Object-Oriented Programming

Object-oriented programming is based on three fundamental concepts: data abstraction, inheritance and dynamic binding.

Inheritance and **danamic binding** make it easier to define new classes that are similar, but not identical, to other classes, and they make it easier for us to write programs that *can ignore the details of how those similar types differ*.

Overviews on OOP

The key ideas in **object-oriented programming** are:

- data abstraction: seperate interface from implementation
- inheritance: define classes that model the relationships among similar types
- dynamic binding: use objects while ignoring the details of how they differ

Typically there is a **base class** at the root of the hierarchy, from which the other classes inherit, directly or indirectly. These inheriting classes are known as **derived classes**. The base class defines those members that are common to the types in the hierarchy. Each derived class defines those members that are specific to the derived class itself.

An example:

```
// base class
class Quote{
    public:
        std::string isbn() const;
        virtual double net_price(std::size_t n) const;
};
```

A base class distinguishes functions that are type dependent from those that it expects its derived classes to inherit withour change. The base class define as `virtural` those functions it expects its derived classes to define for themselves.

A derived class must specify the classes from which it intends to inherit. It does so in a **class derivatino list**, which may have an optional access specifier.

```
// derived class
class Bulk_quote: public Quote{
    public:
        // 派生类必须在其内部必须对所有的虚函数进行声明
        double net_price(std::size_t) const override;// overiide 显示地说明了该函数是重新定义的基类的虚函数(virtual function)
};

**Dynamic Binding**
```

```c++
double print_(ostream &os,const Quote &item,size_t n){
    // 根据传入的形参的对象类型类决定调用 基类的虚函数 还是 派生类重定义的虚函数
    // 即Quote::net_price or Bulk_quote::net_price

    double ret=item.net_prince(n);
    os<<item.isbn() // calls Quote::isbn()
    <<n<<ret<<endl;
    return ret;
}
```

Here, the second paramter of function `print_` is `const Quote &item`, which is an reference to an object of base class `Quote`. We can pass the objects of base classes and the objects of derived class into this parameter. Because `net_price` is a virtual function, and because `print_` calls `net_price` through a reference, the versions of `net_price` that is run will depend on the type of the object that we pass to `print_`.

```
// basic has type Quote; bulk has type Bulk_quote
print_(cout,basic,20);// calls Quote version
print_(cout,bulk,20);// calls Bulk_quote version
```

> Because the decision as to which version to run depends on the type of the argument, that decision can't be made until run time. Therefore, dynamic binding is sometimes known as **run-time binding**.

Note: In c++, dynamic binding happens when a virtual function is called through a reference (or a pointer) to a base class.