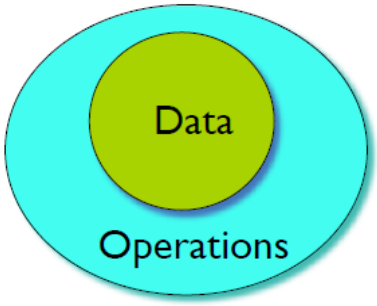
C++ Class

写一个类就写一个默认的构造器放在那里不管有用没用， `A() {}`.

Object-Oriented Programming with C++

Objects = Attributes + Services

- Data: the properties of status
- Operations: the functions



类规范由两部分组成：

- 类声明：以数据成员的方式描述数据部分，以成员函数(方法)的方式描述共有接口——蓝图；
- 类方法定义：描述如何实现类成员函数——细节。

Class declaration

```
// stock00.h -- Stock class interface
#ifndef STOCK00_H_
#define STOCK00_H_
#include <string>
using namespace std;

/* class declaration */
class Stock {
    private:
        string company;
        long shares;
        double share_val;
        double total_val;
        void set_tot() {total_val = share * share_val};
    public:
        void acquire(const string &co, long n, double per); // 获得股票
        void buy(long num, double price);                    // 增持
        void sell(long num, double price);                   // 卖出股票
        void update(double price);                           // 更新股票价格
        void show();                                         // 显示股票信息
};                  // Note semicolon at the end.
```

- 成员函数可以就地定义 (`set_tot()` 叫做**内联函数**)，也可以用原型表示 (其他函数).
- 访问控制：

    - 使用类的对象都可以**直接**访问**公共部分**，但**只能**通过**公有成员函数**(或友元函数)来访问**对象的私有成员**，公有成员函数是程序和对象的私有成员之间的桥梁——防止程序直接访问数据被称为数据隐藏；
    - **公有接口**表示设计的**抽象组件**，将实现**细节放在一起并和抽象分开**被称为**封装**：
        - **Encapsulation**: Bundle data and methods dealing with these data together in an object.
            - Hide the details of the data and the action.
            - Restrict only access to the publicized methods.
            - 数据隐藏是一种封装，类函数定义和类声明放在不同的文件中也是一种封装。
        - Abstract: the ability to ignore details of parts to focus attention on a higher level of a problem.
        - Modularization: the process of dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways.
- 数据项通常放在私有部分，组成类接口的成员函数放在公有部分，但是其实怎么样都是可以的。

Class function implementation

成员函数定义和常规函数相似都有函数头和函数体，有返回类型和参数，但也有其特殊性：

- 定义成员数时使用**作用域解析运算符** `::` 来标识函数所属的类；
- 类方法可以访问类的 `private` **组件。**

```
// stock00.cpp -- implementation the Stock class
#include <iostream>
#include <stock00.h>

void Stock::acquire(const string &co, long n, double per) {
    company = co;
    if ( n < 0 ) {
        std::cout << "Number of shares can't be nagetive;"
                  << company << " shares set to 0.\n";
        shares = 0;
    }
    else shares = n;
    share_val = pr;
    set_tot();
}

void Stock::buy(long num, double price) {
    ......
}
```

