School of Electrical Engineering and Computing

SENG2200/6220 Programming Languages & Paradigms

Topic 4
C++ vs Java – Part 3
Polymorphism

Dr Nan Li
Office: ES222
Phone: 4921 6503
Nan.Li@newcastle.edu.au

Topic 4 Overview

- The 4 attributes of true O-O Programming
  - E(ncapsulation)
  - I(nformation)H(iding)
  - I(nheritance)
  - POLYMORPHISM
- Abstract Classes
- Virtual Functions and C++ Polymorphism
- Specification vs Implementation
- Pure Virtual Functions
- Java Polymorphism
- Pure Abstract Classes and Interfaces

The 4 attributes of true O-O Programming

- Encapsulation
  - Placing related Data (attributes) and their related Operations (methods) together (e.g. C++ structs).
- Information Hiding
  - Making these related data attributes private to the "outside world" and only accessible by way of its own public interface, a set of operations (methods) which are the only way that a client can request changes to the attribute data (eg. C++, Java, C#, etc. classes) .
- Inheritance
  - Allowing automatic building of one set of objects (class) from another, without the need to modify the original (class) definition.
- POLYMORPHISM
  - Allowing an object to respond to a method call according to its own (class's) definition of what this method call should produce, irrespective of how that object is known to the program. Method calls are implemented as strict message passing.

Polymorphism

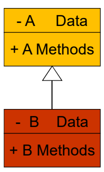
The basic idea is that rather than invoking a method in an object, what we are actually doing is sending a message to the object.
- Consequently, the object should respond to the message according to its own definition, and this should *not be influenced at all by the means by which the object is known to the program (how it is referenced)*.

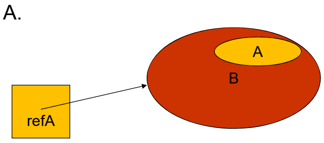
A message is of the form method_name(params) which is sent to the object, and the object then returns a response which is the return value for the method. The return value can be simple or complicated
- void (no response),
- a primitive value, or
- a reference (to an object).

The effect of the method (the side-effects, which may be most of what the method is to achieve) can be by way of
- changes to the object's attribute data,
- changes to the attribute data of any of the reference parameter objects,
- but also may include
  - output to an output stream (or input from an input stream),
  - throwing of an exception.

The Key to Polymorphism

The IS A relationship of inheritance
public class B extends A // in Java, or
public class B : public A // in C++
Because an object of type B IS A object of type A, it can be referred to (Java), or pointed at (C++) by a reference or pointer of type A.

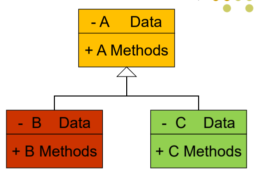




Polymorphism Key (cont.)

If we extend this to having classes B and C both inheriting from class A:

A reference of type A (say refA) can refer to either a B object or a C object, as well as a type A object.
- An array of type A can therefore even contain objects of type B, or type C, because a reference of type A can refer to a B or C object as well as an A object.
  - Exercise: Draw an array of type A in Java
  - Note: This does NOT ask for an array of type A objects. It can include B and C objects as well.
  - Draw the equivalent array in C++



The *actual* type of the object that refA refers to, or is referred to by any element of our array object, can only be determined by looking at the actual object as the program is being executed, ie. at *run-time*.

Polymorphism and Late Binding

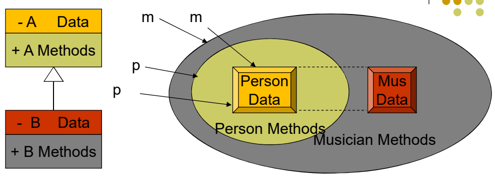
Polymorphism allows an object to respond to a message according to its own definition of what should be done.

- Rather than responding according to the way it is currently referred to in the program.

This means that the compiler won't be always able to determine which method definition to use.

- The program won't know exactly what type of object is being sent the message until run-time.
- This is termed ***late binding***, binding a function call (message) to a particular function (method definition) as the program executes (at runtime).

