynamic binding. However, we are making the call via an object, *h*, not via a pointer or reference, so this call is also resolved at compile-time. The `Herbivore::eats` body is used and we print "plants" for the food.

❹ Next we call the functions via the pointer *phh*. Now, `name()` was not declared as a virtual function. So it's not eligible for dynamic binding. This call is resolved by compile-time binding which means that the compiler looks at the data type of the pointer (*Herbivore**) and uses that to choose the function body. The `Herbivore::name` body is used and we print "Herbivore" for the name.

`eats()`, on the other hand, *is* declared as virtual, so the class designer is allowing dynamic binding. We are making the call via a pointer, so this call is resolved by dynamic binding. That means that the compiled code, at run-time, follows the pointer out to the heap and uses the data type of the object it actually finds there to determine the choice of function body. In this case, *phh* actually points to an *Herbivore*, so the `Herbivore::eats` body is used and we print "plants" for the food.

❺ Next we call the functions via the pointer *pah*. Now, `name()` was not declared as a virtual function. So it's not eligible for dynamic binding. This call is resolved by compile-time binding which means that the compiler looks at the data type of the pointer (*Animal**) and uses that to choose the function body. The `Animal::name` body is used and we print "Animal" for the name.

`eats()`, on the other hand, *is* declared as virtual, so the class designer is allowing dynamic binding. We are making the call via a pointer, so this call is resolved by dynamic binding. That means that the compiled code, at run-time, follows the pointer out to the heap and uses the data type of the object it actually finds there to determine the choice of function body. In this case, *pah* actually points to an *Herbivore*, so the `Herbivore::eats` body is used and we print "plants" for the food.

This, for the first time, shows that dynamic binding can make a different decision than we would have arrived at via compile-time binding.

❻ Finally, we call the functions via the pointer *par*. Now, `name()` was not declared as a virtual function. So it's not eligible for dynamic binding. This call is resolved by compile-time binding which means that the compiler looks at the data type of the pointer (*Animal**) and uses that to choose the function body. The `Animal::name` body is used and we print "Animal" for the name.

`eats()`, on the other hand, *is* declared as virtual, so the class designer is allowing dynamic binding. We are making the call via a pointer, so this call is resolved by dynamic binding. That means that the compiled code, at run-time, follows the pointer out to the heap and uses the data type of the object it actually finds there to determine the choice of function body. In this case, *par* actually points to a *Ruminant*, so the `Ruminants::eats` body is used and we print "grass" for the food.

When does dynamic binding take us to a different function body that would compile-time binding? Whenever a pointer or references points to a value that is of a subtype of the pointer's declared target type.

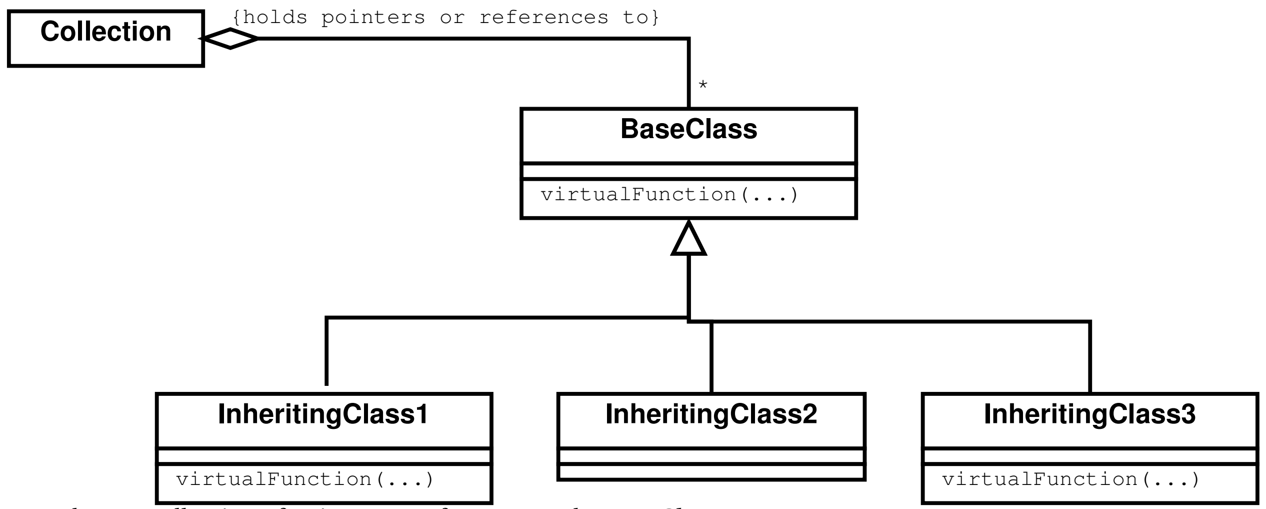
3 Why is Dynamic Binding Important?

