

C/C++ Programming Language

CS219 Spring

Feng Zheng

Lecture 9



南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY



Content

- Brief Review
- Objects and classes
 - Two programming styles
 - Classes in C++
 - Access control
 - Function implementations
 - Constructors and destructors
 - This pointer
 - Class scope
- Summary

Brief Review



Review

- A header file
- Header File Management (**guarding scheme**)
- Scope and Linkage
- 1. Automatic Storage Duration
- 2. Static Duration Variables:
 - External, Internal and No Linkage
 - Specifiers and Qualifiers
 - Functions Linkage
- 3. Storage Schemes and Dynamic Allocation



Objects and Classes



Procedural and Object-Oriented Programming

- Procedural Programming
 - Firstly concentrate on the **procedures** you will follow
 - Then think about how to represent the **data**
 - Put the **functions** together
- Object-Oriented Programming
 - Begin by thinking about the **data**
 - ✓ Concentrate on the **object** as the user perceives it
 - ✓ Describe the object and the **operations** that will describe the user's interaction with the data
 - ✓ Decide how to implement the **interface** and data **storage**
 - Put together a program to use your new design



What Is a Type?

- Specifying a basic type does three things
 - It determines **how much memory** is needed for a data object
 - It determines how the **bits** in memory are **interpreted** (**long** vs. **float**)
 - It determines what **operations**, or methods, can be performed using the data object (integer vs. pointer)
- For built-in types
 - The information about operations is built in to the **compiler**
- For user-defined types in C++
 - Have to provide the same kind of information **yourself**



Classes in C++

- A class is a C++ vehicle for translating an **abstraction** to a **user-defined type**
 - Include **data** representation
 - Include **methods** for manipulating that data
- A class specification has two parts
 - A **class declaration**, which describes the **data** component, in terms of data members, and the public **interface**, in terms of member functions, termed methods
 - The **class method definitions**, which describe how certain class member functions are implemented



Access Control

- Describe access control for class members
 - Any program that uses an object of a particular class can **access the public portions** directly
 - A program can access the **private** members of an object only by **using the public member functions**

keyword `private` identifies class members that can be accessed only through the public member functions (data hiding)

keyword `class` identifies class definition

the class name becomes the name of this user-defined type

class members can be data types or functions

```
class Stock
{
private:
    char company[30];
    int shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }
public:
    void acquire(const char * co, int n, double pr);
    void buy(int num, double price);
    void sell(int num, double price);
    void update(double price);
    void show();
};
```

keyword `public` identifies class members that constitute the public interface for the class (abstraction)



Components

- **Abstraction component:** the public interface
- **Encapsulation component:** gather the implementation details and separate them from the abstraction
 - Data hiding: **insulation** of data from direct access by a program is called
 - Data hiding is an instance of encapsulation
 - ✓ **Prevent** you from **accessing** data directly
 - ✓ **Absolve** you from **needing to know** how the data is represented
 - ✓ By default, the members are **private** (in structure type: public by default)



Implementing Class Member Functions

- Provide **code** for those **member functions** represented by a prototype in the class declaration
 - Use the **scope-resolution operator** (`::`) to identify the class to which the function belongs
 - Access the **private** components of the class
 - Has **class scope** (the same name for multi-class)

- Inline function:

- Any function with a definition in the class **declaration** automatically
- Define a member function **outside** the class declaration and still make it inline

```
class Stock
{
private:
    ...
    void set_tot(); // definition kept separate
public:
    ...
};

inline void Stock::set_tot() // use inline in definition
{
    total_val = shares * share_val;
}
```



Which **Object** Does a Method Use?

- Contain **storage** for **its own** internal variables, the class members
- But all objects of the same class share the same set of class methods, with just **one copy of each method**
- **Questions?**
 - What about a **static** member of a class?
 - What about a **static** member function of a class?
 - What about a **static** variable for member functions of a class?

