ESS201: Programming-II
Module: C++

Jaya Sreevalsan Nair
jnair@iiitb.ac.in
International Institute of Information Technology, Bangalore

Term I: 2018-19; Lecture-08 on November 19, 2018

Reference in popular culture (and a great analogy) to polymorphism:

A beautiful lady in one of the StarTrek movies changes to a hideous troll. Capt. Kirk is astonished at the change, but the lady says,

"Didn't you know that I am a polymorph?"

From C++ Annotations

Polymorphism

Different types of inter-relationships are possible between two classes. One of which is *inheritance*, which encodes *is-a* relationship. One form of inheritance is called *polymorphism*.

- *Liskov Substitution Principle* (LSP) can be applied in an is-a relationship, where a derived class object may be passed to or used by code expecting a pointer or reference to base class object.
- However, LSP is implemented using polymorphism, where the base class pointer may behave like or perform actions of that of the derived class of the object it actually points. e.g. a Vehicle *vp can be pointing to a Car object, and polymorphism allows vp to behave like a Car eventually.
- Polymorphism is implemented using *late binding*. This means that the decision of which function to call is done at *run-time* and not at *compile-time*. The function could be in the base class or in the derived class.

Let us look at an example (`example2.cpp`) before proceeding further:

```
#include <iostream>

class Base{
  public:
     void process() { hello() ; return ;     }


  protected:
   void hello() { std :: cout << "base hello" << std :: endl ; return  ; }
} ;
```

```
class Derived : public Base {
   protected:
     void hello() { std :: cout << "derived hello" << std :: endl ; return ; }
} ;

int main() {
   Derived derived ;
   derived.process() ;

   Base &base = derived ;
   base.process() ;
   return 0 ;
}
```

Output from running this code:

```
base hello
base hello
```

Our observations:

- `Derived` object can call `process` from `Base` interface, but has no direct access to the function `Base::hello()`.
- It turns out that `Derived` does not have access to `Derived::hello()` either.
- How did that happen? The default setting in `C++` is *early* or *static binding*, which means that `Base::process` calls `Base::hello` and this has been already determined by the compiler.
- However, sometimes, the expectation here is that the when a `Derived` object calls `Base::process`, which internally calls the `hello` function, then `Derived::hello` must get implemented.

Summary: We are able to use inheritance to resolve the need for generating `Derived` class which *is-a* `Base` class. However, inheritance alone gives access to `Derived` class to only `Base` class member implementations within the `Base` class. `C++`, thus, offers polymorphism as a feature, which allows us to redefine (in a derived class) members of a base class, and further allows the use of the redefined (derived class) members from the interface of the base class.

The essence of LSP is that, the `public` inheritance should be used:

- not in the cases of *reuse* of the base class members in the derived classes;
- but, in cases of *reuse* of the base class members, by the base class itself, by *polymorphically* using derived class members, which are re-implementations of the base class members.

Polymorphism allows us to do the following:

1. Re-implement base class members and use the re-implemented members in code expecting references or pointers of the base class;
2. Re-use existing code by derived class by overloading the appropriate members of their class.

How does polymorphism work? This is done with the use of the keyword `virtual`.

Polymorphism has to be *explicitly* stated.

- The reason why polymorphism has to be explicitly stated is because, *late binding* is not the default setting for function calls. The default is *static or early binding*, where the compiler determines which function needs to be called based on the class types of objects, and pointers or references to the object.
- Late binding is comparatively slower than early binding, since it is a decision made at run-time as opposed to compile-time.
- Unlike most other OOP languages where only late binding is offered, C++ offers both early as well as late binding. Hence, care must be taken in which choice is being made, and the optimizations done with respect to the choices.

Our example code is modified as follows, where the keyword `virtual` is used;

```
class Base {
  public:
     void process() { hello() ; return ; }
  protected:
     virtual void hello() { std :: cout << "base hello" << std :: endl ;}
} ;
```

Recompile the modified code snippet, and run it to produce the expected result of derived `hello` in derived process. The output from running this is as follows:

```
derived hello
derived hello
```

Thus, late or dynamic binding is allowed in `C++` using `virtual` member functions, which is different from early or static binding. The latter is the default, where the behavior of the member function, which is called via a pointer or reference, is determined by the implementation of that function in the class of the pointer or reference. e.g. `Vehicle*` calls the `Vehicle` member functions *only*, even when pointing to an object of a derived class.