The Need for Polymorphism - C++ Pointers and Inheritance




```
Musician* m = new(….);
Person*  p = m;    // but the object itself is really just the data
// So what happens when we invoke the method ---
m->getAddrLabel(); // will get the Musician class implementation
p->getAddrlabel(); // will get the "wrong" one (the Person class one)
// But a Musician IS A Person, and we should be able to refer to a
// Musician as a Person and still have the object respond properly
// according to the rules of Polymorphism.
```

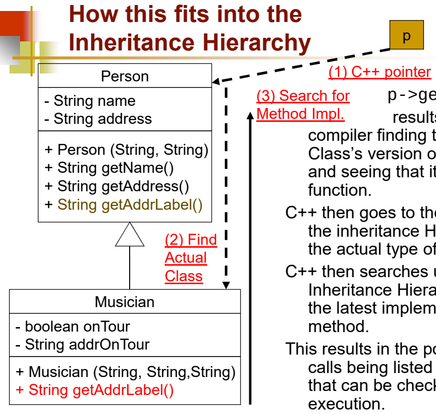

The C++ Answer - VIRTUAL

C++ introduces the keyword ***virtual*** to allow differentiating between functions (methods) which are to be directly called and those which may be re-implemented in a derived class, even when the object is referred to as if it were a member of an ancestor class.

```
// in Person class
virtual char* getAddrLabel() { return … }
// in usage example
p->getAddrlabel(); //
```

in previous example now returns the on-tour address label if the Musician's `onTour` boolean data is true, because the call is bound to the Musician-method at run-time even though it is a Person-type reference invoking the call.

How this fits into the Inheritance Hierarchy



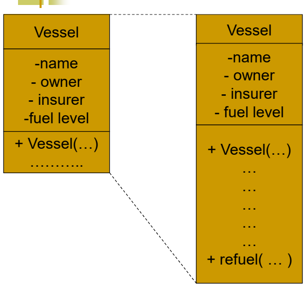
`p->getAddrLabel()` results in the C++ compiler finding the Person Class's version of getAddrLabel() and seeing that it is a virtual function.

C++ then goes to the top (bottom) of the inheritance Hierarchy to find the actual type of the object.

C++ then searches up the Inheritance Hierarchy until it finds the latest implementation of the method.

This results in the possible method calls being listed in a skip table that can be checked during execution.

Examples



Can contain implementations of certain methods

Any class that is derived from Vessel will be able to use these methods

- No matter what sort of vessel it is, it will always be refueled in the same way.
- Data items could be protected rather than private

Vessel is a <u>concrete</u> class because we know how to go about DOING <u>all</u> of the methods specificied

Specification vs Implementation of a Method

There are two parts to every method and we now must be able to differentiate between them:

- <u>Specification</u>: Includes everything needed to be able to <u>call</u> a method as well as process whatever it might return
  - The name of the method
  - The list of parameters that any call to the method must have
  - The type and contents of the answer that is returned
- <u>Implementation</u>: Everything needed to be able to actually execute the call
  - The set of operations in their specific order, given any particular set of parameters, that are used in the call

Vessel is a <u>concrete</u> class because we know how to go about DOING <u>all</u> of the methods defined

- That is, we have both specification and implementation of all of this class's methods.

Now it gets interesting

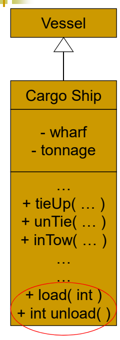
And this is one place where I lose some students who never get to understand the following points:

There are times when we know how to specify a particular method but cannot determine (as yet) how to implement it
- We can know the name of the method
- We can know the set of parameters we want to pass in
- We know what type of answer we are expecting

But we don't know exactly HOW to go about doing the method in this general sense (e.g. 3 or 4 different types of Cargo Ship).

If we _can_ SPECIFY any Method for a class but _cannot_ IMPLEMENT the method (because things are too general at this time), then the class becomes an ABSTRACT Class, we cannot have objects belonging to this class but the specification of it is still VERY useful.

An Abstract Class – Cargo Ship

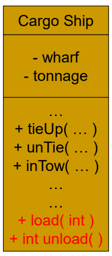


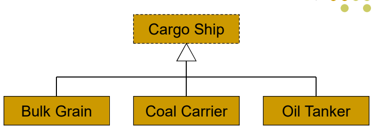
No matter what sort of cargo the ship carries, it will always be tied up and untied in the same way, so these methods can be implemented in this class.

No matter what sort of cargo, the ship will be taken in tow by a tug in the same way, and so this method can also be implemented.

But the ways in which loading and unloading is done will depend on the cargo type and so these cannot be implemented and the class is *Abstract*

Concrete Classes - Cargo Ships





Cargo Ship is an Abstract Class

Bulk Grain, Coal Carrier, Oil Tanker each contain their own implementations of
- `void load(int)`
- `int unload()`

```
or
new BulkGrain(…)  or
new CoalCarrier(…)
```

and are therefore concrete classes

They can therefore be instantiated, e.g.
```
CargoShip* c = new OilTanker(…)
```
but no matter which concrete class constructor is used a `CargoShip` pointer can still be used to reference it.

