

# 面向对象程序设计

## 课程简介

2022 年 秋

耿楠

计算机科学系  
信息工程学院

西北农林科技大学  
NORTHWEST A&F UNIVERSITY

中国 · 杨凌



## ▶ 预修课程:

- ▶ C 语言程序设计 (无法回避的**指针**)
- ▶ 高等数学 (永远的**根**)
- ▶ 英语 (要求不高, 但你得**不断用**)





### ► 预修课程:

- ▶ C 语言程序设计 (无法回避的指针)
- ▶ 高等数学 (永远的根)
- ▶ 英语 (要求不高, 但你得不断用)

### ► 教材:

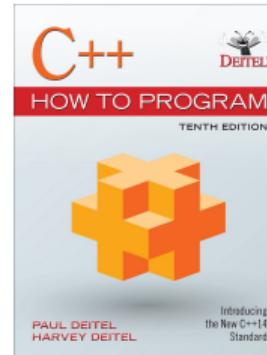
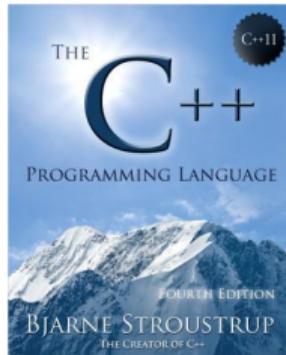
- ▶ C++ 面向对象程序设计 (第 2 版). 龚晓庆, 付丽娜, 朱新懿, 李康著. 北京: 清华大学出版社. 2011.





## ▶ 参考资料:

- ▶ The C++ Programming Language, 4th Edition. Stroustrup, Bjarne. Addison-Wesley. 2013.
- ▶ C++ How to Program, 10th Edition. Paul Deitel, Harvey Deitel. Prentice Hall. 2016.
- ▶ Programming Abstractions in C++. Eric Roberts. Prentice Hall. 2013.





## ► 考勤 (扣分制度)

- ▶ 课堂随机考勤
- ▶ 实习课随机考勤

## ► 成绩评定

- ▶ 结业 (期末) 考试—70%
  - ▶ 笔试 (闭卷, 拟定于 XX 周)
- ▶ 平时成绩—30%
  - ▶ 考勤
  - ▶ 在线评阅系统成绩 (<http://202.117.179.201/contest.php?cid=1047>)
  - ▶ 大作业 (论文分析或采用面向对象方法开发一个实用小软件)



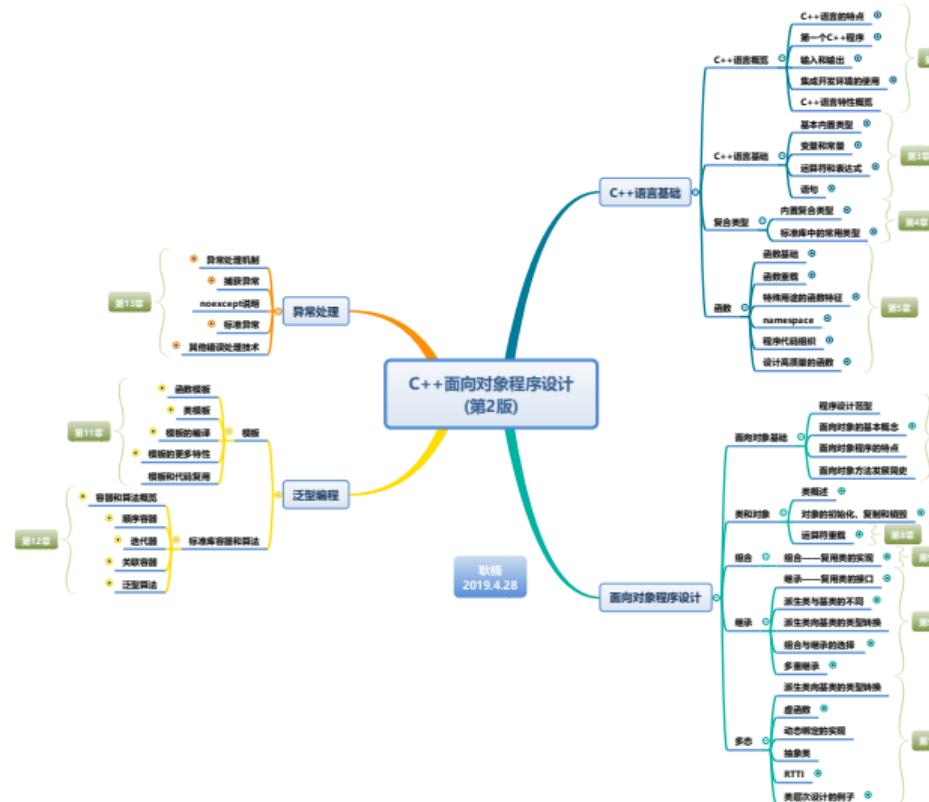


- ▶ 第 1 章 面向对象基础
- ▶ 第 2 章 C++ 语言概览
- ▶ 第 3 章 C++ 语言基础
- ▶ 第 4 章 复合类型
- ▶ 第 5 章 函数
- ▶ 第 6 章 类和对象
- ▶ 第 7 章 对象的初始化、复制和销毁
- ▶ 第 8 章 运算符重载
- ▶ 第 9 章 组合与继承
- ▶ 第 10 章 虚函数与多态性
- ▶ 第 11 章 模板与泛型编程
- ▶ 第 12 章 标准库容器和算法
- ▶ 第 13 章 异常处理



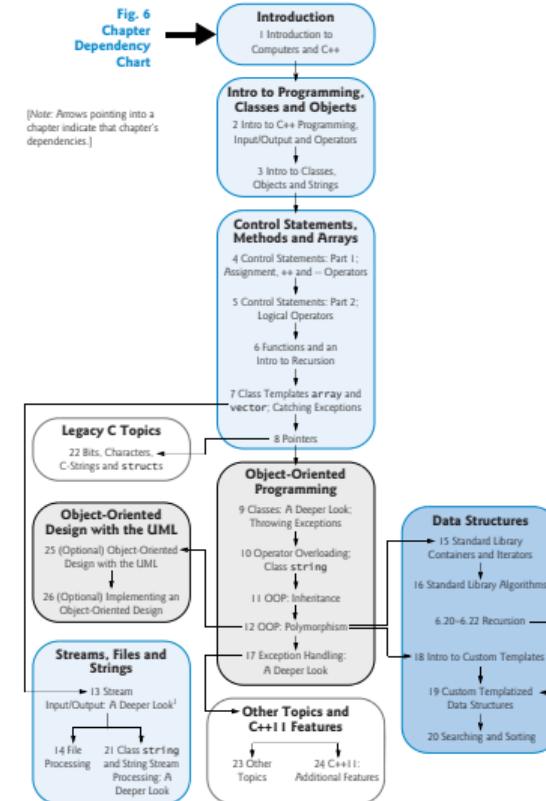


## ▶ 思维导图





## ▶ 学习路线





# 耿楠

教授

---

- 🌐 <http://cie.nwsuaf.edu.cn/>
  - 💻 信息工程学院
  - ✉ 1234567890@nwafu.edu.cn
  - ⌚ <https://github.com/registor/>
- 





## ▶ 预修课程:

- ▶ C 语言程序设计 (无法回避的**指针**)
- ▶ 高等数学 (永远的**根**)
- ▶ 英语 (要求不高, 但你得**不断用**)





### ▶ 预修课程:

- ▶ C 语言程序设计 (无法回避的指针)
- ▶ 高等数学 (永远的根)
- ▶ 英语 (要求不高, 但你得不断用)

### ▶ 教材:

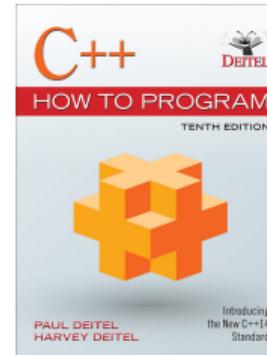
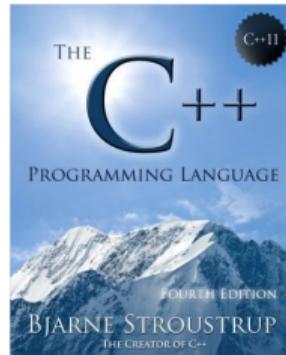
- ▶ C++ 面向对象程序设计 (第 2 版). 龚晓庆, 付丽娜, 朱新懿, 李康著. 北京: 清华大学出版社. 2011.





## ▶ 参考资料:

- ▶ The C++ Programming Language, 4th Edition. Stroustrup, Bjarne. Addison-Wesley. 2013.
- ▶ C++ How to Program, 10th Edition. Paul Deitel, Harvey Deitel. Prentice Hall. 2016.
- ▶ Programming Abstractions in C++. Eric Roberts. Prentice Hall. 2013.





## ► 考勤 (扣分制度)

- ▶ 课堂随机考勤
- ▶ 实习课随机考勤

## ► 成绩评定

- ▶ 结业 (期末) 考试—70%
  - ▶ 笔试 (闭卷, 拟定于 XX 周)
- ▶ 平时成绩—30%
  - ▶ 考勤
  - ▶ 在线评阅系统成绩 (<http://202.117.179.201/contest.php?cid=1047>)
  - ▶ 大作业 (论文分析或采用面向对象方法开发一个实用小软件)



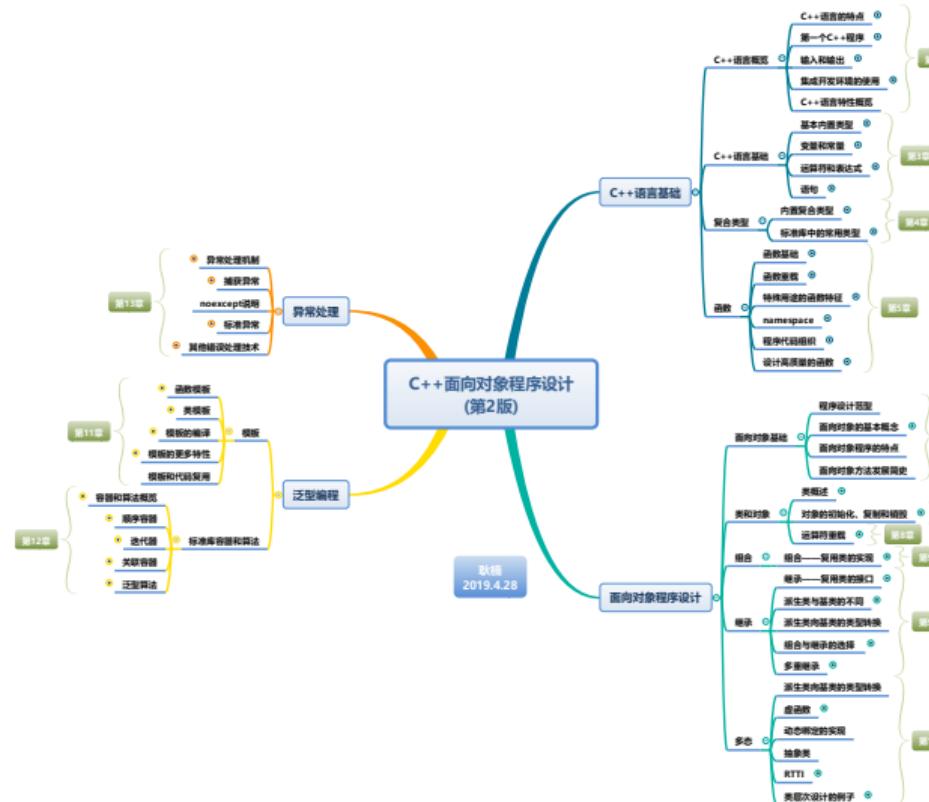


- ▶ 第 1 章 面向对象基础
- ▶ 第 2 章 C++ 语言概览
- ▶ 第 3 章 C++ 语言基础
- ▶ 第 4 章 复合类型
- ▶ 第 5 章 函数
- ▶ 第 6 章 类和对象
- ▶ 第 7 章 对象的初始化、复制和销毁
- ▶ 第 8 章 运算符重载
- ▶ 第 9 章 组合与继承
- ▶ 第 10 章 虚函数与多态性
- ▶ 第 11 章 模板与泛型编程
- ▶ 第 12 章 标准库容器和算法
- ▶ 第 13 章 异常处理



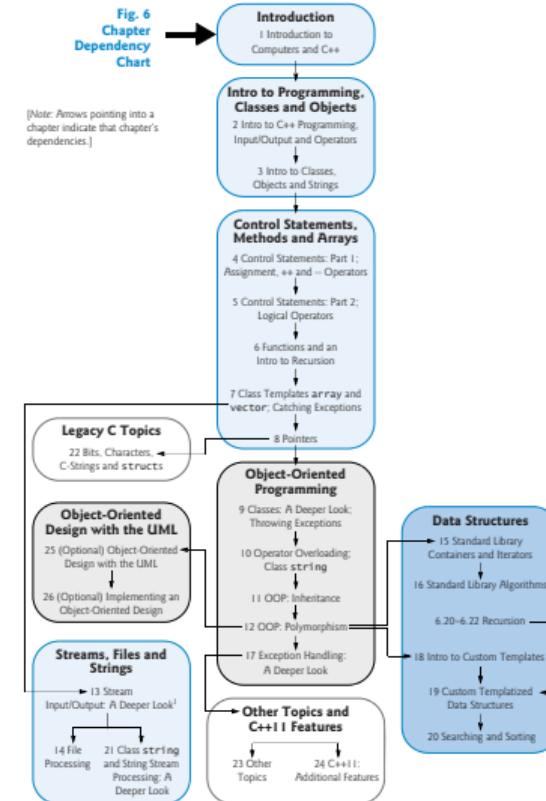


## ▶ 思维导图





## ▶ 学习路线





# 耿楠

教授

---

- 🌐 <http://cie.nwsuaf.edu.cn/>
  - 💻 信息工程学院
  - ✉ 1234567890@nwafu.edu.cn
  - ⌚ <https://github.com/registor/>
- 





## ► 对象

- ▶ 目标
- ▶ 恋爱的对象
- ▶ 描写或写实的人或物

## ► Object

- ▶ Something perceptible by one or more of the senses, especially by vision or touch; a material thing
- ▶ The purpose or goal of a specific action
- ▶ .....





### ▶ 对象（现实世界）

- ▶ 现实世界中的某个具体的事物（实物或实体）
- ▶ 一个封装了属性(attribute)与行为(behavior)的实体

### ▶ 对象（计算机世界）—**数据结构**

- ▶ 数据域
- ▶ 方法
- ▶ 相互作用关系





### ▶ 属性

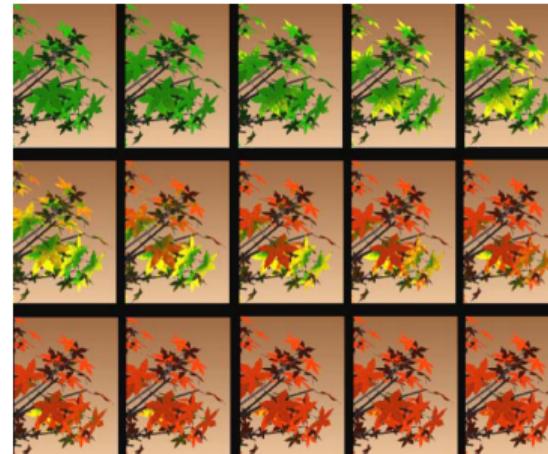
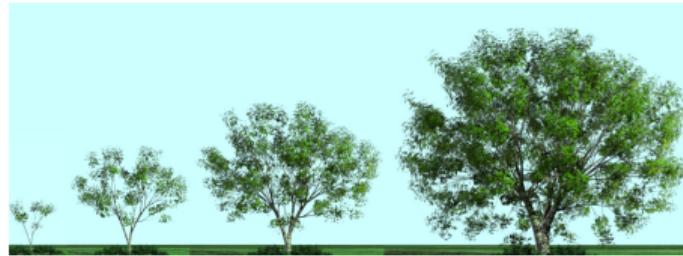
- ▶ 树干树枝、树叶、树龄、纹理、荷尔蒙.....





### ▶ 方法 (行为、操作)

- ▶ 生长、四季变化、修剪、运动、叶枝摩擦、声音……



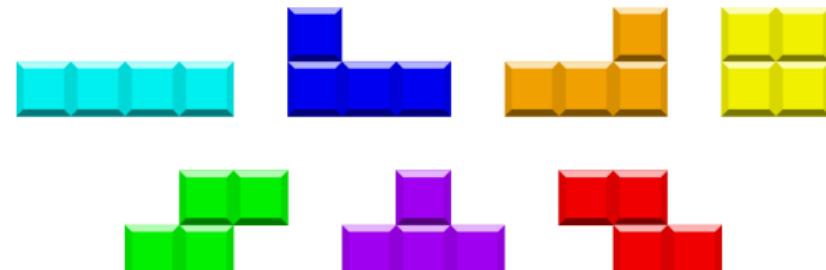
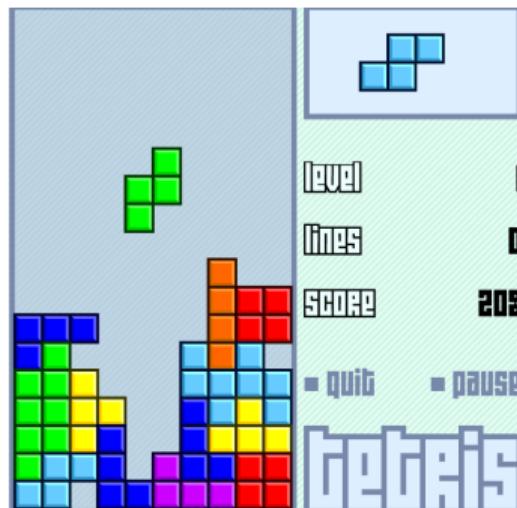


### ▶ 属性

- ▶ 颜色，四个方块的布局，大小，位置，速度，.....

### ▶ 行为

- ▶ 平移，旋转，加速，碰撞检测，显示，.....





### ▶ 类

```
CTetromino{  
    属性: 颜色, 四个方块的布局, 大小, 位置, 速度, ...  
    行为: 平移, 旋转, 加速, 碰撞检测, 显示, ...  
};
```

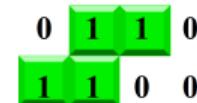
### ▶ 类对象

```
CTetromino I, J, L, O, S, T, Z;
```

### ▶ 操作对象

- ▶ 向对象发送消息
- ▶ 通过调用对象的操作实现

```
CTetromino S(绿色, 01101100);  
S. 平移 (-1, 0);
```





### ▶ 类

```
CTetromino{  
    属性: 颜色, 四个方块的布局, 大小, 位置, 速度, ...  
    行为: 平移, 旋转, 加速, 碰撞检测, 显示, ...  
};
```

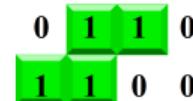
### ▶ 类对象

```
CTetromino I, J, L, O, S, T, Z;
```

### ▶ 操作对象

- ▶ 向对象发送消息
- ▶ 通过调用对象的操作实现

```
CTetromino S(绿色, 01101100);  
S. 平移 (-1, 0);
```

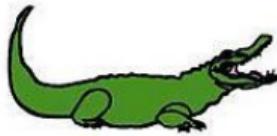
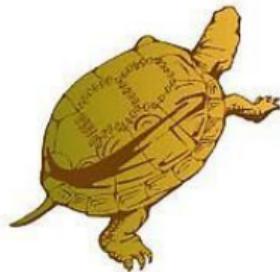




### ▶ 抽象

- ▶ 抽取共性
- ▶ 问题空间的事物 ⇒ 解空间的面向对象的概念

### ▶ 示例 1：两栖动物



### ▶ 示例 2：斑马





- ▶ 对象的属性和服务结合成一个独立的系统单位
- ▶ 隐蔽对象的内部细节
- ▶ 保留有限的对外接口
- ▶ 代码安全性
- ▶ 示例：





### ► 类 (三栏矩形)

- ▶ 类名
- ▶ 属性
- ▶ 操作
- ▶ 对外可见性
  - ▶ “+”—公有
  - ▶ “-”—私有

类名
- 属性 1
- 属性 2
+ 操作 1()
+ 操作 2()

### ► 对象 (矩形)

- ▶ 对象名称
- ▶ 类型
- ▶ 属性值

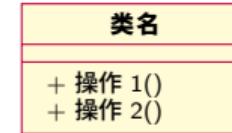
Circle
- radius : double
- center_x : int
- center_y : int
+ area() : double
+ perimeter() : double
+ move(in newx : int, in newy : int) : void
+ scale(in factor : double) : void





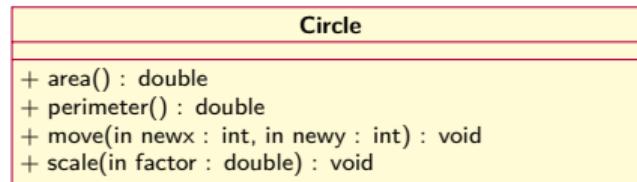
### ▶ 类 (三栏矩形)

- ▶ 类名
- ▶ 属性
- ▶ 操作
- ▶ 对外可见性
  - ▶ “+”—公有
  - ▶ “-”—私有



### ▶ 对象 (矩形)

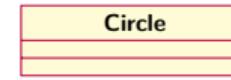
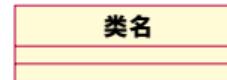
- ▶ 对象名称
- ▶ 类型
- ▶ 属性值





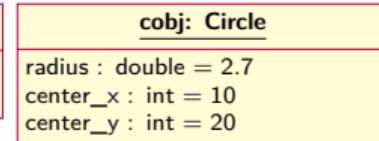
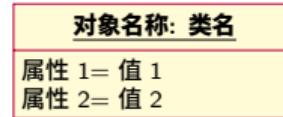
### ▶ 类 (三栏矩形)

- ▶ 类名
- ▶ 属性
- ▶ 操作
- ▶ 对外可见性
  - ▶ “+”—公有
  - ▶ “-”—私有



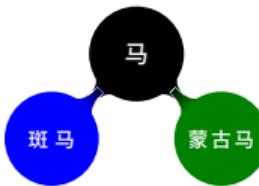
### ▶ 对象 (矩形)

- ▶ 对象名称
- ▶ 类型
- ▶ 属性值





- ▶ 一般类 ⇒ 特殊类 (具备全部属性与行为)
- ▶ 代码重用
- ▶ 示例 1:





### ► 示例 2(伪代码):

```
Animal 类 {
    Animal 类 (const string& name) : name(name) {}
    virtual string talk() = 0;
    const string name;
};

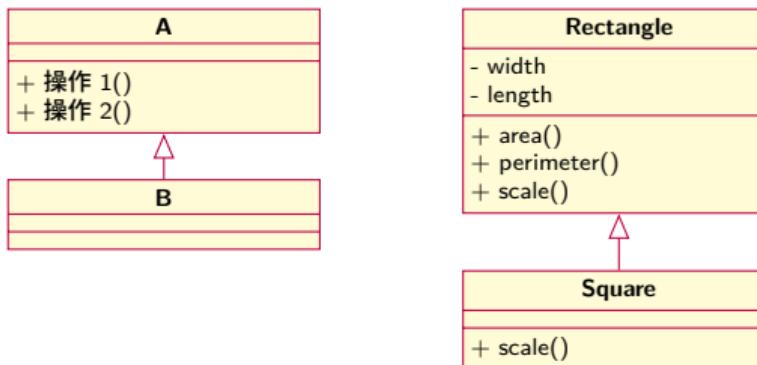
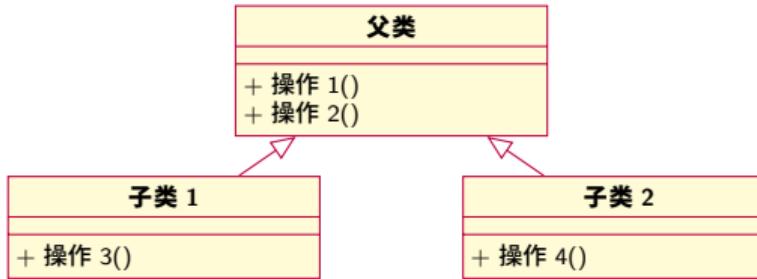
Cat类 : Animal 类 {
    Cat 类 (const string& name): Animal 类 (name) {}
    string talk() { return "喵喵!"; }
};

Dog类 : Animal 类 {
    Dog 类 (const string& name): Animal 类 (name) {}
    string talk() { return "汪汪!"; }
};
```



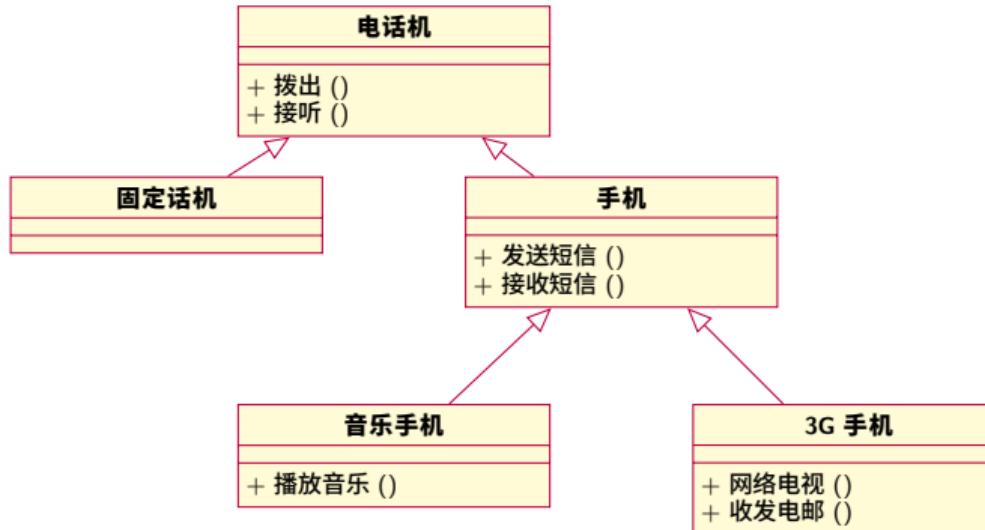


### ▶ 类的继承





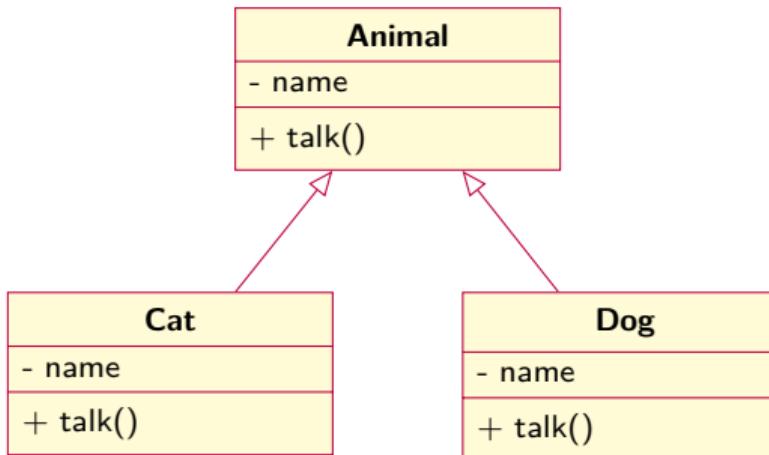
### ▶ 类的继承





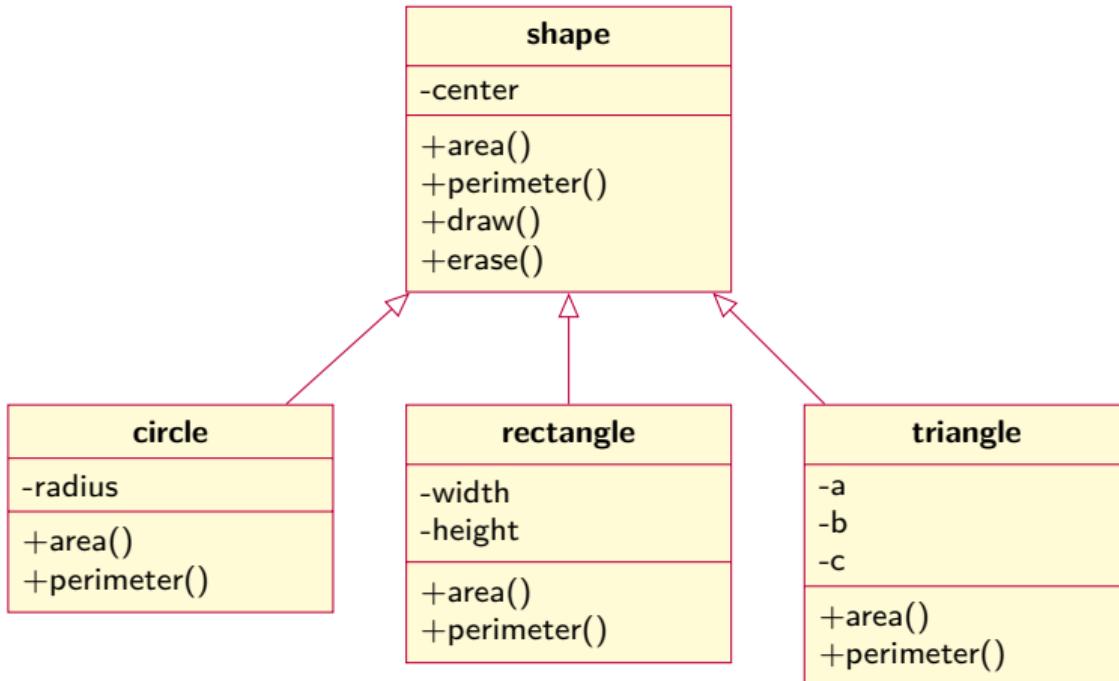
- 在一般类中定义的属性或行为，被特殊类继承之后，可以具有不同的数据类型或表现出不同的行为

```
Animal 类 *animalCat = new Cat 类 ("爱丽丝");  
Animal 类 *animalDog = new Dog 类 ("旺财");  
  
cout << animalCat->name << ":" << animalCat->talk() << endl;  
cout << animalDog->name << ":" << animalDog->talk() << endl;
```





