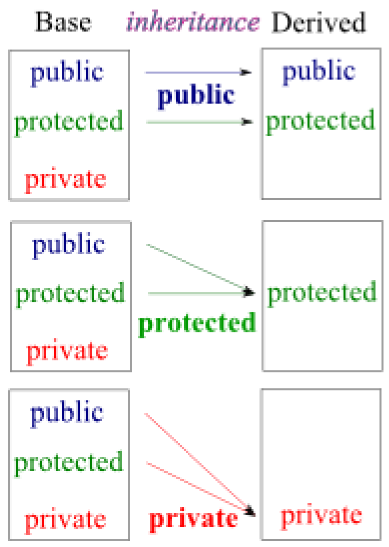
Private Inheritance



Private Inheritance is one of the ways of implementing the **has-a** relationship.

With private inheritance, public and protected member of the base class become private members of the derived class. That means the methods of the base class do not become the public interface of the derived object. However, they can be used inside the member functions of the derived class.

Because there are implications of using private inheritance, in his book, "Effective C++ 55 ...", Scott Meyers gave a separate item for private inheritance: item 39: "Use private inheritance judiciously."
Let's briefly look at the head of that section showing not **is-a** but **has-a** property of private inheritance.

```
class Person {};
class Student:private Person {}; // private
void eat(const Person& p){}             // anyone can eat
void study(const Student& s){}          // only students study

int main()
{
       Person p;     // p is a Person
       Student s;    // s is a Student
       eat(p);              // fine, p is a Person
       eat(s);              // error! s isn't a Person
       return 0;
}
```

He explains, in contrast to public inheritance, compilers will generally **not** convert a derived class object (**Student**) into a base class object (**Person**) if the inheritance relationship between the classes is private. That's why the call to **eat()** fails for the object **s**.

With public inheritance, the public methods of the base class become public methods of the derived class. In other words, the derived class **inherits** the base-class **interface** (the interface is still visible to outside and can use it). This is the **is-a** relationship. But with the private inheritance, the public methods of the base class become private methods of the derived class, even if they were protected or public in the base class. So, the derived class **does not inherit** the base-class **interface**.

But we should be careful when we talk about private inheritance. Sometimes it is very confusing. The **inherit** does not mean **"own"**. Suppose, a parent gave a child a secret recipe for a candy under the condition of not releasing the recipe. The child can give variety of candies to other people but not the recipe. With private inheritance, the derived class does enjoy(implement) the inherited interface but does not own the method. Therefore, derived class cannot show the interface to outside world. The only thing that they can show off to the outside is the product whose inner secret workings are hidden.

A class **does inherit** the **implementation** with private inheritance.

In his new book, "Programming Principles and Practice Using C++", Stroustrup described the implementation and interface as follows: The **interface** is the part of the class's declaration that its users access directly. The **implementation** is the part of the class's declaration that its users access only indirectly through the interface.

Let's look at the following example:

```
#include <iostream>
using namespace std;
class Engine
{
 public:
       Engine(int nc){
               cylinder = nc;
       }

       void start() {
              cout << getCylinder() <<" cylinder engine started" <<
endl;
       };

       int getCylinder() {
              return cylinder;
       }

private:

       int cylinder;

 };



 class Car : private Engine
{    // Car has-a Engine
 public:
    Car(int nc = 4) : Engine(nc) { }
    void start() {
        cout << "car with " << Engine::getCylinder() <<
                  " cylinder engine started" << endl;
        Engine:: start();
    }
 };

int main( )
{
      Car c(8);
      c.start();
      return 0;
}
```

The output is:

```
car with 8 cylinder engine started
8 cylinder engine started
```

As we see from the example, the **Car** class winds up with an inherited **Engine** component such as **cylinder** and the **Car** method can use the Engine method, **getCylinder()**, internally to access the Engine component, **cylinder**.

In short, private inheritance **does** acquire the **implementation**, but **does not** acquire **interface**.

From Scott Meyers'book :

1. "Private inheritance is most likely to be a legitimate design strategy when you're dealing with two classes **not** related by **is-a** where one either needs access to the protected members of another or needs to redefine one or more its virtual functions."

2. "Private inheritance means **is-implemented-in-terms-of**. It's usually inferior to composition"

3. "If you make a class **D** privately inherit from a class **B**, you do so because you are interested in taking advantage of some of the features available in class **B**, not because there is any conceptual relationship between objects of types **B** and **D**."

4. "Private inheritance means nothing during software **design**, only during **software implementation**."

