Dynamic (late) binding and Virtual Function

An example without virtual function:

Sale Class

```
class Sale
{
public:
        Sale();
        Sale(double thePrice);
        double GetPrice() const;
        void SetPrice(double newPrice);
        double  Bill () const;
        double Savings (const Sale & other) const;
private:
        double price;
};

Sale::Sale() : price (0)
{}

Sale :: Sale(double thePrice)
{       price = thePrice;
}

double Sale::Bill()
{       return price;
}

double Sale::GetPrice{} const
{       return price;
}

void Sale::SetPrice(double newPrice)
{       price = newPrice;
}

double Sale::Savings(const Sale & other) const
{       return (Bill() – other.Bill());
}
```

DiscountSale Class

```
class DiscountSale : public Sale
{
public:
        DiscountSale();
```

```
        DiscountSale(double thePrice, double theDiscount);

        double GetDiscount() const;
        void    GetDiscount(double newDiscount);
        double  Bill()  const;
private:
        double discount;
};

DiscountSale::DiscountSale() : Sale(0), discount (0)
{}

DiscountSale::DiscountSale(double thePrice, double theDiscount)
                :Sale(thePrice), discount(theDiscount)
{}

double DiscountSale::GetDiscount() const
{       return discount;
}

void DiscountSale::SetDiscount(double newDiscount)
{       discount = newDiscount;
}

double DiscountSale::Bill() const
{       double fraction = discount / 100;
        return (1-fraction)*getPrice();
}
```

client program:

```
bool operator < (const Sale& first, const Sale & second);

int main()
{
        Sale            simple(10.00);
        DiscountSale    disc(11.00, 10);

        if (disc < simple)
        {
                cout << "Discounted item is cheaper";
                cout << "Saving is $" << simple.saving(discount) << endl;
        }
        else
                cout << "Discounted item is not cheaper."
}

bool operator < (const Sale& first, const Sale& second)
```

```
{
        return (first.Bill() < second.Bill())
}
```

program output:
        Discounted item is not cheaper.

Why?
For operator < : "disc" which is an object of DiscountedSale class, is cast back to "Sale" class during compile time – early binding (binding done at compile time)
As a result, the bill method associated with "first" is the "Bill" method defined for Sale class, not the Bill method for DiscountSale class.

How to make an object always use its own methods, e.g., not being bound by methods defined in base classes?
Answer: **virtual functions**

If a method is defined to be virtual, it is to tell the compiler, " I do not know how this function is implemented. Wait until it is used in a program, and then get the implementation from the object instance" **-- wait until run time to determine the implementation of a function – late binding (examples including overloaded function, function passed as parameter…)**

Change the above program to :

```
class Sale
{
public:
        Sale();
        Sale(double thePrice);
        double GetPrice() const;
        void SetPrice(double newPrice);
        virtual double  Bill () const;   // ← this is the only change necessary,
                                         /*  (a) if a method is defined to be virtual, then all
                                                 new definitions of the function in the derived class will
                                                 automatically be virtual
                                             (b) virtual modifier is not necessary in the
                                                 implementation file.  */
        double Savings (const Sale & other) const;
private:
        double price;
};

/* output:
Discounted item is cheaper
Savings is $0.10
*/
```

In most cases, what we need is to have the object of a derived class to use method defined for its own class. Why don't we make all methods virtual functions? Or make the compiler treat all member functions as virtual?

Answer: having "virtual function" introduces large overhead, e.g., more storage is needed, which slows down the program execution. So, C++ gives this flexibility to the programming to decide which functions should be made virtual. If a function does not need to take advantage of being virtual, then not making it virtual makes the program runs more efficiently.

Additional rules:

1. **constructors can not be virtual**
2. **destructor can, and should be virtual, ➔ guarantees that future descendants of the object can de-allocate themselves correctly.**
3. **virtual function's return type can not be overridden**