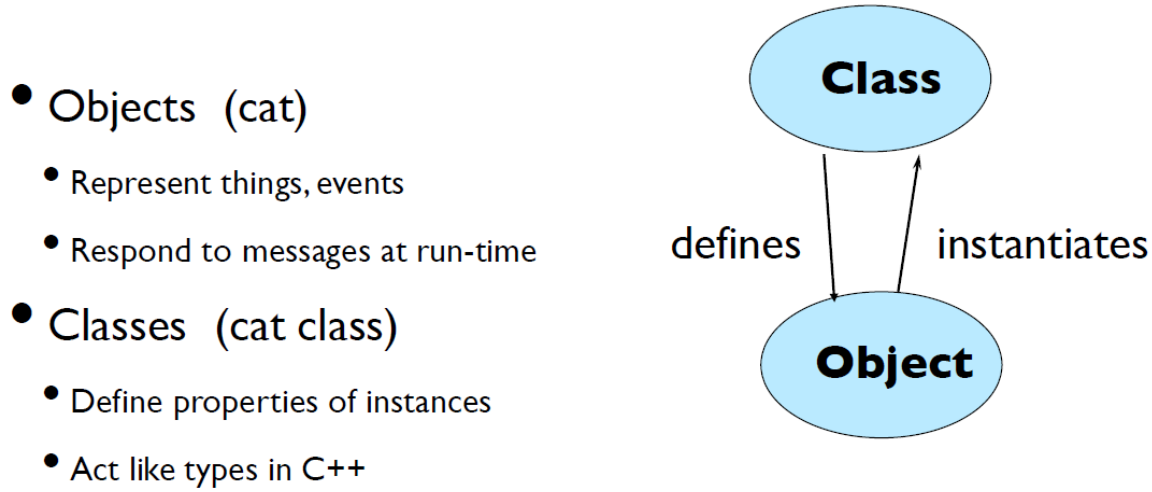
- 首先成员函数的函数头作用使用作用域解析运算符 `::` 来指出函数所属的类：

```
void Stock::update(double price);    // 我们定义的update()是Stock类的成员
void Buffoon::update()               // 可以将另一个类的成员函数也命名为update()
```

- 类方法的完整名称中包含类名：
    - `Stock::update()`：是函数的限定名**(qualified name)**；
    - `update()`：是全名的缩写，即非限定名**(unqualified name)**，只能在类作用域中使用。
- 类方法可以**访问类的私有数据成员**；

- **内联函数**（`eg. set_tot()`）：
  - 可以在类声明中直接定义，`void set_tot() {total_val = shares * share_val;}`；
  - 也可以在类声明之外定义，在 `private` 部分写 `void set_tot()`，然后在**方法实现**中使用 `inline` **限定符** `inline void Stock::set_tot() {...}`.
    - 内联函数要求在每一个使用它们的文件中都对其进行定义，所以我们经常讲其放在定义类的头文件中.
- 所创建的每一个类的**对象**都有**自己的存储空间**用于存储其内部变量和类成员，但同一个类的所有对象**共享同一组类方法**，即每种方法只有一个副本。

Object vs. Class



OOP Characteristics

1. Everything is an object.
2. A program is a bunch of objects telling each other what to do by sending messages.
3. Each object has its own memory made up of other objects.
4. Every object has a type.
5. All objects of a particular type can receive the same massages.

Definition of a Class

- In C++, separated `.h` and `.cpp` files are used to define one class.
- **Class declaration** and **prototypes** in that class are in the header **file** `.h`.

```
#ifndef HEADER_FLAG
#define HEADER_FLAG
// Type declaration here...
#endif                          // HEADER_FLAG
```

- All the **bodies of these functions** are in the source **file** `.cpp`.

Class Constructors and Destructors

Guaranteed initialization with the constructor

构造函数(constructor)：初始化类对象的数据成员，只要类的对象被创建就会**自动执行**构造函数。

- 构造函数被用来创建对象，而**不能通过对象来调用**；
- 构造函数的**名字**和**类名**相同，位于类声明的 `public` **部分**；
- 构造函数**没有返回类型**，有一个（可能为空的）参数列表和（可能为空的）函数体；
- 一个类可以包含多个构造函数，和其他重载函数差不多，不同的构造函数之间必须在参数数量或者参数类型上有区别；
- 不同于其他成员函数，**构造函数不能被声明成** `const` **的**，当我们创建类的一个 `const` 对象时，直到构造函数完成初始化过程，对象才能取得其"常量"属性。

```
class Stock {
    private:
        ...
    public:
        Stock(const string & co, long n, double pr);
        ...
}

Stock::Stock(const string & co, long n, double pr) {    // 参数名不能是类成员名.
    ......
}
```

