1 Static vs. Dynamic Binding

In the last lecture, we learned about inheritance. One of the things that we saw was that when one class B inherits from another A, it can overwrite/replace some of the functions that already existed in A. For instance, our DeluxeBag inherited from the more basic IntLinkedListBag, and we changed the print function from one to the other.

One of the nice things that inheritance does for us is that when B inherits from A, it must have all the functionality of an object of type A. Therefore, we can assign objects of type B into variables of type A. In other words, for our example of DeluxeBag inheriting from IntLinkedListBag, the following would be perfectly valid code:

```cpp
IntLinkedListBag *p;
DeluxeBag *q = new DeluxeBag();
p = q;
p->print();
```

Because by our inheritance, every DeluxeBag is-a IntLinkedListBag, we can assign objects of type DeluxeBag into variables of type IntLinkedListBag (but not the other way). And because objects of type IntLinkedListBag have a print function, we can call it.

But now the question arises: since DeluxeBag and IntLinkedListBag both implement different versions of a print function, which one is called in the last line? We could try out two arguments:

1. The compiler knows that p points to an object of type IntLinkedListBag. It cannot really figure out by code analysis that it so happens that this time (not even always), p points to an object that’s actually of type DeluxeBag. So the call must be to the version of print defined in IntLinkedListBag.

2. While the compiler does not know at compile time what exactly p points to, when it comes time to actually call a function (during execution), it must know the precise object that’s pointed to by p, and it so happens to be of type DeluxeBag. Therefore, the call will be to the version of print defined in DeluxeBag.

It turns out that both are by themselves valid arguments. The first version is the default. If we do not put any additional instructions in the code, C++ will call the version of the function corresponding to the declared type of the variable we are using. This is called static binding.

But sometimes, we would really like the second option to happen — we will see in a moment some examples. We can tell the compiler to do this, by declaring the print function to be virtual. We do this by just adding the word virtual in the definition of the function (both in the parent and child class), as follows:
class IntLinkedListBag {
    ...
    virtual void print();
    ...
}

The keyword virtual tells the compiler to not try to figure out at compile time which version of the function to call, but leave that decision until runtime, when it knows precisely the type of the object. This process — figuring out which version of the function to call at runtime — is called dynamic binding, as opposed to static binding, which is when you figure out at compile time which version to call.

2 Pure virtual functions and abstract classes

Sometimes, we want to declare a function in a class, but not implement it at all; we will see in a moment why. Such a function is called a pure virtual function. The declaration lives entirely in the base classes' header files, and we do not need to care about it at all from the implementation files. The syntax is

```cpp
class A {
    public:
        virtual void print() = 0;
    }
```

The = 0 part tells the compiler that the class does not implement this function. Of course, if a class has one or more pure virtual functions in it, the class cannot be instantiated. Such a class is called an abstract class. No object can be created from an abstract class. After all, if we could create objects, then what would code like the following do?

```cpp
A *obj = new A;
obj->print();
```

What code should be executed when we get to obj->print()? There is no function to call.

3 Why and how to use virtual functions

The main reason to create pure virtual functions is to force inheriting classes to implement a function. For instance, in our example above, if we later implement classes B and C which inherit from A, we have forced them to provide a print() function. So we can safely call the print() function on them.

This kind of stuff is very useful in structuring one's classes, and in avoiding code (and error) duplication. For a standard example, imagine that you are developing a graphics program that lets you draw basic shapes. Since all graphics objects (rectangles, circles, polygons, etc.) share some properties (such as perhaps color, line width, line style, and a few others) and maybe also some functionality, you may want to define a general class GraphicsObject which captures those. You also want each of these objects to provide some functions, such as perhaps draw (drawing itself), resize, or others. But absent a specific definition of an object type (e.g., circle), you don’t know how to implement the functions yet. Yet, you want to force all GraphicsObject objects you later define to provide these functions, so that you can put them all in one big array and process them the same way. The way to do that is to declare pure virtual functions draw(), resize(), and whatever other functions you want. Now, all classes you define later (like Circle, Rectangle, Polygon, etc.) that inherit from GraphicsObject must implement this function in order to have objects generated from them.

This type of thing is also very useful to specify an abstract data type, and separate its functionality from its implementation. After all, we have said before that an abstract data type is characterized by the functions it supports, and there may be many ways to implement them. For instance, a while back, we specified the ADT Bag, which translates as follows. (We are specify a bag of integers here.)
class Bag {
    public :
        virtual void add (int n) = 0;
        virtual void remove (int n) = 0;
        virtual bool contains (int n) = 0;
    }

This only says that any class that wants to implement a Bag must provide these functions. Another programmer can then create, say, a LinkedListBag, a FancyArrayBag, a VectorBag, etc. All these classes would inherit from Bag and implement the functions, though they could do so in very different ways. By inheriting from the abstract class Bag, these other classes have made sure to be very easily interchanged, as follows:

```cpp
Bag * p;
p = new LinkedListBag();
... // some time later
p = new FancyArrayBag();
```

Using dynamic binding here (indicated by the virtual keyword), the code has no problem calling the right version of each function, and we can use different implementations interchangeably.