Pure Virtual Functions in C++

Shows that there is no implementation for the method in this class – so the class must be abstract.
- In class Cargo Ship

```
virtual void load(int)= 0;
virtual int unload() = 0;
```

Within class Person

```
virtual char* getAddrLabel() = 0;
```

- Could be used if Person were to be an Abstract Class with each sub-class now being required to have its own implementation of getAddrLabel()

Abstract Classes in C++

Abstract classes have at least one method that does not have an implementation.

Therefore they cannot be instantiated

An object of a derived concrete class (sub-class), once instantiated, can be referred to as a member of the abstract class (by a pointer or reference).

Abstract classes can only be used as a step in the Inheritance Hierarchy for an object.

But what about an array of CargoShips?
- In Java this is OK – the array is a separate object in its own right
- In C++, this is not possible, you can only have pointers to CargoShips.
- Remember that an array is just a simple container.

Polymorphism Advantages

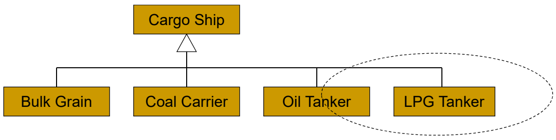
Allows each object to respond according to its own definition of what a particular method can do
- But what does this _really_ mean?

When a new kind of object needs to be added into a system, it will mean that there will be a new class
- with a new definition of what the (polymorphic or virtual) method is supposed to do, as well as any extra attribute data needed.
- the specification is already set out in the abstract class.
- with its own constructor method
- the other concrete classes do not have to be altered at all

A program can therefore be extended to include new capabilities (and new types of objects) without altering existing code (keeping alterations of code to a minimum).

Polymorphism provides SE support for extensibility.

Adding a new class



A new class with new method (& attribute) definitions

Nothing about the other classes needs to be changed

The only changes are where a particular type of cargo ship gets instantiated (the factory design pattern).

How would the program be different without polymorphism?

- The infamous "tentacles of change".
- How would the program be different without object-oriented programming?

Extensibility is difficult

In O-O modelling terms, it is very difficult to determine (forecast) how and where a piece of software is likely to be extended.

If polymorphism and late-binding is so good, why wouldn't you always program like this?

ANSWER: Java does.

In O-O modelling terms, it is very difficult to determine how and where a piece of software is likely to be extended.

So a language that always allows for late binding will always be ready to accept polymorphic-style extensions.

Java Solution

Every method is (in C++ terms) a "virtual" function and so does not need the keyword virtual.

Java always uses late binding to resolve names of methods being called, Java objects are inherently polymorphic.

No matter what type of reference is used in a Java program, when a method is called, Java will find the true type of the object, and work back up the inheritance hierarchy to find the appropriate definition of the method to be called.

At the time that this decision was made, Java was purely interpreted and so the cost in performance was not that great – modern advances bring this under scrutiny (a little). Compiler code optimisation helps at times.

Why does C++ allow other calls?

C++ has strictly compiled code, not interpreted byte code.

Significant performance improvements are possible by early binding of as many method calls as are possible.

- Code branch prediction methods, etc. can save a lot of execution time.
- A bound method call allows these to work effectively.
- Late bound method calls can vary according to the execution profile of the program.

An Aside: C# Inheritance & Polymorphism

Inheritance:

- When a derived class redefines a method in C#, the keyword overrides must be used.

Polymorphism:

- C# returns to the C++ strategy of making the base class method specification to carry the designation of virtual
- The overrides designation for the defined/redefined methods under inheritance remains.
- ie virtual …. overrides …… overrides ….

Pure Abstract Classes

If a class contains ONLY pure virtual methods in C++ then it can be thought of as a pure abstract class.

- It contains NO implementation of any of its methods at all
- It therefore contains ONLY specification of behaviour

The class exists as a source of behaviour specifications for inheritance.

It may also be used as a basis for the reference of different types of objects according to a common reference specification (a C++ pointer or reference, or a Java reference).

Pure Abstract Classes and Attribute Data

25

Attribute data in a pure abstract class can only be either protected (or public for class constants), as private data have no method implementations to make use of them, and once used as a basis of inheritance, any private data becomes inaccessible.

It would be rare for an abstract class to have no method implementations but then to also have attribute data associated with it.

