

## Intro to Polymorphism

Polymorphism 1

### Definition

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

### Derived Class View

- Another Derived Class View
- So, at Runtime:
- Abstract Classes
- Pure Virtual Function Example
- Using the Abstract Class

## Definition

Polymorphism 2

polymorphism: the ability to manipulate objects of distinct classes using only knowledge of their common properties without regard for their exact class (Kafura)

Note that polymorphism involves both algorithms and types of data (classes).

## Base Class: Number

Polymorphism 3

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

```
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

Polymorphism 4

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

```
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() {
    if (Count < Limit) {
        Count++;
        return true;
    }
    return false;
}
```

## Derived Class: Cyclor

Polymorphism 5

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

```
class Cyclor : public Counter {
public:
    Cyclor(int L = 10, int C = 0);
    void Reset();
    bool Next();
};

Cyclor::Cyclor(int L, int C) {
    Limit = (L > 1 ? L : 10);
    Start = Count = C % Limit;
}

void Cyclor::Reset() {
    Count = Start;
}

bool Cyclor::Next() {
    Count = (Count + 1) % Limit;
    return true;
}
```

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

## Testing Counter

Polymorphism 6

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

```
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:

```
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
```

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

## Testing Cyclor

Polymorphism 7

If we pass a Cyclor object to the same function...

```
Cyclor D(10, 0);
Run(D, cout);
```

... 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
```

The Cyclor 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 Cyclor object (and that would produce Cyclor-like behavior).

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

## Polymorphism Revisited

Polymorphism 8

Actually the results are not really acceptable. There are times when we definitely want to be able to write a single function, and pass it objects of related but different types, and have the resulting behavior reflect the type of the actual parameter.

For example, suppose that we derived a new type, say OTHourlyEmployee, from HourlyEmployee (from the last chapter of notes), where an employee of the new type will receive an increased pay rate for any hours beyond some floor. This might be reflected by having each type now incorporate a function to compute and return the correct pay amount...

```
double HourlyEmployee::Pay() { return (Hours * Rate); }
```

```
double OTHourlyEmployee::Pay() {
    if (Hours <= 40)
        return (Hours * Rate);
    return (40 * Rate + (Hours - 40)*1.5*Rate);
}
```

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

## Polymorphism Demanded

Polymorphism 9

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 that 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:

```
void PrintChecks(LinkList E) {
    HourlyEmployee thisEmp;
    E.gotoHead();
    while ( E.moreList() ) {
        thisEmp = E.getCurrentData();
        PrintACheckFor(thisEmp);
        E.Advance();
    }
}
```

`PrintACheck()` will call the member function `Pay()` and that had better be the right one, whether `thisEmp` is an `HourlyEmployee` or an `OTHourlyEmployee`!

But how can we make this happen automatically?

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

## C++ Support for Polymorphism

Polymorphism 10

In C++, polymorphic behavior can be attained by combining:

- an inheritance hierarchy
- object accesses via pointers
- use of virtual member functions in base classes

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

Now, passing a `Cycler` object will produce an infinite loop.

```
void Run(Number* C, ostream& Out) {
    do {
        Out << "C:" << setw(5) << C->Value() << endl;
    } while ( C->Next() );
}
```

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

## Virtual Functions and Binding

Polymorphism 11

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.

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

## Early Binding

Polymorphism 12

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

```
Counter C(100, 0);
C.Next();
cout << C.Value() << endl;
```

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 is always used if the invocation is direct (via the name of an object using the dot operator), whether virtual functions are used or not.

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

## Invocation via a Pointer w/o Virtuality

Polymorphism 13

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).

```
Cyclor CC(10, 5);
CC.Next();

Cyclor* pC = &CC;
CC.Next();
pC->Reset();

Number* pN = &CC;
CC.Next();
pN->Reset();
```

This call binds to the local implementation of `Reset()` given in the class `Cyclor` 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.

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

## Enabling Polymorphism with Virtual Functions

Polymorphism 14

However, when a function call is made using a pointer, and virtual functions are involved, the binding of the call to an implementation is based upon the type of the target object (not the declared type of the pointer).

Modify the declaration of `Number` to make `Reset()` and `Next()` virtual functions:

```
class Number {
protected:
    int Count;
public:
    Number(int InitCount = 0);
    virtual void Reset();
    int Value() const;
    virtual bool Next(); // does nothing
};
```

Note this doesn't change the implementation of the function `Reset()`.

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

## Invocation via a Pointer with Virtuality

Polymorphism 15

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:

```
Cyclor CC(10, 5);
CC.Next();

Cyclor* pC = &CC;
CC.Next();
pC->Reset();

Number* pN = &CC;
CC.Next();
pN->Reset();
```

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

If you don't think that's cool...

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

## Late Binding

Polymorphism 16

When the binding of call to implementation takes place at runtime we say we have late binding (aka dynamic binding).