Someone asked in class what would happen if we try to define something like a non-virtual unimplemented function:

```cpp
void add (int n) = 0;
```

This results in an error. There is no point in having a non-virtual non-defined function. If it’s not defined, it cannot be called. And if it is not virtual, then when we overwrite it later, the compiler cannot figure out to call the overwriting version, unless the object is already of type B, in which case there is no need for inheriting.

Another classic reason for using virtual functions is that we can often avoid rewriting very similar code. For instance, imagine that we are implementing an abstract data type, and several of the functions all require access to one primitive routine, such as looking up an element, or expanding an array. (We will see an example of this in Homework 4.) But all that the other functions do can be described in terms of those key routines, and they look the same regardless of the internal implementation of the key routines. So the code would look something like the following:

```cpp
class IncompleteBag {
    public:
        void add (int n);
        void remove (int n);
        bool contains (int n);

    protected:
        virtual int first() = 0;
        virtual int next() = 0;
    }
```

Imagine that the functions first() and next() somehow help us go through the entire bag, and are crucial to implement add, remove, and contains. But let’s also imagine that those functions don’t need to know anything else about the implementation (such as whether it’s based on an array or linked list) than that it implements first and next. Then, we could implement the above three functions once and for all, and reuse the code in this way. Maybe, the contains function looks as follows:
bool IncompleteBag::contains (int n)
{
    int STOP_CONDITION = -1; // arbitrary stop condition
    for(int *p = first(); p != STOP_CONDITION; p = next()) {
        if(p == n) return true;
    }
    return false;
}

Then, our LinkedListBag and FancyArrayBag and VectorBag wouldn’t have to implement each of the functions add, remove, and contains from scratch, but just to provide the missing pieces first and next. Because these functions are virtual, whenever in the course of executing contains, the program runs into first or next, it determines at runtime which version to call.

This concept — determining which version of a class member function to call at runtime — is called polymorphism, which literally means “many forms”: the object stored in a variable could be one of many forms, and the execution will do “the right thing” for the current object.

4 The ADT List

In thinking about data structures, we should always think about which operations exactly we want supported, rather than necessarily how to support them. For instance, a Dictionary (which we haven’t introduced formally as a data type yet, but gave as an example in Week 1) allows us to add pairs of a word and definitions, remove such pairs, and look up the definition of a word. Similarly, a Bag lets us add and remove elements, and look up whether the bag already contains the element.

While from our human experience, we like our dictionaries to be sorted (so we can look up things fast), this is really not part of the functionality, but part of the solution. Perhaps inside a computer, we could implement a fast dictionary entirely without sorting. So for a dictionary, sorting is not essential, nor is accessing elements by index: we don’t need to be able to look up word number 1729. (Of course, we might want that functionality in order to perform Binary Search, but again, that’s part of the solution, not part of the specification of what is needed.)

On the other hand, there are times when we really do care about having data in a particular order. For instance, when implementing a playlist in an MP3 player, or an ordered to-do list, it may really matter to us whether the first movement of a symphony comes before or after the second movement, or whether cleaning our apartment comes before or after buying groceries.

Motivated by these applications, the functionality that we would typically want would be to insert or remove elements (at any position), and to read and overwrite elements at any specific position. We may want a few other functions (such as testing for emptiness, getting the number of elements in the list, etc.), but those are not that interesting in terms of implementation, and could be easily added. So our first cut of specification would look as follows:

```cpp
template <class T>
class List
{
    void insert (int pos, const T & data);
    // inserts the data immediately before position pos.
    void remove (int pos);
    // removes the data at position pos.
    T get (int pos)
    // returns the data stored at position pos.
};
```

1In fact, we can: this will occupy much of the second half of the semester.
void set (int pos, const T & data)
       // sets the entry at position pos to data.
}

This version is a bit under-specified. In particular, we haven’t said what should happen when pos lies outside the current range of indices in the list. Should the list expand to include this index (for instance, for set or insert)? Or should there be an error? The intent is that the only way the list can grow is by inserting elements, and then, it only grows by one. So if a position pos is out of bounds, some kind of error should happen.

Then, there are more obvious problem cases, such as calling get on a position into which nothing has been written. In that case, almost certainly, some kind of error should happen. We’ll see more about lists, including how to implement them, next class.