1 Virtual Functions

This worksheet will try to explain why virtual functions are interesting. Before we talk about virtual functions, let's review what we know about inheritance.

Inheritance is useful because our inherited subclasses can borrow the functionality of the parent. This is useful because we don't need to work as hard. Let's take a trivial example:

```
#include <iostream>
class Student {
 public:
  void sleep() { cout << "Snooze\n"; }
  void eat() { cout << "Yum\n"; }
  void study() { cout << "I'm studying\n"; }
};

class BerkeleyStudent : public Student {
 public:
  void sleep() { cout << "What's sleep?\n"; }
  void eat() { cout << "mmmm... coffee...\n";}
};
```

We don't need to redefine studying for `BerkeleyStudent`, since we borrow the definition of `Student`.

However, the relation can go the other way: we can write code that works with a parent class, and have it work properly with subclasses. More truthfully, we can write code that works with a base (parent) class pointer, and allow these pointers to point to instances of derived classes.

For example, it would be very nice to be able to write the function:

```
#include "student.h"
#include <iostream>
void liveLife(Student *s) {
  s->eat();
  s->sleep();
  s->study();
}

int main() {
  Student* s1 = new Student();
  Student* s2 = new BerkeleyStudent();
  liveLife(s1);
  cout << "\n";
  liveLife(s2);
}
```

and have that work on all students. We can later derive different type of students, and expect `liveLife()` to work with all of them.

As it stands, `liveLife()` doesn't quite work yet. Here's the output of running `main()`:

```
Yum
Snooze
I'm studying

Yum
Snooze
I'm studying
```

What's going on? Why aren't `BerkeleyStudent`'s methods being called? There is a reason for this problem: `liveLife()` only knows that it's working with a `Student` instance — that is, it has no idea that we pass it a `BerkeleyStudent`.

Can we have things both ways? Can we write functions that work with `Student*` base pointers, yet have the methods do subclass-specific stuff? Yes, and that's where virtual methods come in.

What we'll need to do is make the system aware that certain function calls need to check for the run-time type of that base pointer and call the appropriate method. This is called a virtual function call. Let's see what it looks like:

```
#include <iostream>
class Student {
 public:
  virtual void sleep() { cout << "Snooze\n"; }
  virtual void eat() { cout << "Yum\n"; }
  virtual void study() { cout << "I'm studying\n"; }
};

class BerkeleyStudent : public Student {
 public:
  virtual void sleep() { cout << "What's sleep?\n"; }
  virtual void eat() { cout << "mmmm... coffee...\n";}
};
```

So all we need to do is prepend the declaration with the word `virtual`. Once we do this, everything works ok:

```
Yum
Snooze
I'm studying

mmmm... coffee...
What's sleep?
I'm studying
```

If you have experience with Java, you'll notice that all Java methods are virtual. The reason why C++'s methods aren't virtual by default is because it's a little more efficient not to do the runtime virtual call. However, today's computers are fast enough to handle the minimal cost of doing a virtual function call.

Let's try an extended example of using virtual functions.

2 A Mapping Function

In our code, we often find ourselves writing something like:

```
for(int i = 0; i < myvector.size(); i++)
{
    results[i] = some_function(i);
}
```

If you've taken Scheme, you'll notice that this is like using a functional `map`:

```
(map function list)
```

As a concrete example, let's try to write our own "mapping" function in C++: we'd like this mapper to take in a vector of things and a "function". To simplify the example, we'll write functions that map along integer vectors — to make this more general, we would use templates.

In C++, it's a bit ugly to pass functions around — it involves working with function pointers. The C++ way of passing "functions" is to pass around an instance of a virtual class. Let's define this now:

```
class IntFunction {
public:
  virtual int operator()(int x) = 0;
};
```

We've declared that any `IntFunction` will define its own implementation of the `operator()` method. We choose `operator()` for convenience. Now we can define a few subclasses to do some things:

```
#include "intfunction.h"
class SquareFunc: public IntFunction {
 public:
  virtual int operator()(int x) {
    return x * x;
  }
};

class DoubleFunc: public IntFunction {
 public:
  virtual int operator()(int x) {
    return x * 2;
  }
};

class ScalarMulFunc: public IntFunction {
 public:
  ScalarMulFunc(int c) : c(c) {}
  virtual int operator()(int x) { return x * c; }
 private:
  int c;
};
```

Now that we have these subclasses, we can finally define `mymap()`.

```
// map.cc
#include "intfunction2.h"
#include <vector>

// Apply an IntFunction on every element of L and return the results
// in a vector.
vector<int> mymap(IntFunction* f, vector<int> L) {
  vector<int> results;
  for(unsigned i = 0; i < L.size(); i++) {
    results.push_back((*f)(L[i]));
  }
  return results;
}

// A small utility to make it transparent to print out vectors with
// opening/closing braces.
template <class T>
ostream& operator<<(ostream &out, const vector<T> &L) {
  out << "[";
  for(unsigned i = 0; i < L.size() - 1; i++)
    out << L[i] << ", ";
  if(L.size() > 0)
    cout << L[L.size() - 1] << "]";
  return out;
}

int main() {
  vector<int> L;
  for(int i = 0; i < 10; i++) {
    L.push_back(i);
  }
  SquareFunc sq;
  DoubleFunc db;
  ScalarMulFunc neg(-1);

  cout << L << "\n";
  cout << mymap(&sq, L) << "\n";
  cout << mymap(&db, L) << "\n";
  cout << mymap(&neg, L) << "\n";
}
```

Finally, let's see how it works:

```
[dyoo@einfall virtual]$ ./a.out
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

I hope this helps you see how virtual function calls are useful — they allow us, through base class extension, to write generic functions that can handle different situations easily. If you have any questions, you can post to the `ucb.class.cs9f` (`cs9f-tutor@hkn.eecs.berkeley.edu`) newsgroup or email me at `dyoo@hkn.eecs.berkeley.edu`.