- **使用构造函数：**
  - ```
    // Approach 1: call the constructor explicitly
    Stock food = Stock("World Cabbage", 250, 1.25);

    // Approach 2: call the constructor implicitly
    Stock food("World Cabbage", 250, 1.25);

    // Approach 3: use new
    Stock* pstock = new Stock("Electroshock Games", 18, 19.0);

    // Approach 4: use list initialization
    Stock s1 = {"Derivative Plus", 12, 0.1};
    Stock s2 = {"Hahaha"};      // 只要提供与某个构造函数参数列表匹配的内容.
    Stock s3 = {};
    ```
    - 在第三种情况下，虽然对象没有名字，但是可以使用指针来管理该对象。

- 默认构造函数：在**未提供显式初始值**时，用来创建对象的构造函数，它**没有参数**
  - **当且仅当没有**定义任何构造函数时，编译器才会提供默认构造函数；
    - 如果提供了**非默认**的构造函数，那就**必须**提供初始值：

      ```
      // 没有自己定义构造函数时：
      Stock stock1;           // 对的
      // 但是当定义了自己的构造函数 Stock::Stock(const string & co, ...)
      Stock stock2;           // 错的
      ```
  - 如果想要创建对象，而不显式地初始化，那么必须定义一个不接受任何参数的默认构造函数
      ```

- 方法一是给**已有的**构造函数的所有参数提供**默认值**：

```
Stock(const string & co="Error", int n=0, double pr=0.0);
```

- 方法二是通过**函数重载**来定义另一个构造函数——**一个没有参数的构造函数**：

```
Stock::Stock(){         // Default constructor
    company = "no name";
    shares = 0;
    share_val = 0.0;
    total_val = 0.0;
}
```

  - ```
    Stock first;    // calls it implicitly, 没有圆括号注意
    Stock first = Stock();          // calls it explicitly
    Stock* prelief = new Stock;     // calls it implicitly
    ```

The destructor

In C++, clean up is as important as initialization and is therefore guaranteed with the **destructor (析构函数)**.

- 析构函数的名称是在类名称前面加上 `~`；
- 特殊的是析构函数**没有参数**，返回值和声明类型是可有可无的；
- 如果构造函数使用 `new` 来分配内存，那么析构需要使用 `delete` 来释放这些内存；如果没有使用 `new`，那么析构函数可以不执行任何操作。

```
Stock::~Stock() {}      // No parameter.
```

- When is a destructor called: the destructor is called automatically by the compiler when the object goes out of scope.
  - 静态存储对象：析构函数将在程序结束时自动调用；(全局变量)
  - 自动存储对象：析构函数将在程序**执行完代码块(对象在其中定义)**自动被调用；(定义的变量)
  - `new` 创建对象：将驻留在栈内存或自由存储区，当使用 `delete` 释放内存的时候被调用。

Improving the `Stock` Class

```
// stock10.h -- Stock class declaration with constructor and destructor added
class Stock {
    private:
        std::string company;
        long shares;
        double share_val;
        double total_val;
        void set_tot() { total_val = shares * share_val; }
    public:             // two constructors
        Stock();        // default constructor
        Stock(const string &co, long n=0, double pr=0.0);
        ~Stock();       // destructor
        void buy(long num, double price);
        void sell(long num, double price);
        void update(double price);
```
```

```
        void show();
};
```

```
// // stock10.cpp -- Stock class with constructor and destructor added
#include <iostream>
#include "stock10.h"

Stock::Stock() {
    std::cout << "Default constructor called.\n";
    company = "no name";
    shares = 0;
    share_val = 0.0;
    total_val = 0.0;
}
Stock::Stock(const string &co, long n, double pr) {
    std::cout << "Constructor using " << co << " called.\n";
    company = co;
    ......
}
```

