ESS201: Programming-II
Module: C++

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

Term I: 2018-19; Lecture-06 on November 05, 2018

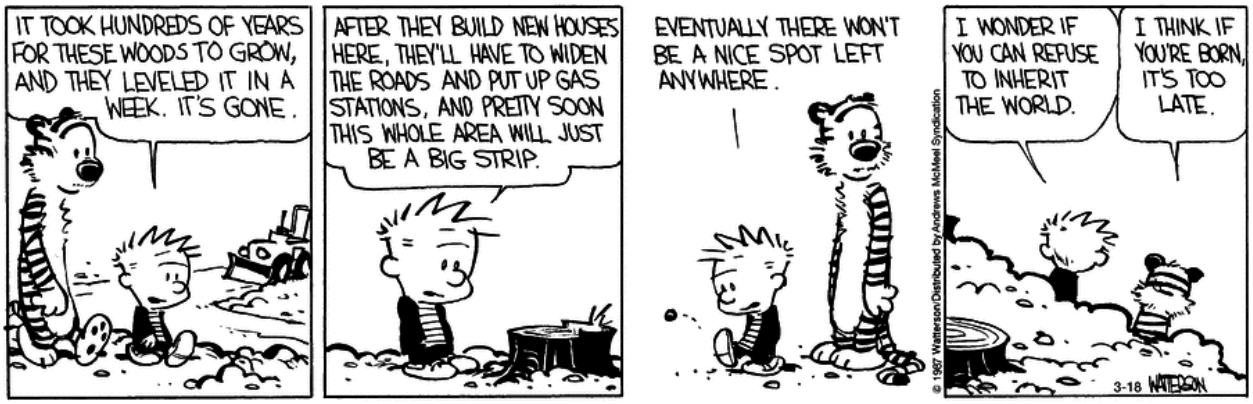


Image courtesy: Bill Watterson/ Calvin and Hobbes

1 Inheritance

C++ offers features where one can define relationship between different related classes. Earlier, we have seen inter-class relationships such as *composition*, *aggregation*, and *association*. Yet another relationship that we will explore here is that of a class being defined in terms of an older, pre-existing, class. This produces a new class having all the functionality of the older class, and additionally defining its own specific functionality. Instead of *composition*, where a given class contains another class, we here introduce **derivation** or **inheritance**, where a given class *is-a(-type-of)* or *is-implemented-in-terms-of* another class.

The existing class in an inheritance is called the **base class**, while the new class is called the **derived class**. Properties of inheritance:

- The derived class inherits the functionality of the base class,
- The base class does not appear as a data member in the interface of the derived class.

Demonstrating how `Land` vehicle is derived from base class `Vehicle`.

```
class Vehicle {
    size_t _mass;
  public:
    // constructors/destructor //
    Vehicle();
    Vehicle(size_t mass):_mass(mass);
    ~Vehicle();

    // get/set functions //
    size_t mass() const;
    void setMass(size_t mass);
};
```

Let us look at the class `Land_derv`, which *inherits* (or, *is derived*) from Vehicle:

```
class Land_derv: public Vehicle {
    size_t _speed;
  public:
    // constructors/destructor //
    Land_derv();
    Land_derv(size_t mass, size_t speed) ;

    // get/set functions //
    void setSpeed(size_t speed);
    size_t speed() const;
};
```

Having access to public methods in `Vehicle`, `setMass` from a `Land_derv` will invoke the member function in `Vehicle` and set value of private data member `_mass` that belongs to `Vehicle`
Compare this with a composition:

```
class Land_comp {
      Vehicle d_v ; // composed Vehicle
      size_t _speed ;

   public:
      // constructors/destructor //
      Land_comp() ;
      Land_comp(size_t mass, size_t speed) ;
      ~Land_comp() ;

      // get/set functions //
      void setMass(size_t mass) ;
} ;

void Land::setMass(size_t mass) {
     d_v.setMass(mass) ;
}
```

Using composition, the `Land_comp::setMass` function only passes its argument on to `Vehicle::setMass`. Thus, as far as handling the variable `mass` is concerned, `Land::setMass` introduces no extra functionality, just extra code.

Thus, in the above example, the intended relationship of *being-a-type-of* or *is-a* is represented better by inheritance than by composition. A rule of thumb for choosing between inheritance and composition distinguishes between *is-a* and *has-a* relationships. A truck is a vehicle, so it is best if `Truck` *derives* from `Vehicle`. On the other hand, a truck has an engine; which may be better represented by *composing* an `Engine` class inside the `Truck` class.