Points to remember when using `virtual`:

- The keyword `virtual` makes a specific member function a *virtual member function*. This is not a default setting, but has to be explicitly defined in C++ (unlike in OOP languages, where late binding is the default).
- If a member function in base class is declared `virtual`, it remains virtual in all (nested) derived classes. The keyword `virtual` should not be mentioned in the derived class, where the members are declared virtual in the base class.
- A member function may be declared `virtual` anywhere in a class hierarchy, but this defeats the underlying polymorphic class design, as the original base class is no longer capable of completely covering the redefinable interfaces of derived classes. e.g.

  ```
  size_t mass() const ; // for Vehicle,
  ```

  and

  ```
  virtual size_t mass() const ; // for Truck
  ```

This means that the late binding for the function is available only for `Truck` objects and objects for classes derived from `Truck`, and not for the entire hierarchy. Static binding is used until the point of `Truck` in the class hierarchy.

Enhancing reusability: Good class design has a *redefinable* interface to enhance *reusability.*

- A redefinable interface allows derived classes to fill in their own implementation, without affecting the user interface.
- At the same time the user interface will behave according to the expected behavior of the derived class, and not just to the default implementation of the base class.
- Members of the reusable interface should be declared in the `private` section of the classes, as they conceptually belong to their own classes.
- Those functions should not be called by code using the base class, but they exist to be overridden by derived classes using polymorphism to redefine the behavior of the base class.
- Separating user interface from the redefinable interface allows us to fine-tune the user interface (this is desirable as it has only one point of maintenance), while at the same time allowing us to standardize the expected behavior of the members of the redefinable interface.

Virtual destructors: Consider the following:

```
Vehicle *vp = new Four_wheeler(3000, 200, "Toyota") ;
delete vp ;
```

It is legal for a base class pointer to point to a `new` derived class object. Now, the `delete` is applied to a base class pointer. This obviously leads to memory leak. Moreover, a destructor not only frees allocated memory, but also performs actions which are necessary when an object ceases to exist. e.g. when a bank account is deleted, the remaining balance is credited to the account holder before deletion. Thus, calling the base class destructor in this case does not work in this case.

This issue is resolved in `C++` by the use of *virtual destructors.* Salient points about virtual destructors:

- When a base class destructor is declared `virtual` then the destructor of the actual class pointed to by a base class pointer `b_ptr` is going to be called when `delete b_ptr` is executed.
- Late binding is realized for destructors even though the destructors of derived classes have unique names. In this specific manner, virtual destructors are different from other virtual member functions, where the functions are overloaded and hence the function names have to be the same.
- It also means that the destructor of the appropriate class performs as it usually does. i.e., `Truck` destructor finishes execution, and internally calls the destructors of its base classes, one at a time, up the class hierarchy, i.e. `Four_wheeler`, `Land_derv`, and finally, `Vehicle`.
- The rule of thumb is that the destructors should always be defined `virtual` in classes designed as a base class from which other classes are going to be derived.
- Virtual destructors, as the non-virtual ones, must be in the `public` section of the class interface, as opposed to the recommendation for redefinable interface for enhancing reusability.
- But, do not define virtual destructors (even empty ones) inline as this complicates class maintenance.
  - In general, virtual members/functions should *never* be implemented `inline`. This is because the internal implementation of polymorphism expects physical addresses of the virtual functions of the class. Since inline functions are not explicitly called, the compiler overlooks the functions, thus causing linkage problems at run-time.

The usage of virtual destructor is as follows:

```
class Vehicle {
  public:
    virtual ~Vehicle() ; // all derived class destructors are virtual.
}
```

Pure virtual functions: In several base class designs, the virtual member functions do not have to be implemented in the base class.

- That just means that the derived classes are required to provide the missing implementations.
- This feature is called *interface* in some languages (e.g. Java), and is defined as a *protocol* in C++.
  - Derived classes must obey the protocol by implementing these as-yet-not-implemented members.
- If a class contains at least one member whose implementation is missing, then no objects of that class can be defined.
  - Such incompletely defined classes are always base classes.
  - They enforce a protocol by merely declaring names, return values and arguments of some of their members.

  These classes are called **abstract classes** or **abstract base classes**. Derived classes become non-abtract classes by implementing the as yet not implemented members.
  - Abstract base classes are the foundation of many design patterns, especially templates.
- Members that are merely declared in base classes are called **pure virtual functions**.
  - A virtual member becomes a pure virtual member by postfixing = 0 to its declaration (i.e., by replacing the semicolon ending its declaration by '= 0;').
  - All classes derived from the abstract base class must implement the the member function marked as *pure virtual* in the base class, or the objects of the derived class cannot be constructed.
  - Abstract base classes need not have data members. However, once a base class declares a pure virtual member, it must be declared identically in derived classes.