Most pure abstract classes therefore have (virtual in C++) method specifications and no attribute data.

This is basically what an ***interface*** is in Java – with Java adding some extra requirements on how they are used.

Java Interfaces

26

```
public interface Queue {
   void enqueue(Object);
   Object dequeue();
   Object first();
   int size();
   boolean isEmpty();
}
```

Java does allow a class to be listed specifically as abstract, in which case not all methods need to be implemented.

This only contains specification of behaviour

Including it (effectively inheriting from it) into another class can therefore not run into the C++ diamond inheritance problem.

Java stipulates that any class inheriting from an interface must provide implementations of ALL methods in the interface and consequently introduces the keyword ***implements*** to show this.

C++ “Interfaces”?

27

```
class Queue {
  public:
   virtual void
      enqueue(Object)=0;
   virtual Object
      dequeue()=0;
   virtual Object first()=0;
   virtual int size()=0;
   virtual boolean
      isEmpty()=0;
};
```

Whatever you can do in Java in an O-O sense you can do in C++ as well.

In this case by use of Pure Abstract Classes

In these cases, C++ does not have a diamond (multiple) inheritance problem either.

C++ does not protect you from misuse of the facility. It does not restrict the way these classes are used and relies on good behaviour by the programmer.

Standard Java Interfaces

28

There are many groups of operations which might be useful on all types of objects, e.g.

- Comparable – where the objects can be ranked into a specific order.
- Serializable – where the objects need to be “flattened” into a byte stream for writing to disk or sending over a network.
- Cloneable – where an object needs to be copied rather than aliased.

There are standard ways of doing many of these operations that work for “most” objects and so implementations of them exist within the (abstract) Object class in Java, so that if they are not specifically implemented, then the standard (Object) one is used.

Back to Multiple Inheritance - Aspects and Adjectival Types

29

Interfaces are not a cure-all to the basic problems of C++ and diamond inheritance.

It is possible to have an in-between case, where behaviour *and data* that might otherwise be thought of as an interface, have a possibly powerful use in O-O terms.

These are the so-called “adjectival types” or “attribute types”.

They have resulted in what is known as Aspect-Oriented Programming (referred to Cross-Cutting Concerns).

C++ can effectively handle aspects via multiple inheritance, but without any specific protection from within the language to make sure that only aspects are included.

Java requires a new language (e.g. AspectJ) to allow these capabilities to be implemented.

Adjectival Types – An Example: Rentable

30

Note that “Rentable” is an adjective.

```
Class Rentable {
private:
  - String customerName;
  - Date returnDate;
  - Money costPerTime;
  ………
public:
  + void rentOut(Date);
  + Money returnItem(Date);
  ………
};
```

A Car for Avis or Hertz

Add Rentable to a Car and get a Rent-A-Car.

```
Class RentACar : public Car, Rentable {
  …
  …
  …
};
```

The same facility can easily make a Rentable DVD for VideoEzy or a Rentable House for Dial Dowling R/E.

Rentable adds data and behaviour, but both the data and behaviour are guaranteed (by the programmer) not to interfere with the other base class (the noun), only to qualify it.

The special thing to note is that there is no meaningful “IS A” relationship between a rental car, a rented DVD, or a rented house.

Cross-Cutting Concerns

This is a major term used in Aspect-Oriented Programming

- We have seen with inheritance that data and methods appearing in a number of classes can be “Factored Out” to form a Base (Parent) class in an inheritance hierarchy

What about “Rentable” appearing in classes like Rented-Car, Rented-House and Rented-DVD?

- They appear in a collection of classes
- BUT, there is no obvious relationship between the classes that can factor these to a higher point in the inheritance hierarchy
- They “Cut-Across” the inheritance hierarchy
- Houses, Cars and DVDs simply can’t be made into a hierarchy

Aspects allow these to be factored “across” the inheritance hierarchy, implemented once, and then added back into relevant classes where needed.

Other Aspects

The most popular Aspects to add into objects are

- Security for: SecureFile, SecureChannel
- Transactions for: collecting a wide variety of actions
- Synchronisation into a single atomic action
- Monitoring for: measuring just about anything

But it is likely that wherever there is an adjective in your problem statement, that you may be able to make use of an Aspect-Style structure in your program, especially where the same adjective might be applied to a number of very different classes (classes where no inherent or meaningful inheritance relationship seem to exist).

In some cases it will not be worth the extra effort, but as software systems scale – extra real benefits may appear.