Interpretation of `class Land_derv: public Vehicle` is that the derived class `Land_derv` now contains all the functionality of its base class `Vehicle` as well as its own features. Following are the features of derivation exhibited here:

1. The private data member of the base class, `_mass` is not mentioned as a member in the interface of `Land_derv`. Nevertheless it is used through the (public member) function of the base class `mass()`. This member function is an implicit part of the class, inherited from its "parent", `Vehicle`.

2. Although the derived class `Land_derv` contains the functionality of base class, the private members of the base class remain private, i.e., they can only be accessed by the member functions and friends (`friend` members and functions) of the base class. This means that member functions of `Land_derv` must use member functions of `Vehicle` (i.e. `mass()` and `setMass`) to access or manipulate the `_mass` data member. Here there is no difference between the access rights granted to `Land_derv` and the access rights granted to other code outside of the class `Vehicle`. The idea here is that the base class encapsulates its own specific characteristics, and *data hiding* is used for realizing encapsulation.

Encapsulation as a *core* principle of good class design gives the following benefits:

1. Encapsulation reduces the dependencies among classes, thus, improving the maintainability and testability of classes. It allows modification of classes without the need to modify the code for dependent classes.

2. By strictly complying with the principle of data hiding the internal data organization of a class may change without requiring depending code to be modified as well.

Designing inheritance: When using inheritance to implement an is-a relationship, the "direction of use" must be determined correctly. That implies that the design of the inheritance must follow from the design of the base class.

- The base class facilities are not there to be used by the derived class.

- On the contrary, the derived class facilities must redefine (reimplement) the base class facilities using polymorphism (using `virtual` functions). *Note: Polymorphism will be discussed at length in a later lecture.*

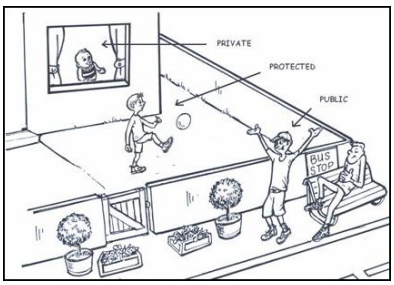
- This design principle allows code to use the derived class facilities polymorphically through the base class.

- Composition over inheritance: Overall, when designing classes, always aim at the lowest possible coupling. When the interface of a class is only partially or, worse, minimally used and if the derived class is implemented in-terms-of, or broadly, is-part-of (instead of an is-a relationship) another class, composition is a better design choice than inheritance. Define the appropriate interface members in terms of the members offered by the composed objects (as shown in the example of `Land_comp`).

Regarding compilation: Whenever the internal data organization changes in the base class, the derived class need not be modified (ideally in a good class design), but *needs* to be re-compilled. The only scenario where re-compilation of derived class is *not* required with respect to modification of base class is when *only* new member functions are added to the base class, which are not yet used in derived class. *Note: the afore-mentioned point of re-compilation of derived class with respect to addition of new member functions in hte base class is not applicable if the new function is the first virtual member function of the base class.*

1.1 Data Hiding: Access Rights

Figure 1 shows an analogy of access rights, providing an understanding of **data hiding**. In C++, data hiding restricts control over the data of an object to the members of its class, and encapsulation is used to restrict access to the functionality of objects. Both principles are invaluable tools for maintaining *data integrity*.



Access Specifier	Same Class	Derived Class	Any Other Class	Friend Function	Friend Class
Private	Yes	No	No	Yes	Yes
Protected	Yes	Yes	No	Yes	Yes
Public	Yes	Yes	Yes	Yes	Yes

Figure 1: (Left) Analogy of access rights; (right) Access rights in class inheritance in C++. Image courtesy: stackoverflow, startertutorials

The keyword `private` is the main tool for data hiding. On the other hand, the `public` members allow users to communicate with objects but the objects decide on how requests sent to objects are handled. In a well-designed class, only its objects are in full control of their data.