```
// usestock1.cpp -- using the Stock class
#include <iostream>
#include "stock10.h"
using namespace std;
int main () {
    Stock stock1("NanoSmart", 12, 20.0);            // explicitly call.
    stock1.show();
    Stock stock2 = Stock("Boffo Object", 2, 2.0);   // implicitly call.
    stock2.show();

    cout << "Assigning stock1 to stock2:\n";
    stock2 = stock1;
    cout << "Listing stock1 and stock2:\n";
    stock1.show();  stock2.show();

    cout << "Using a constructor to reset an object:\n";
    stock1 = Stock("Nifty Foods", 10, 50.0);        // temp objct.
    cout << "Revised stock1:\n";
    stock1.show();
}
```

- `stock2 = stock1` 说明可以将一个对象**赋值**给另一个**同类型**的对象，在默认情况下，给类对象复制时将把一个对象的成员赋值给另一个对象；
- `stock1 = Stock("Nifty Foods", 10, 50.0);` 说明构造函数不仅仅可用于初始化新对象，还可以创建一个**新的临时变量**，然后将其内容**赋值给已有的**对象 `stock1`，随后程序调用 `destructor` 来**删除**该临时变量：
    - 但是初始化的效率更加高一点。

Initializer list

```
class Point {
    private:
        const float x, y;
    public:
        Point(float xa, float ya) : y(ya), x(xa) {}
};
```

- Can initialize any type of data
  - pseudo-constructor calls for built-ins.
  - No need to perform assignment within body of ctor**(大括号中不用写任何东西)**
- Order of **initialization** is order of **declaration**
  - Not the order in the initializer list!
  - Destroyed in the reverse order.

Initialization vs. assignment

```
/* Initialization */
Student::Student(string s) : name(s) {} // before constructor body
```

```
/* Assignment */
Student::Student(string s) {name=s;}    // inside constructor body
                                // string must have a default constructor
```

Const member function and `this`

Const member function

```
const Stock land = Stock("Kruskal");    // It is a const object.
land.show();                            // Not allowed.
```

对象本身是 `const` 类型，所以不能保证调用 `show()` 不会改变对象的值，所以编译无法通过。

普通的成员函数不能 access 到 internals(private部分)，要确保函数不会修改调用对象——**declare member functions const: 将 `const` 关键词放在函数的括号后面**

```
// When declare
void show() const;
// When implementation
void Stock::show() const { ... }
```

- Repeat the const keyword in the definition as well as the declaration.
- Function members that do not modify data should be declared const.

对于在 `class` 声明里面的 `const`，比如说

```
class A { const int i; };
```

- It has to be initialized in initializer list of the constructor.
- ……

`this`: the hidden parameter

`this`: a hidden parameter for **all member functions**, with the type of the class:

```
void Point::print()    <==>    void Point::print(Point* this)
```

To call the function, you must specify a variable:

```
Point a;
a.print();      <==>      Point::Print(&a);
```

- Inside member functions, you can use `this` as the pointer to the variable that calls the function.
- `this` is a natural local variable of all class member functions that you **can not define**, but can **use it directly**.
- `this`**指针**：用来指向调用成员函数的对象（`this` 被作为隐藏参数传递给函数）.

想要程序知道类的某一个 `private` 成员数据，通常使用**内联函数**返回那一个值，如：

```
class Stock {
    private:
        ...
        double total_val;
    public:
        double total() const { return total_val; };
        ...
};
```

如果我们现在想比较两个 `Stock` 对象，并返回股价较高的那个对象的引用，做法如下：

- 要比较两个股票，必须将第二个对象作为参数传递给比较函数，可以**按引用来传递参数** `const Stock &`；
- 将比较结果传回给调用程序，可以让方法**返回一个引用**，这个引用指向股价高的那个对象；

  ```
  const Stock & topval(const Stock & s) const;    // 比较方法的原型
  ```

  - 该函数显式地访问一个对象 `s`，隐式地访问自己这个对象，并返回其中一个对象的引用；
  - 括号中的 `const` 表明该函数不会修改被显式访问的对象，括号后的 `const` 表明不会修改被隐式访问的对象——由于该函数返回两个 `const` 对象之一的引用，所以返回类型也是 `const` 引用。

    ```
    const Stock & Stock::topval(const Stock & s) const {
        if ( s.total() > this->total() ) return s;
        else return *this;      // this是指针，所以要解引用 * 得到对象
    }
    ```

An array of Object

声明对象数组的方法与声明标准类型数组相同，可以使用**列表初始化**来初始化对象数组(如果类包含多个构造函数就可以对不同的元素使用不同的构造函数)：