Virtual destructors as pure virtual functions: The virtual destructors cannot be pure virtual functions.

- This is because not all derived classes need to have a separate implementation of the destructor; and such destructors are provided by default.
- if it is a pure virtual member, its implementation in the base class does not exist. However, derived class destructors eventually call their base class destructors, by default. Thus, the default action will not be possible.

```
class Base {
  size_t _base_data ;
  virtual void pure_imp(void) = 0 ;

  public:
    virtual ~Base() ;
} ;

class Derived: public Base {
   public:
```

```
     void pure_imp(void) ;
} ;

void Derived :: pure_imp(void) {
   std :: cout << "Derived::pure_imp() called" << std :: endl ;
}
```

Implementing pure virtual member functions:

- Pure virtual member functions may be implemented. To implement a pure virtual member function, the definition has the expected "= 0;" specification, but this may be implemented the base class as well.
- Since the "= 0;" ends in a semicolon, the pure virtual member member is always at most a declaration in the abstract base class. An implementation maybe provided in the base class outside of its interface, or using `inline`.
- Pure virtual member functions may be called from derived class objects or from its class or derived class members by specifying the base class and scope resolution operator together with the member to be called.
- *Implementing* a pure virtual member in the base class has limited use.
  - One could argue that the implementation of the pure virtual member function may be used to perform tasks that can already be performed at the base class level.
  - However, there is no guarantee that the base class virtual member function is actually going to be called.
  - Therefore, base class specific tasks could as well be offered by a separate member, without blurring the distinction between a member doing some work and a pure virtual member enforcing a protocol.

```
#include <iostream>
class Base {
  public:
   virtual ~Base();
   virtual void pureimp() = 0;
};

Base::~Base()
{}

void Base::pureimp() {
  std::cout << "Base::pureimp() called\n";
}
class Derived: public Base {
  public:
   virtual void pureimp();
};

inline void Derived::pureimp() {
  Base::pureimp();
  std::cout << "Derived::pureimp() called\n";
}
```

```
int main() {
  Derived derived;
  derived.pureimp();
  derived.Base::pureimp();
  Derived *dp = &derived;
  dp->pureimp();
  dp->Base::pureimp();
}

// Output:
// Base::pureimp() called
// Derived::pureimp() called
// Base::pureimp() called
// Base::pureimp() called
// Derived::pureimp() called
// Base::pureimp() called
```

Pure virtual functions as `const` members:

- Even though optional, quite often pure virtual member functions are `const` member functions. This is done to allow the construction of `const` derived class objects.
- The general rule for const member functions also applies to pure virtual functions: if the member function alters the data members of the object, it cannot be a `const` member function.
- This rule is strictly applied through the class hierarchy starting from the abstract class. Hence, if any of its derived class will have the need to alter the data members of the derived class object through the implementation of polymorphised member function, then the function must not be declared `const` at the base class.

Internal working of polymorphism: Knowing the internal workings of polymorphism allows one to understand the penalty in memory usage and efficiency when using polymorphism.

- Appropriate function call in polymorphism is determined at run-time, and not at compile-time.
- Given this requirement, the address of the function must be available/accessible to the object (calling the function), so that it can be looked up prior to the actual call.
- An object containing virtual member functions also contains, usually as its first data member a hidden data member, pointing to an array containing the addresses of the virtual member functions of the class. This hidden data member is usually called the **vpointer**, which gives the array of virtual member function addresses, called the **vtable**.
- The vtable of a class is shared by all its objects. The overhead of polymorphism in terms of memory consumption is therefore:
  - one vpointer data member per object, pointing to,
  - one vtable per class.

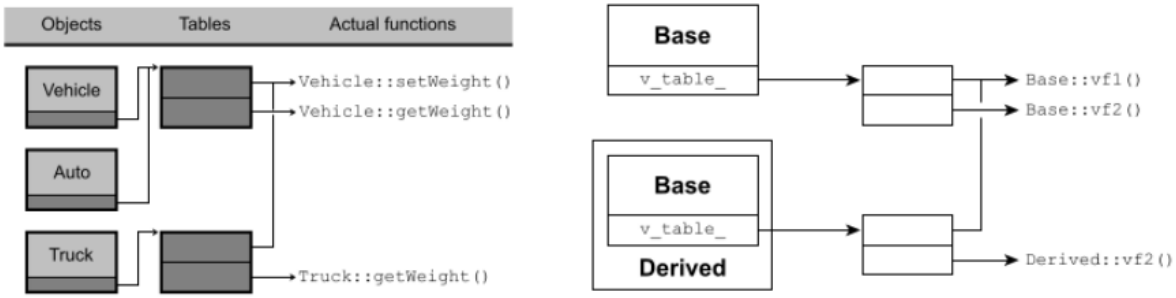

Figure 1: Vtables in base and derived classes. Image courtesy: C++ Annotations.