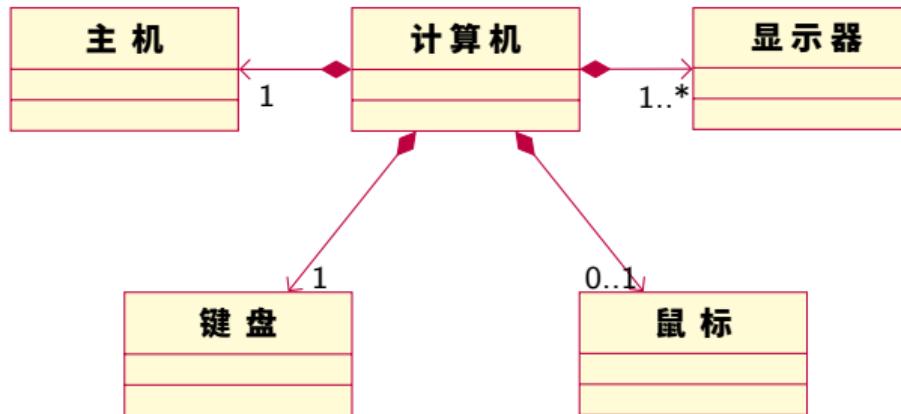
### ▶ 形状类





### ▶ 组合

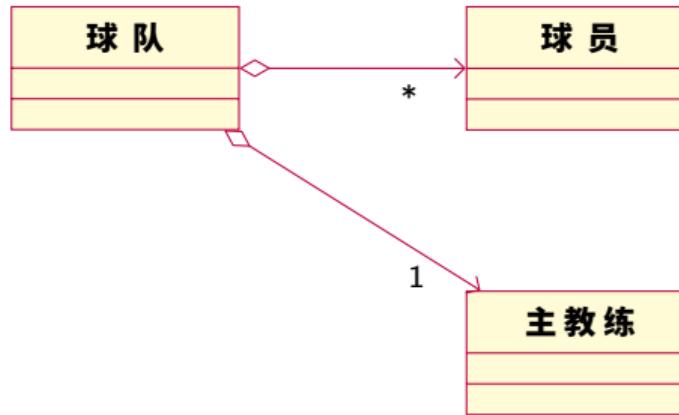
- ▶ 整体和部分关系
- ▶ 同时存在
- ▶ 同时销毁





### ▶ 聚合

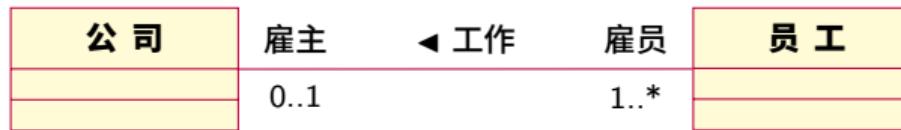
- ▶ 包含关系
- ▶ 共享成员
- ▶ 组合的泛化





### ▶ 关联

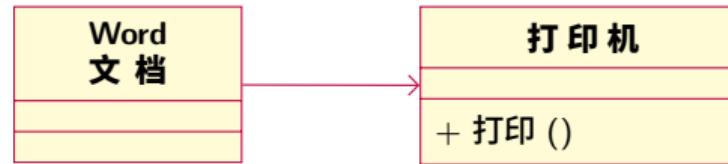
- ▶ 消息传递链
- ▶ 聚合的泛化





### ▶ 依赖

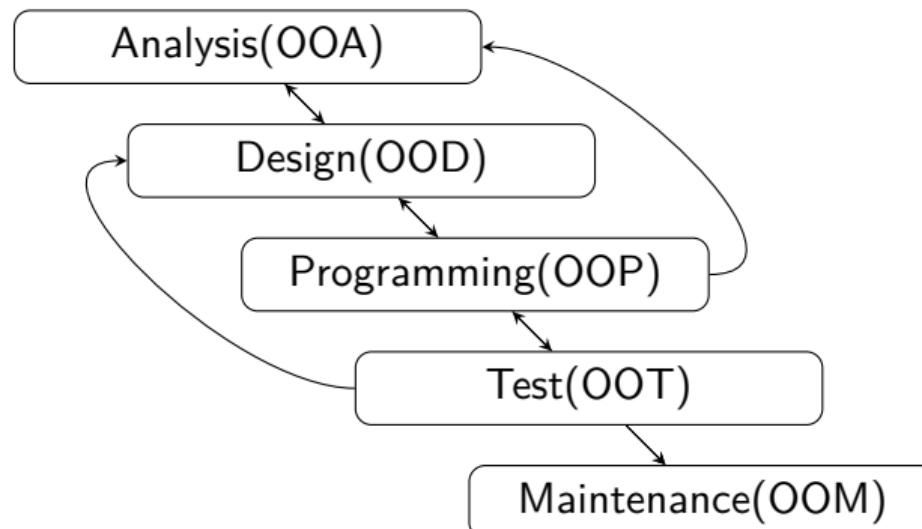
- ▶ 短时使用关系
- ▶ 非永久关系





► 使用**对象**去设计程序编写代码的一种方法

- ▶ 数据抽象
- ▶ 信息隐藏
- ▶ 代码重用





## ▶ 过程式程序设计

- ▶ 数据和过程分离，可维护性与可重用性差
- ▶ 不适于大型程序开发

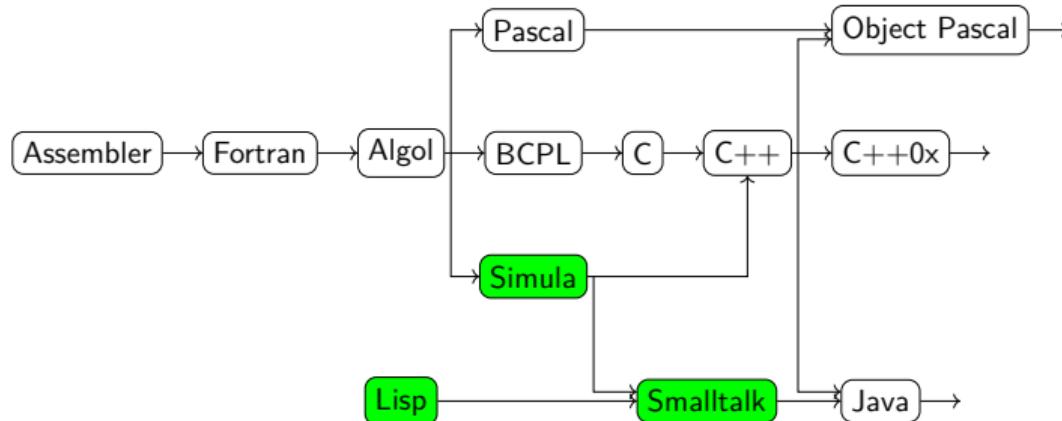
## ▶ 面向对象程序设计

- ▶ 现实世界的直接模拟
- ▶ 可重用性好，可维护性好
- ▶ 适合大型程序开发





## ► 族谱 (Family tree)

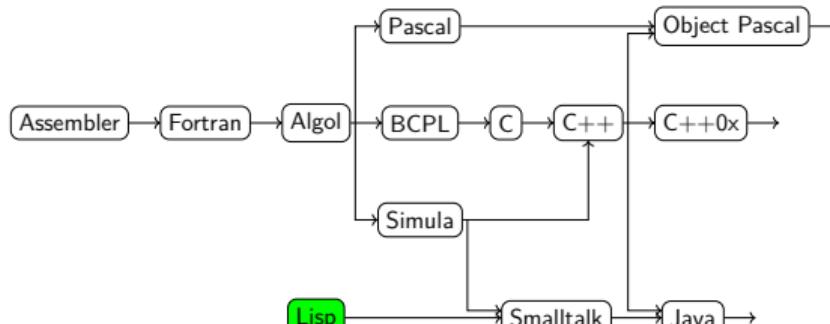




## ► Lisp

- ▶ 1958
- ▶ 人工智能语言
- ▶ 引入了对象的概念 (Identified items with attributes)
- ▶ 语法特点

```
( let( (a 6) (b 4) (+ a b) ) )
```



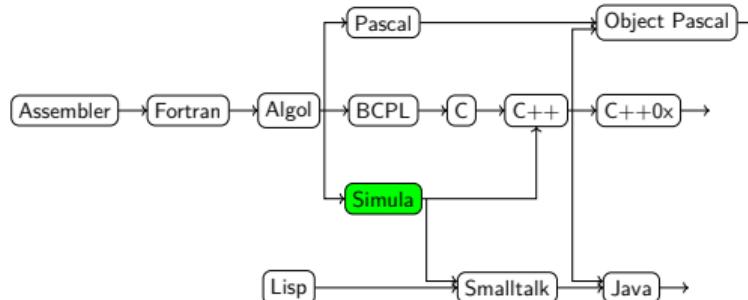
John McCarthy



## ▶ Simula

- ▶ 1967
- ▶ 被认为是第一个面向对象程序设计语言
- ▶ 第一次引入了对象、数据抽象和类的定义及继承机制
- ▶ 语法特点

```
Begin
  Class Char(c);
  Character c;
  Begin
    Procedure print;
    OutChar(c);
  End;
End;
```

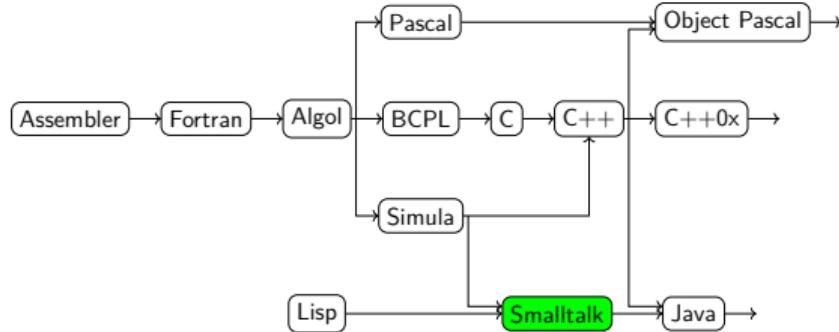


Ole-Johan Dahl and  
Kristen Nygaard



## ► Smalltalk

- 1972
- 第一个纯 (Pure) 面向对象程序设计语言
- 语法特点
  - Primitives such as character and punctuation is treated as an object  
**例如: 3+4**  
Sends the message “+” to the receiver 3 with argument 4



Alan Kay



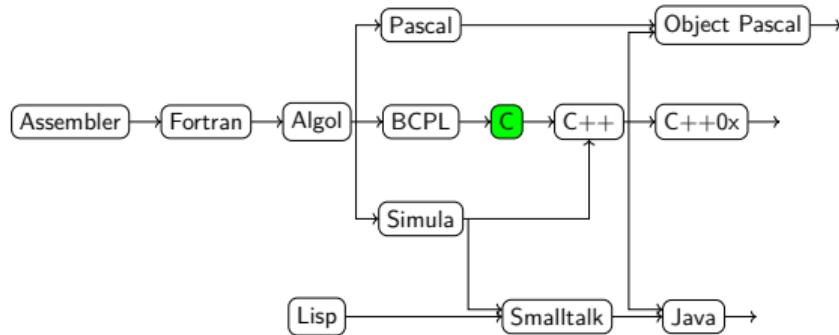


C

发展史

## ▶ C

- ▶ 1972
- ▶ C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language **efficient enough to displace assembly language**, yet **sufficiently abstract and fluent** to describe algorithms and interactions in a wide variety of environments.



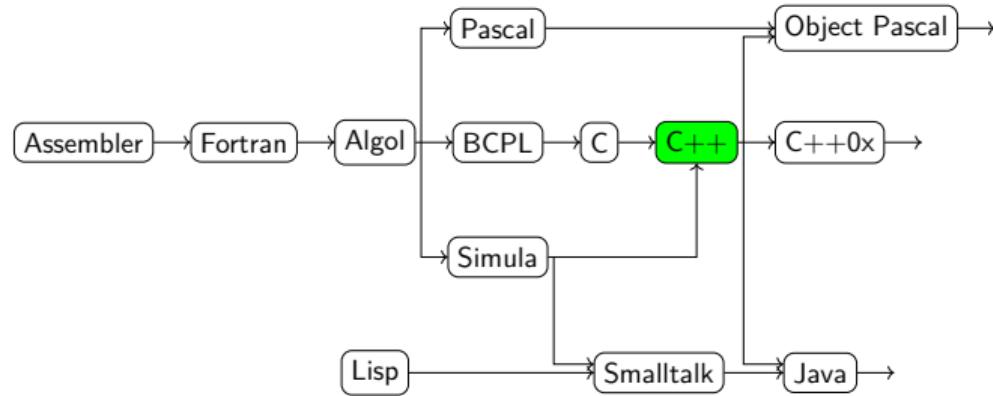
Dennis Ritchie





## ► C++

► 1985



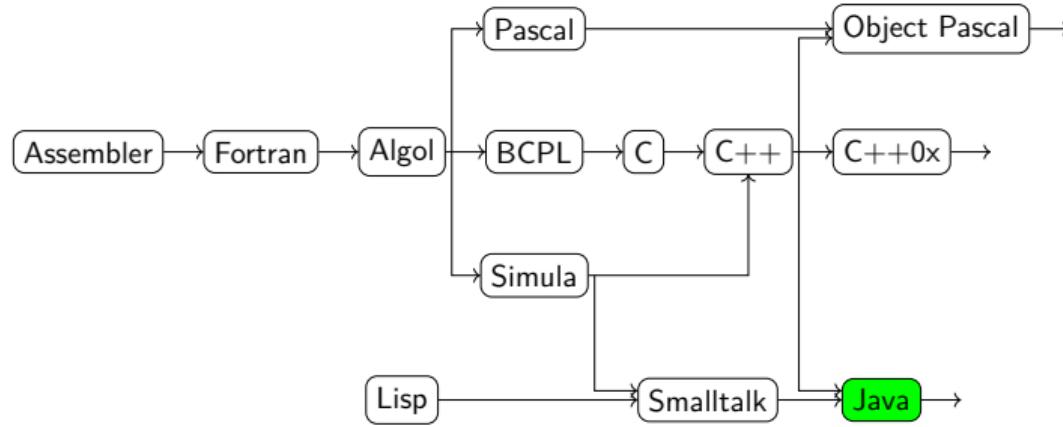
Bjarne Stroustrup  
(本贾尼·斯特劳斯特卢普)





## ► JAVA

► 1995



James Gosling





- ▶ Java  
    网络计算 (Network computing)
- ▶ C++  
    系统程序设计 (Systems programming)



Bjarne Stroustrup

<http://www2.research.att.com/~bs/>





- ▶ Adobe 系统: Photoshop, Illustrator 等
- ▶ Alias|Wavefront: Maya, Autodesk 等
- ▶ Apple OS X 重要的部分
- ▶ 游戏: 星际争霸, 暗黑破坏神, 魔兽争霸等
- ▶ Telecommunications
- ▶ Google (网络搜索引擎等)
- ▶ Microsoft applications and GUIs
- ▶ Linux tools and GUIs
- ▶ Amazon.com: 大型电子商务应用软件
- ▶ ....





- ▶ Java、C# 和 Objective C
- ▶ 计算机图形学及计算机动画
- ▶ 图形图像处理
- ▶ 虚拟现实技术
- ▶ ....





# ▶ 开发流程

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP  
Nine, G.





### ► 辅助程序

### ► 工具

- 编辑器
- 编译器／解释器
- 自动建立工具
- 调试器
- 版本控制器
- GUI 设计器
- .....





## ▶ DEV C++

The screenshot shows the Dev-C++ 4.9.9.2 IDE interface. The menu bar includes File, Edit, Search, View, Project, Execute, Debug, Tools, CVS, Window, and Help. The toolbar has icons for New, Insert, Toggle, and Goto. The main window has tabs for Project, Classes, and Debug, with main.cpp, rectangle.h, and rectangle.cpp selected. The code editor displays the following C++ code:

```
#include <cstdlib>
#include <iostream>
#include "rectangle.h"

using namespace std;

int main(int argc, char *argv[])
{
    double l, w;

    cout << "Please input the length and the width of a rectangle";
    cout << "Length=";
    cin >> l;
    cout << "Width=";
    cin >> w;

    cout << "The perimeter of the rectangle is " << GetPerimeter(l, w);
    cout << "The Area of the rectangle is " << GetArea(l, w) << endl;
}
```

<http://sourceforge.net/projects/orwelldevcpp/>





### ► Visual Studio 2005

The screenshot shows the Microsoft Visual Studio 2005 IDE interface. The title bar reads "VSTest - Microsoft Visual Studio". The menu bar includes File, Edit, View, Project, Build, Debug, Tools, Window, Help. The toolbar has icons for New, Open, Save, Cut, Copy, Paste, Find, Replace, and others. The Solution Explorer on the left shows a single project "VSTest" with files stdafx.h, stdafx.cpp, VSTest.cpp, and ReadMe.txt. The Properties window is visible on the right. The main code editor window displays the "VSTest.cpp" file:

```
// VSTest.cpp : 定义控制台应用程序的入口点。
//

#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
    cout << "Hello world!" << endl;

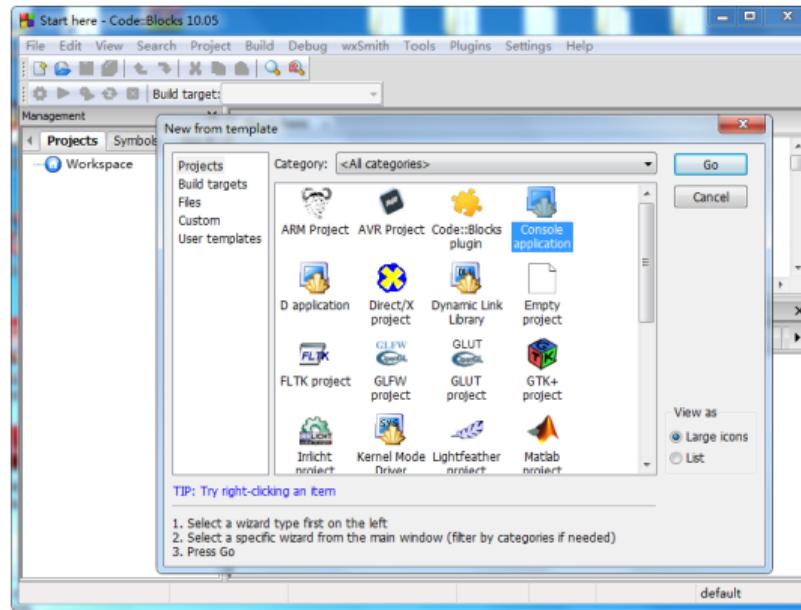
    return 0;
}
```

The status bar at the bottom shows "生成成功" (Build Succeeded), "行 12 列 1 Ch 2 Ins", and a URL "http://www.visualstudio.com/".





### ► Code::Blocks

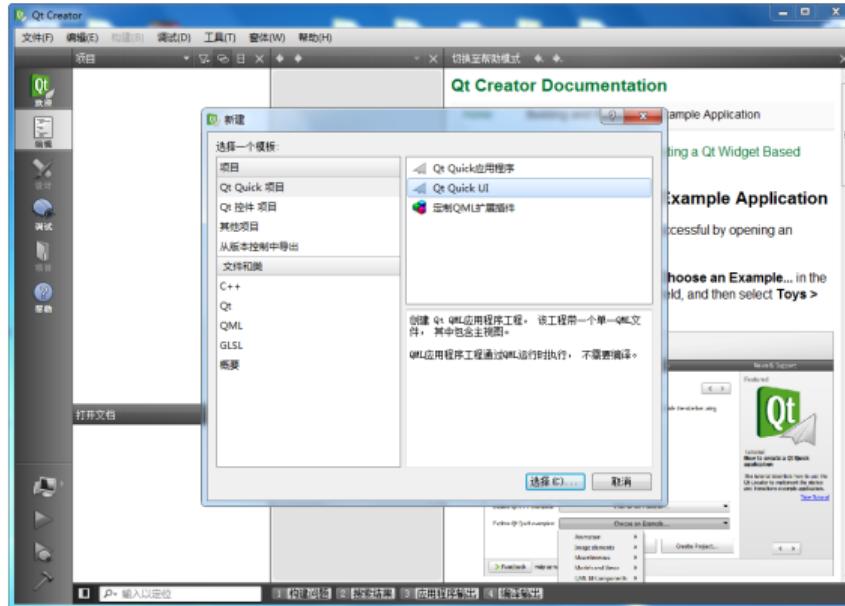


<http://www.codeblocks.org/>





► QT



<http://qt-project.org/>





```
// 例 01-01: ex01-01.cpp
#include <iostream>
using namespace std;
int main()
{
    char strName[32];
    cin >> strName;
    cout << "Hello, " << strName << "!" << endl;
    return 0;
}
```

## ▶ 名字空间 (namespace)

```
using namespace std;
```

## ▶ 输入/输出

### ▶ cin 对象

```
cin >> 对象 1 >> 对象 2 >> ... >> 对象 n;
```

### ▶ cout 对象

```
cout << 对象 1 << 对象 2 << ... << 对象 n;
```





### ▶ 两个整数的加法

```
// 例 01-02: ex01-02.cpp
// 计算两个整数的和
#include <iostream> // 输入输出（流）
// main 函数是程序的入口
int main()
{
    // 变量声明
    int number1 = 0; // 第 1 个整数（初始化为 0）
    int number2 = 0; // 第 2 个整数（初始化为 0）
    int sum = 0; // 和（初始化为 0）

    std::cout << "Enter first integer: "; // 提示输入数据
    std::cin >> number1; // 读入数据到 number1

    std::cout << "Enter second integer: "; // 提示输入数据
    std::cin >> number2; // 读入数据到 number2

    sum = number1 + number2; // 加，并将结果存入 sum

    std::cout << "Sum is " << sum << std::endl; // 显示 sum，并显示 end line
    return 0;
}
```



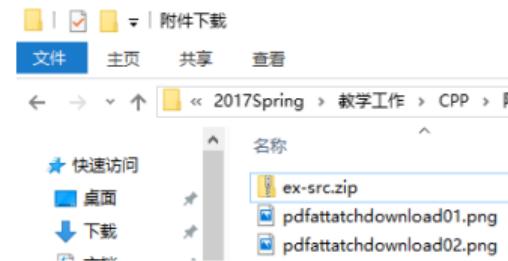
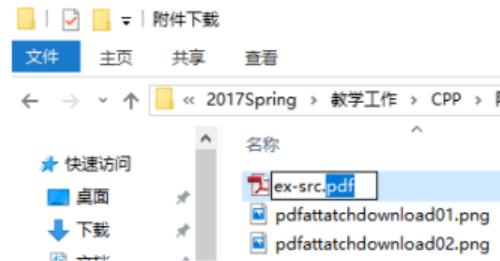
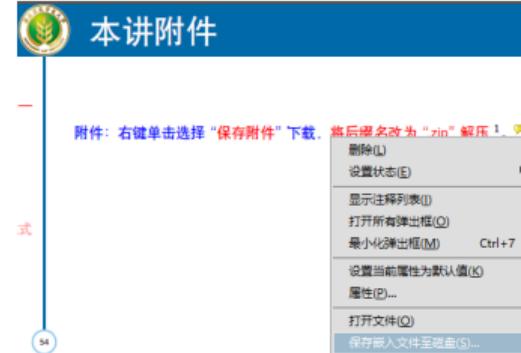
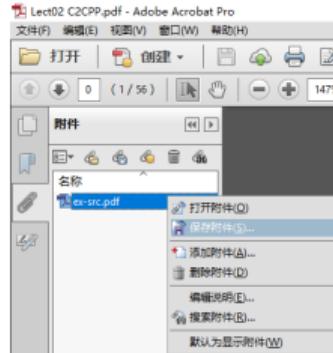


# 本讲附件

| 附件

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP  
Nine, G.

附件：右键单击该链接，选择“保存附件”下载，将后缀名改为“.zip”解压<sup>1 2</sup>。



<sup>1</sup>请退出全屏模式后点击该链接。

<sup>2</sup>以 Adobe Acrobat Reader 为例。

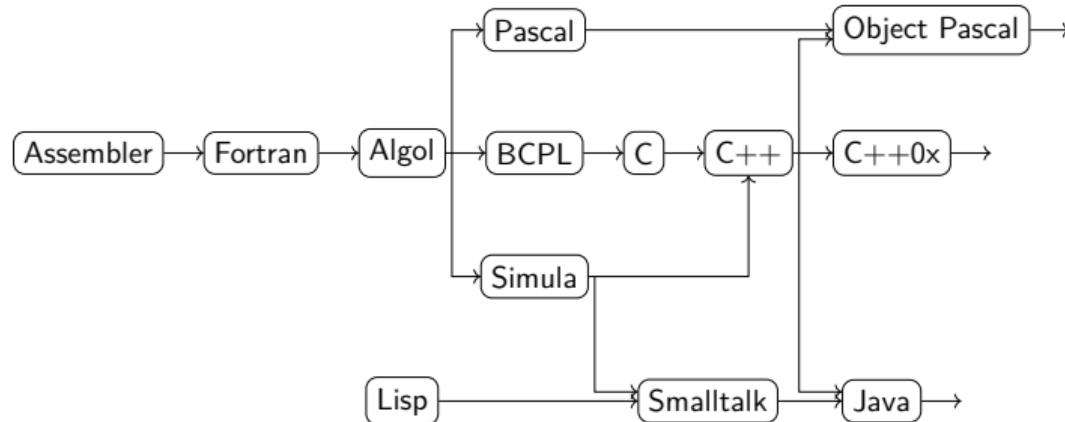


- ▶ 对象、类及其特性
  - ▶ 什么是对象
  - ▶ 什么是类
  - ▶ 四大特性（数据抽象、封装、继承和多态）
- ▶ 面向对象程序设计语言发展史
- ▶ 基本 C++ 程序
  - ▶ `cin` 和 `cout`
  - ▶ `using namespace`





## ► 族谱 (Family tree)





### ► Hello, your name!

```
// 例 01-01: ex01-01.cpp
#include <iostream>
using namespace std;
int main( )
{
    char strName[32];
    cin >> strName;
    cout << "Hello, " << strName << "!" << endl;
    return 0;
}
```

### ► 名字空间

- **using namespace**

### ► 文件后缀名

- Windows: .cpp
- Unix/Linux: .cpp, .cc or .c





- ▶ 格式化输入输出
- ▶ 基本数据类型与表达式
- ▶ 控制结构
- ▶ 构造数据类型
- ▶ 函数
- ▶ 大型程序结构控制





## ► cin/cout

```
// 例 02-01: ex02-01.cpp
// 求三个数的平均值，演示 C++ 简单 I/O
#include <iostream>
using namespace std;
int main()
{
    float num1, num2, num3;           // 定义三个数

    cout << "Please input three numbers:" ;
    cin >> num1 >> num2 >> num3;

    cout << "The average of " << num1 << ", " << num2 << "and " << num3;
    cout << " is: " << (num1 + num2 + num3) / 3 << endl;

    return 0;
}
```

The terminal window shows the following output:

```
Please input three numbers:101 201 300
The average of 101, 201and 300 is: 200.667

Process returned 0 (0x0)  execution time : 10.174 s
Press ENTER to continue.
```





## ▶ 格式控制

操作符	作用	说明
oct	数据以 8 进制形式输出	32cm 作用范围为后续输出的整数，对小数不起作用
dec	数据以 10 进制形式输出（默认）	
hex	数据以 16 进制形式输出	
endl	换行并刷新输出流	
setw(n)	设置输出宽度	22cm 作用范围为后续对象
setprecision(n)	设置输出精度（默认为 6）	

注意:

- ▶ `#include <iomanip>`
- ▶ 默认情况下, `setprecision(n)` 仅对带有小数的数有效, n 为整数与小数位数之和



### ▶ 设置输出格式

```
// 例 02-02: ex02-02.cpp
// 求三个数的平均值，演示 C++ 简单 I/O 格式控制
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    float num1, num2, num3; // 定义三个数

    cout << "Please input three numbers:" ;
    cin >> num1 >> num2 >> num3;

    cout << setw(20) << setprecision(12);
    cout << "The average of " << num1 << ", " << num2 << " and " << num3;
    cout << " is:" << setw(20) << (num1 + num2 + num3) / 3 << endl;

    return 0;
}
```

```
testCpp
Please input three numbers:101 201 300
The average of 101 , 201 and 300 is:      200.666671753
Process returned 0 (0x0)  execution time : 10.591 s
Press ENTER to continue.
```





## ▶ 基本数据类型

- ▶ **int, float, double, void, char**
- ▶ 布尔型: **bool** (**true**⇒1, **false**⇒0)

## ▶ 变量与常量

### ▶ 变量的定义与赋初值

- ▶ **int sum=100; double pi=3.1416; char c='a';**
- ▶ **int sum(100); double pi(3.1416); char c('a');**

### ▶ 符号常量与常变量

- ▶ **#define PI 3.1416**
- ▶ **const float PI=3.1416;**
- ▶ **PI = 3.1415926535898; // 错误!**





## ▶ 常量表达式

```
const int size = 20;           // size 是常量表达式
const int limits = size + 1;   // limits 是常量表达式
int max = 80;                 // max 不是常量表达式, 80 是字面值常量,
                             // 但 max 不是 const, 不保证运行是不变。
const int lines = get_size();  // lines 不是常量表达式
                             // lines 是常量, 但 get_size() 运行时才能确定
```