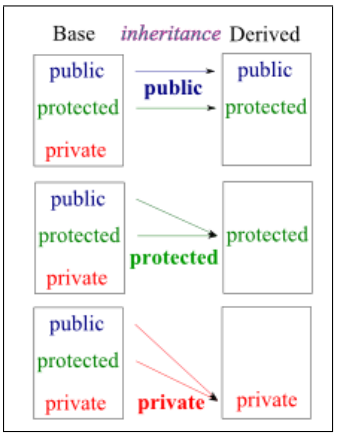
Inheritance adheres to these principles, including the way the `private` and `protected` keywords operate.

1. A derived class does not have access to `private` members of the base class.
2. The keyword `protected` is used for members of the base class which are not in its `public` interface, but can be accessed by derived classes. However, the use of `protected` must be done judiciously, as its overuse can lead to unnecessary tight coupling of the base and derived class – which is *bad* class design.
3. If a derived class (and not other users of the base class) must have access to the `private` data of the base class, then a better practice to use member functions (such as, accessors and modifiers) which are declared protected in the interface of the base class. This enforces the intended restricted access without resulting in tightly coupled classes.

Figure 1 shows the *scope of influence* of the keywords `public`, `protected`, and `private` in base class with respect to a derived class.

1.2 Different Modes of Inheritance

Just as there are access rights to members of a class, there are different modes in which inheritance can be defined. The keywords `public`, `protected` and `private` are used as different *modes* of inheritance. Figure 2 shows the scope of access rights in different modes of inheritance.



Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Figure 2: Access rights in different modes of inheritance. Image courtesy: bogotobogo, geeksforgeeks.

Definition of the different modes of inheritance:

- **`public` inheritance**: When `public` derivation is used the access rights of the interface of the base class remains unaltered in the derived class. With inheritance, public derivation is frequently used. Its usage is:

  ```
  class Derived: public Base
  ```

- **`protected` inheritance**:

  ```
  class Derived: protected Base
  ```

  When `protected` derivation is used, all `public` and `protected` members of the base class become `protected` members in the derived class. The derived class may access all the `public` and `protected` members of the base class. But (nested derived) classes that are in turn derived from the derived class view the members of the base class as `protected`. Any other code (outside of the inheritance tree) is unable to access the members of the base class.

- **`private` inheritance**: When `private` derivation is used, all the members of the base class become `private` members in the derived class. The derived class members may access all base class `public` and `protected` members but base class members cannot be used elsewhere.

When to use what?

- `private` inheritance is the default mode of inheritance. In cases where the mode of inheritance is not specified, the default mode is used. It is usually needed in situations where a derived class object is defined *in-terms-of* the base class, but where composition cannot be used.

- `public` inheritance should be used to define an *is-a* relationship between a derived class and a base class: the derived class object *is-a* base class object allowing the derived class object to be used polymorphically as a base class object in code expecting a base class object.

- `protected` inheritance is not widely used, but one could maybe encounter `protected` inheritance when defining a base class that is itself a derived class and needs to make its base class members available to classes derived from itself.

From the perspective of users of base and derived classes: When `private` or `protected` derivation is used, users of derived class objects are denied access to the base class members. `private` inheritance denies access to all base class members to users of the derived class, `protected` inheritance does the same, but allows classes that are in turn derived from the derived class to access the `public` and `protected` members of the base class.

1.3 Nested Inheritance

Inheritance can be re-thought in the form of a tree data structure. Presence of a second level of depth in the tree indicates nested derivation or inheritance. e.g. if we have `Truck` as a class derived from `Land_derv`, which itself is derived from `Vehicle`, then this occurrence is called **nested inheritance/derivation**.

However, nested inheritance must be used with caution. Repeatedly deriving classes from classes quickly results in big, complex class hierarchies that are hard to understand, hard to use and hard to maintain. Data hiding is hard to be adhered to in complex class hierarchies.

1.4 Multiple Inheritance

When a class is derived from a single base class, it is called single inheritance. In addition to single inheritance, C++ also supports **multiple inheritance**. In multiple inheritance a class is derived from several base classes and hence inherits functionalities from multiple parent classes at the same time.

When using multiple inheritance, the newly derived class would be considered as an instantiation of all base classes. If all the derived class does not have a strong *is-a* relationship with both the base classes, composition is more appropriate. In general, single inheritance is more widely used than the multiple one.