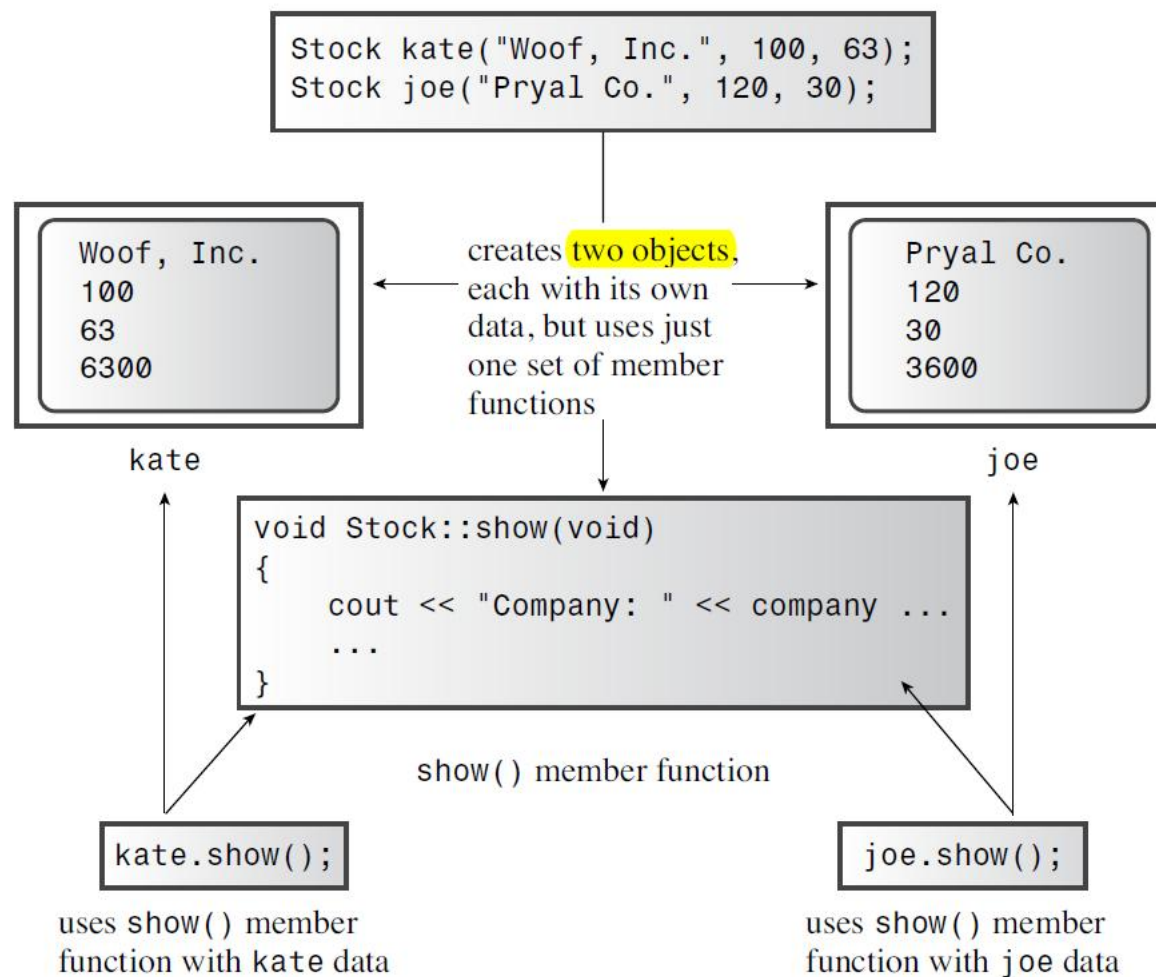


Using Classes

- A program example:
create and use objects
of a class

- Run

- stock00.h
- stock00.cpp
- usestock0.cpp





Reviewing Our Story to Date

- Specify a **class design** is to provide a class declaration
- Specify a class design is to implement the class member **functions**
- Create an **object**, which is a particular example of a class



Constructor Declaration and Definition

- A program **automatically invokes** the constructor when it declares an object
 - Have **NO** return value and has **NO** declared type
 - Avoid **confusion**: between member variables and arguments
- Using constructors
 - Call the constructor **explicitly**

```
Stock food = Stock("World Cabbage", 250, 1.25);
```
 - Call the constructor **implicitly**

```
Stock garment("Furry Mason", 50, 2.5);
```
 - Constructors are used **differently** from the other class methods