▶ **constexpr**类型 (验证是不是常量表达式)

```
constexpr int size = 20;        // 20 是常量表达式
constexpr int limits = size + 10; // size + 10 是常量表达式
constexpr int max = length();   // 取决于 length() 函数是不是常量函数
```

▶ **constexpr**与**const**

```
constexpr int a = length(); // 必须在编译时能计算出 length() 返回值
const int b = length();    // b 的值可以在运行时获得, 之后不再改变
```





### ► auto类型说明符

```
auto x = 5;           // 5 是 int 类型, x 则是 int 类型
auto size = sizeof(int); // size 是表示内存字节数的类型
auto name = "world";   // name 是保存字符串的类型
cout << "hello " << name; // 可以使用 name
auto a;               // 错误! 没有初始值无法确定类型
auto r = 1, pi = 3.14; // 错误! 类型混淆
```

### ► decltype类型指示符, 返回操作数类型

```
decltype(sizeof(int)) size; // sizeof(int) 结果的类型
const int ci = 0;           // 是常量表达式
decltype(ci) x = 1;         // const int 类型
decltype(f()) y = sum;      // 函数 f() 的返回值类型
```





### ▶ 运算符与表达式

- ▶ 算术运算符: +(正号), -(负号), \*, /, %(取余)
- ▶ 关系运算符: >, <, >=, <=, ==, !=
- ▶ 逻辑运算符: !, &&, ||
- ▶ 位运算符: ~, <<, >>, &, ^(异或), |
- ▶ 赋值运算符: =, \*=, /=, %=, +=, -=, ...
- ▶ 递增递减运算符: ++, --

$$ax^2 + bx + c = 0$$

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
if(abs(b * b - 4 * a * c) > 1.0e - 10)
{
    x1 = (-b + sqrt(b * b - 4 * a * c)) / (2 * a);
    x2 = (-b - sqrt(b * b - 4 * a * c)) / (2 * a);
}
```





## ▶ 判断

- ▶ `if ... else ...`
- ▶ `if ... else if ... else ...`
- ▶ `switch ... case ...`

## ▶ 循环

- ▶ `for(exp1;exp2;exp3){...}`
- ▶ `while(exp){...}`
- ▶ `do ... while(exp);`

## ▶ 转移

- ▶ `break`
- ▶ `continue`
- ▶ `goto`





## ► 范围for语句

```
for(declaration : expression){  
    statement;  
}  
// 其中:  
//   expression 必须是一个序列 (列表、数组、vector、string 等),  
//   能返回 begin 和 end 对象。  
//   declaration 是一个变量, 序列中每个元素都能够转换为该类型,  
//   常用 auto 声明
```

## ► 范围for示例

```
// 累加 20 以内的素数  
int sum = 0;  
for(int e : {2, 3, 5, 7, 11, 13, 17, 19}) // 用 auto 类型更合理  
    sum += e;  
cout << sum << endl; // 输出结果 77  
int arr[] = {1, 3, 5, 7, 9}; // 声明数组 arr, 初始化为 5 个奇数  
for(auto ele : arr){ // 声明 ele, 与数组 arr 关联在一起, 用了 auto  
    ele = ele * 2; // 修改数组每个元素的值  
    cout << ele << " "; // 输出 ele, 2 6 10 14 18  
}  
cout << endl;  
for(auto ele : arr)  
    cout << ele << " "; // 没有改变: 1 3 5 7 9  
cout << endl;
```





## ▶ 指针

```
int a=255;
int *p;
p=&a;
```

24 &a bitheight=4 []10&p→ 0x00ff ffee

0x00ff fffb 600FF 4212

bitheight=4 []10

6.....

a bitheight=4 []10&a→ 0x00ff 4212

0x00ff 420f 60000 00FF

```
float x[5];
float *p = x;
double sum = 0.0;
for (int i = 0; i < 5; i++)
{
    sum += *p++;
}
```





### ► 动态内存分配

- ▶ `malloc` 和 `free`
- ▶ `new` 和 `delete`

```
// C 语言
float *x = (float *)malloc(n*sizeof(float));
free (x);
```

```
// C++ 语言
float *x = new float[n];
delete []x;
```

```
int **mat;
int m, n;
mat = new int *[m];

for (i = 0; i < m; i++)
    mat[i] = new int[n];

for (i = 0; i < m; i++)
    delete [] mat[i];
delete [] mat;
```





### ▶ 定位new表达式

#### ▶ 语法: new (指针) 类型

```
#include <iostream>
#include <new> // 必须包含该头文件

using namespace std;

char * buf = new char[1000]; // 预分配空间

int main()
{
    int * pi = new (buf) int; // 在 buf 中创建一个 int 对象

    return 0;
}
```





### ▶ 指针常量

```
int a = 2, b = 3;
int * const p = &a; //定义时必须赋初值
p = &b;           //错误，地址不能被修改
*p = b;          //正确，内容可以被修改
```

### ▶ 常量指针

```
int a = 2, b = 3;
const int * p;
p = &b;           //正确，地址可以被修改
*p = b;          //错误，内容不可以被修改
```

### ▶ 常指针常量

```
int a = 2, b = 3;
const int * const p = &a; //定义时必须赋初值
p = &b;           //错误，地址不能被修改
*p = b;          //错误，内容不可以被修改
```





## ▶ 引用是已存在的变量的别名

```
int i = 3;
int &j = i;    //引用必须赋初值
int &j = 3;    //错误，初值必须为变量
j = 4;
```

## ▶ 引用和指针的区别与联系

```
int i = 3;
int &j = i;
int *k = &i;
cout << &i << endl;
cout << &j << endl;
cout << &k << endl;
```

6...

6... 24 bitheight=1 [10  
60x0016 FE04 [10k → 0x0016 FDEC  
6... bitheight=1 [10  
6bitheight=00B [10<sup>1</sup>] → 0x0016 FE04  
bitheight=1 [10 ...





## ▶ 引用作为函数参数（例 1）

```
// 例 02-03: ex02-03.cpp
#include <iostream>
using namespace std;

void swap(int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
}

int main()
{
    int m = 3, n = 4;

    cout << "before swap:";
    cout << m << "," << n << endl;

    swap(m, n);

    cout << "after swap:";
    cout << m << "," << n << endl;

    return 0;
}
```

```
// 例 02-04: ex02-04.cpp
#include <iostream>
using namespace std;

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main()
{
    int m = 3, n = 4;

    cout << "before swap:";
    cout << m << "," << n << endl;

    swap(&m, &n);

    cout << "after swap:";
    cout << m << "," << n << endl;

    return 0;
}
```



## ▶ 引用作为函数参数（例 2）

```
struct StuNode
{
    int ID;
    char name[128];
    bool gender;
    int age;
    struct StuNode *next;
};
```

```
// 例 02-05: ex02-05.cpp
void CreateHeadNode(StuNode **pHead)
{
    StuNode *p;
    p = new StuNode;
    if (p == NULL) return;
    p->next = NULL;
    *pHead = p;
}

int main()
{
    StuNode *pHead = NULL;
    CreateHeadNode(&pHead);

    return 0;
}
```





## ▶ 引用作为函数参数（例 2）

```
struct StuNode
{
    int ID;
    char name[128];
    bool gender;
    int age;
    struct StuNode *next;
};
```

```
// 例 02-06: ex02-06.cpp
void CreateHeadNode(StuNode *&pHead)
{
    StuNode *p;
    p = new StuNode;
    if (p == NULL) return;
    p->next = NULL;
    pHead = p;
}

int main()
{
    StuNode *pHead = NULL;
    CreateHeadNode(pHead);
    return 0;
}
```





## ▶ 常引用

```
int i = 100;
const int &r1 = i;           //正确
const int &r2 = i;           //必须初始化
r2 = 200;                   //错误
```

## ▶ 常引用参数

```
int fun(const int &a, const int &b)
{
    return (a + b) / 2;        //参数不能被修改
}
```





## ▶ 常引用

```
int i = 100;
const int &r1 = i;           //正确
const int &r2 = i;           //必须初始化
r2 = 200;                   //错误
```

## ▶ 常引用参数

```
int fun(const int &a, const int &b)
{
    return (a + b) / 2;        //参数不能被修改
}
```





### ▶ begin() 和 end()

#### ▶ 语法

---

```
begin(数组名)
end(数组名)
```

---

### ▶ 运算

- ▶ 解引用
- ▶ 自增、自减
- ▶ 加或减整数、
- ▶ 指针相减
- ▶ 指针比较

```
#include<iterator> // 迭代器运算头文件
...
int ia[5] = {1, 2, 3, 4, 5};
int *pb = begin(ia);
int *pe = begin(ia);
while(pb != pe && *pb >= 0)
{
    ++pb;
}
```





## ▶ 字符数组和 C 风格字符串

- ▶ 以'\\0'结束字符串
- ▶ 需要使用头文件:#include<cstring>

## ▶ C++ 的 string 类

- ▶ 需要使用头文件:#include<string>
- ▶ 丰富的字符串处理函数
- ▶ 便捷的运算符重载
- ▶ 单字符处理
  - ▶ 需要使用头文件:#include<cctype>
  - ▶ 基本循环
  - ▶ 范围 for



▶ 标准类型 `vector`

- ▶ 同种类型对象的集合
- ▶ 长度可变

## ▶ 定义和初始化

- ▶ 语法:`vector<元素类型> 变量名;`
- ▶ 初始化方法

## 方 法

```
vector<T> v1
vector<T> v2(v1)
vector<T> v2 = v1
vector<T> v3(n, val)
vector<T> v4(n)
vector<T> v5{a,b,c,...}
vector<T> v5={a,b,c,...}
```

## 说 明

v1 为空，元素是 T 类型，默认初始化  
声明 v2 向量，用 v1 初始化，是 v1 的副本  
等价于 `v2(v1)`  
v3 有 n 个 T 类型的**重复**元素，每个元素的值都是 val  
v4 有 n 个**重复**默认值初始化的元素  
v5 元素个数为初始化式中元素个数  
等价于 `v5{a,b,c,...}`





### ▶ 强制类型转换

#### ▶ C 风格

```
float x = 3.5;  
int roundX = (int)(x + 0.5);
```

#### ▶ C++ 风格: castname<类型名>(表达式)

- ▶ static\_cast
- ▶ dynamic\_cast
- ▶ const\_cast
- ▶ reinterpret\_cast

```
int roundX = static_cast <int> (x+0.5);
```

### ▶ 强制指针类型转换

```
char *p;  
void *q = malloc(sizeof(char)*1024);  
p = q; //错误! 无法从“void *”转换为“char *”, C++  
p = (char *)q;
```





- ▶ 函数调用执行过程
- ▶ 内联函数
- ▶ 带默认形参的函数
- ▶ 函数重载
- ▶ 函数模板
- ▶ 系统函数





# 函数调用执行过程

| 函数

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP  
Nine, G.

- ▶ 函数调用
  - ▶ 参数和函数返回地址入栈
- ▶ 执行函数体
  - ▶ 寄存器进出栈，通过栈访问参数
- ▶ 函数返回
  - ▶ 返回到调用函数的下一条语句执行

```
int increase(int a)
{
    return ++a;
}
int main()
{
    int x = 3,y;
    y = increase(x);
    return 0;
}
```

```
1: int x=3,y;
00E813FE mov     dword ptr [x],3
2: y = increase(x);
00E81405 mov     eax,dword ptr [x]
00E81408 push    eax
00E81409 call    increase (0E81154h)
00E8140E add    esp,4
00E81411 mov     dword ptr [y],eax
```

```
1: int increase(int a) {
00E813A0 push    ebp
00E813A1 mov     ebp,esp
00E813A3 sub    esp,0C0h
00E813A9 push    ebx
00E813AA push    esi
00E813AB push    edi
00E813AC lea     edi,[ebp-0C0h]
00E813B2 mov     ecx,30h
00E813B7 mov     eax,0CCCCCCCCCh
00E813BC rep stos dword ptr es:[edi]
2: return ++a;
00E813BE mov     eax,dword ptr [ebp+8]
00E813C1 add    eax,1
00E813C4 mov     dword ptr [ebp+8],eax
00E813C7 mov     eax,dword ptr [ebp+8]
3: }
00E813CA pop    edi
00E813CB pop    esi
00E813CC pop    ebx
00E813CD mov     esp,ebp
00E813CE pop    ebp
00E813D0 ret
```



## ▶ 普通函数调用缺陷

- ▶ 时间开销

## ▶ 内联函数

- ▶ 在编译时将函数体代码插入到调用处
- ▶ 适用于代码短、频繁调用的场合

## ▶ 定义

---

```
inline 函数类型 函数名 (参数表)
{
    函数体;
}
```

---





## ► 本质是预处理后展开

```
inline int increase(int a)
{
    return ++a;
}
int main()
{
    int x = 3,y;
    y = increase(x);
    return 0;
}
```



```
int main()
{
    int x = 3,y;
    int a = x;
    y = ++a;
    return 0;
}
```

## ► 效率测试

```
inline float getCos_inline(int &x)
{
    float r;
    x = rand();
    r = cos(2 * 3.1416 * x / (float)RAND_MAX);
    return r;
}
```

```
Time for inline:763214
Time for called function:4057411
Process returned 0 (0x0)   execution time : 4.821
Press ENTER to continue.
```

测试环境 (Code:Blocks GCC Release)  
效率比 = 4057411ms/763214ms = 5.3

例 02-07: ex02-07.cpp





## ▶ 注意事项

- ▶ 不能出现递归
- ▶ 代码不宜过长
- ▶ 不宜出现循环
- ▶ 不宜含有复杂控制语句如switch等
- ▶ 有些编译器会智能处理是否为内联函数





## ▶ 语法

### ▶ constexpr 函数 (常量表达式函数)

## ▶ 基本要求

- ▶ 只有一句**return**可执行语句，可有别名、**using**等
- ▶ 必须有返回类型，返回类型不能是**void**
- ▶ 使用前必须定义 (不只是声明)
- ▶ **return**中不能有非常量表达式





## ► 是编译时求值，不是运行时调用

```
constexpr int data() // 错误，函数体只能有一条 return 可执行语句
{
    const int i = 1;
    return i;
}
constexpr int data() // 正确
{
    return 1;
}
constexpr void f() // 错误，无法获得常量
{
}
```

## ► 是函数使用（编译时），不是函数调用（运行时）

```
constexpr int f(); // 只有 constexpr 函数的声明，没有定义
int a = f(); // 正确，可以将编译时的计算转换为运行时的调用
const int b = f(); // 正确，编译器将 f() 转换为一个运行时的调用
constexpr int c = f(); // 错误，c 是 constexpr，要求使用 f()，在编译时计算
constexpr int f() // constexpr 函数的定义
{
    return 1;
}
constexpr int d = f(); // 正确，f() 已定义，可以使用 f()
```



► **return** 中不能包含运行时才能确定的函数

```
constexpr int e()
{
    return 1;
}
constexpr int g()
{
    return e();      // 错误，调用了非 constexpr 函数
}
constexpr int e()
{
    return 1;
}
constexpr int g()
{
    return e();      // 正确，函数 e() 是常量表达式函数
}
```

► 用 **constexpr** 函数初始化 **constexpr** 变量

```
constexpr int new_sz()
{
    return 100;
}
constexpr int size = new_sz();
```





► 在函数定义或说明中为形参赋默认值

► 作用

- 若调用给出实参值，则形参采用实参值
- 若调用未给出实参值，则调用默认参数值

---

```
// 例 02-08: ex02-08.cpp
void SetNetCamera (char *UserName = "guest", char *Password = "321",
                    char *URL = "219.145.198.100", char *ServerName = "654",
                    float Zoom = 0.2, float Alpha = 10.0, float Beta = 15.0)
{
    cout << UserName << " ";
    cout << Password << " ";
    cout << URL << " ";
    cout << ServerName << " ";
    cout << Zoom << " ";
    cout << Alpha << " ";
    cout << Beta << endl;
}
```

---





## ► 基本要求

- ▶ 调用函数时，如省去某个实参，则该实参右边所有实参都要省略
- ▶ 默认形参必须**自最右向左连续**定义
- ▶ 若函数声明(原型)中给出默认形参值，则函数定义时不能重复指定

```
// 例 02-09: ex02-09.cpp
void SetNetCamera (char *UserName = "guest", char *Password = "321",
                    char *URL = "219.145.198.100", char *ServerName = "654",
                    float Zoom = 0.2, float Alpha = 10.0, float Beta = 15.0)
{
    cout << UserName << " " << Password << " "
        << URL << " " << ServerName << " "
        << Zoom << " " << Alpha << " " << Beta << endl;
}
int main()
{
    SetNetCamera();
    SetNetCamera("Xinji", "class1&2", "219.145.198.105", "654");
    SetNetCamera("Administrator", "nwsuaf", "219.145.198.108", "654",
                1.0, 15.0, 30.0);
    return 0;
}
```





## ► 中间参数不能省略

```
// 例 02-10: ex02-10.cpp
void SetNetCamera (char *UserName = "guest", char *Password = "321",
                    char *URL = "219.145.198.100", char *ServerName = "654",
                    float Zoom = 0.2, float Alpha = 10.0, float Beta = 15.0)
{
    cout << UserName << " " << Password << " "
        << URL << " " << ServerName << " "
        << Zoom << " " << Alpha << " " << Beta << endl;
}
int main()
{
    SetNetCamera();
    SetNetCamera("Xinji", "class1&2", "219.145.198.105", "654");
    SetNetCamera("Administrator", "nwsuaf", , , 1.0, 15.0, 30.0);
    return 0;
}
```





## ► 不可重复指定参数默认值

```
// 例 02-11: ex02-11.cpp
void SetNetCamera (char *UserName = "Guest", char *Password = "321",
                    char *URL = "219.145.198.100", char *ServerName = "654",
                    float Zoom = 0.2, float Alpha = 10.0, float Beta = 15.0);

int main()
{
    SetNetCamera();
}

void SetNetCamera (char *UserName, char *Password, char *URL,
                  char *ServerName, float Zoom, float Alpha, float Beta)
{
    cout << UserName << " " << Password << " "
        << URL << " " << ServerName << " "
        << Zoom << " " << Alpha << " " << Beta << endl;
}
```





## ► 默认形参必须自最右向左连续定义

```
// 例 02-12: ex02-12.cpp
void SetNetCamera (char *UserName = "Guest", char *Password = "321",
                    char *URL = "219.145.198.100", char *ServerName = "654",
                    float Zoom, float Alpha, float Beta)
{
    cout << UserName << " " << Password << " "
        << URL << " " << ServerName << " "
        << Zoom << " " << Alpha << " " << Beta << endl;
}
```





## ► 默认形参必须自最右向左连续定义

```
// 例 02-13: ex02-13.cpp
void SetNetCamera (char *UserName = "Guest", char *Password = "321",
                    char *URL, char *ServerName,
                    float Zoom = 0.2, float Alpha = 10.0, float Beta = 15.0)
{
    cout << UserName << " " << Password << " "
        << URL << " " << ServerName << " "
        << Zoom << " " << Alpha << " " << Beta << endl;
}
```





## ► 必须为无默认值的参数提供实参

```
// 例 02-14: ex02-14.cpp
void SetNetCamera (char *UserName, char *Password,
                    char *URL = "219.145.198.100", char *ServerName = "654",
                    float Zoom = 0.2, float Alpha = 10.0, float Beta = 15.0)
{
    cout << UserName << " " << Password << " "
        << URL << " " << ServerName << " "
        << Zoom << " " << Alpha << " " << Beta << endl;
}
int main()
{
    SetNetCamera();
    SetNetCamera("Guest", "321");
    SetNetCamera("Xinong", "class1 & 2", "219.145.198.105", "654");
    SetNetCamera("Administrator", "nwsuaf");
    return 0;
}
```





## ► 形参的初始化可以是函数

```
// 例 02-15: ex02-15.cpp
#include <iostream>
using namespace std;
float RadianToAngle(float radian)
{
    return radian * 180.0 / 3.1416;
}
void SetNetCamera (char *UserName, char *Password,
                  char *URL = "219.145.198.100", char *ServerName = "654",
                  float Zoom = 0.2, float Alpha = RadianToAngle(0.174),
                  float Beta = RadianToAngle(0.262))
{
    cout << UserName << " " << Password << " "
        << URL << " " << ServerName << " "
        << Zoom << " " << Alpha << " " << Beta << endl;
}
int main()
{
    SetNetCamera("Xinji", "class1&2", "219.145.198.105", "654");
    return 0;
}
```





► 重载：同一符号或函数名对应多种操作

- 操作符重载
- 函数重载

► 函数重载

- 共性函数拥有相同函数名字

```
int sum_int(int *a, int size);
float sum_float(float *a, int size);
double sum_double(double *a, int size);
```

```
int sum(int *a, int size);
float sum(float *a, int size);
double sum(double *a, int size);
```





## ► C++ 函数重载实现机理

- ▶ 函数名
- ▶ 参数类型
- ▶ 参数个数

## ► 参数个数不同情况下的实现重载

```
float dis_2d(float x0, float y0, float x1, float y1);
float dis_3d(float x0, float y0, float z0,
            float x1, float y1, float z1);
```

```
float dis(float x0, float y0, float x1, float y1);
float dis(float x0, float y0, float z0,
          float x1, float y1, float z1);
```





## ▶ 注意事项

### ▶ 避免二义性

```
void my_fun(int a, int b);  
void my_fun(int &a, int &b);
```

### ▶ 避免将不适宜重载的函数重载

如果不同的函数名所提供的信息可使程序更容易理解，则不必使用重载

```
void rotate(float r);  
void translate(float x, float y);
```

```
void transform(float r);  
void transform(float x, float y);
```





► 用一个函数表示逻辑功能相同但参数类型不同的函数

```
int sum(int *a, int size);
float sum(float *a, int size);
double sum(double *a, int size);
```

► 定义

```
template <class 类型名 1, class 类型名 1, ...> 返回类型 函数名 (形参表)
{
    函数体;
}

// 例 02-16: ex02-16.cpp
template <class _T>
_T sum(_T *a, int size)
{
    _T result = 0;
    for (int i = 0; i < size; i++)
    {
        result += a[i];
    }
    return result;
}
```





## ► 带有两个通用类型的函数模板

```
// 例 02-17: ex02-17.cpp
template <class T1, class T2>
void myfunc(T1 x, T2 y)
{
    cout << x << " " << y << endl;
}

int main()
{
    myfunc(10, "I hate C++");
    myfunc(98.6, 19L);
}
```





## ▶ 优先级别

- ▶ 如果同时定义重载函数，将优先使用重载函数，若不能找到精确匹配，再使用函数模板

## ▶ 应用

- ▶ 数据结构中的链表、堆栈等
- ▶ C++ 的标准模板库 (排序等)
- ▶ 通用类





## ► 应用 (例：冒泡排序)

```
// 例 02-18: ex02-18.cpp
template <class _T>
void bubble(_T *items, int count)
{
    register int a, b;
    _T t;
    for(a = 1; a < count; a++)
        for(b = count - 1; b >= a; b--)
            if(items[b - 1] > items[b])
            {
                t = items[b - 1];
                items[b - 1] = items[b];
                items[b] = t;
            }
}
```





- ▶ cmath

- ▶ iostream

- ▶ 包含 ctype.h, string.h, memory.h, stdlib.h 等 isdigit(), strcpy(), memcpy(), atoi(), rand() 等

- ▶ ctime

- ▶ **time\_t**, clock() 等





## ► extern

- ▶ 大型程序设计中所有模块共同使用的全局变量 (函数)
- ▶ 在一个模块中定义全局变量 (函数), 其它模块中用**extern**说明“外来”的全局变量 (函数)

---

```
// 例 02-19-01: ex02-19-01.cpp
#include <iostream>
using namespace std;
extern int G;
void p1dispG()
{
    G = 11;
    cout << "in p1 G=" << G << endl;
}

// 例 02-19-02: ex02-19-02.cpp
#include <iostream>
using namespace std;
extern int G;
extern int g;
void p2dispG(){
    G = 22;
    cout << "in p2 G=" << G << endl;
}
void p2dispg(){
    g = 33;
    cout << "in p2 g=" << g << endl;
}
```

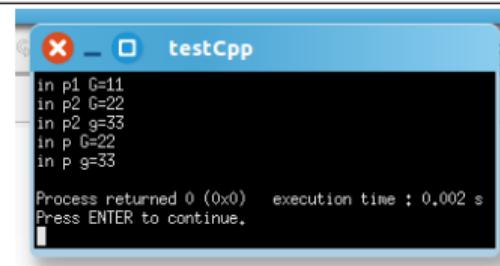
---



---

```
// 例 02-19-00: ex02-19-00.cpp
#include <iostream>
using namespace std;
extern void p1dispG();
extern void p2dispG();
extern void p2dispg();
int G = 0, g = 0;
int main(){
    p1dispG();
    p2dispG();
    p2dispg();
    cout << "in p G=" << G << endl;
    cout << "in p g=" << g << endl;
    return 0;
}
```

---





## ► static

- static 可用来声明全局静态变量和局部静态变量。当声明全局静态变量时，全局静态变量只能供本模块使用，不能被其它模块再声明为 extern 变量

---

```
// 例 02-20-01: ex02-20-01.cpp
extern void p1dispG();
static int G=0;

int main() {
    p1dispG();
    cout<<"in p G="<<G<<endl;
    return 0;
}
```

---

---

```
// 例 02-20-02: ex02-20-02.cpp
extern int G;

void p1dispG(){
    G=11;
    cout<<"in p1 G="<<G<<endl;
}
```

---





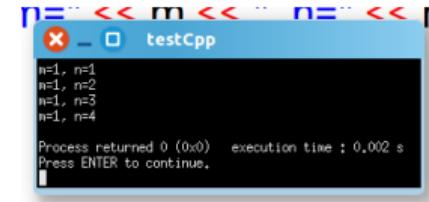
## ► static

- 当一个局部变量声明为static变量，它既具有局部变量的性质，又具有全局变量的性质

```
// 例 02-21: ex02-21.cpp
#include <iostream>
using namespace std;

void fun()
{
    static int n = 0;
    int m = 0;
    n++;
    m++;
    cout << "m=" << m << ", n=" << n << endl;
}

int main()
{
    for (int i = 0; i < 4; i++)
        fun();
    return 0;
}
```





- ▶ 多文件操作使用#include
- ▶ #include < 系统文件 >
- ▶ 到编译器指定的文件包含目录查找
- ▶ #include "| 自定义文件.h"





## ► 同一程序在不同的编译条件下得到不同的目标代码

```
// 例 02-22: ex02-22.cpp
#define USA 0
#define CHINA 1
#define ENGLAND 2
#define ACTIVE_COUNTRY USA

#if ACTIVE_COUNTRY == USA
char *currency = "dollar";
#elif ACTIVE_COUNTRY == ENGLAND
char *currency = "pound";
#else
char *currency = "yuan";
#endif
int main ()
{
    float money;
    cin >> money;
    cout << money << currency << endl;
    return 0;
}
```





## ► 便于程序移植或跨平台

---

```
// 例 02-23-01: ex02-23-01.cpp
#if defined _WIN32
#include <windows.h>
...
#elif defined __APPLE__
...
#else
#include <unistd.h>
...
#endif
```

---

---

```
// 例 02-23-02: ex02-23-02.cpp
#ifndef __cplusplus
    #include <iostream>
#else
    #include <stdio.h>
#endif
```

---





### ► 避免重复包含头文件如 “MyHeader.h”

```
// 例 02-24: ex02-24.cpp
#ifndef __MYHEADER_H
#define __MYHEADER_H
...
#endif
```

### ► 便于调试程序

```
// 例 02-25: ex02-25.cpp
#define _DEBUG
#ifdef _DEBUG
...
#endif
```





- ▶ 不同程序员撰写的软件模块可能使用相同标志符
- ▶ 为避免冲突，将可能存在相同标志符的程序模块放入名字空间
- ▶ 定义：

---

```
namespace 名称
{
    成员变量或成员函数;
}
```

---





► 声明方式（作用域分辨符）  
名字空间名::成员变量或成员函数

```
// 例 02-25-02: ex02-25-02.cpp
//Xinong.h
#ifndef XINONG_H_INCLUDED
#define XINONG_H_INCLUDED

namespace Xinong
{
    int year = 2011;
    char name[] = "Xinong";
    void ShowName()
    {
        cout << name << " " <<
            year << endl;
    }
}

#endif // XINONG_H_INCLUDED
```

```
// 例 02-25-01: ex02-25-01.cpp
#include <iostream>
using namespace std;
#include "Xinong.h"

int main()
{
    Xinong::ShowName();
    return 0;
}
```





► 声明方式（作用域分辨符）  
名字空间名::成员变量或成员函数

```
// 例 02-26-02: ex02-26-02.cpp
//Xilin.h
#ifndef XILIN_H_INCLUDED
#define XILIN_H_INCLUDED

namespace Xilin
{
    int year = 2011;
    char name[] = "Xilin";
    void ShowName()
    {
        cout << name << " " <<
            year << endl;
    }
}