Why is Dynamic Binding Important?
Dynamic binding lets us write application code for the superclass that can be applied to the subclasses, taking advantage of the subclasses' different methods.

. . . .

3.1 The Key Pattern to All OOP

Collections of Pointers/References to a Base Class
Suppose we have an inheritance hierarchy:



and that we have a collection of pointers or references to the *BaseClass*.

. . . .

The Key Pattern to All OOP

Then this code:

```
BaseClass* x;
for (each x in collection) {
   x->virtualFunction(...);
}
```

uses dynamic binding to apply subclass-appropriate behavior to each element of a collection.

- Each time around the loop, we extract a pointer from the collection. Thanks to subtyping, that pointer could be pointing to something of type *BaseClass* or to any of its subclasses.
- But when we call `virtualFunction` through that pointer, the runtime system uses the data type of the thing pointed to

determine which function body to invoke. If we have enough subclasses, we could wind up doing a different function body each time around the loop.

Study this pattern. Once you understand this, you have grasped the essence of OOP!

. . . .

3.2 Examples of the key pattern

Examples of the key pattern

There are lots of variations on this pattern. We can use almost any data structure for the collection.

. . . .

Example: arrays of Animals

```
Animal** animals = new Animal*[numberOfAnimals];
   ⋮
for (int i = 0; i < numberOfAnimals; ++i)
   cout << animals[i]->name() << " "
       << animals[i]->eats() << endl;
```

. . . .

Example: Linked Lists of Animals (C++)

```
struct ListNode {
   Animal* data;
   ListNode* next;
};
ListNode* head; // start of list
   ⋮
for (ListNode* current = head; current != 0; current = current->next)
   cout << current->data->name() << " "
        << current->data->eats() << endl;
```

. . . .

Example: vector of Animals

```
vector<Animal*> animals;
   ⋮
for (int i = 0; i < animals.size(); ++i)
   cout << animals[i]->name() << " "
        << animals[i]->eats() << endl;
```

. . . .

Example: Trees of Animals

```
struct TreeNode {
   Animal* data;
   TreeNode* leftChild;
   TreeNode* rightChild;
};
TreeNode* root;

void printTree (const TreeNode* t)
{
  if (t != 0) {
    printTree(t->leftChild);
    cout << t->data->name() << " "
         << t->data->eats() << endl;
    printTree(t->rightChild);
  }
}
   ⋮
printTree(root);
```

. . . .

This example is a touch more subtle. There's no loop, but the essential idea is the same. We are still iterating over a collection (in this case, using recursive calls), obtaining at each step a pointer that can point to any of several types in an inheritance hierarchy, and using that pointer to invoke a virtual function.

4 Examples

4.1 Example: Spreadsheet – Rendering Values

Example: Spreadsheet – Rendering Values

Continuing our earlier example:

- Every *Cell* holds a Value.
- Every *Value* can be rendered into a string of a given max width. (See the render function in `value.h`.
- Pairs of *Value*s can be compared for equality
- Numeric, String, and Error values are some of the possible *Value*s

. . . .

Displaying a Cell

Here is the code to draw a spreadsheet on the screen.

```
void NCursesSpreadSheetView::redraw() const
{
  drawColumnLabels();
  drawRowLabels();

  CellRange shown = showing();
  for (CellName cn = shown.first();
       shown.more(cn); cn = shown.next(cn))
    drawCell(cn);
}
```

After drawing the column and row labels, a call is made to `showing()`. That function returns a rectangular block of cell names (a *CellRange*) representing those cells that are currently visible on the screen, taking into account the window size, where we have scrolled to, etc. We have a loop that goes through the collection of cell names, invoking `drawCell` on each one.

- This is not a virtual call. But if we look *inside* `drawCell`...

. . . .

drawCell

```
void NCursesSpreadSheetView::drawCell
    (CellName name) const
{
  string cellValue;
  Cell* c = sheet.getCell(name);
  const Value* v = c->getValue();
  if (v != 0)
   {
    cellValue = v->render(theColWidth);
   }
  centerStringInWidth (cellValue,
                       theColWidth);
  // . . . show cellValue on screen . . .
}
```

. . . .

Here we can see that, from the spreadsheet, we get the cell with the given name. Then from that cell we get a pointer to a value. From that pointer we call `render`.

render()

Now `render` in `value.h` is virtual, and various bodies implementing it can be found in classes like

```
std::string NumericValue::render (unsigned maxWidth) const
  // Produce a string denoting this value such that the
  // string's length() <= maxWidth (assuming maxWidth > 0)
  // If maxWidth==0, then the output string may be arbitrarily long.
  // This function is intended to supply the text for display in the
  // cells of a spreadsheet.
{
  char buffer[256];
  for (char precision = '6'; precision > '0'; --precision)
    {
      if (maxWidth > 0)
    {
      sprintf (buffer, "%.1u", maxWidth);
    }
      else
    buffer[0] = 0;
      string format = string("%") + buffer + "." + precision + "g";
      int width = sprintf (buffer, format.c_str(), d);
      if (maxWidth == 0 || width <= maxWidth)
    {
      string result = buffer;
      result.erase(0, result.find_first_not_of(" "));
      return result;
    }
    }
  return string(maxWidth, '*');
}
```