Default Constructors

- Do you remember the default functions?
- Create an object when you **don't provide explicit** initialization values
 - Have the members been initialized?
- How to provide default constructors
 - One is to provide **default** values for all the arguments to the existing constructor
 - The second is to use **function overloading** to define a second constructor, one that has no arguments
 - You can have **only one** default constructor



Destructors

- **When** program expires, **when** the program exits the block of code in which an object is defined or **when** you use `delete` to free the memory dynamically allocated for an object
 - Destructor: a **special member function** is called
 - **Clean** up all variables
 - Use **new** to create variables in constructor and use **delete** to free them
- Destructor form
 - Be formed from the **class name** preceded by a tilde (~)
 - Have **NO** return value and has **NO** declared type
 - **Must** have **NO** arguments
- Run `stock01.h`, `stock01.cpp`, `usestock01.cpp`



Initialization and const

- C++11 list initialization (followed **program 2**)

```
Stock hot_tip = {"Derivatives Plus Plus", 100, 45.0};  
Stock jock {"Sport Age Storage, Inc"};  
Stock temp {};
```

- **const** member functions

➤ A function promises **NOT** to modify the **invoking object**

```
void show() const;           // promises not to change invoking object  
  
void stock::show() const    // promises not to change invoking object
```

```
#define VALUE 100
```

```
const int value = 100;
```

```
const int * p_int;  
int const * p_int;
```

```
int * const p_int;
```

```
void func(const int *);  
void func(const int &);
```