Good class design dictates that a class should have a single, well described responsibility and that principle often conflicts with multiple inheritance where we can state that objects of class `Derived` are both `Base1` and `Base2` objects. This implies `Derived` is-a `Base1` type as well as is-a `Base2` type. e.g. Smartphone is-a phone and is-a camera.

```
class Smartphone: public Phone, public Camera {
   public:
     Smartphone();
};
```

The keyword `public` is present before both base class names as the default is `private` derivation. The keyword `public` must be repeated before each of the base class specifications. However, it is not required that all base classes use the same mode of inheritance. e.g. one could be `public` and the other `private`. In situations where two base classes offer identically named members special provisions need to be made to prevent ambiguity:

```
Smartphone smartphone() ;
smartphone.Phone::setSensor("XYZ") ;
smartphone.Camera::setSensor("XYZ") ;
```

When to use? Many a times the use of multiple inheritance is not for introducing additional functionality of its own, but to merely combine two existing classes into a new aggregate class. Thus, C++ offers the possibility to simply sweep multiple simple classes into one more complex class. Given this definition also fits for association, one can make a judicious call of which design concept is required for a specific scenario.

2 Member Functions of Derived Classes

Constructors: Since a derived class object is constructed or built on top of the base class object, the base class must be constructed *before* the derived class is initialized. Since a constructor is where data members of an object are initialized, the constructor for derived class is responsible for proper initialization of base class.

```
class Vehicle {
     size_t _mass ;
     ...
} ;

class Land_derv: public Vehicle {
     size_t _speed ;
} ;

Land_derv :: Land_derv(size_t mass, size_t speed) {
     setMass(mass) ;
     setSpeed(speed) ;
}
```

This implementation has the following disadvantages:

- The base class constructor is *always* called before any action is implemented with respect to the derived class.
- Similar to the argument for the use of parametrized inputs for constructors, the afore-mentioned assignment *after* initialization is inefficient as well as impossible where data `const` members or base class reference must be initialized. In such cases, it is best to use a special constructor of base class must be used instead of the default one.

Hence, there is a more efficient implementation that can be usedm where a base class constructor is called within the derived class initializer clause/list. This specific implementation is called the *base class initializer*.

- However, the base class initializer must be called *before* initializing any of the data members of the derived class.
- When using the base class initializer, none of the data members of the derived class may be used.
- When constructing a derived class object, only *after* the base class object is constructed successfully, the derived class data members are available for initialization.

```
Land_derv :: Land_derv(size_t mass, size_t speed):
                      Vehicle(mass), _speed(speed) {}
```

Derived class by default calls the default constructor of the base class, which must be corrected in the case of copy constructors. Hence, copy constructor of the base class is called, and then the copy constructors of the data members of the derived class are called.

```
Land_derv :: Land_derv(Land_derv const &other):
                      Vehicle(other), _speed(other.speed) {}
```

Inheriting Constructors: Derived classes are allowed to be constructed *without* explicitly defining constructors of the derived classes. In such cases, the available *base class constructors* are called, i.e., the construction of derived class objects is *delegated* to the base class constructor(s). The following syntax can be used in such cases:

```
class MyBaseClass {
   public:
   // Constructors //
} ;
class MyDervClass: public MyBaseClass {
   public:
   using MyBaseClass::MyBaseClass ;
}
```

However, this feature cannot be used in the case of *multiple inheritance* where not all base class signatures are unique. This is best avoided in the case of multiple inheritance in general, given its complexity.
Destructors: When an object is destroyed, destructors are automatically called. For derived class objects, destructor of both the derived class and base class are called. Constructors and destructors are called in a stack format.

- When derived class is constructed, the appropriate base class constructor is called, followed by the appropriate derived class constructor.
- When the derived class object is destroyed, its destructor is called first, automatically followed by the activation of the base class destructor. A derived class destructor is always called *before* its base class destructor is called.

```
class MyBaseClass {
    public:
      ~MyBaseClass() ;
} ;

class MyDervClass : public MyBaseClass {
    public:
      ~MyDervClass() ;
} ;

int main() {
    MyDervClass l_derv ;
}
```

At the end of `main` function, the destructor of `MyDervClass` is called first for `l_derv`, followed by that of `MyBaseClass`.

- The only case the base class destructor is *not* called is when the derived class constructor throws an exception and does not complete construction completely.