Composition

As you may know, the **private** inheritance is a variant of **composition**, **aggregation**, or **containment**. So, the **has-a** relationship can be achieved using **composition** as in the example below.

```
#include <iostream>

using namespace std;

class Engine
{
 public:
       Engine(int nc){
               cylinder = nc;
       }

       void start() {
              cout << getCylinder() <<" cylinder engine started" <<
endl;
       };

       int getCylinder() {
              return cylinder;
       }

private:
       int cylinder;
 };

 class Car
{
 public:
       Car(int n = 4): eng(n) { }

       void start() {
          cout << "car with " << eng.getCylinder() <<
                " cylinder engine started" << endl;
              eng.start();
       }
private:
       Engine eng;   // Car has-a Engine
 };

int main( )
{
       Car c(8);
```

```
        c.start();
        return 0;
}
```

This produced the same output as the example for private inheritance.

```
car with 8 cylinder engine started
8 cylinder engine started
```

Composition vs. Private Inheritance

So, what are the similarities and differences between **private inheritance** and **composition**?

1. **Similarities**

1. In both cases, there is exactly one **Engine** member object contained in every **Car** object.

2. In both cases the **Car** class has a **start()** method that calls the **start()** method on the contained **Engine** object.

3. In neither case can users (outsiders) convert a **Car*** to an **Engine***.

2. **differences**

1. The **composition** is needed if you want to contain several **Engine**s per **Car**.

2. The **private inheritance** can introduce unnecessary multiple inheritance.

3. The **private inheritance** allows members of **Car** to convert a **Car*** to an **Engine***.

4. The **private inheritance** allows access to the **protected** members of the base class.

5. The **private inheritance** allows **Car** to override **Engine**'s **virtual** functions.

Let's look at the following example which shows the transitions of designs from private inheritance to composition.

Private inheritance lets us inherit the functionality, but not the public interface of another class. In the following code, **Circle** does not expose any of the member functions of **Ellipse**.

```
class Circle : private Ellipse
{
public:
      Circle();
      explicit Circle(float r);

      void setRadius(float r);
      float getRadius() const;
};
```

But objects of type **Circle** cannot be passed to code that accepts an **Ellipse** because the **Ellipse** base type is not publicly accessible. If we really want to expose a public or protected method of **Ellipse** in **Circle**, then we can do this as in the example below:

```
class Circle : private Ellipse
{
public:
      Circle();
      explicit Circle(float r);

      using Ellipse::getMajorRadius;
      using Ellipse::getMinorRadius;

      void setRadius(float r);
      float getRadius() const;
};
```

However, preferred is to use composition. This simply means that instead of class A inheriting from B, A declares B as a private data member (has-a) or A declares a pointer or reference to B as a member variable (holds-a):

```
class Circle
{
public:
      Circle();
      explicit Circle(float r);

      void setRadius(float r);
      float getRadius() const;

private:
      Ellipse mEllipse:
};


void Circle::setRadius(float r)
{
      mEllipse.setMajorRadius(r);
      mEllipse.setMinorRadius(r);
}

float Circle::getRadius()const
{
      return mEllipse.getMajorRadius();
}
```

The interface for **Ellipse** is not exposed in the interface for **Circle**, however, **Circle** still builds upon the functionality of **Ellipse** by creating a private instance of **Ellipse**. Therefore, **composition** provides the functional equivalent of private inheritance.

Use **compostion** over **private inheritance** because inheritance produces a more tightly coupled design.

Which should I prefer: composition or private inheritance?

Given that we can achieve a **has-a** relationship either with composition or with private inheritance, which should we use? Use **composition** when you can, but use **private** inheritance when you have to.

When we look at the class declaration, we see explicitly named objects representing the contained classes, and our code can refer to these objects **by name**. While the private inheritance makes the relationship seem more abstract. Also, inheritance can raise problems of **multi-inheritance**.

In short, we're less likely to run into trouble if we use composition. Also, as we summarized in the previous section, composition allows us to include **several objects**.

On the other hand, private inheritance does offer features that composition can't. Suppose, for example, that a class has **protected** members. Such members are available to derived classes but not to the world. If we include such a class in another class by using composition, the new class becomes part of that world, not a derived class. So, it can't access protected members. But by using inheritance makes the new class a derived class, so it can access protected members.

Another case that gives private inheritance advantage is if we want to redefine **virtual functions**. This is a privilege awarded to a derived class but not to a containing class.

Summary

I'll use Scott Meyers' statements as a summary.
"Private inheritance means **is-implemented-in-terms-of**. If you make a class **D** privately inherit from a class **B**, you do so because you are interested in taking advantage of some of the features available in class **B**, not because there is any conceptual relationship between objects of types **B** and **D**. As such, private inheritance is purely an implementation technique. (That's why everything you inherit from a private base class becomes private in your class: it's all just implementation detail.)
....
private inheritance means that implementation **only** should be inherited; interface should be ignored. If **D** privately inherits from **B**, it means that **D** objects are implemented in terms of **B** objects, nothing more. Private inheritance means nothing during software **design**, only during software **implementation**."