#endif // XILIN_H_INCLUDED
```

```
// 例 02-26-01: ex02-26-01.cpp
#include <iostream>
using namespace std;
#include "Xilin.h"

int main()
{
    Xilin::ShowName();
    return 0;
}
```



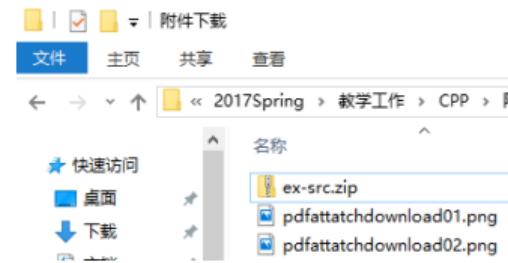
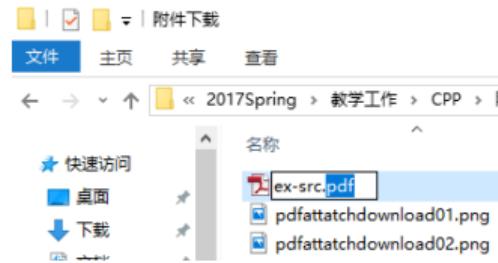
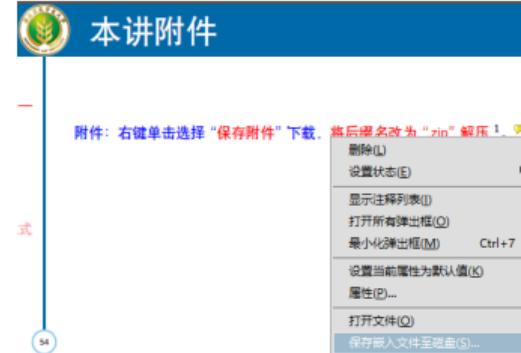
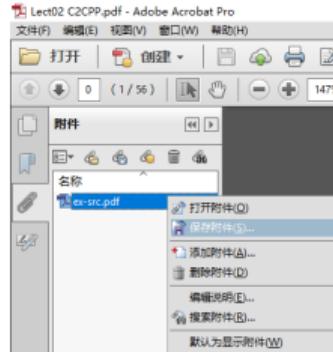


# 本讲附件

| 附件

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP  
Nine, G.

附件：右键单击该链接，选择“保存附件”下载，将后缀名改为“.zip”解压<sup>3 4</sup>。



<sup>3</sup>请退出全屏模式后点击该链接。

<sup>4</sup>以 Adobe Acrobat Reader 为例。



### ▶ 将不同数据类型组合成一个整体

[[]] 语法 struct | 结构类型名 | |  
数据类型 1 成员变量 1; | 数据类  
型 2 成员变量 2; | … | 数据类型  
n 成员变量 n; ;

### ▶ 结构体的实际大小

有时 ≠ 结构体内部成员变量所占内存之和

[[]] 字节数 sizeof(StuNode) =  
| 40 |;  
2\*sizeof(int)  
+sizeof(char)\*(20+| 12 |) =  
| 40 |;

定义 struct StuNode int ID;  
char name[20]; char gen-  
der[12]; int age; ;

定义 struct StuNode int ID;  
char name[20]; char gen-  
der[12]; int age; ;





## ▶ 结构体的实际大小

有时  $\neq$  结构体内部成员变量所占内存之和

[1] 字节数 sizeof(StuNode) =

| 44 |;

2\*sizeof(int)

+sizeof(char)\*(20+| 13 |)

| 41 |;

[1] 字节数 sizeof(StuNode) =

| 44 |;

2\*sizeof(int)

+sizeof(char)\*(20+| 15 |)

| 43 |;

定义 struct StuNode int ID;  
char name[20]; char gen-  
der[13]; int age; ;

定义 struct StuNode int ID;  
char name[20]; char gen-  
der[15]; int age; ;





## ► 结构体的实际大小

SumSize = 结构体内部成员变量所占内存之和

L = 结构体内基本类型长度的最大值

SumSize % L == 0 ? SumSize : (L \* (SumSize / L) % + L)

[||] 字节数 sizeof(StuNode) =  
| 44 |;

2\*sizeof(int)  
+sizeof(char)\*(20+| 16 |) =  
| 44 |;

定义 struct StuNode int ID;  
char name[20]; char gen-  
der[16]; int age; ;

设置内存对齐方式: #pragma pack(value)

#pragma pack(1)





## ► 结构体的使用

[[]] 语法 struct | 结构类型名结构变量名  
| = | 成员 1 初始值 |, | 成员 2 初始值 |,  
|...|, | 成员 n 初始值 | ;

定义 struct StuNode int  
ID; char name[20]; char  
gender[16]; int age; ;

声明 struct StuNode myNode0; struct StuNode myNode =  
101, "Tom", "male", 35;





## ► 结构体中可以有函数

语法 struct | 结构类型名 | | 数  
据类型 1 成员变量 1|; |...|; | 数据类  
型 n 成员变量 n|; |函数返回类型函  
数名 (参数列表)|; ;

```
// 例 03-01: ex03-01.cpp
// 结构体定义及其中函数的实现

struct StuNode
{
    int ID;
    char name[20];
    char gender[16];
    int age;
    void Set(int id, char *n, char *g, int a);
};

// 赋值函数, 为结构体中的数据赋值
void StuNode::Set(int id, char *n, char *g, int a)
{
    ID = id;
    strcpy(name, n);
    strcpy(gender, g);
    age = a;
}
```





## ▶ 结构体中可以有函数

```
// 例 03-02: ex03-02.cpp
// 结构体的定义

struct StuNode
{
    int ID;
    char name[20];
    char gender[16];
    int age;

    // 结构体中的赋值函数
    void Set(int id, char *n, char *g, int a)
    {
        ID = id;
        strcpy(name, n);
        strcpy(gender, g);
        age = a;
    }
};
```





## ▶ 新结构体的使用

```
struct 结构类型名 {  
    数据类型 1 成员变量 1;  
    数据类型 2 成员变量 2;  
    ...;  
    数据类型 n 成员变量 n;  
    函数返回类型 函数名 (参数列表);  
};  
  
[struct] 结构类型名 变量名;
```

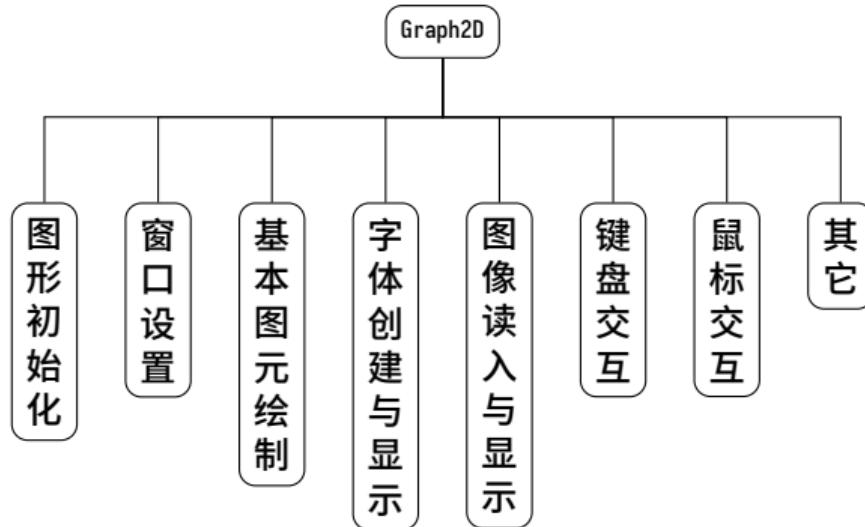
C++ 中可以省略

```
// 例 03-03: ex03-03.cpp  
  
struct StuNode  
{  
    int ID;  
    char name[20];  
    char gender[16];  
    int age;  
  
    // 赋值函数, 以同类型的对象为参数  
    void Set(StuNode inNode)  
    {  
        ID = inNode.ID;  
        strcpy(name, inNode.name);  
        strcpy(gender, inNode.gender);  
        age = inNode.age;  
    }  
};  
int main()  
{  
    StuNode myNode0;  
}
```



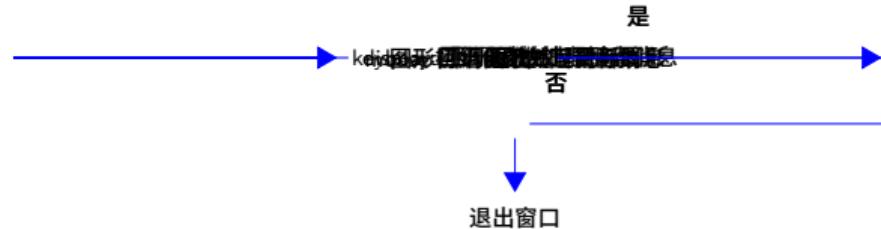


### ► Graph2D 功能结构<sup>5</sup>



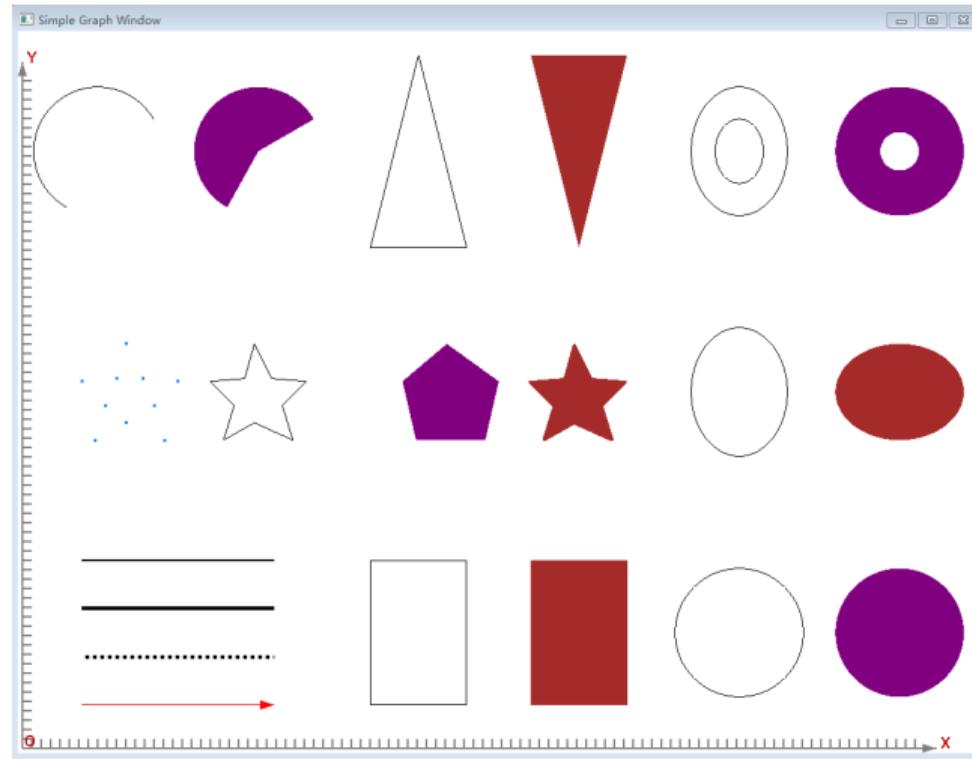


### ► Graph2D 运行机制





### ► Graph2D 坐标系统（二维笛卡尔右手坐标系）





### ▶ 图形库的安装

#### ▶ Graph2D 库包含三个文件

- ▶ graph2d.h
- ▶ libgraph2d.a
- ▶ graph2d.dll

#### ▶ 安装

- ▶ 手动安装：参见 Readme.txt 文件
- ▶ 自动安装：运行 InstallGraph2D.exe





### ► 图形库在 Code::blocks 中的配置

- 在菜单 Settings>Compiler>Link settings 中添加 libgdi32、libopengl32、libglu32、libfreeglut 和 libgraph2d

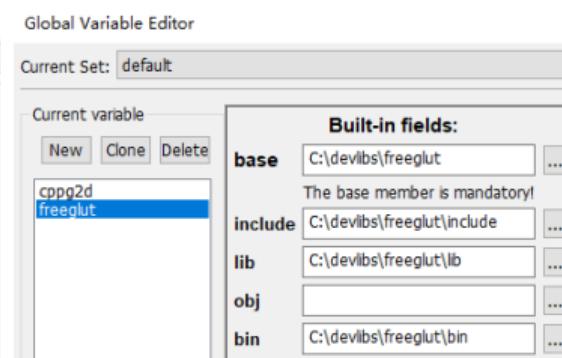
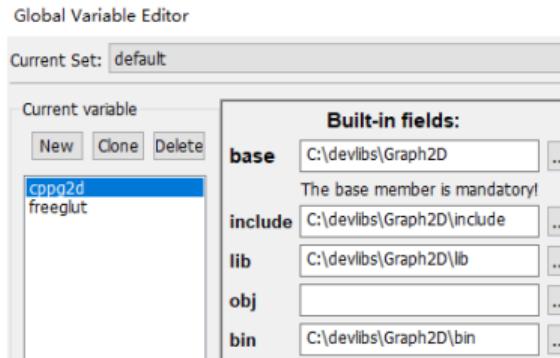
设置链

[scale=.35,unit=1mm]chap03/03graph2dCBsetting



▶ Graph2D 图形库 Code::Blocks 开发向导<sup>6</sup>

- ▶ 将 devLibs 文件夹及其中所有文件**保持目录结构** 拷贝到任意位置 (如 C 盘根目录)
- ▶ 将 Graph2D 文件夹和 config.script 复制到 Code::Blocks 安装目录的 wizard 文件夹<sup>7</sup>
- ▶ 在菜单 Settings>Global variables... 中添加 **cppg2d**和**freeglut**环境变量<sup>8</sup>



<sup>6</sup>详见: <https://gitee.com/registor/Graph2D>

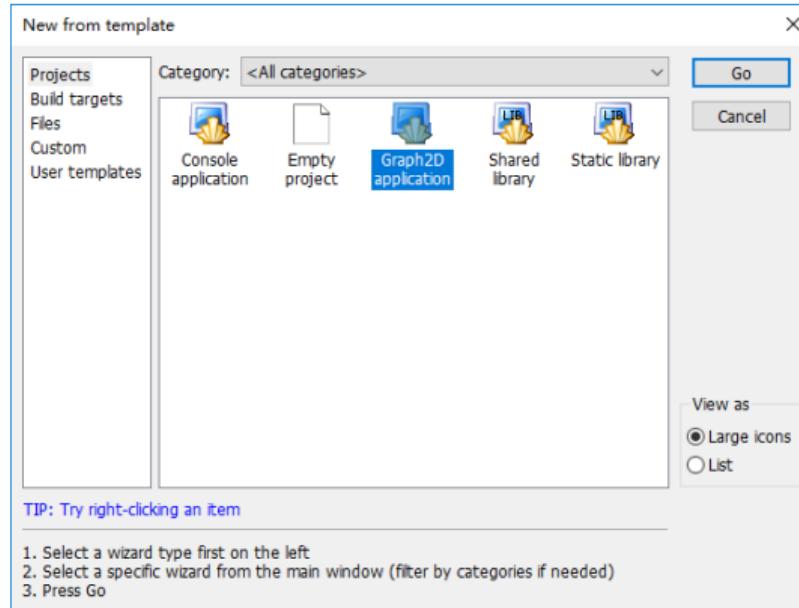
<sup>7</sup>如: C:\Program Files (x86)\CodeBlocks\share\CodeBlocks\templates\wizard

<sup>8</sup>注意要与 devLibs 在自己硬盘上的路径一致



### ► Graph2D 图形库 Code::Blocks 开发向导<sup>9</sup>

#### ► 在向导中选择 Graph2D application



<sup>9</sup>详见: <https://gitee.com/registor/Graph2D>



## ► Graph2D 骆驼式命名法

- ▶ 第一个单词小写
- ▶ 后面的单词首字母大写
- ▶ 示例: showCoordinate

---

```
// 例 03-35: ex03-35.cpp
// C++ 基本绘图函数的演示

#include <Graph2D.h>

using namespace graph;

void display()
{
    circle(512, 384, 100);
    putText(480, 384, "Hello World!");
}

int main(int argc, char *argv[])
{
    initGraph(display);
    return 0;
}
```

---

[scale=.15,unit=1mm]chap03/04gra





## ▶ Graph2D 简单图形库的使用

---

```
// 例 03-03-01: ex03-03-01.cpp
```

```
#include <Graph2D.h>
using namespace graph;
struct myRect
{
    double x;
    double y;
    double width;
    double height;

    void Draw()
    {
        setColor(RED);
        fillRectangle(x, y,
                      x + width, y + height);
    }

    void Move()
    {
        x += 10;
        y += 10;
    }
};
```

---

---

```
// 例 03-03-02: ex03-03-02.cpp
```

```
myRect r = {10.0, 10.0,
            200.0, 100.0};

void display()
{
    r.Draw();
}

void keyboard(unsigned char key)
{
    r.Move();
}

int main(int argc, char *argv[])
{
    initGraph(display, keyboard);

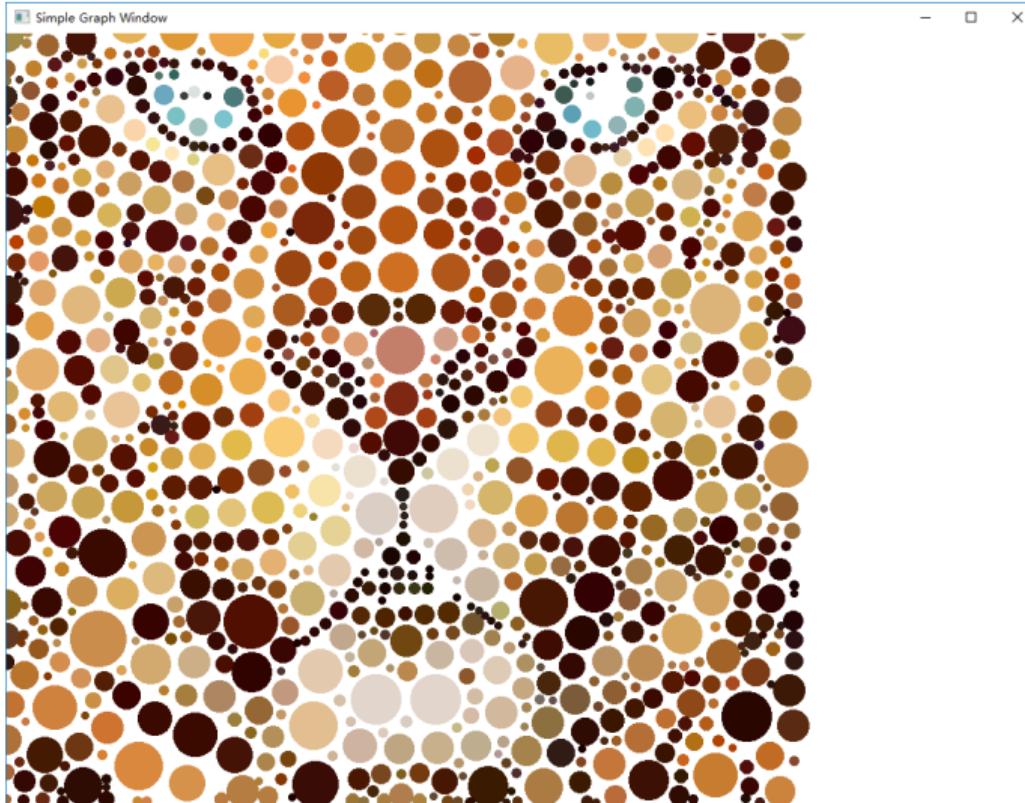
    return 0;
}
```

---





### ► Graph2D 简单图形库的使用





### ▶ 面向过程程序设计

- ▶ 程序模块由函数（过程）组成
- ▶ 数据与函数分离，通过参数传递给函数

### ▶ 面向对象程序设计

- ▶ 程序模块由类组成
- ▶ 函数与数据是一个整体（封装）
- ▶ 访问限制
- ▶ 继承与多态性





### ► 一个包含函数的结构体

```
class 类名
{
public: //存取权限控制符
    公有数据成员或公有函数成员的定义;
protected:
    保护数据成员或保护函数成员的定义;
private:
    私有数据成员或私有函数成员的定义;
};
```

类定义结束符





### ► 实例

```
// 例 03-13: ex03-13.cpp
// 定义类 Stash, 对其中成员（数据或函数）
// 根据需要赋予不同的权限

class Stash
{
private:
    int size; // Number of a spaces
    int next; // Next empty space
    int *a; // Dynamically allocated arrays
    void inflate(int increase);
public:
    void initialize();
    void cleanup();
    int add(int element);
    int fetch(int index);
    int count();
};
```





## ▶ 成员变量（数据成员）

```
// 例 03-14: ex03-14.cpp
// 数据成员为结构体类型的类

struct Record
{
    char name[20];
    int score;
};

class Team
{
private:
    int num; //基本类型
    Record *p; //构造类型
};
```

```
// 例 03-15: ex03-15.cpp
// 关于类中数据成员定义为类类型的示例

class Team; //已定义的类
class Grade
{
    Team a; // 已定义的类类型
    Grade *p; // 正在定义的类类型定义指针成员
    Grade &r; // 正在定义的类类型定义引用成员
    Grade b; // 错误! 使用了未定义完整的类
};
```

数据成员描述了类对象所包含的数据类型，数据成员的类型可以是 C++**基本数据类型**，也可以是**构造数据类型**。





### ▶ 成员函数

- ▶ 对类中的数据成员实施的操作
- ▶ 消息传递

---

```
// 例 03-16: ex03-16.cpp
// 类 Stash 成员函数 add 的实现

// 在 a 的尾部添加元素 element, 若 a 中已无空间, 则调用
// inflate 函数增加 increment 个单位
// 函数返回 a 中最后一个元素的 index

int Stash::add(int element)
{
    if(next >= size)
        inflate(increment);
    a[next] = element;
    next++;
    return (next - 1);
}
```

---





### ▶ 成员函数

- ▶ 既可以放在类外定义，也可放在类中定义（内联函数）

```
// 例 03-17: ex03-17.cpp
// 类中成员（数据或函数）默认权限为 private
// 成员函数也可以在类定义中实现

class Stash
{
    int size; // Number of a spaces
    int next; // Next empty space
    int *a; // Dynamically allocated arrays
    void inflate(int increase);

public:
    int add(int element)
    {
        if(next >= size)
            inflate(increment);

        a[next] = element;
        next++;
        return(next - 1);
    }
};
```





### ▶ 成员函数

#### ▶ 成员函数中可以使用该类定义变量

```
// 例 03-18: ex03-18.cpp
// 类类型的变量作为函数参数, 返回值的演示

class Clock
{
private:
    int H, M, S;
public:
    Clock AddTime(Clock C2)      // 形参为 Clock 类型的变量
    {
        Clock T;                // 函数体中定义了 Clock 类型的变量
        ...
        return T;                // 返回类型为 Clock 类型
    }
};
```





### ► 与新结构体的区别与联系

- The only difference between a **class** and a **struct** is that by default all members are **public** in a struct and **private** in a class.





- ▶ 类是包含函数的自定义数据类型，它**不占内存**，是一个抽象的“虚”体
- ▶ 使用已定义的类建立对象与用数据类型定义变量一样
- ▶ 对象建立后，对象**占据内存**，变成了一个“实”体
- ▶ 类与对象的关系就像数据类型与变量的关系一样





## ▶ 对象的建立

类名 对象名;

## ▶ 对象的使用

对象名 . 属性;

对象名 . 成员函数名 (参数);

```
// 例 03-19: ex03-19.cpp
// 创建 Stash 的实例, 调用相关函数进行处理
```

```
int main()
{
    Stash s;

    s.initialize();

    for(int i = 0; i < 100; i++)
        s.add(i);

    for(int j = 0; j < s.count(); j++)
        cout << "No." << j << " = "
            << s.fetch(j) << endl;

    s.cleanup();
}
```





- ▶ 成员的存取控制
- ▶ 存取控制属性

- ▶ 公有类型 (**public**): 适用于完全公开的数据
- ▶ 私有类型 (**private**): 适用于不公开的数据
- ▶ 保护类型 (**protected**): 适用于半公开的数据

存取属性	意义	可存取对象
<b>public</b>	公开 (公有)	该类及基所有对象
<b>protected</b>	保护	该类及其子类成员
<b>private</b>	私有	该类成员





## ▶ 成员的存取控制

---

```
// 例 03-20: ex03-20.cpp
// 类中成员（数据或函数）protect 权限的示例
```

```
class Stash
{
    int size; // Number of a spaces
    int next; // Next empty space
    int *a; // Dynamically allocated arrays
protected:
    void inflate(int increase);
public:
    void initialize();
    void cleanup();
    int add(int element);
    int fetch(int index);
    int count();
};
```

---

---

```
Stash s;
int len = s.size(); pzd56
s.inflate(100); pzd56
s.initialize(); pzd52
```

---





### ▶ 句柄类

- ▶ 公有接口
- ▶ 私有指针指向实际数据

```
#ifndef HANDLE_H_INCLUDED
#define HANDLE_H_INCLUDED

class Handle{
public:
    void initialize();
    void cleanup();
    int read();
    void change(int);
private:
    // 前向引用说明
    struct Inner;
    // 指向数据成员
    Inner * pointer;
};

#endif // HANDLE_H_INCLUDED
```

```
#include "handle.h"

struct Handle::Inner
{
    int i;
};

void Handle::initialize()
{
    pointer = new Inner;
    pointer->i = 0;
}

void Handle::cleanup()
{
    delete pointer;
}

int Handle::read()
{
    return pointer->i;
}

void Handle::change(int x)
{
    pointer->i = x;
}
```





### ► 初始化和清理工作

```
// 例 03-21: ex03-21.cpp
// 用类定义栈类型

class ch_stack
{
private:
    char *s; // 栈的内容保存在 s 中
    int tp; // 栈顶指示器, 栈空为 -1
    int size;
public:
    void init();
    void release();
    void push(char c);
    char pop();
    ...
};
```

```
ch_stack s;
s.init();      初始化
...
s.release();   清理
```

若无构造和析构函数, 需显式调用函数





```
ch_stack s;  
s.init();  
...  
s.release();
```

```
void ch_stack::init()  
{  
    s = (char *)malloc(STACK_INIT_SIZE);  
    tp = EMPTY;  
    size = STACK_INIT_SIZE;  
}
```

```
void ch_stack::release()  
{  
    if(size != 0)  
    {  
        free(s);  
    }  
}
```

八





## ▶ 构造函数

- ▶ 系统自动调用的初始化函数
- ▶ 函数名与类名相同
- ▶ 无返回值
- ▶ 公有函数

## ▶ 委托构造

- ▶ 委托其他构造函数完成初始化

## ▶ 析构函数

- ▶ 析构函数没有返回值
- ▶ 析构函数没有任何参数，不能被重载
- ▶ 析构函数在对象消失时由系统自动调用

## ▶ 拷贝构造函数

- ▶ 在建立新对象时将已存在对象的数据成员值拷贝给新对象
- ▶ 拷贝构造函数的形参是本类的对象的引用

## ▶ 浅拷贝和深拷贝

- ▶ 在默认拷贝构造函数中，直接将原对象的数据成员值依次拷贝给新对象中对应的数据成员
- ▶ 对于需要动态分配内存的场合，浅拷贝会出错





## ► 构造函数

- ▶ 系统自动调用的初始化函数
- ▶ 函数名与类名相同
- ▶ 无返回值
- ▶ 公有函数

---

```
ch_stack s;
s.init();
...
```

---

---

```
ch_stack s;
...
...
```

---

---

```
class ch_stack
{
    char *s;
    int tp;
    int size;
public:
    ch_stack()
    {
        init();
    }
    void init();
    ...
};
```

---

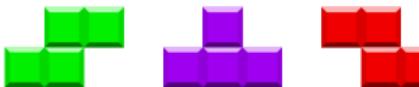
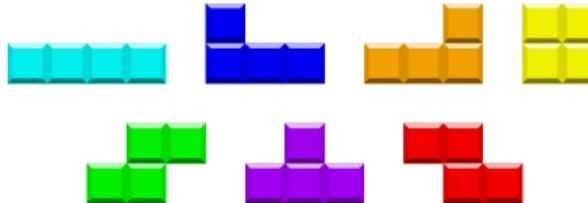
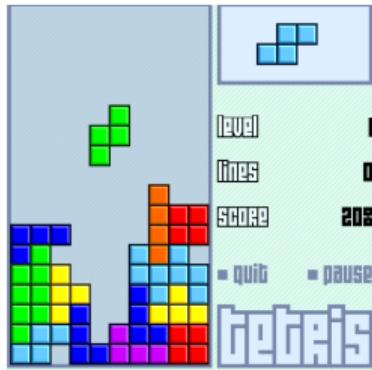




# 实例：俄罗斯方块

| 构造与析构

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP  
Nine, G.



## ▶ 类

CTetromino

{

类

属性：颜色，四个方块的布局，大小，位置，速度，…  
行为：平移，旋转，加速，碰撞检测，显示，…  
};

CTetromino I, J, L, O, S, T, Z;

对象





```
// 例 03-23: ex03-23.cpp
#include "Cie2DGraph.h"
using namespace Cie2DGraph;
class CTetromino
{
    unsigned long color;
    int vxInit, vyInit;
    int vx[4], vy[4];
    int size;
    float speed;
public:
    CTetromino();
    CTetromino(char inType,
               unsigned long inColor,
               int inSize = 20,
               float inSpeed = 1);
    void Translate(int xOffset, int yOffset);
    void Draw();
    ...
};

```

构造函数可以重载

带默认形参的构造函数

人





# 构造函数

# 构造与析构

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP

Nine, G.