3 Redefining Member Functions

Consider the following code snippet:

```
class Vehicle {
   size_t _mass ;
   public:
     void setMass(size_t mass) ;
} ;

class Land_derv: public Vehicle {
   size_t _speed ;
   public:
     void setSpeed(size_t speed) ;
} ;

class Four_wheeler: public Land_derv {
   std :: string _brand ;
   ...
} ;

class Truck: public Four_wheeler {
   size_t _mass ;
   public:
     void setMass(size_t tractor_mass, size_t trailer_mass) ;
     size_t mass() const ;
} ;

void Truck::setMass(size_t tractor_mass, size_t trailer_mass) {
   _mass = tractor_mass + trailer_mass;
   Four_wheeler::setMass(tractor_mass); // note: Four_wheeler:: is
                                        // required
   return ;
}
...
truck.Four_wheeler::setMass(x) ;
```

In this example, the derived class `Truck` has a data member `_mass` which has the same name as the base class `Four_wheeler` has (through its inheritance from `Vehicle`).

- The redefinition of `setMass` is allowed; however, it *hides* `Four_wheeler::setMass`.
- For `Truck`, only `setMass` with two `size_t` parameters will work.
- The usage of `Four_wheeler::setMass` must be used with the scope resolution operator (i.e., ':'), both in function implementations of `Truck` as well as outside of the class.

Alternatives to avoid the scope resolution operator outside class definitions:

1. The function prototype as in the base class can be redefined and implemented `inline` to call the base class member. As the function is defined `inline`, no overhead of an additional function call is involved.

   ```
   // in the interface of class Truck
   void setMass(size_t tractor_mass) ;
   ```

```
// outside, but below, the interface
inline Truck::setMass(size_t tractor_mass) {
    _mass = mass() - Four_wheeler::mass() + tractor_mass ;
// Substituting mass() with _mass gives erroneous results.
Four_wheeler::setMass(tractor_mass) ;
}
```

2. A `using` declaration may be added to the derived class interface to prevent hiding the base class members.

   ```
   class Truck: public Four_wheeler {
      public:
         using Four_wheeler::setMass;
         void setMass(size_t tractor_mass, size_t trailer_mass);
   };
   ```

   A `using` declaration imports (all overloaded versions of) the mentioned member function directly into the derived class interface. We can now use from the above, both `truck.setMass(5000)` as well as `truck.setMass(5000, 2000)`.

   - However, if a base class member has a signature that is identical to a derived class member then compilation fails (a `using Four_wheeler::mass` declaration cannot be added to interface of `Truck`).
   - The `using` declarations must *obey* access rights. To prevent non-class members from using `setMass(5000)` without a scope resolution operator but allowing derived class members to do so, the `using Four_wheeler::setMass` declaration should be put in the `private` section of Truck interface.

     ```
     class Truck: public Four_wheeler {
         public:
           void setMass(size_t tractor_mass, size_t trailer_mass);

         private:
           using Four_wheeler::setMass;
     };
     ```

Overloading functions with same input parameters: In this example, member function `mass()` is redefined to return the `private` data member of the derived class, as opposed to the same of the base class.

This has to be *explicitly* done, or else the function definition of the base class is used. It must be noted that the function definition of the base class accesses the data member of the base class, as the data member has the same name.

```
size_t Truck::mass() const {
   return _mass;
}
```

Given the above definition, `truck.mass()` would return the data member `_mass` of class `Truck`. If this function is *not redefined* as above then, `truck.mass()` will still work, except, it is now calling the function `_mass` of its base class `Four_wheeler`, which in turns passes it on to its base class, `Land_derv` and which again passes it on to its base class, `Vehicle`. The variable returned here, then is `Vehicle::_mass`, and not `Truck::_mass`.

Inheritance+Composition: Taking the point of view of a `Truck` *is-a* `Four_wheeler` (tractor) containing a `Vehicle` (trailer), we can try a redesign of aforementioned example:

```
class Truck: public Four_wheeler{ // for the tractor
    Vehicle _trailer;
    public:
       Truck();
       Truck(size_t tractor_mass, size_t speed, char const *name,
       size_t trailer_mass);
       void setMass(size_t tractor_mass, size_t trailer_mass);
       void setTractorMass(size_t tractor_mass);
       void setTrailerMass(size_t trailer_mass);
       size_t tractorMass() const;
       size_t trailerMass() const;

       // consider:
       Vehicle const &trailer() const;
};
```

4 Example Codes and Practicals

Folder specifications:

1. `v0`: Derived class constructors and destructor.
2. `v1`: Overloading members and functions in derived classes with same member and function names as base class.
3. `v2`: Overloading member functins in derived classes with same input parameters as base class.