```
const int STKS = 10;
Stock stocks[STKS] = {
    Stock("NanoSmart", 12.5, 10),
    Stock(),
    Stock("Monolithic Obelisks", 130, 3.25),
};
```

- 由于该声明只初始化了数组的部分元素，因此余下的7个元素将使用默认构造函数进行初始化；
- 初始化对象数组：
  - 首先使用默认构造函数创建数组元素；
  - 然后花括号中的构造函数将创建临时对象；
  - 然后将临时对象的内容复制到相应的元素中。

Fields, parameters, local variables

- All three kinds of variable are able to store a value that is appropriate to their defined type.
- Fields are defined outside constructors and methods.
- Fields are used to store data that persists throughout the life of an object. As such, they maintain the current state of an object. They have a lifetime that lasts as long as their object lasts.
- Fields have class scope: their accessibility extends throughout the whole class, and so they can be used within any of the constructors or methods of the class in which they are defined.

Class scope

- 类中定义的名称(包括数据成员名和类成员函数名)的作用域都是整个类，作用域为整个类的名称只在该类中是已知的，在类外是不可知的；
- 只有在类声明或成员函数定义中才可以直接使用未修饰的成员名称(未限定)；在其他情况下使用类成员名时，必须根据情况使用 `.` 或 `->` 或 `::`；

Class Scope Constants

创建一个由所有对象共享的常量，就要求使符号常量的作用域为类，有两种方式

1. 在类中声明一个枚举，在**类声明**中声明的**枚举**的作用域为**整个类**：

```
class Bakery {
    private:
        enum {Months = 12};
        double costs[Months];
        ...
};
```

   - [x] 这种方式声明枚举并不会创建类数据成员，即所有对象都不包含枚举；`Months` 只是一个符号名称，在作用域为整个类的代码中碰到编译器将会用 12 替换它；

- [x] 这里使用枚举只是为了创建符号常量，并不打算创建枚举类型变量，所以**不用枚举名。**

2. 使用关键字 `static`：

```
private:
    static const int Months = 12;
    double costs[Months];
    ...
```

- [x] 这个常量将与**其他静态变量存储在一起**，而不是存储在对象中，所以只有一个 `Months` 常量被所有对象共享。

- [x]
```
class HasArray {
  const int size;
  int array[size]; // ERROR!
  ...
};
```

- Make the const value static:

```
- static const int size = 100;
- static indicates only one per class (not one per object)
```

- Or use "anonymous enum" hack :

```
class HasArray{
    enum { size = 100 };
    int array[size];  // OK!
    …
}
```

```
private:
    const int Months = 12;      // false
    double costs[Months];
```

直接这样是错的，因为声明类只是描述了对象的形式，并没有创建对象，因此在创建对象之前没有存储值的空间。

Scoped Enumerations (C++11)

C++11提供了一种新枚举，其枚举量的作用域为**类 `class`(也可以是 `struct`)**：

```
// declaration
enum class egg {Small, Medium, Large, Jumbo};
enum class t_shirt {Small, Medium, Large, Xlarge};
// usage
egg choice = egg::Large;        // the Large enumerator of the egg enum
t_shirt Floyd = t_shirt::Large; // the Large enumerator of the t_shirt enum
```

- 枚举量的作用域为类之后，不同枚举量定义中的枚举量就不会发生名称冲突了；
- 提高了作用域内枚举的类型安全，有些情况下常规枚举将自动转换为整型，如将其赋值给 `int` 变量；

---

c++中允许在结构体当中定义函数，它的用法和类的用法很像，不过与类有一个区别在于：

- `struct` 中定义的函数和变量都是默认为 `public` 的，但 `class` 中的则是默认为 `private`.