```
// 例 03-24: ex03-24.cpp
// 类 CTetromino 默认构造函数的实现

CTetromino::CTetromino()
{
    size = 20;
    speed = 1;
    color = 0xFF0000;
    vxInit = 600 / (2 * size) - 2;
    vyInit = 800 / size - 1;
    vx[0] = 0; vy[0] = 0;
    vx[1] = 1; vy[1] = 0;
    vx[2] = 2; vy[2] = 0;
    vx[3] = 3; vy[3] = 0;
}
```

```
// 例 03-25: ex03-25.cpp
// 类 CTetromino, 有参数构造函数实现

CTetromino::CTetromino(char inType,
                      unsigned long inColor,
                      int inSize, float inSpeed)
{
    size = inSize;
    speed = inSpeed;
    color = inColor;
    vxInit = 600 / (2 * size) - 2;
    vyInit = 800 / size - 1;
    switch (inType)
    {
        case 'I': case 'i':
            vx[0] = 0; vy[0] = 0;
            vx[1] = 1; vy[1] = 0;
            vx[2] = 2; vy[2] = 0;
            vx[3] = 3; vy[3] = 0;
            break;
        .
        .
        case 'T': case 't':
            vx[0] = 0; vy[0] = 0;
            vx[1] = 1; vy[1] = 0;
            vx[2] = 2; vy[2] = 0;
            vx[3] = 1; vy[3] = 1;
            break;
        default:
            vx[0] = 0; vy[0] = 0;
            vx[1] = 1; vy[1] = 0;
            vx[2] = 2; vy[2] = 0;
            vx[3] = 3; vy[3] = 0;
            break;
    }
}
```





## ▶ 初始化列表实现

- ▶ 初始化列表唯一
- ▶ 保证参数一致性

```
// 例 03-25-01-delegating : ex03-25-01.cpp
#include <iostream>

using namespace std;

class X
{
public:
    X(int _a, int _b, int _c):a(_a), b(_b), c(_c)
    {cout << "X(int, int, int)" << endl;}
    X(int _a, int _b):X(_a, _b, 0)
    {cout << "X(int, int)" << endl;}
    X(int _a):X(_a, 0, 0)
    {cout << "X(int)" << endl;}
    X():X(1, 1)
    {c = 1; cout << "X()" << endl;}
public:
    int a, b, c;
};

int main(){
    cout << "1: " << endl;
    X one(1, 2, 3);
    cout << "2: " << endl;
    X two(1, 2);
    cout << "3: " << endl;
    X three(1);
    cout << "4: " << endl;
    X four;
}
```





- ▶ 对象消失时的清理工作（如释放内存单元）
- ▶ 定义

```
~类名 ();  
  
// 例 03-26: ex03-26.cpp  
// 演示类中析构函数的定义及实现  
  
class ch_stack  
{  
    char *s;  
    int tp;  
    int size;  
public:  
    ~ch_stack()  
    {  
        release();  
    }  
    void release();  
    ...  
};
```





- ▶ 析构函数没有返回值
- ▶ 析构函数没有任何参数，不能被重载
- ▶ 析构函数在对象消失时由系统自动调用

---

```
// 例 03-27: ex03-27.cpp
int main
{
    ch_stack s;
    s.init();
    ...
    s.release();
    return 0;
}
```

---

---

```
// 例 03-28: ex03-28.cpp
int main
{
    ch_stack s;
    ...
    return 0;
}
```

---

八





- ▶ 在新建对象时将已有对象的数据成员值拷贝给新对象
- ▶ 拷贝构造函数的形参是**本类的对象的引用**

---

类名 (类名& 对象名)  
{  
:  
};

---

---

// 例 03-29: ex03-29.cpp  
// 类默认构造函数的定义

```
class ch_stack
{
    char *s;
    int tp;
    int size;
public:
    ch_stack();
    ch_stack(ch_stack &s0bj);
    ...
};
```

---





### ▶ 拷贝构造函数调用时机

- ▶ 用类的一个对象去初始化该类的另一个对象时
- ▶ 调用函数中，将对象作为实参传递给形参时
- ▶ 如函数返回值是类的对象，函数执行完成后将返回值返回时
  
- ▶ 在默认的拷贝构造函数中，是直接将原对象的数据成员值依次拷贝给新对象中对应的数据成员
  
- ▶ 在对于需要动态分配内存的场合，默认的拷贝构造函数会出错





### ► 浅拷贝

```
// 例 03-30: ex03-30.cpp
// 类拷贝构造函数的实现（浅拷贝）

ch_stack::ch_stack(ch_stack & s0bj)
{
    size = s0bj.size;
    tp = s0bj.tp;
    s = s0bj.s;
}
```

// 例 03-31: ex03-31.cpp
// 类拷贝构造函数的效果演示

```
int main()
{
    ch_stack s;

    for(int i = 0; i < 7; i++)
        s.push('a' + i);

    // 自动调用拷贝构造函数
    ch_stack sCopy(s);

    while(!s.empty()) s.pop();

    for(int i = 0; i < 7; i++)
        s.push('1' + i);

    while(!sCopy.empty)
        cout << sCopy.pop();
    cout << endl;
}
```





## ► 深拷贝

```
// 例 03-32: ex03-32.cpp
// 类拷贝构造函数的实现（深拷贝）

ch_stack::ch_stack(ch_stack & s0bj)
{
    size = s0bj.size;
    tp = s0bj.tp;
    s = NULL;
    if (size != 0)
    {
        s = (char *)malloc(size);
        for (int i = 0; i <= tp; i++)
            s[i] = s0bj.s[i];
    }
}
```

// 例 03-31: ex03-31.cpp  
// 类拷贝构造函数的效果演示

```
int main()
{
    ch_stack s;

    for(int i = 0; i < 7; i++)
        s.push('a' + i);

    // 自动调用拷贝构造函数
    ch_stack sCopy(s);

    while(!s.empty()) s.pop();

    for(int i = 0; i < 7; i++)
        s.push('1' + i);

    while(!sCopy.empty)
        cout << sCopy.pop();
    cout << endl;
}
```





### ► 相同点

- ▶ 系统自动建立默认构造函数和析构函数
- ▶ 系统自动调用
- ▶ 无返回值
- ▶ 必须是公有函数

### ► 相异点

- ▶ 功能不同
- ▶ 函数名不同
- ▶ 构造函数可被重载，也可带（默认）形参，析构函数不可以





### ▶ 对象指针指向对象存放的地址

### ▶ 定义与使用

---

类名 \* 对象指针名；

对象指针名 -> 数据成员；

对象指针名 -> 数据函数；

---

### ▶ 优点

▶ 地址传递

▶ 使用对象指针效率高





- ▶ 对象引用与被引用对象共享地址空间
- ▶ 定义与使用

---

类名 **6** 对象指针名；  
对象指针名 **7** 数据成员；  
对象指针名 **8** 数据函数；

---

- ▶ 示例

```
CTetromino tetrObj('I', 0xFF0000, 30, 0.5);  
CTetromino &tetrRef=tetrObj;
```

---





- ▶ 以与对象为元素的数组
- ▶ 定义与初始化

类名 对象数组名 [n];  
初始化：数组名 [n] = {类名 (初始值1, 初始值2, …),  
                  类名 (初始值1, 初始值2, …),  
                  ⋮  
                  类名 (初始值1, 初始值2, …)  
};

- ▶ 示例

```
CTetromino tetrObj[2] = {CTetromino('I', 0xFF0000, 30, 0.5),  
                          CTetromino('T', 0x00FF00, 30, 0.5)  
};
```





- ▶ 动态建立的对象
- ▶ 定义与初始化

```
类名 对象指针名 = new 类名 (初始值1, 初始值2, ...);  
delete 对象指针名;
```

```
类名 对象指针名 = new 类名 [n];  
delete [] 对象指针名;
```

- ▶ 示例

```
CTetromino *tetrObj;  
tetrObj = new CTetromino('I', 0xFF0000, 30, 0.5);
```

```
CTetromino *tetrObj;  
tetrObj = new CTetromino
```

```
tetrObj->Draw();  
tetrObj[0].Draw();
```



► 系统预定义指针，指向当前对象 (即**当前对象的地址**)

```
// 例 03-33-01: ex03-33-01.cpp
// 构造函数重载, 理解 this 指针

class Point2D
{
    float x, y;
public:
    Point2D()
    {
        x = y = 0;
    }
    Point2D(int x, int y)
    {
        this->x = x;
        this->y = y;
    }

    void Output()
    {
        cout << this << endl;
    }
};
```

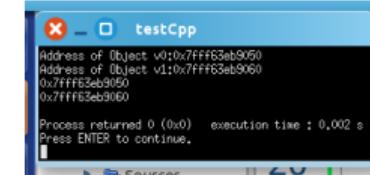
```
// 例 03-33-02: ex03-33-02.cpp
// this 指针的值及其意义
```

```
int main()
{
    Point2D v0, v1;

    cout << "Address of Object v0:"
        << &v0 << endl;
    cout << "Address of Object v1:"
        << &v1 << endl;

    v0.Output();
    v1.Output();

    return 0;
}
```





## ► 利用 this 指针明确成员函数当前操作的数据成员是属于哪个对象

```
// 例 03-33-01: ex03-33-01.cpp  
// 构造函数重载, 理解 this 指针
```

```
class Point2D  
{  
    float x, y;  
public:  
    Point2D()  
    {  
        x = y = 0;  
    }  
    Point2D(int x, int y)  
    {  
        this->x = x;  
        this->y = y;  
    }  
    void Output()  
    {  
        cout << this << endl;  
    }  
};
```

```
// 例 03-34: ex03-34.cpp
```

```
int main()  
{  
    Point2D v0, v1;  
    ...  
    return 0;  
}
```

```
CPoint2D v0, v1;
```

```
0x011B14DE lea    ecx,[v0]  
0x011B14E1 call   CPoint2D::CPoint2D  
0x011B14E6 lea    ecx,[v1]  
0x011B14E9 call   CPoint2D::CPoint2D
```





# this 指针

## | 对象的使用

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP  
Nine, G.

```
int main()
{
    Point2D v0;
    Point2D v1;
    .
    .
    return 0;
}
```

ecx = 002bf964

入

6	16	bitheight=1	[]10
bitheight=1	[]10	this	→ 0x002bf874
6...		bitheight=1	[]10
bitheight=1	[]10	v1	→ 0x002bf954
6y	bitheight=1	[]10	0x002bf958
6	bitheight=1	[]10	0x002bf95C
6	bitheight=1	[]10	0x002bf960
bitheight=1	[]10	v0	→ 0x002bf964
6y	bitheight=1	[]10	0x002bf968
6		bitheight=1	[]10
		bitheight=1	[]10





# this 指针

## | 对象的使用

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP  
Nine, G.

```
int main()
{
    Point2D v0;
    Point2D v1;
    .
    .
    return 0;
}
```

ecx = 002bf964

1

6	bitheight=1	10	bitheight=1	10
bitheight=1	10	this	→	0x002bf874
6...				...
bitheight=1	10	v1	→	0x002bf954
6y	bitheight=1	10	0x002bf958	
6	bitheight=1	10	0x002bf95C	
6	bitheight=1	10	0x002bf960	
bitheight=1	10	v0	→	0x002bf964
60	bitheight=1	10	0x002bf968	
6			bitheight=1	10
			bitheight=1	10





# this 指针

## 对象的使用

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP  
Nine, G.

```
int main()
{
    Point2D v0;
    Point2D v1;
    .
    .
    return 0;
}
```

ecx = 0x002bf954

bitheight=1	bitheight=1	bitheight=1	bitheight=1
6	16	bitheight=1	bitheight=1
bitheight=1	bitheight=1	0x002bf874	0x10
6...	bitheight=1	0x002bf954	...
bitheight=1	bitheight=1	0x002bf958	0x10
6y	bitheight=1	0x002bf95C	bitheight=1
6	bitheight=1	0x002bf960	bitheight=1
bitheight=1	bitheight=1	0x002bf964	0x10
60	bitheight=1	0x002bf968	bitheight=1
6	bitheight=1	0x002bf96C	bitheight=1
6	bitheight=1	0x002bf96E	bitheight=1





# this 指针

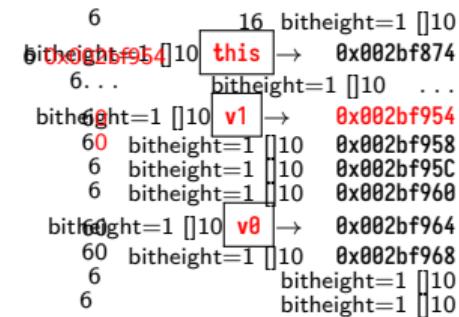
## | 对象的使用

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP  
Nine, G.

```
int main()
{
    Point2D v0;
    Point2D v1;
    ...
    return 0;
}
```

ecx = 0x002bf954

1



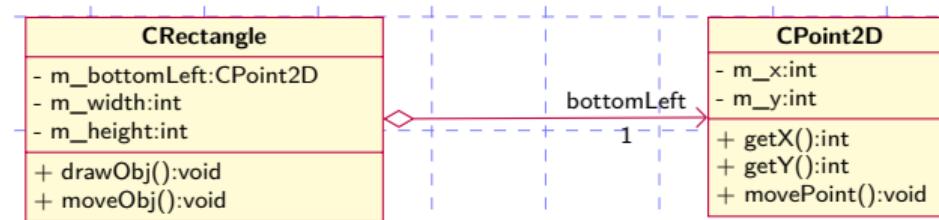


▶ 组合类：类中含有其它类的对象作为成员

▶ 组合对象

▶ 组合类的定义

▶ 先定义成员类，再定义组合类





## ► CPoint2D 类

```
// 例 03-36: ex03-36-point.h
// CPoint2D 类的定义
#ifndef CPOINT2D_H
#define CPOINT2D_H
class CPoint2D
{
public:
    CPoint2D();
    CPoint2D(int x, int y);
    CPoint2D(CPoint2D &pt);
    int getX(){
        return m_x;
    }
    int getY(){
        return m_y;
    }
    void movePoint();
private:
    int m_x;
    int m_y;
};
#endif // CPOINT2D_H
```

```
// 例 03-36: ex03-36-point.cpp
// 重载构造函数及不同构造函数的实现
#include <iostream>
#include "point.h"
using namespace std;

// 默认构造函数
CPoint2D::CPoint2D(){
    m_x = 0;
    m_y = 0;
}

// 带参构造函数
CPoint2D::CPoint2D(int x, int y):
    m_x(x), m_y(y){
}

// 拷贝构造函数
CPoint2D::CPoint2D(CPoint2D &pt){
    m_x = pt.m_x;
    m_y = pt.m_y;
}
void CPoint2D::movePoint(){
    m_x += 10;
    m_y += 10;
}
```





## ► CRectangle 类

```
// 例 03-36: ex03-36.cpp
// CRectangle 类的定义

#ifndef CRECTANGLE_H
#define CRECTANGLE_H
#include "point.h"
class CRectangle{
public:
    CRectangle();
    CRectangle(int x, int y, int width, int height);
    CRectangle(CPoint2D &pt, int width, int height);
    void drawObj();
    void moveObj();
private:
    CPoint2D m_bottomLeft;
    int m_width;
    int m_height;
};

#endif // CRECTANGLE_H
```





## ► CRectangle 类

```
// 例 03-36: ex03-36.cpp
// CRectangle 类的实现

#include <Graph2D.h>
#include "rectangle.h"
using namespace graph;
// 默认构造函数
CRectangle::CRectangle():m_bottomLeft(0, 0)
{
    m_width = 0;
    m_height = 0;
}
// 带参构造函数
CRectangle::CRectangle(int x, int y, int width, int height):
    m_bottomLeft(x, y), m_width(width), m_height(height)
{
}
// 带参构造函数
CRectangle::CRectangle(CPoint2D &pt, int width, int height):
    m_bottomLeft(pt), m_width(width), m_height(height)
{
}

void CRectangle::drawObj()
{
    setColor(RED);
    fillRectangle(m_bottomLeft.getX(), m_bottomLeft.getY(),
                  m_bottomLeft.getX() + m_width,
                  m_bottomLeft.getY() + m_height);
}
void CRectangle::moveObj()
{
    m_bottomLeft.movePoint();
}
```

初始化列表





## ▶ 使用类对象

```
// 例 03-36: ex03-36.cpp
// C++ 基本绘图函数的演示

#include <Graph2D.h>
#include "rectangle.h"
using namespace graph;
CRectangle rect(100, 100, 200, 100);
void display()
{
    rect.drawObj();
}
void keyboard(unsigned char key)
{
    rect.moveObj();
}
int main(int argc, char *argv[])
{
    initGraph(display, keyboard);
    return 0;
}
```

使用 graph 名字空间

定义 rect 对象，并初始化

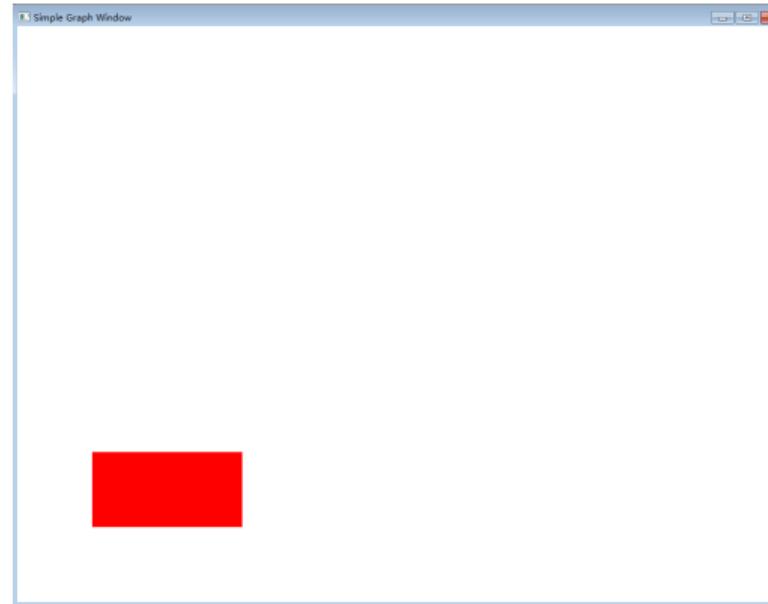
调用成员函数绘图

调用成员函数移动

函数指针 (回调函数)



## ▶ 使用类对象





- ▶ 非成员函数 (或外部函数) 访问私有成员 (保护成员)

- ▶ 设置访问控制属性为**public**
- ▶ 破坏了类的封装性和隐藏性

- ▶ 友元

- ▶ 友元函数
- ▶ 友元类

- ▶ 友元不是类的成员，但能够访问类中被隐蔽的信息





## ▶ 友元函数

```
// 例 ex03-38: ex03-38.cpp
class A
{
private:
    int i ;
    void MemberFun(int) ;
    friend void FriendFun(A * , int);
} ;

void FriendFun(A * ptr , int x)
{
    ptr -> i = x ;
}
void A::MemberFun( int x )
{
    i = x ;
}
```

友元函数的声明

友元函数的定义

人





## ▶ 友元函数

```
// 例 ex03-39: ex03-39.cpp
class A
{
private:
    int i ;
    void MemberFun(int) ;
    friend void FriendFun(A * , int) ;
} ;

friend void A::FriendFun(A * ptr , int x)
{
    ptr->i = x ;
}
void A::MemberFun( int x )
{
    i = x ;
}
```

friend必须出现在类中，  
定义时不能使用域操作





## ▶ 友元函数

```
// 例 03-40: ex03-40.cpp
// 类友元函数的定义及实现

class A
{
private:
    int i ;
    void MemberFun(int);
    friend void FriendFun(A * , int);
};

void FriendFun(A * ptr , int x)
{
    i = x;
}
void A::MemberFun(int x)
{
    i = x;
}
```

friend不能直接访问成员变量  
必须通过参数传递访问私有成员

人





## ► 求两点间的距离

// 例 03-41: ex03-41.cpp

// 类友元函数的定义

```
//point2d.h
#ifndef POINT2D_INCLUDED
#define POINT2D_INCLUDED
class CPoint2D
{
private:
    float x, y;
public:
    CPoint2D()
    {
        x = y = 0;
    }
    CPoint2D(float x, float y)
    {
        this->x = x;
        this->y = y;
    }
    friend float Distance(CPoint2D p1,
                          CPoint2D p2);
};

#endif // POINT2D_INCLUDED
```

// 例 ex03-41-main: ex03-41-main.cpp

```
#include <iostream>
#include <cmath>
#include "point2d.h"
using namespace std;
float Distance(CPoint2D p1,
                CPoint2D p2){
    float dx = p1.x - p2.x;
    float dy = p1.y - p2.y;
    return sqrt(dx * dx + dy * dy );
}
int main(){
    CPoint2D v0(1, 1), v1(4, 5);
    cout << Distance (v0, v1) << endl;
    return 0;
}
```

不用通过对象调用友元函数





- ▶ 友元函数不能直接访问成员变量，但可以通过参数传递操作对象中的所有成员
- ▶ 友元函数在类中声明，但在类外定义时勿需域操作符
- ▶ 友元函数调用时勿需通过对象





## ▶ 前向引用声明

// 例 03-42: ex03-42.cpp

```
class B; // 前向声明
class A
{
public:
    void funA(B b);
};

class B
{
public:
    void funB(A b);
};
```

```
// 例 03-43: ex03-43.cpp
// 类的前向声明, 参数为引用型的函数示例

class Matrix; // 类的前向声明
class Vector{
    float e[4];
    ...
    friend Vector Multi(const Matrix&, const Vector&);
};
class Matrix{
    Vector v[4];
    ...
    friend Vector Multi(const Matrix&, const Vector&);
};
Vector Multi(const Matrix &m, const Vector &v){
    Vector r;
    for(int i = 0; i < 4; i++){ //r[i]=m[i]*v;
        r.e[i] = 0;
        for(int j = 0; j < 4; j++)
            r.e[i] += m.v[i].e[j] * v.e[j];
    }
    return r;
}
```





## ▶ 友元类

- ▶ 如果 A 类声明为 B 类的友元，则将 A 类称为 B 类的**友元类**
- ▶ 若 A 类为 B 类的友元类，则 A 类的所有成员函数都是 B 类的友员函数
- ▶ 定义

---

```
class B
{
    :
    friend class A;
}
```

---





## ▶ 公有数据成员

```
// 例 03-44: ex03-44.cpp
// CPoint2D 的定义
```

```
class CPoint2D
{
public:
    float x, y;
    CPoint2D()
    {
        x = y = 0;
    }
    CPoint2D(float x, float y)
    {
        this->x = x;
        this->y = y;
    }
    void Translate(float x, float y);
    void Scale(float r);
    void Rotate(float angle);
};
```

```
// 例 03-44: ex03-44.cpp
// CRectangle 的定义
```

```
class CRectangle
{
    unsigned long color;
    CPoint2D wPos, lv1, lv2, lv3, lv4;
public:
    CRectangle();
    CRectangle( unsigned long color,
                CPoint2D w,
                float width,
                float height);
    void Translate(float xOffset,
                  float yOffset);
    void Rotate(float angle);
    void Scale(float r);
    void Draw();
};
```





## ▶ 友元类

---

// 例 03-45: ex03-45.cpp

```
class CPoint2D
{
    float x, y;
public:
    CPoint2D()
    {
        x = y = 0;
    }
    CPoint2D(float x, float y)
    {
        this->x = x;
        this->y = y;
    }
    void Translate(float x, float y);
    void Scale(float r);
    void Rotate(float angle);
    friend class CRectangle;
};
```

---

// 例 03-44: ex03-44.cpp

// CRectangle 的定义

```
class CRectangle
{
    unsigned long color;
    CPoint2D wPos, lv1, lv2, lv3, lv4;
public:
    CRectangle();
    CRectangle( unsigned long color,
                CPoint2D w,
                float width,
                float height);
    void Translate(float xOffset,
                  float yOffset);
    void Rotate(float angle);
    void Scale(float r);
    void Draw();
};
```

---





## ► 利用友元类访问类的私有成员

```
// 例 ex03-46: ex03-46.cpp
void CRectangle::Draw()
{
    SetPenColorHex(color);
    DrawFillBox(wPos.x + lv1.x, wPos.y + lv1.y,
                wPos.x + lv2.x, wPos.y + lv2.y,
                wPos.x + lv3.x, wPos.y + lv3.y,
                wPos.x + lv4.x, wPos.y + lv4.y);
}
```

访问私有成员





▶ 友员关系是非传递的

▶ Y 类是 X 类的友员，Z 类是 Y 类的友员，但 Z 类不一定是 X 类的友员

▶ 友员关系是单向的

▶ 若 Y 类是 X 类的友员，则 Y 类的成员函数可以访问 X 类的私有和保护成员，反之则不然

▶ 友员提高了数据的共享性，但在一定程度上削弱了数据的隐藏性





## ▶ 常对象

```
// 例 03-47: ex03-47.cpp
// CPoint2D 的实现

class CPoint2D
{
    float x, y;
public:
    CPoint2D()
    {
        x = y = 0;
    }
    CPoint2D(float x, float y)
    {
        this->x = x;
        this->y = y;
    }
    void Translate(float x, float y)
    {
        this->x = this->x + x;
        this->y = this->y + y;
    }
    //...
};
```

```
// 例 03-47: ex03-47.cpp

int main()
{
    const CPoint2D p1(1, 2);
    p1.Translate(1, 1);p1dz56

    CPoint2D p2;
    p1 = p2;p2dz56
}
```





## ► 常数据成员能通过初始化列表获得初值

---

// 例 03-48: ex03-48.cpp

```
class A
{
private:
    const int& r;           //常引用数据成员
    const int a;             //常数据成员
    static const int b;     //静态常数据成员
public:
    A(int i): a(i), r(a)
    {
        cout << "constructor!" << endl;
    };
    void display()
    {
        cout << a << "," << b << "," << r << endl;
    }
};
```

---





- ▶ 使用**const**关键字修饰的用于访问类的常对象的函数
- ▶ 定义

返回类型 成员函数名 (参数表) **const**;

```
// 例 03-49: ex03-49.cpp
// CPoint2D 的定义

class CPoint2D{
    float x, y;
public:
    CPoint2D(){
        x = y = 0;
    }
    CPoint2D(float x, float y){
        this->x = x; this->y = y;
    }
    float GetX() const {
        return x;
    }
    float GetY() const;
    void Set(float x, float y){
        this->x = x; this->y = y;
    }
};
```

```
// 例 03-49: ex03-49.cpp

float CPoint2D::GetY() const
{
    return y;
}
int main()
{
    const CPoint2D p(1, 2);
    float x = p.GetX();
    float y = p.GetY();
}
```



## ► 使用

```
// 例 ex03-50-constfun.cpp
class CPoint2D{
    float x, y;
public:
    CPoint2D(){
        x = y = 0;
    }
    CPoint2D(float x, float y){
        this->x = x; this->y = y;
    }
    float GetX() const {
        return x;
    }
    float GetY() const;
    void Set(float x, float y) const {
        this->x = x; this->y = y;pzd56
    }
};
```

常成员函数不能更新对象的数据成员，也不能调用对象的非常成员函数

```
// 例 03-49: ex03-49.cpp
float CPoint2D::GetY() const
{
    return y;
}
int main()
{
    const CPoint2D p(1, 2);
    float x = p.GetX();
    float y = p.GetY();
}
```





## ▶ 使用

```
// 例 ex03-51-constfun.cpp
class CPoint2D{
    float x, y;
public:
    CPoint2D(){
        x = y = 0;
    }
    CPoint2D(float x, float y){
        this->x = x; this->y = y;
    }
    float GetX() const {
        return x;
    }
    float GetY() const;
    void Set(float x, float y) {
        this->x = x; this->y = y;
    }
};
```

常对象可以调用常成员函数

```
// 例 ex03-51-main.cpp
float CPoint2D::GetY() const
{
    return y;
}
int main()
{
    const CPoint2D p(1, 2);
    float x = p.GetX();
    float y = p.GetY();
}
```

人



▶ **mutable**易变量

- ▶ 任意可变量
- ▶ 不受**const**常对象限制

```
// 例 03-51-mutable : ex03-51-01.cpp
class Data
{
    // mutable 成员
    mutable bool cache_valid;
    mutable string cache;
    void compute_cache_value() const;
    // ...
public:
    // ...
    string string_rep() const;
};

// 常函数中可以修改数据成员
string Data::string_rep() const{
    if(!cache_valid){
        compute_cache_value();
        cache_valid = true;
    }
    return cache;
}
```





- ▶ 不同对象之间数据成员和函数的共享
- ▶ 在内存中只有一个对应的存储区域
- ▶ 定义
  - ▶ **static** 数据类型静态成员名;
  - ▶ **static** 返回类型函数名 (){};
- ▶ 静态成员的初始化
  - ▶ 数据类型类名::静态成员名 = 初始值





## ► 例子：统计点的个数

```
// 例 03-37: ex03-37-class.cpp
class CPoint2D
{
    int x, y;
    static int num;
public:
    CPoint2D(){
        x = y = 0;
        num++;
    }
    CPoint2D(int x, int y){
        this->x = x;
        this->y = y;
        num++;
    }
    ~CPoint2D(){
        num--;
    }
    static int GetCounter(){
        return num;
    }
};
```

静态成员变量的初始化

```
// 例 03-37: ex03-37-main.cpp
int CPoint2D::num = 0;

int main()
{
    CPoint2D v0, v1;
    int num1, num2;
    num1 = CPoint2D::GetCounter();
    num2 = v0.GetCounter();
}
```



### ▶ 静态成员变量的初始化

- ▶ 类名::静态成员变量名

### ▶ 在建立对象之前通过类就可操作静态成员

### ▶ 静态成员函数中**没有this指针**

- ▶ 不能直接访问类中的非静态成员变量

- ▶ 不能调用非静态成员函数

- ▶ 不能声明为**const**或**volatile**





### ▶ 单件模式

- ▶ 保证一个类仅有一个实例
- ▶ 利用**private**访问控制与**static**成员实现

```
// 例 03-37-01-singleton : ex03-37-01.cpp
#include <iostream>

using namespace std;

class Singleton
{
private:
    int num;
    static Singleton* sp;
    Singleton(int _num){num = _num;}
    Singleton(const Singleton &){}
public:
    static Singleton * getInstance(int _num);
    void handle()
    {
        if(num > 0)
            {num -= 1;}
        else
            {cout << "num is zero!" << endl;}
    }
};
```

```
// 例 03-37-01-singleton : ex03-37-01.cpp
Singleton* Singleton::sp = nullptr;

Singleton* Singleton::getInstance(int _num){
    if(sp == nullptr) {sp = new Singleton(_num);}
    return sp;
}

int main(){
    Singleton* sp = Singleton::getInstance(1);
    sp->handle();
    Singleton* st = Singleton::getInstance(10);
    st->handle();
}
```





## ▶ 语法

### ▶ 声明

成员类型 类名::\*指向数据成员的指针;

### ▶ 使用

对象.\*指向数据成员的指针;

对象指针->\*指向数据成员的指针;

## ▶ 示例