, *NumericValue*,

```
std::string StringValue::render (unsigned maxWidth) const
  // Produce a string denoting this value such that the
  // string's length() <= maxWidth (assuming maxWidth > 0)
  // If maxWidth==0, then the output string may be arbitrarily long.
  // This function is intended to supply the text for display in the
  // cells of a spreadsheet.
{
  if (maxWidth == 0 || maxWidth > s.length())
    return s;
  else
    return s.substr(0, maxWidth);
}
```

, *StringValue*, and

```
std::string ErrorValue::render (unsigned maxWidth) const
  // Produce a string denoting this value such that the
  // string’s length() <= maxWidth (assuming maxWidth > 0)
  // If maxWidth==0, then the output string may be arbitrarily long.
  // This function is intended to supply the text for display in the
  // cells of a spreadsheet.
{
  string s = theValueKindName;
  if (maxWidth == 0 || maxWidth > s.length())
    return s;
  else
    return s.substr(0, maxWidth);
}
```

, *ErrorValue*, .

- So that `render` call will be resolved by dynamic binding, sending us to the proper function body depending on just what kind of value is actually stored in the cell.
- Combine that with the loop in the `redraw` function, and we have a loop going through a collection of pointers (the value pointers inside the cells inside the spreadsheet) and using each pointer to invoke a virtual function.

. . . .

4.2 Example: Evaluating a Cell

Example: Evaluating a Cell

- After evaluating a formula in a spreadsheet cell

```
const Value* Cell::evaluateFormula()
{
  Value* newValue = (theFormula == 0)
    ? new StringValue()
    : theFormula->evaluate(theSheet) ;

  if (theValue != 0 && *newValue == *theValue )
    delete newValue;
  else
    {
      delete theValue;
      theValue = newValue;
      notifyObservers();
    }
  return theValue;
}
```

we check to see if the value obtained is equal to *theValue* already stored in that cell.

  – If the values are equal, we simply discard the newly computed value. We don't need it.

  – But if they are not equal, we need to save the new value in place of the old one and trigger the re-evaluation of any cells that mention this one in *their* formulas.

. . . .

(The exact mechanism for how that trigger works will be explored later.)

operator==

Look at the implementation of `operator==` in `value.h`

- It calls `isEqual`, which is a virtual function in *Value*.
  - This gives us an opportunity to select subclass-specific behavior for how to compare two values for equality, which you can see in

```
bool NumericValue::isEqual (const Value& v) const
  //pre: valueKind() == v.valueKind()
  //  Returns true iff this value is equal to v, using a comparison
  //  appropriate to the kind of value.
{
  const NumericValue& vv = dynamic_cast<const NumericValue&>(v);
  return d == vv.d;
}
```

,

```
bool StringValue::isEqual (const Value& v) const
  //pre: valueKind() == v.valueKind()
  //  Returns true iff this value is equal to v, using a comparison
  //  appropriate to the kind of value.
{
  const StringValue& vv = dynamic_cast<const StringValue&>(v);
  return s == vv.s;
}
```

, and

```
bool ErrorValue::isEqual (const Value& v) const
  //pre: valueKind() == v.valueKind()
  //  Returns true iff this value is equal to v, using a comparison
  //  appropriate to the kind of value.
```

```
{
  return false;
}
```

.

- Can you see how the key pattern is manifested here?
  - What is the container and what are the contained values?

....

Appendices

A value.h

```
#ifndef VALUE_H
#define VALUE_H

#include <string>
#include <typeinfo>

//
// Represents a value that might be obtained for some spreadsheet cell
// when its formula was evaluated.
//
// Values may come in many forms. At the very least, we can expect that
// our spreadsheet will support numeric and string values, and will
// probably need an "error" or "invalid" value type as well. Later we may
// want to add addiitonal value kinds, such as currency or dates.
//
class Value
{
public:
  virtual ~Value() {}


  virtual std::string render (unsigned maxWidth) const = 0;
  // Produce a string denoting this value such that the
  // string's length() <= maxWidth (assuming maxWidth > 0)
  // If maxWidth==0, then the output string may be arbitrarily long.
  // This function is intended to supply the text for display in the
  // cells of a spreadsheet.
  virtual Value* clone() const = 0;
  // make a copy of this value

protected:
  virtual bool isEqual (const Value& v) const = 0;
  //pre: typeid(*this) == typeid(v)
  //  Returns true iff this value is equal to v, using a comparison
  //  appropriate to the kind of value.

  friend bool operator== (const Value&, const Value&);
};

inline
bool operator== (const Value& left, const Value& right)
{
  return (typeid(left) == typeid(right))
    && left.isEqual(right);
}

#endif
```