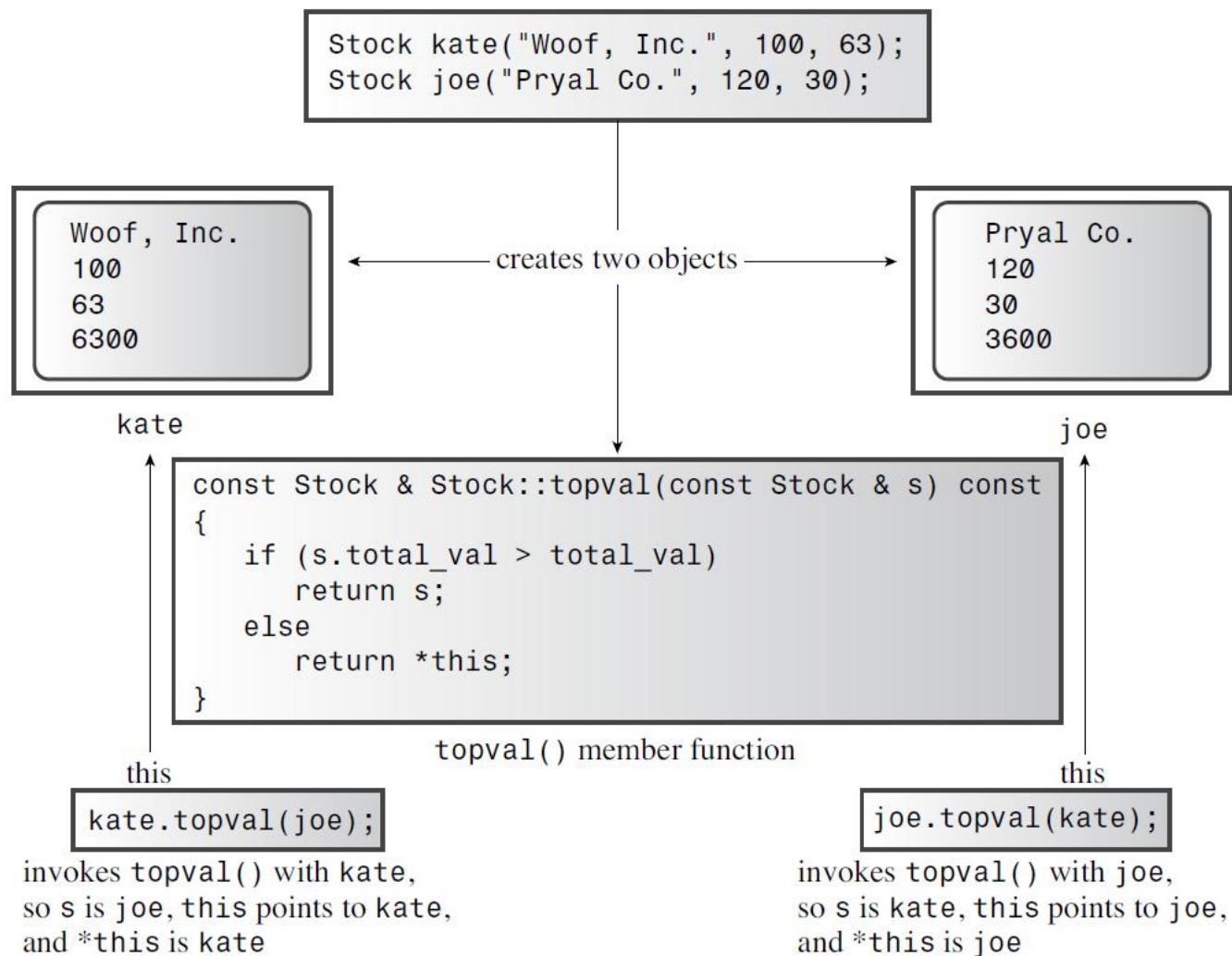
Review of Constructors and Destructors

- Constructor (**special** member function of the **same name**)
 - Have **more than one constructor** with the same name, provided that each has its **own** signature
 - Have **NO declared** type
 - Have **NO arguments** for a **default** constructor
- Destructor (**special** member function of the name preceded by a **tilde**)
 - Invoke a destructor when an object is **destroyed**
 - You can have **only one** destructor per class
 - Have **no** return type (not even void) and **no** arguments
 - Use delete become necessary when class constructors use new



Knowing Your Objects: The **this** Pointer

- The **this** pointer points to the **object** used to invoke a member function
- In **general**, all class methods have a **this pointer** set to the address of the object that invokes the method
 - **Why** is it general and **what** is special?





An Array of Objects

- Create **several objects** of the **same class**
 - Declare an array of objects **the same way** you declare an array of any of the standard types
 - ✓ Either: the class explicitly defines **no constructors** at all, in which case the implicit do-nothing default constructor is used
 - ✓ Or: an **explicit default constructor** be defined
 - ✓ More: use a **constructor** to initialize the array elements

- Run `stock02.h`, `stock02.cpp`,
`usestock02.cpp`

```
const int STKS = 4;
Stock stocks[STKS] = {
    Stock("NanoSmart", 12.5, 20),
    Stock("Boffo Objects", 200, 2.0),
    Stock("Monolithic Obelisks", 130, 3.25),
    Stock("Fleep Enterprises", 60, 6.5)
};
```



Class Scope

- Review scope
 - **Global** (or file) scope
 - **Local** (or block) scope
 - **Function** names can have **global** scope but they never have local scope
- **Class scope** applies to names defined **in a class**
 - The names of class **data** members
 - Class member **functions**
 - **Can't** directly access members of a class from the **outside** world
 - ✓ Direct membership operator (.)
 - ✓ Indirect membership operator (->)
 - ✓ Scope-resolution operator (::)



Class Scope Constants

- Problem: until you create an object, there's **no place** to store a value

- A **symbolic constant**: declare an enumeration within a class
- A constant within a class—using the keyword **static**

1. Constants: symbol and literal
2. Defining Constants: macros
3. Enumerated Constants

```
class Bakery
{
private:
    const int Months = 12;    // declare a constant? FAILS
    double costs[Months];
    ...
}
```

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

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




Scoped Enumerations (C++11)

- Problem: enumerators from two different **enum** definitions can conflict
- Have **class scope** for its enumerators

```
enum egg {Small, Medium, Large, Jumbo};  
enum t_shirt {Small, Medium, Large, Xlarge};
```

```
enum class egg {Small, Medium, Large, Jumbo};  
enum class t_shirt {Small, Medium, Large, Xlarge};
```