```
// 例 03-38-01 : ex03-38-01-01.cpp
#include <iostream>

using namespace std;

class Data
{
public:
    int a, b, c;
    void printf() const
    {
        cout << "a = " << a
            << ", b = " << b
            << ", c = " << c << endl;
    }
};
```

```
// 例 03-38-01 : ex03-38-01-02.cpp
int main()
{
    Data d, *dp = &d;

    int Data::*pmInt = &Data::a;
    // pmInt 指向 Data 的 int 成员
    dp->*pmInt = 12;
    pmInt = &Data::b;
    d.*pmInt = 24;
    pmInt = &Data::c;
    dp->*pmInt = 36;
    dp->printf();

    return 0;
}
```





## ▶ 语法

### ▶ 声明

返回类型 (类名::\*指向成员函数的指针)(形参表);

### ▶ 使用

指向成员函数的指针 = &类名::函数成员名称;

## ▶ 示例

```
// 例 03-38-02 : ex03-38-02-01.cpp
#include <iostream>
using namespace std;
class FuncTable
{
    void f(int) const {cout << "FuncTable::f()\n";}
    void g(int) const {cout << "FuncTable::g()\n";}
    void h(int) const {cout << "FuncTable::h()\n";}
    void i(int) const {cout << "FuncTable::i()\n";}
    static const int cnt = 4;
    void (FuncTable::*fptr[cnt])(int) const; // 函数表
public:
    FuncTable() { // 初始化函数表
        fptr[0] = &FuncTable::f;
        fptr[1] = &FuncTable::g;
        fptr[2] = &FuncTable::h;
        fptr[3] = &FuncTable::i;
    }
}
```

```
// 例 03-38-02 : ex03-38-02-02.cpp
void select(int i, int j){
    if(i < 0 || i >= cnt)
        return;
    (this->*fptr[i])(j);
}
int count() {return cnt;}
};

int main()
{
    FuncTable ft;
    for(int i = 0; i < ft.count(); i++){
        ft.select(i, 20);
    }

    return 0;
}
```





- ▶ 类只是一个类型，除静态数据成员外，在没有实例化成对象前不占任何内存
- ▶ 对象的存储
  - ▶ 数据段：全局对象、静态对象
  - ▶ 代码段：成员函数、静态函数等
  - ▶ 栈：局部对象、参数传递时的临时对象
  - ▶ 堆：动态内存分配





## ▶ 测试代码

```
// 例 03-52: ex03-52.cpp
// static 成员的示例

#include <iostream>
using namespace std;
class CPoint2D{
    int x, y;
    // 静态数据成员
    static int num;
public:
    CPoint2D(){
        x = y = 0;
        num++;
    }
    CPoint2D(int x, int y){
        this->x = x;
        this->y = y;
        num++;
    }
    ~CPoint2D(){
        num--;
    }
    // 静态函数成员
    static int GetCounter(){
        return num;
    }
    void Output();
};
```

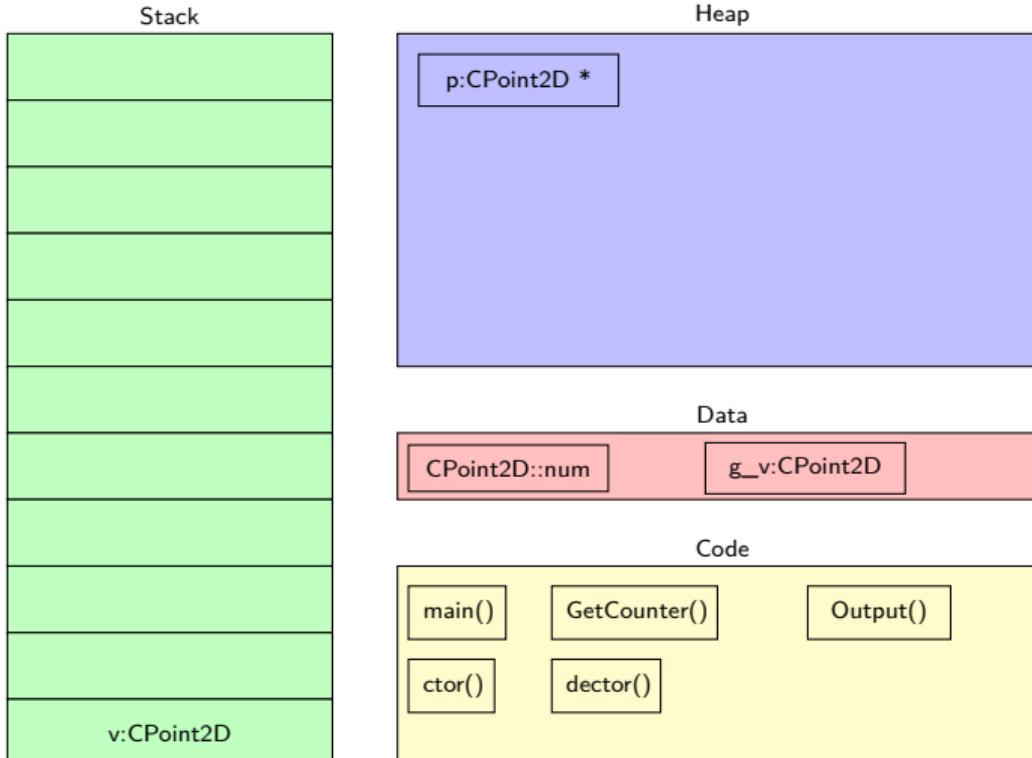
```
// 例 03-52: ex03-52.cpp
// 不同作用域对象的对比，类静态成员的地址查看
```

```
void CPoint2D::Output(){
    cout << "静态变量地址:" << &num << endl;
    cout << "静态函数地址:" << GetCounter << endl;
}
int CPoint2D::num = 0;
CPoint2D g_v(1, 1);
int main(){
    CPoint2D v;
    CPoint2D *p;
    p = new CPoint2D;
    p->GetCounter();
    cout << "全局对象地址:" << &g_v << endl;
    cout << "局部对象地址:" << &v << endl;
    cout << "动态对象地址:" << p << endl;
    p->Output();
    delete p;
    return 0;
}
```



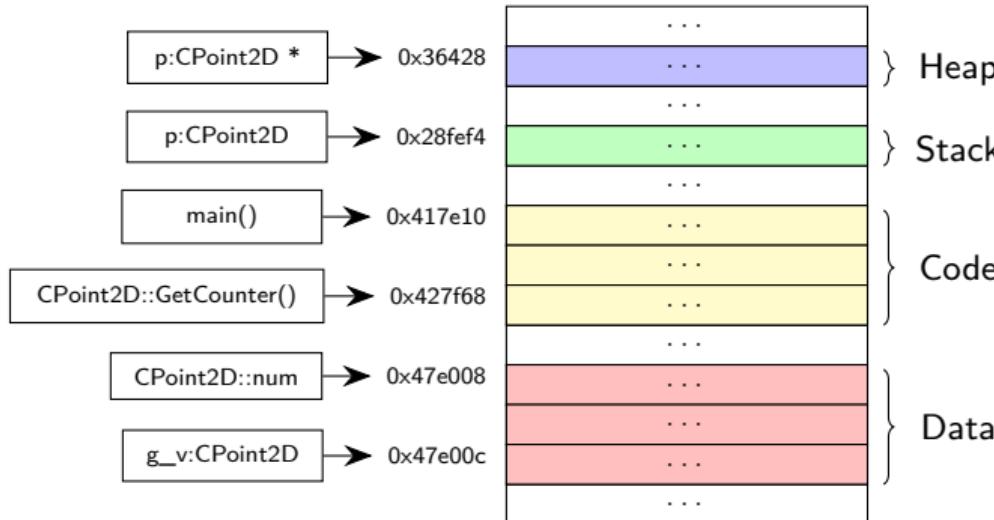


### ▶ 内存分配示意图



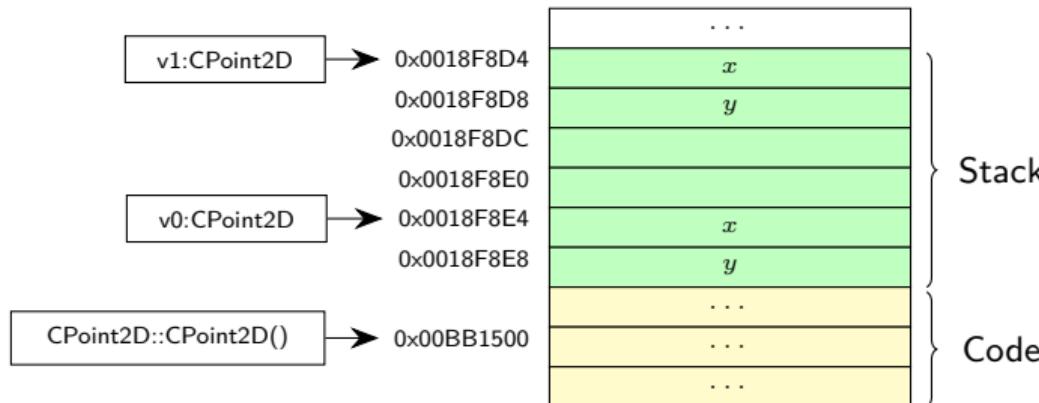


## ▶ 内存地址示意图





- ▶ 类的不同对象的数据成员拥用各自独立的内存空间
- ▶ 类的成员函数为该类的所有对象共享





- ▶ 全局对象数据成员占有的内存空间在程序结束时释放
- ▶ 局部对象与实参对象数据成员的内存空间在函数调用结束时释放
- ▶ 动态对象数据成员的内存空间要使用**delete**语句释放
- ▶ 对象的成员函数的内存空间在该类的所有对象生命周期结束时自动释放





## ► 对象构造函数与析构函数调用顺序

```
// 例 03-53: ex03-53.cpp
// 构造函数, 析构函数

#include <string>
#include <iostream>
using namespace std;
class base{
    string mark;
public:
    base(string str){
        mark = str;
        cout << "Constructor "
            << mark << endl;
    }
    ~base(){
        cout << "Destructor "
            << mark << endl;
    }
};
```

```
// 例 ex03-53-ctor-main.cpp
base g1("g1");
base g2("g2");
int main()
{
    base l1("l1");
    base l2("l2");
    base *d1, *d2;
    d1 = new base("d1");
    d2 = new base ("d2");
    delete d1;
    delete d2;
    return 0;
}
```





## ► 对象构造函数与析构函数调用顺序

```
// 例 ex03-53-ctor-main.cpp
base g1("g1");
base g2("g2");
int main()
{
    base l1("l1");
    base l2("l2");
    base *d1, *d2;
    d1 = new base("d1");
    d2 = new base ("d2");
    delete d1;
    delete d2;
    return 0;
}
```

[scale=0.5,unit=1mm]chap03/07ctorse





## ▶ 主要内容

- ▶ 初始化方式
  - ▶ 设计构造函数
- ▶ 销毁操作
  - ▶ 析构函数
- ▶ 操作
  - ▶ 成员函数
- ▶ 属性
  - ▶ 数据结构
  - ▶ 类型转换
  - ▶ 动态分配
- ▶ 辅助函数

## ▶ 不适合用类表示的问题

- ▶ 无所不能的类
- ▶ 只有数据没有行为了的类
- ▶ 只有行为没有数据的类

## ▶ UML 图





- ▶ 类是对逻辑上相关的函数与数据的封装，它是对问题的抽象描述
- ▶ 类中有数据成员与成员函数，成员的访问控制属性有 **private**、  
**protected** 和 **public**
- ▶ **类是“型”，不占内存，使用类建立对象后，对象占有内存空间**
- ▶ 建立对象时**调用构造函数初始化对象的数据成员**，一个类提供默认的构造函数与默认的拷贝构造函数（**浅拷贝**）
- ▶ 对象使用完后系统**自动调用析构函数**
- ▶ 建立**对象指针、对象引用**均没有建立对象，不调用构造函数
- ▶ 通过对**象指针使用对象成员**使用“->”，通过对**象引用使用对象成员**使用“.”
- ▶ 对象数组的定义、赋值、引用与普通数组一样。**对象数组初始化需要通过构造函数完成**
- ▶ 建立动态对象使用语句“**new**”，建立动态对象数组使用语句“**new []**”





- ▶ “**this**” 指针是一个系统预定义的**指向当前对象的指针**
- ▶ 组合类是含有类对象的类，组合类对象称为组合对象。定义组合对象时调用构造函数的顺序为**类中成员对象定义的顺序**
- ▶ 静态数据成员是类的数据成员，**独立于类存在**。静态成员函数属于整个类，是该类的**所有对象共享的成员函数**
- ▶ 友元函数**不是类的成员**，但它**可以访问类的任何成员**。一个类的友元类可以访问该类的任何成员
- ▶ 常对象与常成员



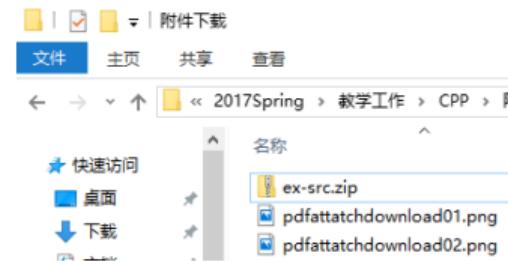
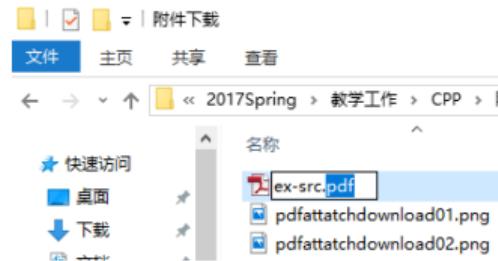
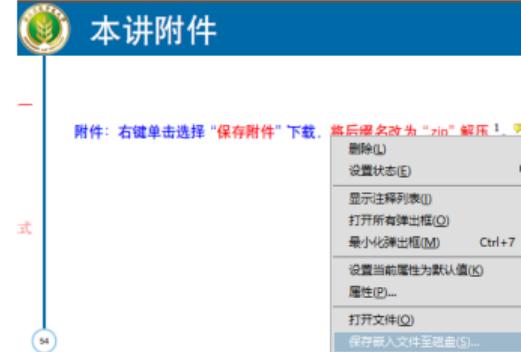
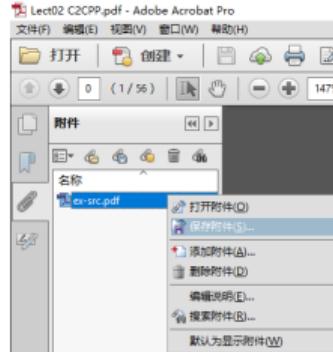


# 本讲附件

| 附件

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP  
  
Nine, G.

附件：右键单击该链接，选择“保存附件”下载，将后缀名改为“.zip”解压。<sup>10 11</sup>



<sup>10</sup>请退出全屏模式后点击该链接。

<sup>11</sup>以 Adobe Acrobat Reader 为例。



### ▶ 使用一致接口 (uniform interface) 处理不同数据

- ▶ 函数重载
- ▶ 运算符重载 (“+”)

```
3.14 + 0.0015 = 3.1415;  
[1, 2, 3] + [4, 5, 6] = [1, 2, 3, 4, 5, 6];  
[3 + 4i] + [1 + 5i] = [4 + 9i];  
"coffee" + " tea" = "coffee tea";
```

- ▶ 虚函数 (继承与派生)





### ▶ 函数重载

- ▶ 功能相同、数据不同 (类型或个数) 的同名函数

### ▶ 运算符重载

- ▶ 同一个运算符作用于不同类型数据的操作
- ▶ 运算符 ⇒ **函数名**





### ▶ 运算符重载是 C++ 的一大亮点

- ▶ 函数调用更简洁、直观
- ▶ 增加程序可读性 (readability)

### ▶ 实际应用

- ▶ 复数操作:  $(a+bi)+(c+di) = (a+c) + (b+d)i$
- ▶ 字符串操作: `string1 == string2`
- ▶ 向量操作:  $(a_1, \dots, a_n)*s = (a_1*s, \dots, a_n*s)$
- ▶ 矩阵乘法:  $M = N*K$
- ▶ ...





- ▶ 位置信息：纬度 (latitude) 和经度 (longitude)
- ▶ 经纬度决定地球上一个地点的精确位置

0.95 已知：

北京天安门：(39.907306, 116.391264)

求：

西南方向偏移 (-5.642159, -8.323558) 后的新位置？

解：

$$39.907306 + (-5.642159)$$

$$116.391264 + (-8.323558)$$

答案：

(34.265147 108.067706) ⇒ 西北农林科技大学行政楼位置





- ▶ 位置信息：纬度 (latitude) 和经度 (longitude)
- ▶ 经纬度决定地球上一个地点的精确位置





- ▶ 位置信息：纬度 (latitude) 和经度 (longitude)
- ▶ 经纬度决定地球上一个地点的精确位置

```
// 例 04-01-01: ex04-01-01.cpp
// 重载 + 操作符

#include <iostream>
#include <iomanip>
using namespace std;
class Clocation{
    double latitude, longitude;
public:
    Clocation(){}
    Clocation(double lt, double lg){
        latitude = lt;
        longitude = lg;
    }
    void show(){
        cout << setprecision(9) << latitude << " ";
        cout << longitude << endl;
    }
    Clocation operator+(Clocation op2);
};
Clocation Clocation::operator+(Clocation op2){
    Clocation temp;
    temp.latitude = op2.latitude + latitude;
    temp.longitude = op2.longitude + longitude;
    return temp;
}
```

+ 运算符的重载

```
// 例 04-01-02: ex04-01-02.cpp
// 重载操作符的测试

int main()
{
    Clocation Tiananmen(39.907306, 116.391264);
    Clocation Offset(-5.642159, -8.323558);
    Clocation Yangling;

    Tiananmen.show();

    Yangling = Tiananmen + Offset;

    Yangling.show();
    return 0;
}
```

北京天安门:  
(39.907306, 116.391264)

西南方向偏移:  
(-5.642159, -8.323558)

西北农林科技大学  
行政楼位置



### ▶ 可重载的运算符

+	-	*	/	%	^	&		~
!	,	=	<	>	<=	>=	++	--
<<	>>	==	!=	&&		+=	-=	/=
%=	^=	&=	=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []			

### ▶ 单目运算符和双目运算符

### ▶ 不可重载的运算符

.	.*	::	?:
---	----	----	----





### ▶ 重载规则

- ▶ 重载后运算符的优先级和结合性不变
- ▶ 运算符操作数的个数不能改变
- ▶ 不能重载 C++ 中不支持的运算符 (@、#、\$ 等)
- ▶ 保持运算符的语义





### ▶ 重载为类的成员函数

#### ▶ 定义

---

```
返回类型 [类名::]operator 运算符 (形参表) {}
```

```
CLocation operator+(CLocation op2);
```

---

### ▶ 重载为类的非成员函数 (一般为友员函数)

#### ▶ 定义

---

```
friend 返回类型 operator 运算符 (形参表) {}
```

---





### ▶ 语法：返回类型 [类名::]operator 运算符 (形参表) {}

- ▶ 返回类型一般是一个对象
- ▶ 可以省略一个形参

```
CLocation operator+(CLocation op2);  
CLocation operator-(CLocation op2);  
CLocation operator=(CLocation op2);  
CLocation operator++();
```

注意：用成员函数实现 + 运算符重载，只有一个参数！

```
CLocation loc1(30,100), loc2(5,4);  
loc1 = loc1 + loc2;
```





### ▶ 重载函数定义

```
// 例 04-02-01: ex04-02-01.cpp
// 重载 - , = , ++ 操作符

CLocation CLocation::operator-(CLocation op2){
    CLocation temp;
    temp.latitude = latitude - op2.latitude;
    temp.longitude = longitude - op2.longitude;
    return temp;
}

CLocation CLocation::operator=(CLocation op2){
    longitude = op2.longitude;
    latitude = op2.latitude;
    return *this;
}

CLocation CLocation::operator++(){
    longitude++;
    latitude++;
    return *this;
}
```

```
// 例 04-02-02: ex04-02-02.cpp
// 重载操作符的测试

int main(){
    CLocation ob1(39.907306,
                  116.391264);
    CLocation ob2, ob3;
    ob1.show();
    ob2 = ++ob1;
    ob2.show();
    // multiple assignment
    ob3 = ob2 = ob1;
    ob1 = ob1 - ob2;
    ob1.show();
    ob2.show();
    ob3.show();
    return 0;
}
```

利用\*this指针返回运算后的对象



### ▶ 前缀运算符: `++i`, `--i`

---

```
CLocation operator++();
```

---

```
CLocation operator--();
```

### ▶ 后缀运算符: `i++`, `i--`

---

```
CLocation operator++(int x);
```

---

```
CLocation operator--(int x);
```

0.4 哑元参数, 仅表示重载后缀运算符





### ▶ 前缀与后缀

```
// 例 04-03-01: ex04-03-01.cpp  
// + 及 ++ 作为成员函数的重载
```

```
// Overload prefix ++ for CLocation.  
CLocation CLocation::operator++()  
{  
    longitude++;  
    latitude++;  
    return *this;  
}
```

```
// Overload postfix ++ for CLocation.  
CLocation CLocation::operator++(int x)  
{  
    longitude++;  
    latitude++;  
    return CLocation(latitude - 1, longitude - 1);  
}
```

返回运算后的对象

```
// 例 04-03-02: ex04-03-02.cpp  
// 重载操作符的测试
```

```
int main()  
{  
    CLocation ob1(39.907306,  
                 116.391264);  
    CLocation ob2, ob3;  
  
    ob2 = ++ob1;  
    ob3 = ob2++;  
  
    ob1.show();  
    ob2.show();  
    ob3.show();  
}
```





- ▶ 语法: **friend** 返回类型 **operator** 运算符 (形参表) {}

- ▶ 友元函数**没有 this 指针**, 需给出所有传递参数

```
friend CLocation operator+(CLocation op1, CLocation op2);
```

- ▶ 若使用单目运算符且修改成员数据, **需使用引用传递**

```
friend CLocation operator++(CLocation &op1);
friend CLocation operator++(CLocation &op1, int x);
```

哑元参数表示后缀





### ▶ 重载为类的友元函数

```
// 例 04-04: ex04-04.cpp
// 操作符 + , ++ 的重载

CLocation operator+(CLocation op1, CLocation op2)
{
    CLocation temp;
    temp.longitude = op1.longitude + op2.longitude;
    temp.latitude = op1.latitude + op2.latitude;
    return temp;
}
CLocation operator++(CLocation &op1)
{
    op1.longitude++;
    op1.latitude++;
    return op1;
}
CLocation operator++(CLocation &op1, int x)
{
    return CLocation(op1.latitude++, op1.longitude++);
}
```





### ▶ 操作符左右为不同类型数据

#### ▶ CLocation loc1;

```
loc1 = loc1 * 1.01; ... (1)  
loc1 = 1.01 * loc1; ... (2)
```

#### ▶ 如何实现?

▶ 重载为类的成员函数无法实现!





## ► 操作符左右为不同类型数据

```
// 例 04-05-01: ex04-05-01.cpp
// 重载 * 操作符

friend CLocation operator*(CLocation op1,
                           double s);
friend CLocation operator*(double s,
                           CLocation op1);
CLocation operator*(CLocation op1, double s)
{
    CLocation temp;
    temp.longitude = s * op1.longitude;
    temp.latitude = s * op1.latitude;
    return temp;
}
CLocation operator*(double s, CLocation op1)
{
    CLocation temp;
    temp.longitude = s * op1.longitude;
    temp.latitude = s * op1.latitude;
    return temp;
}
```

```
// 例 04-05-02: ex04-05-02.cpp
// 重载 * 操作符的测试

int main()
{
    CLocation ob1(39.907306,
                  116.391264);
    CLocation ob2, ob3;

    ob2 = ob1 * 1.01;
    ob3 = -1.01 * ob1;

    ob1.show();
    ob2.show();
    ob3.show();
}
```





- ▶ 一般单目运算符重载为类的成员函数，双目运算符重载为类的友元函数
- ▶ `=`, `()`, `[]`, `->` 双目运算符不能重载为类的友元函数
- ▶ “`>>`” 和 “`<<`” 只能重载为类的友元函数





- ▶ “>>” 和 “<<”

只能以友元函数方式重载

- ▶ “=”

- ▶ “[ ]”

- ▶ “( )”

- ▶ “->”

- ▶ “new”、“delete”、“new[]” 和 “delete []”

- ▶ “,”

只能以成员函数方式重载



## ► 重载运算符“>>”和“<<”

```
// 例 04-06-01: ex04-06-01.cpp
// 重载输入输出操作符

class CLocation
{
    double latitude, longitude;
public:
    CLocation (double lt = 0, double lg = 0)
    {
        latitude = lt;
        longitude = lg;
    }
    friend ostream& operator<<( ostream& out , CLocation loc);
    friend istream& operator>>( istream& in , CLocation & loc);
};
```





## ► 重载运算符“>>”和“<<”

```
// 例 04-06-02: ex04-06-02.cpp
// 重载输入输出操作符的实现及测试

ostream &operator<<(ostream &out, CLocation loc){
    out << loc.latitude << " " << loc.longitude << endl;
    return out;
}
istream& operator>>(istream& in, CLocation& loc){
    in >> loc.latitude >> loc.longitude;
    if(loc.latitude < -90 || loc.latitude > 90)
        cout << "Error latitude input!" << endl;
    if(loc.longitude < -180 || loc.longitude > 180)
        cout << "Error longitude input!" << endl;
    return in;
}
int main(){
    CLocation loc;
    cin >> loc ;
    cout << loc ;
    return 0;
}
```





## ► 重载运算符 “=”

```
ch_stack operator=(ch_stack & s0bj)
{
    size = s0bj.size;
    tp = s0bj.tp;
    s = s0bj.s;
}
```

浅拷贝

```
ch_stack operator=(ch_stack & s0bj)
{
    if(s != NULL)
        free(s);
    size = s0bj.size;
    tp = s0bj.tp;
    s = NULL;
    if (size != 0)
    {
        s = new char[size];
        for (int i = 0; i <= tp; i++)
            s[i] = s0bj.s[i];
    }
}
```

深拷贝



## ► 自赋值检测

```
ClassName &ClassName::operator=(ClassName &s)
{
    if (this == &s)
        return *this;
    ...
}
```

## ► 引用计数与浅拷贝

- 深拷贝安全但某些情况下易浪费空间。
- 缺点：增加复杂度。





## ▶ 重载运算符“[]”

### ▶ 防止数组越界

```
// 例 04-07-01: ex04-07-01.cpp
// 重载 [] 操作符

class atype{
    int a[3];
public:
    atype(int i, int j, int k){
        a[0] = i;
        a[1] = j;
        a[2] = k;
    }
    int &operator[](int i){
        if(i < 0 || i > 2)
        {
            cout << "Boundary Error\n";
            exit(1);
        }
        return a[i];
    }
};
```

```
// 例 04-07-02: ex04-07-02.cpp
// 重载 [] 操作符的测试

int main()
{
    atype ob(1, 2, 3);
    cout << ob[1];
    cout << " ";
    ob[1] = 25;
    cout << ob[1];
    ob[3] = 44;
}
```





## ▶ 重载运算符“()”

- ▶ 自动执行和表达式中的使用
- ▶ 函数对象

```
// 例 04-08-01: ex04-08-01.cpp
// +, () 运算符的重载
```

```
class loc
{
    double longitude, latitude;
public:
    ...
    loc operator+(loc op2);
    loc operator()(double i, double j)
    {
        longitude = j;
        latitude = i;
        return *this;
    }
};
```

```
// 例 04-08-02: ex04-08-02.cpp
// 运算符重载的测试
```

```
int main()
{
    loc ob1(10, 20), ob2(1, 1);
    ob1.show();
    ob1(7, 8);
    ob1.show();
    ob1 = ob2 + ob1(10, 10);
    ob1.show();
    return 0;
}
```





## ► 重载运算符“->”(应用于安全指针)

```
ClassName* ClassName::operator->();
```





## ► 重载运算符“new”和“delete”

- 自定义的内存分配与释放（如在堆区无内存可分配的情况下自动使用磁盘空间）

```
void *ClassName::operator new(size_t size);
void ClassName::operator delete(void *p);
```

## ► 使用系统的“new”和“delete”

```
::new
::delete
```





## ▶ 重载运算符“new[]”和“delete[]”

- ▶ 为数组动态分配内存空间

```
void *ClassName::operator new[](size_t size);
void ClassName::operator delete[](void *p);
```

## ▶ 重载运算符“,”

```
ClassName ClassName::operator,(ClassName obj);
```



## ► 类型转换运算符

- 重载“int”、“float”、…(不必指定返回类型)

```
Rational r = 3.5f;
class Rational
{
    int up;
    int down;
public:
    //成员函数 return
    operator float()
    {
        up/(float)down;
    }
};
```





- ▶ 目的
  - ▶ 将表达式传入函数 (函数指针)
- ▶ 本质
  - ▶ 可调用的代码单元
  - ▶ 类似于未命名的**inline**函数
- ▶ 实例