```
Counter C(10, 0);
Cyclor CC(10, 5);

Number* pN;

char ch;
cout << "Enter choice: ";
cin >> ch;
if (ch == 'y')
    pN = &C;
else
    pN = &CC;

pN->Next();
pN->Reset();
```

There's no way to know the type of the target of `pN` until runtime.

However, the calls to `Next()` and `Reset()` will be bound to the correct implementations regardless.

But HOW is this done? See Stroustrup 2.5.5 and 12.2.6.

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

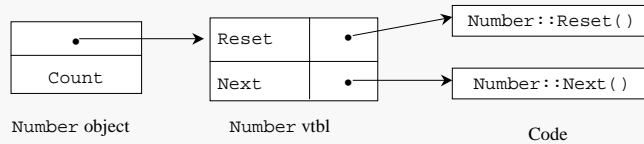
©2000 McQuain WD

## Virtual Function Tables

Polymorphism 17

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.

Computer Science Dept Va Tech January 2000

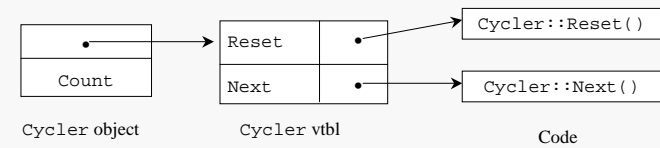
OO Software Design and Construction

©2000 McQuain WD

## Derived Class View

Polymorphism 18

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 vtbl pointers will target the base class implementation....

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

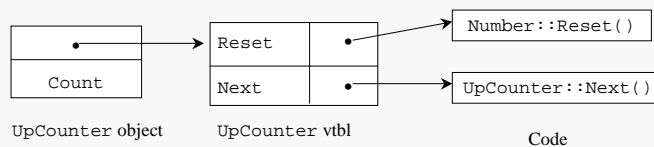
## Another Derived Class View

Polymorphism 19

Consider another derived class:

```
class UpCounter : public Number {
public:
    Counter(int C = 0);
    void Next() {Count++;}
};
```

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



Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

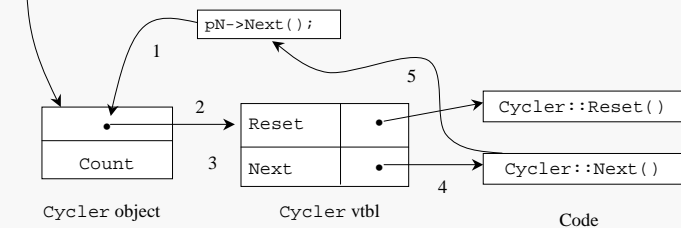
## So, at Runtime:

Polymorphism 20

Resuming the example from slide 16, say the user enters 'y'.

*pN* 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.

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

## Abstract Classes

Polymorphism 21

The root class `Number` does not seem to have any compelling application in its own right. The class serves a useful purpose within the logical specification of an inheritance hierarchy, but instances of it (arguably) do not.

An abstract class is simply a class that exists for high-level organizational purposes, but that cannot ever be instantiated.

In C++, a class is abstract if one or more of its member functions are pure virtual.

A pure virtual member function has no implementation, only a declaration (prototype) in the abstract class declaration.

Pure virtual member functions remain pure virtual in derived classes that do not provide an implementation that overrides the base class prototype.

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

## Pure Virtual Function Example

Polymorphism 22

The class `Number` can be revised into an abstract class by specifying one or more pure virtual member functions.

Question: which member functions should be pure virtual?

One answer: those that do not have sensible implementations at the root level of the hierarchy. For example, `Next()` is different for most, if not all, of the derived classes developed above, so there's not much point in providing a base implementation.

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

A virtual function is "made pure" by the initializer "= 0".

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD

## Using the Abstract Class

Polymorphism 23

Assuming the revised declaration just given, the class `Number` can be derived from, but an attempt to declare an object of type `Number` will generate a compile-time error.

Since the classes `Cycler` and `Counter` presented earlier provide overridings for the pure virtual members of `Number`, their declarations do not need to be changed.

The class `UpCounter`, however, will itself be an abstract class unless it is revised. Why?

Computer Science Dept Va Tech January 2000

OO Software Design and Construction

©2000 McQuain WD