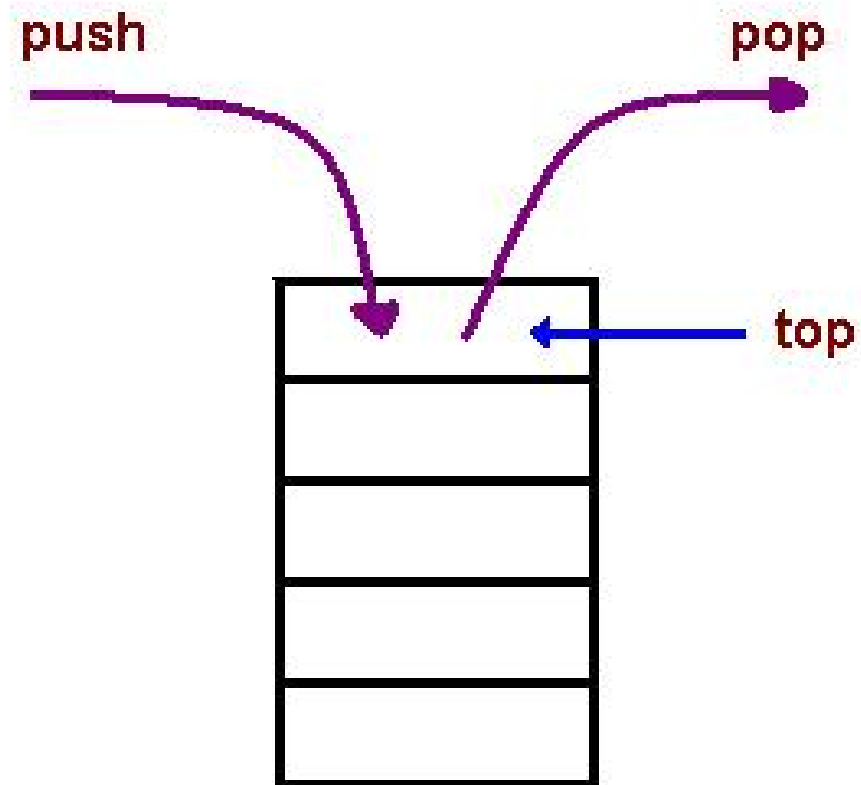
```
enum egg_old {Small, Medium, Large, Jumbo};           // unscoped  
enum class t_shirt {Small, Medium, Large, Xlarge};    // scoped  
egg_old one = Medium;                                // unscoped  
t_shirt rolf = t_shirt::Large;                        // scoped  
int king = one;   // implicit type conversion for unscoped  
int ring = rolf;  // not allowed, no implicit type conversion  
if (king < Jumbo) // allowed  
    std::cout << "Jumbo converted to int before comparison.\n";  
if (king < t_shirt::Medium) // not allowed  
    std::cout << "Not allowed: < not defined for scoped enum.\n";
```

*But you can do an explicit
type conversion*



Abstract Data Types

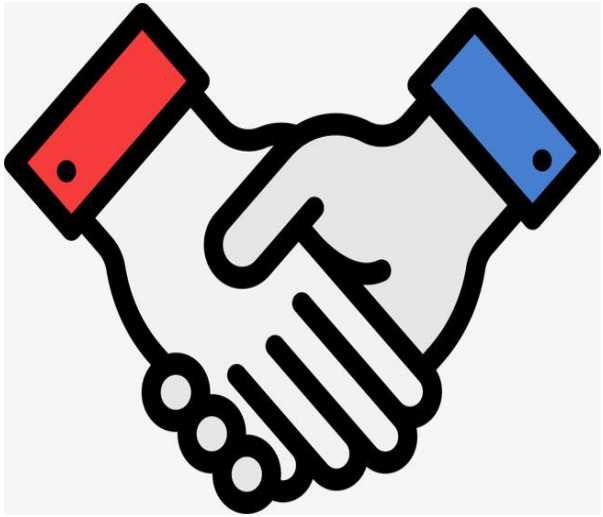
- An example: stack
 - Create an **empty** stack
 - Add an item to the **top** of a stack
 - Remove an item from the **top**
 - Check whether the stack is **full**
 - Check whether the stack is **empty**
- Run `stack.h`, `stack.cpp`, `stacker.cpp`





Summary

- Objects and classes
 - Two programming styles
 - Classes in C++
 - Access control
 - Function implementations
 - Constructors and destructors
 - this pointer
 - Class scope
 - Abstract data type: stack



Thanks



zhengf@sustech.edu.cn