```
#include <iostream>

using namespace std;

int main()
{
    int girls = 3, boys = 4;
    // lambda 函数
    auto totalChild = [] (int x, int y) -> int {return x + y;};
    cout << totalChild(girls, boys) << endl;

    return 0;
}
```

- ▶ 别称
  - ▶ lambda 表达式





## ▶ 完整定义

```
[capture list](params list) mutable exception -> return type{function body}
```

**capture list** 捕获外部变量列表（上下文中的变量）

**params list** 形参列表

**mutable**修饰符 用于说明可否修改外部捕获的外部变量

**exception** 异常设定

**return type** 返回类型

**function body** 函数体

## ▶ 简化定义

- ▶ [capture list](params list)->**return type{function body}**
- ▶ [capture list](params list){function body}
- ▶ [capture list]{function body}





- ▶ 对于任何参数类型，如果仅仅只是读参数的值，应该作为**const**引用用来传递。普通算术运算符、关系运算符、逻辑运算符都不会改变参数，应该以**const &**引用作为主要的参数传递方式。
- ▶ 当运算符函数是类的成员函数时，应该将其定义为**const**成员函数。
- ▶ 返回值的类型取决于运算符的具体含义。如果使用运算符的结果产生一个新值，就需要产生一个作为返回值的新对象，这个对象作为一个常量通过传值方式返回。
- ▶ 如果函数返回的是原有对象，则通常以引用方式返回，根据是否希望对返回的值进行运算来决定是否返回**const**引用。
- ▶ 所有赋值运算符均改变左值。为了使赋值结果能用于链式表达式，如  $a=b=c$ ，应该返回一个改变了的左值的引用。一般赋值运算符的返回值是非**const**引用，以便能够对刚刚赋值的对象进行运算。



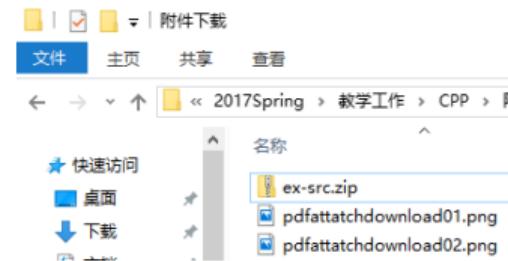
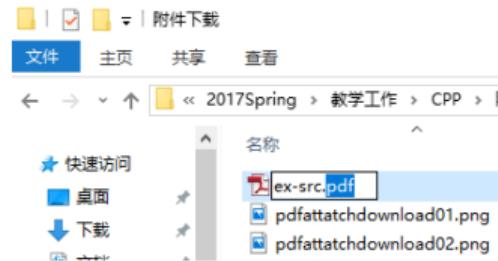
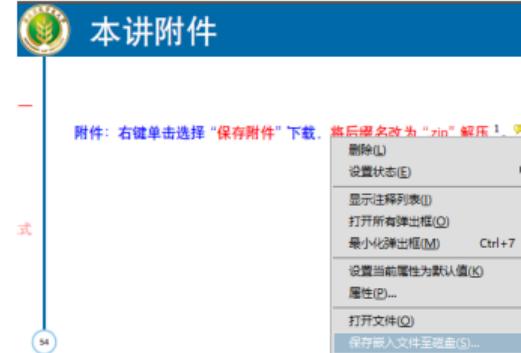
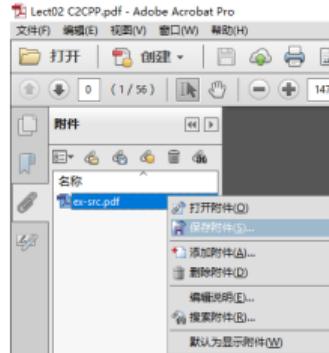


# 本讲附件

| 附件

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP  
Nine, G.

附件：右键单击该链接，选择“保存附件”下载，将后缀名改为“.zip”解压<sup>12 13</sup>。

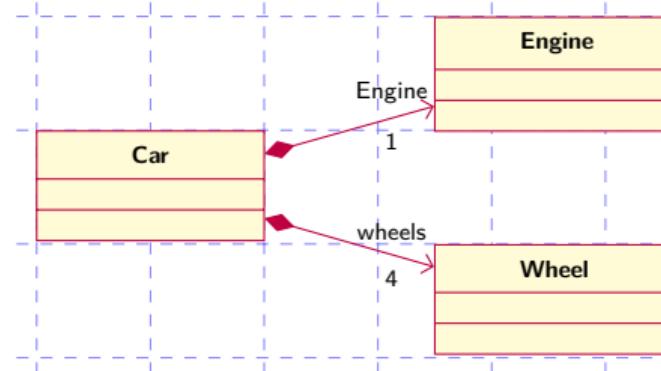


<sup>12</sup>请退出全屏模式后点击该链接。

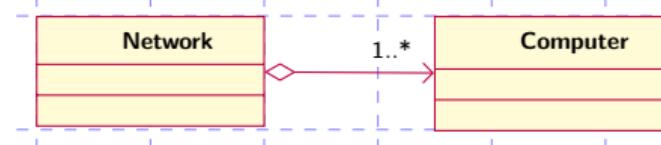
<sup>13</sup>以 Adobe Acrobat Reader 为例。



### ▶ 组合—composition(整体与部分的关系)

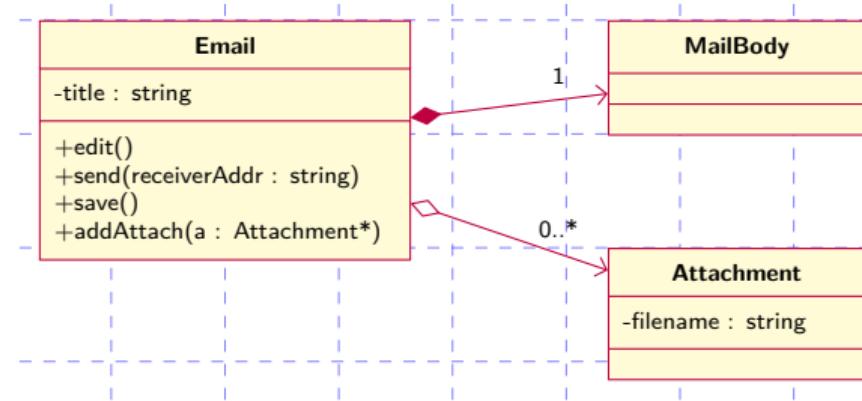


### ▶ 聚合—aggregation(松散关系)



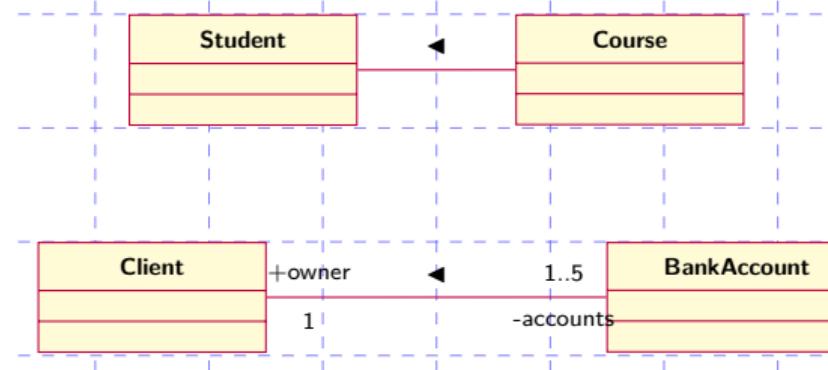


### ▶ 复合关系



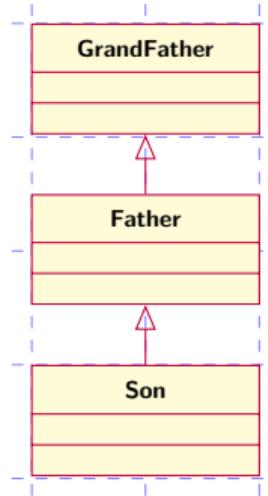


### ▶ 拥有关系





### ► 继承与派生类



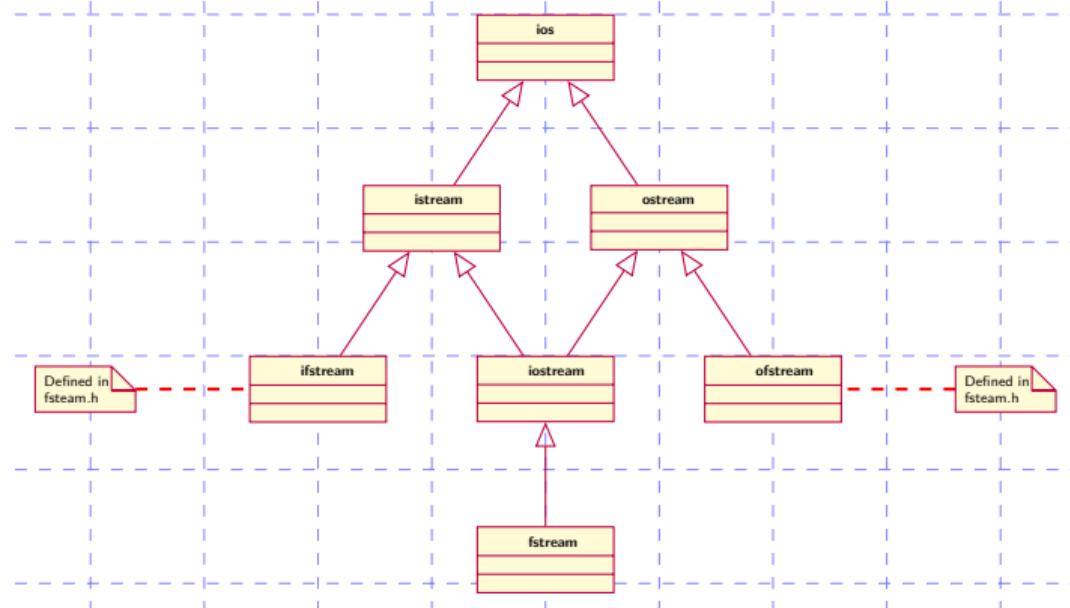


- ▶ 用已有类定义新类，新类拥有原有类的全部特征
  - ▶ 原有类 ⇒ **基类（父类）**
  - ▶ 新类 ⇒ **派生类**
- ▶ 可以**多继承**（一个派生类有多个基类）和**多层派生**（多层次继承）
- ▶ 特点：新类可以**继承**原有类的属性和行为，并且可以**添加**新的属性和行为，或更新原有类的成员
- ▶ 优点：**代码重用**





### ► 例子:C++ 输入输出流类





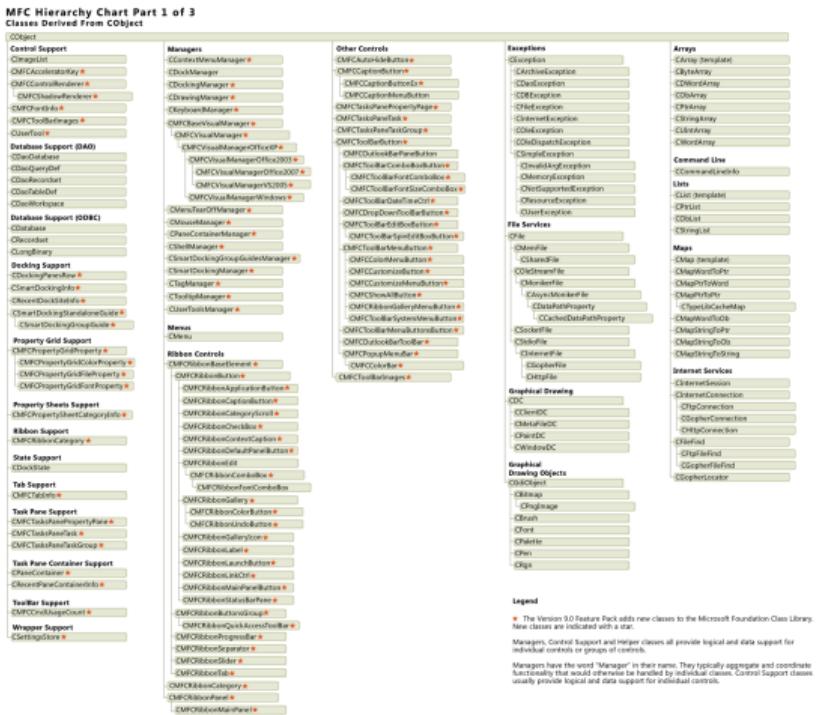
# 应用实例

| 继承与派生

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP

Nine, G.

## ► 例子:MFC 类层次





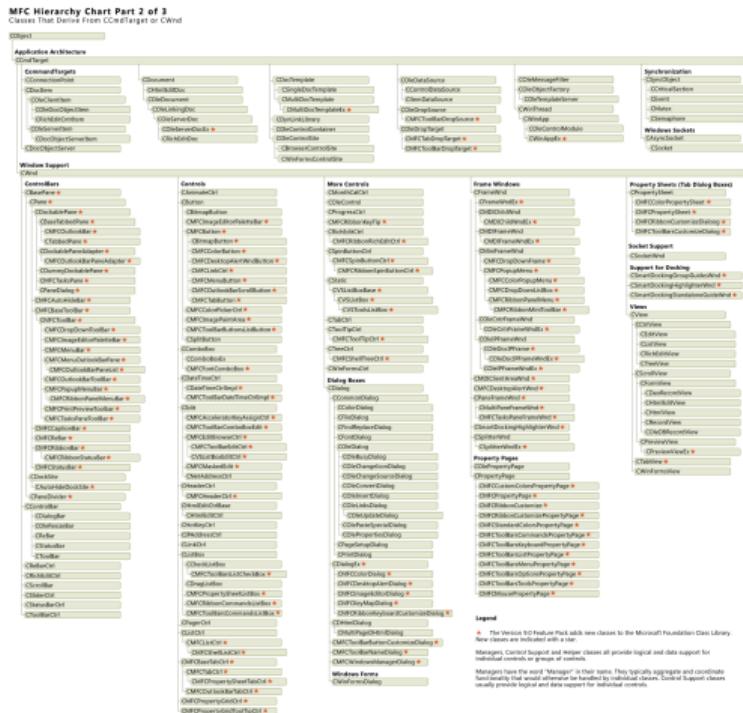
# 应用实例

| 继承与派生

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP

Nine, G.

## ► 例子:MFC 类层次





# 应用实例

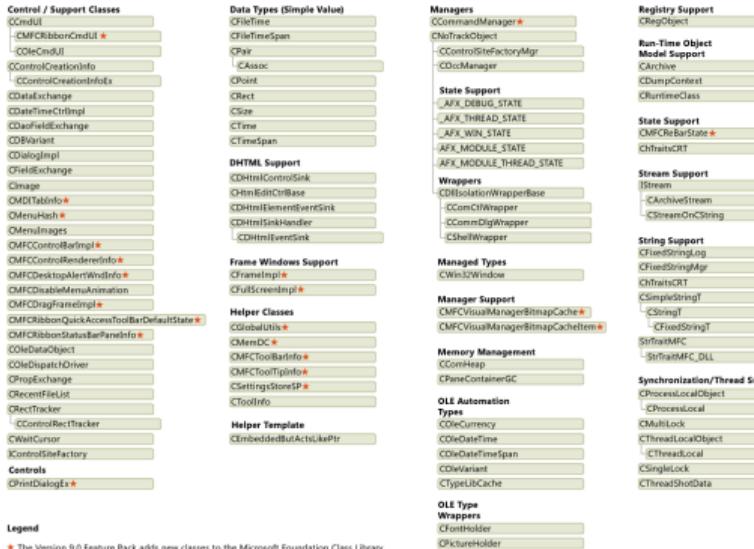
| 继承与派生

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP

Nine, G.

## ► 例子:MFC 类层次

MFC Hierarchy Chart Part 3 of 3  
Classes Not Derived From CObject





## ► 例子: QT 类层次





### ▶ 派生类的定义

```
class 派生类名: 继承方式1 基类名1,  
        继承方式2 基类名2, ...  
{  
    private:  
        派生类的私有数据和函数;  
    public:  
        派生类的公有数据和函数;  
    protected:  
        派生类的保护数据和函数;  
};
```

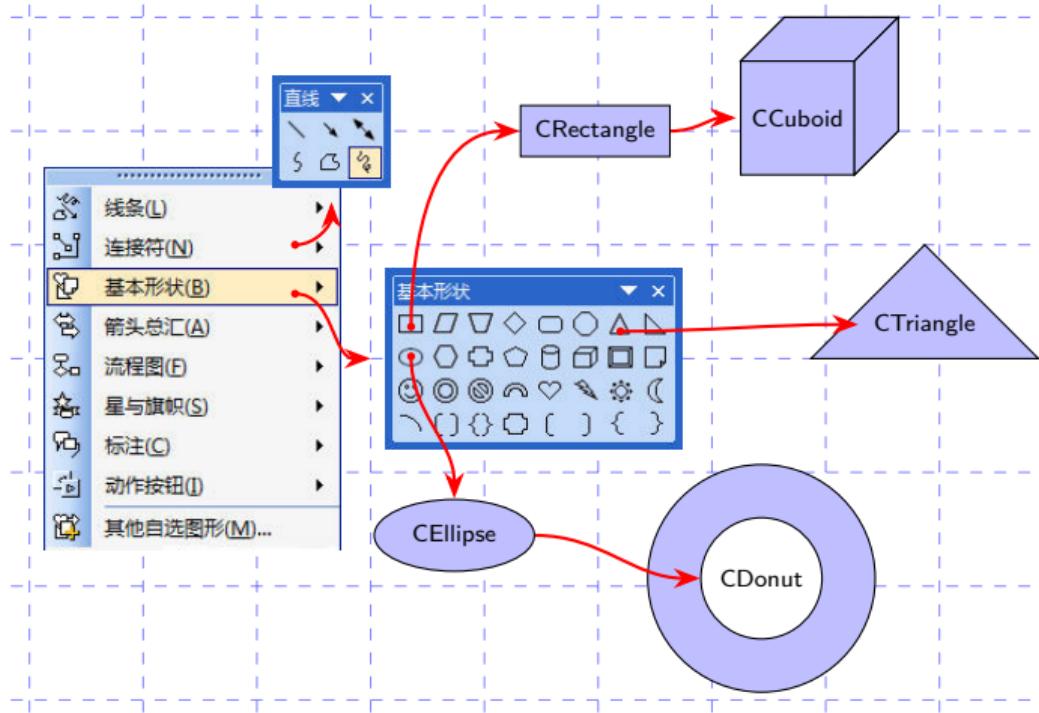
继承方式:

public:	公有继承
private:	私有继承
protected:	保护继承



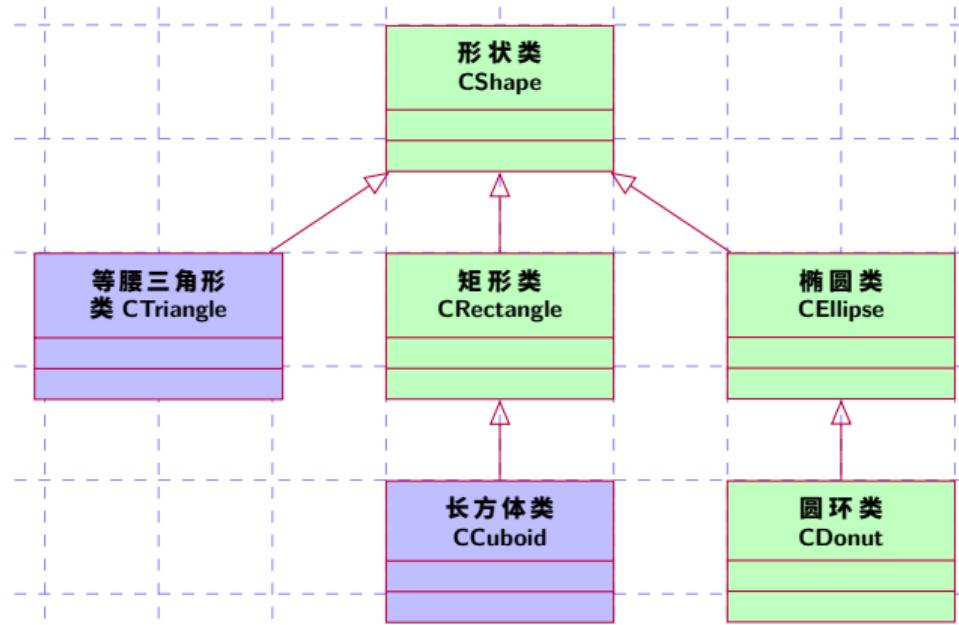


### ▶ 用继承方式实现基本形状类





### ▶ 基类：形状类 (CShape)

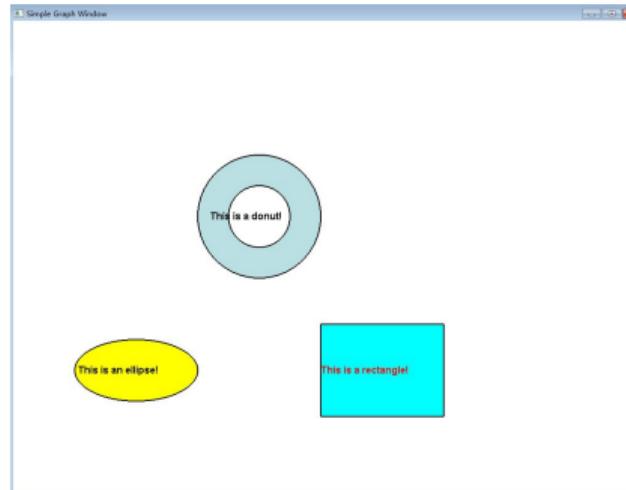




### ▶ 用继承方式实现基本形状类

#### ▶ 功能描述

- ▶ 带文本的基本形状绘制
- ▶ 可更改文本和形状的颜色
- ▶ 可更改形状的大小
- ▶ 可上下左右移动形状





## ▶ 用继承方式实现基本形状类

```
// 例 05-01: ex05-01.cpp
// 定义一个表示二维平面的点的类

class CPoint2D{
    float x, y;
public:
    CPoint2D(){
        x = y = 0;
    }
    CPoint2D(float x, float y){
        this->x = x;
        this->y = y;
    }
    void Translate(float x, float y);
    void Scale(float r);
    void Rotate(float angle);
    friend class CShape;
    friend class CRectangle;
    friend class CEllipse;
    friend class CDonut;
};
```

```
// 例 05-02: ex05-02.cpp

class CShape{
    ULONG textColor;
    char strText[256];
protected:
    CPoint2D wPos;
    ULONG objColor;
public:
    CShape();
    CShape(CPoint2D w, char *strText,
           ULONG objColor = 0xBBE0E3,
           ULONG textColor = 0): wPos(w);
    void Translate(float x, float y);
    void DrawText();
    void ShowPos();
};
```

文本颜色

文本内容

全局坐标

对象颜色

所有基本形状共用数据成员



## ▶ 用继承方式实现基本形状类

```
// 例 05-01: ex05-01.cpp
// 定义一个表示二维平面的点的类

class CPoint2D{
    float x, y;
public:
    CPoint2D(){
        x = y = 0;
    }
    CPoint2D(float x, float y){
        this->x = x;
        this->y = y;
    }
    void Translate(float x, float y);
    void Scale(float r);
    void Rotate(float angle);
    friend class CShape;
    friend class CRectangle;
    friend class CEllipse;
    friend class CDonut;
};
```

友元类

```
// 例 05-02: ex05-02.cpp

class CShape{
    ULONG textColor;
    char strText[256];
protected:
    CPoint2D wPos;
    ULONG objColor;
public:
    CShape();
    CShape(CPoint2D w, char *strText,
           ULONG objColor = 0xBBE0E3,
           ULONG textColor = 0): wPos(w),
    void Translate(float x, float y);
    void DrawText();
    void ShowPos();
};
```

文本颜色

文本内容

全局坐标

对象颜色

所有基本形状共用数据成员



## ► 用继承方式实现基本形状类

```
// 例 05-01: ex05-01.cpp
// 定义一个表示二维平面的点的类

class CPoint2D{
    float x, y;
public:
    CPoint2D(){
        x = y = 0;
    }
    CPoint2D(float x, float y){
        this->x = x;
        this->y = y;
    }
    void Translate(float x, float y);
    void Scale(float r);
    void Rotate(float angle);
    friend class CShape;
    friend class CRectangle;
    friend class CEllipse;
    friend class CDonut;
};
```

```
// 例 05-02: ex05-02.cpp
```

```
class CShape{
    ULONG textColor;
    char strText[256];
protected:
    CPoint2D wPos;
    ULONG objColor;
public:
    CShape();
    CShape(CPoint2D w, char *strText,
           ULONG objColor = 0xBBE0E3,
           ULONG textColor = 0): wPos(w);
    void Translate(float x, float y);
    void DrawText();
    void ShowPos();
};
```

所有基本形状共用函数成员

友元类

平移操作

文本显示

位置输出



## ▶ 用继承方式实现基本形状类—矩形类

```
// 例 05-03: ex05-03.cpp
// 定义类 CRectangle, 该类继承 CShape

#include "Shape.h"
class CRectangle: public CShape{ ← 基类
    CPoint2D lv1, lv2, lv3, lv4;
public:
    CRectangle(): lv1(CPoint2D(-50, -30)), lv2(CPoint2D(50, -30)),
        lv3(CPoint2D(50, 30)), lv4(CPoint2D(-50, 30)) {}
    CRectangle(float length, float width, CPoint2D w, char *strText,
        ULONG objColor = 0xBBE0E3,
        ULONG textColor = 0):
        CShape(w, strText, objColor, textColor){ ← 初始话列表初始化基类数据
            lv1.x = lv4.x = -0.5 * length;
            lv1.y = lv2.y = -0.5 * width;
            lv2.x = lv3.x = 0.5 * length;
            lv3.y = lv4.y = 0.5 * width;
        }
    void Draw();
    void ShowPos();
};
```





## ▶ 用继承方式实现基本形状类—椭圆类

```
// 例 05-04: ex05-04.cpp
// 定义类 CEllipse, 该类继承 CShape

#include "Shape.h"
class CEllipse: public CShape{ ← 基类
protected:
    float x_radius, y_radius;
public:
    CEllipse(){
        x_radius = y_radius = 50;
    }
    CEllipse(float rx, float ry, CPoint2D w, char *strText,
             ULONG objColor = 0xBBE0E3,
             ULONG textColor = 0);
        CShape(w, strText, objColor, textColor){ ← 初始化列表初始化基类数据
    x_radius = rx;
    y_radius = ry;
}
void Draw();
void ShowPos();
};
```





## ▶ 用继承方式实现基本形状类—圆环类

```
// 例 05-05: ex05-05.cpp
// 定义类 CDonut, 该类继承 CEllipse

#include "Ellipse.h"
class CDonut: public CEllipse { ← 基类
    float ratio;
public:
    CDonut()
    {
        ratio = 0.5;
    }
    CDonut( float r, float rx, float ry, CPoint2D w, char *strText,
             ULONG objColor = 0xBBE0E3,
             ULONG textColor = 0 );
    CEllipse(rx, ry, w, strText, objColor, textColor) { ← 初始化列表初始化基类数据
        ratio = r;
    }
    void Draw();
    void ShowPos();
};
```





## ► 吸收基类成员

```
// 例 05-06: ex05-06.cpp
// 定义矩形的 Draw 方法

void CRectangle::Draw()
{
    setColor(CShape::objColor);
    fillRectangle(lv1.x + CShape::wPos.x, lv1.y + CShape::wPos.y,
                  lv3.x + CShape::wPos.x, lv3.y + CShape::wPos.y);
    setColor(0x000000);

    setLineWidth(2);
    rectangle(lv1.x + CShape::wPos.x, lv1.y + CShape::wPos.y,
              lv3.x + CShape::wPos.x, lv3.y + CShape::wPos.y);
    setLineWidth(1);

    CShape::DrawText();
}
```

吸收数据成员

吸收成员函数





## ► 改造基类成员

// 例 05-07: ex05-07.cpp  
// 成员函数实现

```
void CShape::ShowPos()  
{  
    cout << strText << endl;  
    cout << "CShape: (" << wPos.x << ","  
        << wPos.y << ")" << endl;  
}
```

```
void CRectangle::ShowPos()  
{  
    CShape::ShowPos();  
    cout << "CRectangle: (" << lv1.x << ","  
        << lv1.y << "), (" << lv3.x << ","  
        << lv3.y << ")" << endl;  
}
```

CRectangle myRect;  
myRect.ShowPos();

同名覆盖:

当通过派生类对象调用 ShowPos() 时, 将  
自动调用成员函数 CRectangle::ShowPos()





## ► 添加新成员

```
class CRectangle:public CShape{  
    CPoint2D lv1, lv2, lv3, lv4;  
public:  
    void Draw();  
};
```

描述新的属性和行为

- ▶ 数据成员
- ▶ 成员函数

```
class CEllipse:public CShape{  
protected:  
    float x_radius, y_radius;  
public:  
    void Draw();  
};
```





### ► 继承关系是可以传递的

- ▶ 如类 A 派生出类 B，类 B 又派生出类 C，则类 B 是类 C 的直接基类，类 A 是类 B 的直接基类，而类 A 称为类 C 的间接基类

### ► 继承关系不允许循环

- ▶ 在派生中，不允许类 A 派生出类 B，类 B 又派生出类 C，而类 C 又派生出类 A





## ▶ 公有继承 (public)

- ▶ 基类的**公有成员**在派生类中仍然为公有成员，可以由派生类对象和派生类成员函数**直接访问**
- ▶ 基类的**私有成员**在派生类中，无论是派生类的成员还是派生类的对象都**无法直接访问**
- ▶ **保护成员**在派生类中仍是保护成员，可以通过派生类的成员函数访问，但**不能由派生类的对象直接访问**





### ► 实例

```
class CShape
{
    ULONG textColor;
    char strText[256];
protected:
    CPoint2D wPos;
public:
    ULONG objColor;
};
```

```
class CEllipse:public CShape
{
    float x_radius, y_radius;
public:
    void Draw()
    {
        ULONG color1 = CShape::textColor; pzd56
        CPoint2D pos = CShape::wPos; pzd52
        ULONG color2 = CShape::objColor; pzd52
    }
};
```

This is an ellipse!

