Polymorphism

Definition
Base Class: Number
Derived Class: Counter
Derived Class: Cycler
Testing Counter
Polymorphism Revisited
C++ Support for Polymorphism
Virtual Functions and Binding
Early Binding
Invocation via a Pointer w/o Virtuality
Invocation via a Pointer with Virtuality
Late Binding

Polymorphism I

Polymorphism
Definition

Base Class: Number

Derived Class: Counter

Derived Class: Cycler

Testing Counter

Polymorphism Revisited

C++ Support for Polymorphism

Virtual Functions and Binding

Early Binding

Invocation via a Pointer w/o Virtuality

Invocation via a Pointer with Virtuality

Late Binding

Base Class: Number

First we have a base class that encapsulates a single integer value:

```cpp
class Number {
protected:
    int Count;
public:
    Number(int InitCount = 0);
    void Reset();
    int Value() const;
};
Number::Number(int InitCount) {
    Count = InitCount;
}
void Number::Reset() {
    Count = 0;
}
int Number::Value() const {
    return Count;
}
```

Derived Class: Counter

Then a derived class that extends Number to provide a simple capped counter object:

```cpp
class Counter : public Number {
protected:
    int Start; // starting value for counter
    int Limit; // upper limit for counter
public:
    Counter(int L = 0, int C = 0);
    void Reset();
    bool Next();
};
Counter::Counter(int L, int C) {
    Start = Count = C;
    Limit = (L > = C ? L : C);
}
void Counter::Reset() {
    Count = Start;
}
bool Counter::Next() {
    Count = (Count < Limit) ? Count + 1 : 0;
    return true;
}
```

Derived Class: Cycler

And a derived class that extends Counter to provide a circular counter:

```cpp
class Cycler : public Counter {
public:
    Cycler(int L = 10, int C = 0);
    void Reset();
    bool Next();
};
Cycler::Cycler(int L, int C) {
    Limit = (L > 1 ? L : 10);
    Start = Count = C % Limit;
}
void Cycler::Reset() {
    Count = Start;
}
bool Cycler::Next() {
    Count = (Count + 1) % Limit;
    return true;
}
```

Testing Counter

Consider the following function for testing the operation of a Counter object:

```cpp
void Run(Counter C, ostream& Out) {
    do {
        Out << "C:" << setw(5) << C.Value() << endl;
    } while (C.Next());
}
```

If we declare a Counter object and call the function, the results are predictable:

```cpp
Counter C(10, 0);
Run(C, cout);
```

C: 0
C: 1
C: 2
C: 3
C: 4
C: 5
C: 6
C: 7
C: 8
C: 9
C: 10
We may well want to use a single linked list (or other container object) to organize all the HourlyEmployee and OTHourlyEmployee objects, rather than keep them in different data structures.

We can easily accomplish this by having the list nodes store pointers to objects, rather than the objects themselves.

It would also be natural to write a function to print paychecks from the data in the list, perhaps something like:

```cpp
void PrintChecks(LinkList E) {
    HourlyEmployee thisEmp;
    while ( E.moreList() ) {
        E.gotoHead();
        thisEmp = E.getCurrentData();
        PrintACheckFor(thisEmp);
        E.Advance();
    }
    Cycler D(10, 0);
    D.Run(D, cout);
}
```

But how can we make this happen automatically?

### Polymorphism Demanded

The Cycler object is behaving like a Counter object!

Is this a result of slicing??

Is this objectionable? After all, we could write a second test function that expects a Cycler object (and that would produce Cycler-like behavior).

### Virtual Functions and Binding

A member function is declared to be **virtual** by using the keyword `virtual`.

Normally functions are declared virtual in a base class and then overridden in each derived class for which the function should have a specialized implementation.

This modifies the rules whereby a function call is **bound** to a specific function implementation.

In normal circumstances (I.e., what we’ve done before) the compiler determines how to bind each function call to a specific implementation by searching within the current scope for a function whose signature matches the call, and then expanding that search to enclosing scopes if necessary.

With an inheritance hierarchy, that expansion involves moving back up through the inheritance tree until a matching function implementation is found.

The results may be surprising:

```
C: 0
C: 1
C: 2
C: 3
C: 4
C: 5
C: 6
C: 7
C: 8
C: 9
C: 10
```

This call binds to the local implementation of `Next()` given in the class `Counter`.

This call binds to the implementation of `Value()` inherited from the class `Number`.

### Early Binding

When the binding of call to implementation takes place at compile-time we say we have early binding (aka static binding).

This call binds to the implementation of `Value()` given in the class `Counter`.

Early binding is *always* used if the invocation is direct (via the name of an object using the `.` operator), whether virtual functions are used or not.
Invocation via a Pointer w/o Virtuality

When a function call is made using a pointer, and no virtual functions are involved, the binding of the call to an implementation is based upon the type of the pointer (not the actual type of its target).

This call binds to the local implementation of `Reset()` given in the class `Cycler` and sets the counter value to 5.

This call binds to the implementation of `Reset()` inherited from the class `Number` and sets the counter value to 0.

Note: this assumes the original declaration and implementation of the class `Number` from slide 3.

Invocation via a Pointer with Virtuality

Now, if we access objects in this inheritance hierarchy via pointers, we get polymorphic behavior. That is, the results are consistent with the type of the target, rather than the type of the pointer:

Now both calls to `Reset()` bind to the local implementation of `Reset()` given in the class `Cycler`, even though the access is through a base type pointer.

If you don’t think that’s cool...

Virtual Function Tables

When the binding of call to implementation takes place at runtime, the address of the called function must be managed dynamically.

The presence of a virtual function in a class causes the generation of a virtual function table (vtable) for the class, and an association to that table in each object of that type:

This increases the size of each object, but only by the size of one pointer.

Key fact: if a function is virtual in a base class, it’s also virtual in the derived class, whether it’s declared virtual there or not.

Derived Class View

So for a `Cycler` object we’d have:

In this simple case, the derived object has its own implementations to replace each of the virtual functions inherited from the base class.

That’s often NOT the case. Then, one or more of the derived class vtable pointers will target the base class implementation…
Consider another derived class:

```cpp
class UpCounter : public Number {
public:
    UpCounter(int C = 0); // Constructor
    void Next() { Count++ ; } // Increment
};
```

So for an `UpCounter` object we'd have:

- `pN->Next();` stores the address of a `Cycler` object.
- The compiler generates code to follow the vtbl pointer in the target of `pN` (at runtime) to retrieve the address of the appropriate function.

Detail: the vtbl pointer must be at a fixed offset within the target object.