Let us analyze the case of `Vehicle`:

- A statement like `vp->mass` first inspects the hidden data member of the object pointed to by `vp`.
- In our case, `vp->vpointer` points to `vp->vtable` containing addresses of a get function and a set function: one pointer to the function `mass` and one pointer to the function `setMass` (it will be three pointers if the class also defines (as it should) a virtual destructor).
- The address of the function to be executed is determined from this table.
- The base and derived classes share the same vtable, unless the derived class overrides any of the member functions, in which case, it will have its own vtable

Error in vtable: Sometimes, the linker generates an error, such as:

```
In function 'Derived::Derived()':
: undefined reference to 'vtable for Derived'
```

Note: in our common usage, we use g++ for both compiling and linking.

This error implies that:

- Even though a virtual function is mentioned in the interface of the derived class, its implementation is missing in the class implementation of the derived class.
- Usually, the compiler passes such code, leading up to construction of objects of the derived class. However, the linker throws an error as it is unable to find the implementation of the derived class function.

Example code that generates the error:

```
class Base {
  virtual void member();
};

inline void Base::member() {}

class Derived: public Base {
  virtual void member();
  // only declared
};
```

```
int main() {
  Derived d; // Will compile, since all members were declared.
             // Linking will fail, since we donŠt have the
             // implementation of Derived::member()
}
```

Source Code

1. `example/example1.*` shows how a `public` function, `process()`, that is overloaded in both base and derived classes can be invoked through a derived class instantiation.
2. `example/example2.*` does not have the overloading of the `public` function, `process()`, in the derived class. Then, when the derived class object invokes `process()`, the base class implementation of the function occurs.
3. `example/example3.*` show how `example2.h/cpp` can be modified by the use of `virtual`.
4. `v0`: example of virtual function in `Vehicle`
5. `v1`: example of virtual destructor
6. `v2`: example of pure virtual function

References

- C++ Annotations

Source code for `example1.*`

```
#ifndef Example_H_
#define Example_H_

#include <iostream>

class Base {
 public:
  void process() { hello() ; return ; }
 protected:
  void hello() { std :: cout << "base hello" << std :: endl ; return ; }
} ;

class Derived : public Base {
  // overloading base.process //
 public:
  void process() { hello() ; return ; }
 protected:
  void hello() { std :: cout << "derived hello" << std :: endl ; return ; }
} ;

#endif
```

```
#include "example1.h"

int main() {
  Derived derived ;
  // derived.hello() ; // compiler error since it is a protected function //
  derived.process() ;

  // in case of invoking base.process //
  Base &base = derived ;
  base.process() ;

  return 0 ;
}
```

Source code for `example2.*`

```
#ifndef Example_H_
#define Example_H_

#include <iostream>

class Base {
 public:
  void process() { hello() ; return ; }
 protected:
  void hello() { std :: cout << "base hello" << std :: endl ; return ; }
} ;

class Derived : public Base {
 protected:
  void hello() { std :: cout << "derived hello" << std :: endl ; return ; }
} ;

#endif
```

```
#include "example2.h"

int main() {
  Derived derived ;
  // derived.hello() ; // compiler error since it is a protected function //
  derived.process() ;

  // in case of invoking base.process //
  Base &base = derived ;
  base.process() ;

  return 0 ;
}
```

Source code for `example3.*`

```
#ifndef Example_H_
#define Example_H_

#include <iostream>

class Base {
 public:
  void process() { hello() ; return ; }
 protected:
  virtual void hello() { std :: cout << "base hello" << std :: endl ; return ; }
} ;

class Derived : public Base {
 protected:
  void hello() { std :: cout << "derived hello" << std :: endl ; return ; }
} ;

#endif
```

```
#include "example3.h"

int main() {
  Derived derived ;
  // derived.hello() ; // compiler error since it is a protected function //
  derived.process() ;

  // in case of invoking base.process //
  Base &base = derived ;
  base.process() ;

  return 0 ;
}
```