```
CEllipse myEllip;
ULONG color1 = myEllip.textColor; pzd56
CPoint2D pos = myEllip.wPos; pzd56
ULONG color2 = myEllip.objColor; pzd52
```





## ► 私有继承 (Private)

- ▶ 基类的**公有成员和保护成员**被继承后成为**派生类的私有成员**
- ▶ 基类的**私有成员**在派生类中**不能被直接访问**
- ▶ 经过私有继承，所有基类的成员都**成为了派生类的私有成员**，如进一步派生，基类的全部成员将无法在新的派生类中被访问





## ► 实例

```
class CShape
{
    ULONG textColor;
    char strText[256];
protected:
    CPoint2D wPos;
public:
    ULONG objColor;
};
```

```
class CEllipse:private CShape
{
    float x_radius, y_radius;
public:
    void Draw()
    {
        ULONG color1 = CShape::textColor; pzd56
        CPoint2D pos = CShape::wPos; pzd52
        ULONG color2 = CShape::objColor; pzd52
    }
};
```

This is an ellipse!

```
CEllipse myEllip;
ULONG color1 = myEllip.textColor; pzd56
CPoint2D pos = myEllip.wPos; pzd56
ULONG color2 = myEllip.objColor; pzd56
```





## ► 实例

```
class CShape{  
    ULONG textColor;  
    char strText[256];  
protected:  
    CPoint2D wPos;  
public:  
    ULONG objColor;  
};
```

```
class CDonut:public CEllipse{  
    float ratio;  
public:  
    void Draw(){  
        ULONG color1 = CShape::textColor; pzd56  
        CPoint2D pos = CShape::wPos; pzd56  
        ULONG color2 = CShape::objColor; pzd56  
    }  
};
```

```
class CEllipse:private CShape{  
    float x_radius, y_radius;  
public:  
    void Draw();  
};
```

```
CEllipse myEllip;  
ULONG color1 = myEllip.textColor; pzd56  
CPoint2D pos = myEllip.wPos; pzd56  
ULONG color2 = myEllip.objColor; pzd56
```





### ► 保护继承 (Protected)

- 基类的**公有成员和保护成员**被继承后作为派生类的**保护成员**
- 基类的**私有成员**在派生类中**不能**被直接访问
- 如果将派生类作为新的基类继续派生时，基类成员可以沿继承树继续传播





## ► 实例

```
class CShape
{
    ULONG textColor;
    char strText[256];
protected:
    CPoint2D wPos;
public:
    ULONG objColor;
};
```

```
class CEllipse:protected CShape
{
    float x_radius, y_radius;
public:
    void Draw()
    {
        ULONG color1 = CShape::textColor; pzd56
        CPoint2D pos = CShape::wPos; pzd52
        ULONG color2 = CShape::objColor; pzd52
    }
};
```

This is an ellipse!

```
CEllipse myEllip;
ULONG color1 = myEllip.textColor; pzd56
CPoint2D pos = myEllip.wPos; pzd56
ULONG color2 = myEllip.objColor; pzd56
```





## ► 实例

```
class CShape{  
    ULONG textColor;  
    char strText[256];  
protected:  
    CPoint2D wPos;  
public:  
    ULONG objColor;  
};
```

```
class CDonut:public CEllipse{  
    float ratio;  
public:  
    void Draw(){  
        ULONG color1 = CShape::textColor; pzd56  
        CPoint2D pos = CShape::wPos; pzd52  
        ULONG color2 = CShape::objColor; pzd52  
    }  
};
```

```
class CEllipse:protected CShape{  
    float x_radius, y_radius;  
public:  
    void Draw();  
};
```

```
CEllipse myEllip;  
ULONG color1 = myEllip.textColor; pzd56  
CPoint2D pos = myEllip.wPos; pzd56  
ULONG color2 = myEllip.objColor; pzd56
```





## ▶ 基类成员在派生类中的访问控制属性

[width=8em,trim=l] 基类属性继承方式	public	protected	private
green!95!black [gray].8 public	public	protected	不可访问
yellow!95!black [gray].8 protected	protected	protected	不可访问
red!95!black [gray].8 private	private	private	不可访问





## ▶ 派生类构造函数的定义

派生类名 (参数总表): 基类名1(参数表1), …, 基类名 m(参数表 m),

成员对象名1(参数表1), …, 成员对象名 n(参数表 n)

{

派生类新增成员的初始化;

}

初始化列表





### ▶ 形状类派生椭圆类

```
class CEllipse:private CShape
{
    float x_radius, y_radius;
public:
    CEllipse() {x_radius = y_radius = 50;}
    CEllipse(float rx, float ry, CPoint2D w, char *strText, ULONG objColor=0xBBE0E3,
             ULONG textColor=0):CShape(w, strText,objColor, textColor)
    {
        x_radius = rx;
        y_radius = ry;
    }
    void Draw();
    void ShowPos();
};
```

初始化列表





### ► 椭圆类派生圆环类

```
class CDonut:public CEllipse
{
    float ratio;
public:
    CDonut() {ratio = 0.5;}
    CDonut(float r, float rx, float ry, CPoint2D w, char *strText,
           ULONG objColor=0xBBE0E3,
           ULONG textColor=0):CEllipse(rx, ry, w, strText, objColor, textColor)
    {
        ratio = r;
    }
    void Draw();
    void ShowPos();
};
```

初始化列表





## ▶ 单继承的构造与析构

- ▶ 首先调用基类成员类构造函数
- ▶ 然后调用基类构造函数
- ▶ 再调用派生类成员类的构造函数
- ▶ 最后调用派生类构造函数
- ▶ 当派生类对象析构时，各析构函数调用顺序正好相反





## ► 单继承的构造与析构

```
// 例 05-08-01: ex05-08-01.cpp
// 构造函数和析构函数的演示

#include <iostream>

using namespace std;
class memObj
{
    int a;
public:
    memObj(int x)
    {
        a = x;
        cout << "Constructing member object " << a << endl;
    }
    ~memObj()
    {
        cout << "Destructing member object" << a << endl;
    }
};
```





## ► 单继承的构造与析构

```
// 例 05-08-02: ex05-08-02.cpp
// 定义一个简单的类，包含一个属性，
// 构造函数和析构函数

class base
{
    memObj obj1;
public:
    //该构造函数有成员初始化列表
    base():obj1(1)
    {
        cout << "Constructing base\n";
    }
    ~base()
    {
        cout << "Destructuring base\n";
    }
};
```

```
// 例 05-08-03: ex05-08-03.cpp
// 定义类 derived，该类继承 base 类

class derived: public base
{
    memObj obj2;
public:
    derived():obj2(2)
    {
        cout << "Constructing derived\n";
    }
    ~derived()
    {
        cout << "Destructuring derived\n";
    }
};
```





## ► 单继承的构造与析构

```
// 例 05-08-04: ex05-08-04.cpp

int main()
{
    derived ob;
    return 0;
}
```

X - □ testcpp

```
Constructing member object 1
Constructing base
Constructing member object 2
Constructing derived
Destructing derived
Destructing member object2
Destructing base
Destructing member object1

Process returned 0 (0x0) execution time : 0.003 s
Press ENTER to continue.
```





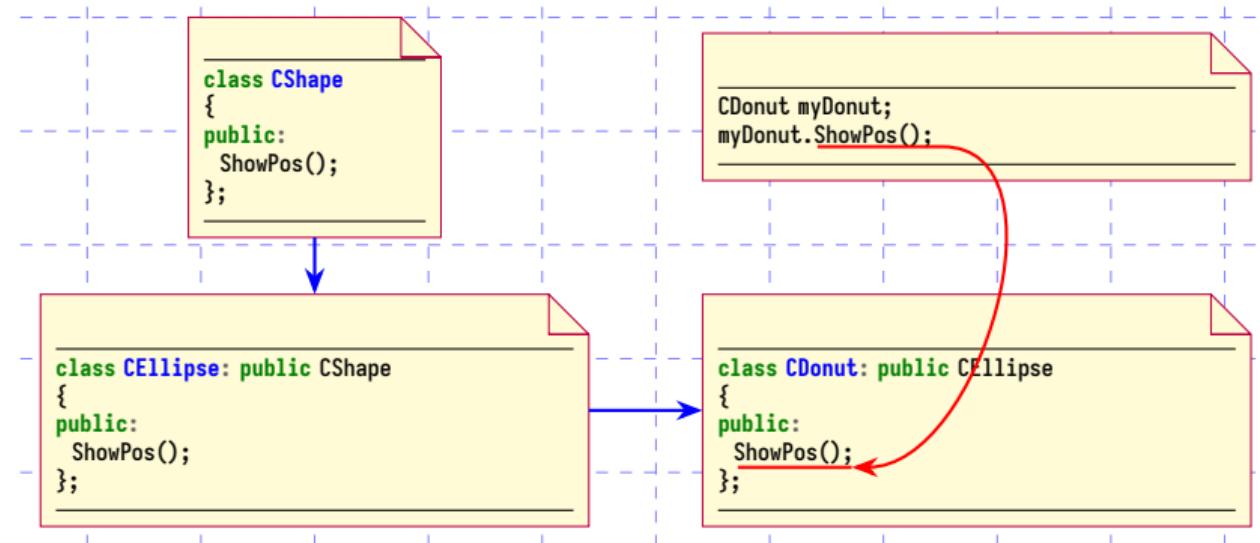
▶ **类型兼容**: 在公有派生的情况下，一个派生类对象可作为基类的对象来使用

- ▶ 派生类对象可以赋值给基类对象
- ▶ 派生类对象可以初始化基类的引用
- ▶ 派生类对象的地址可赋给指向基类的指针



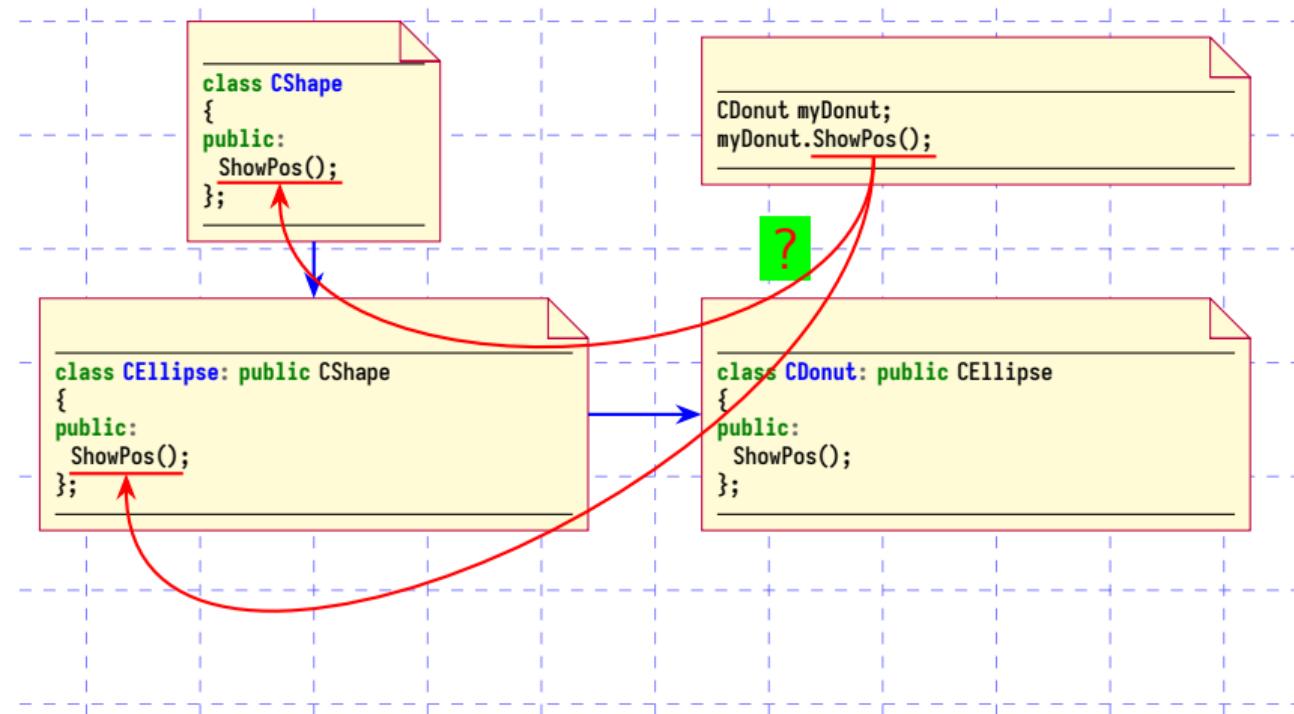


### ▶ 如何通过派生类对象调用基类中被覆盖的成员函数



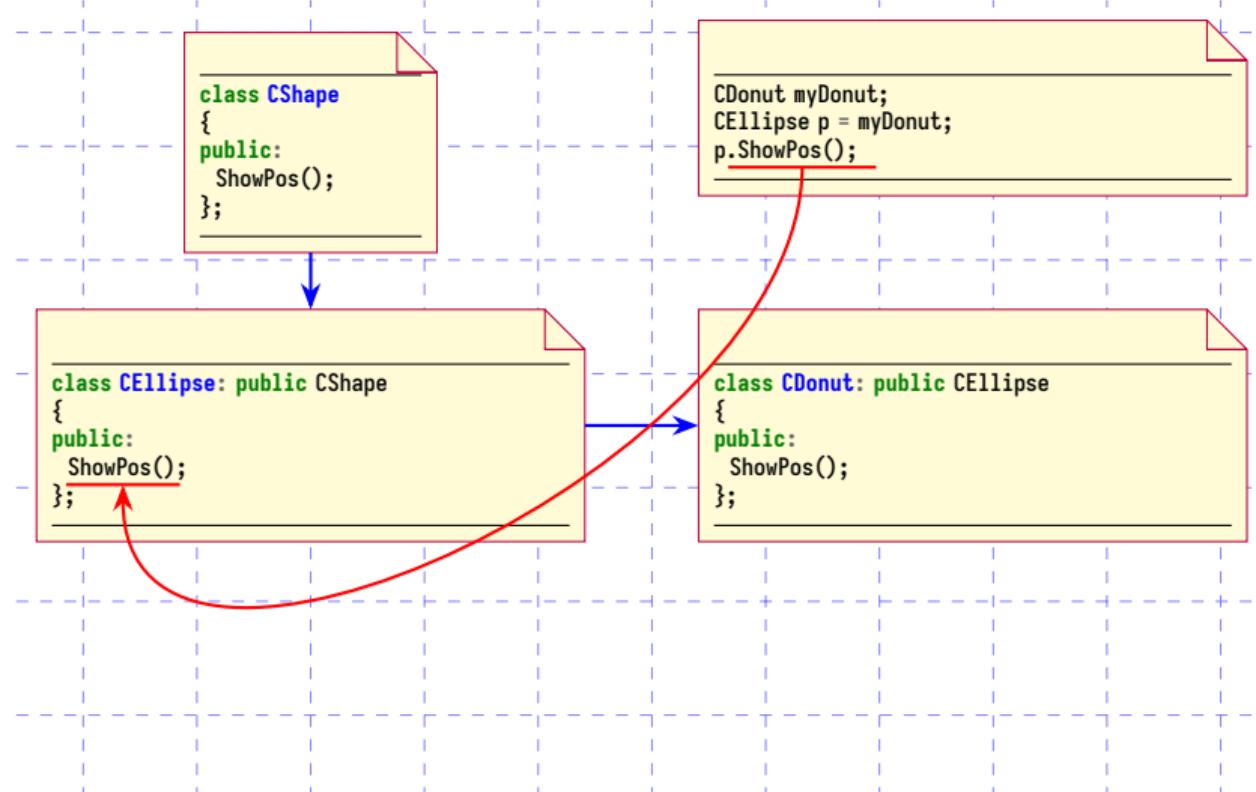


### ▶ 如何通过派生类对象调用基类中被覆盖的成员函数



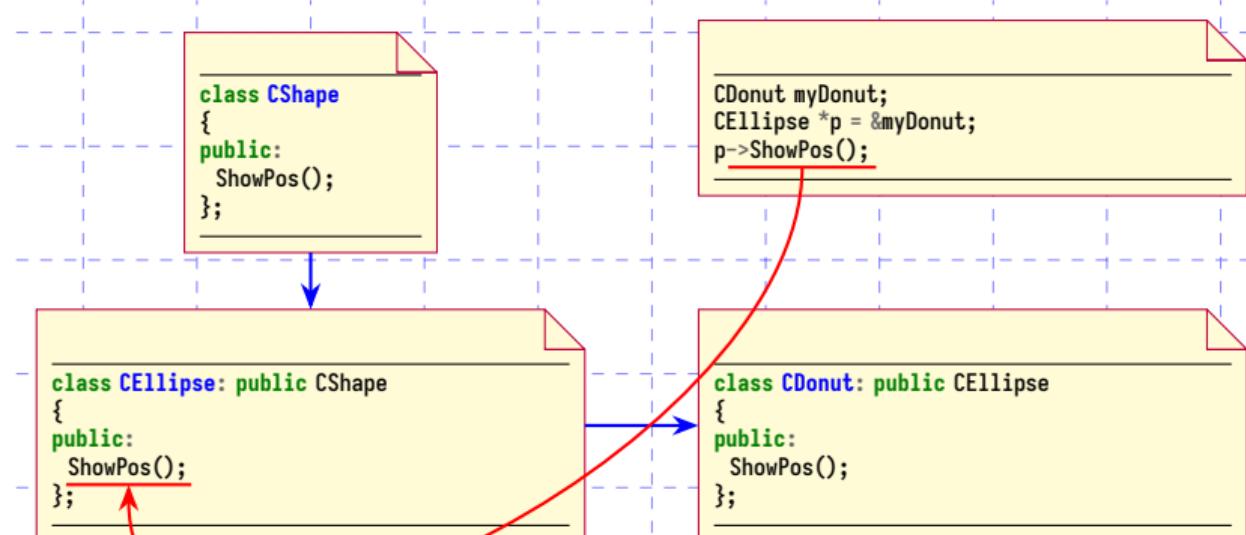


### ▶ 如何通过派生类对象调用基类中被覆盖的成员函数



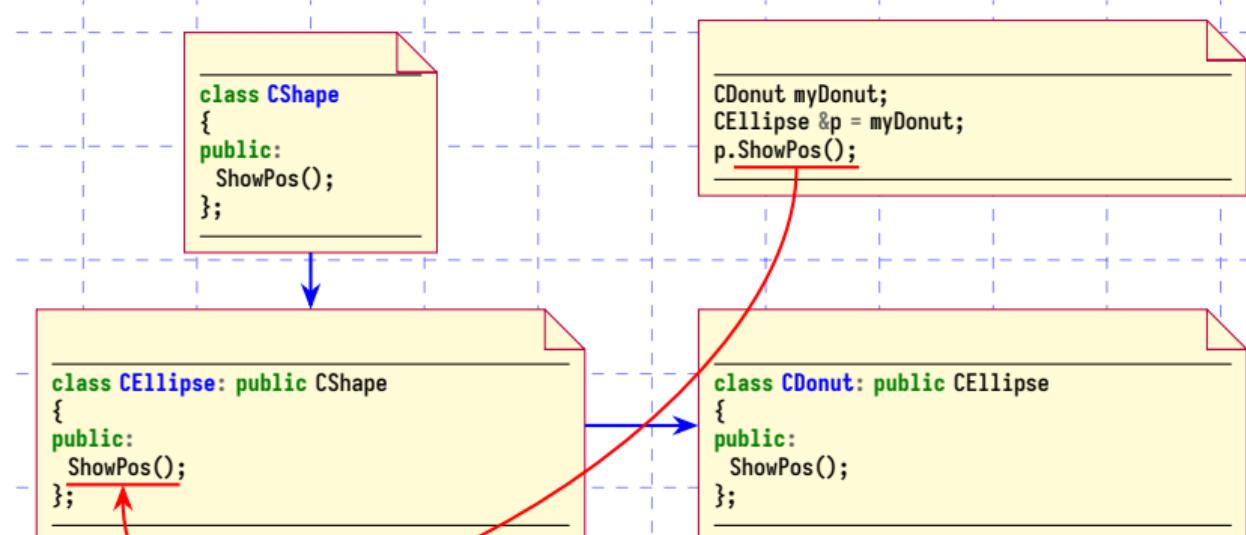


### ▶ 如何通过派生类对象调用基类中被覆盖的成员函数



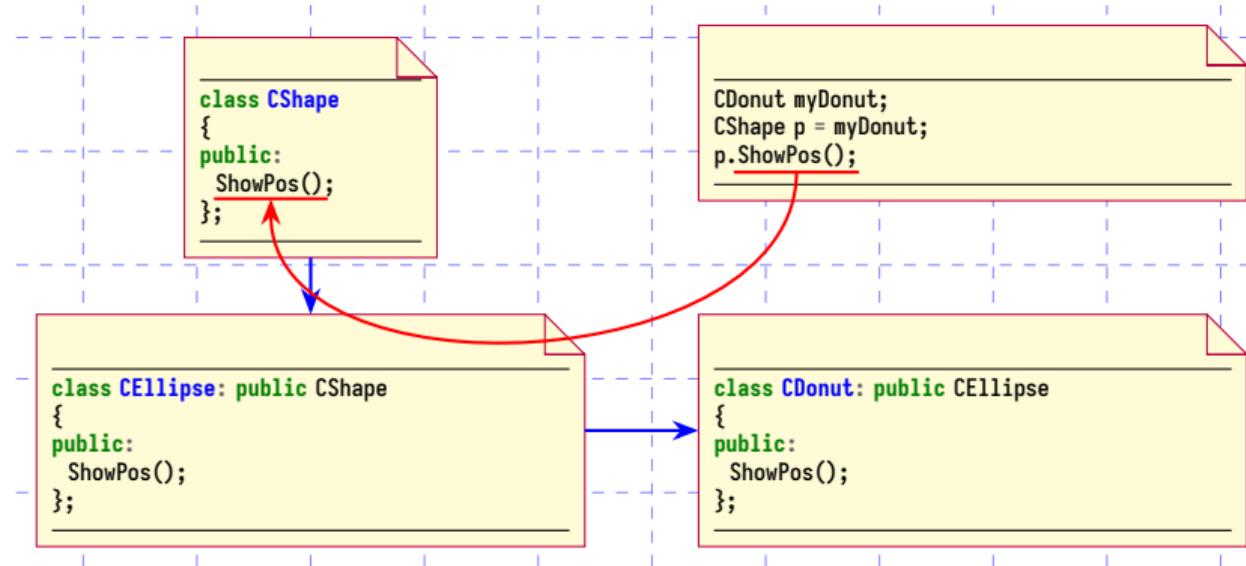


### ▶ 如何通过派生类对象调用基类中被覆盖的成员函数



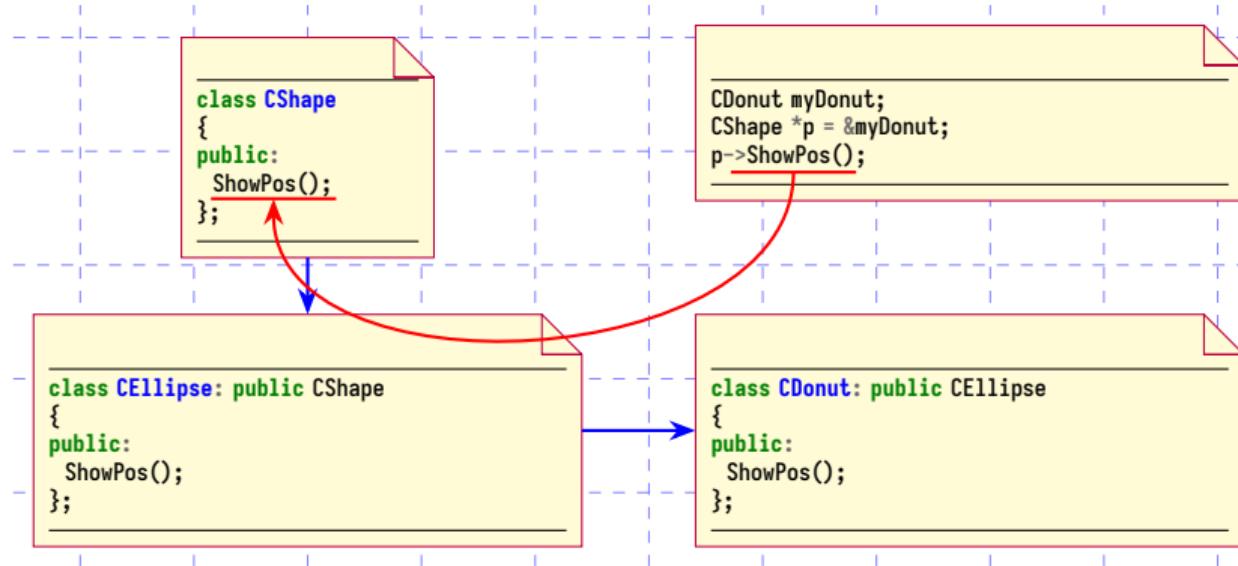


### ▶ 如何通过派生类对象调用基类中被覆盖的成员函数



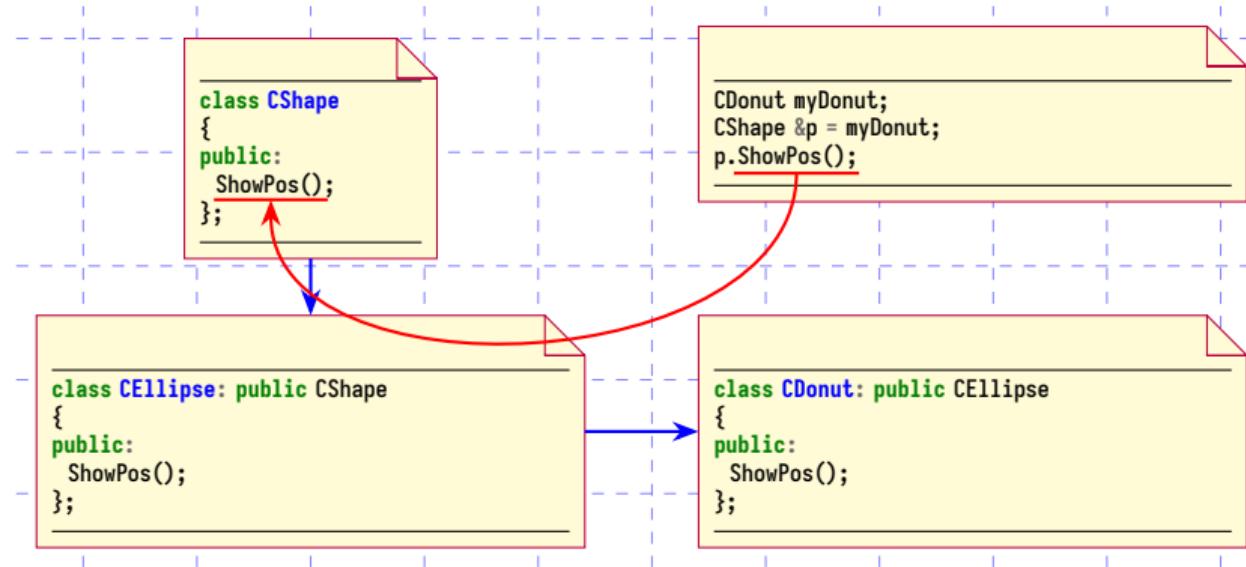


### ▶ 如何通过派生类对象调用基类中被覆盖的成员函数



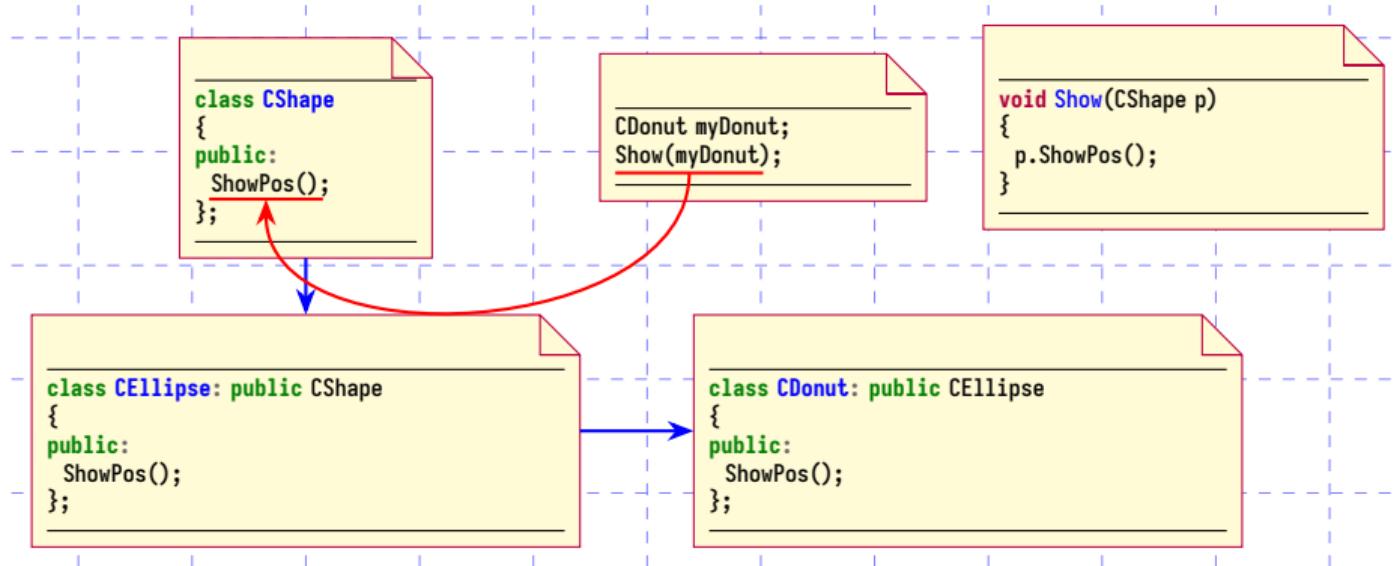


### ▶ 如何通过派生类对象调用基类中被覆盖的成员函数



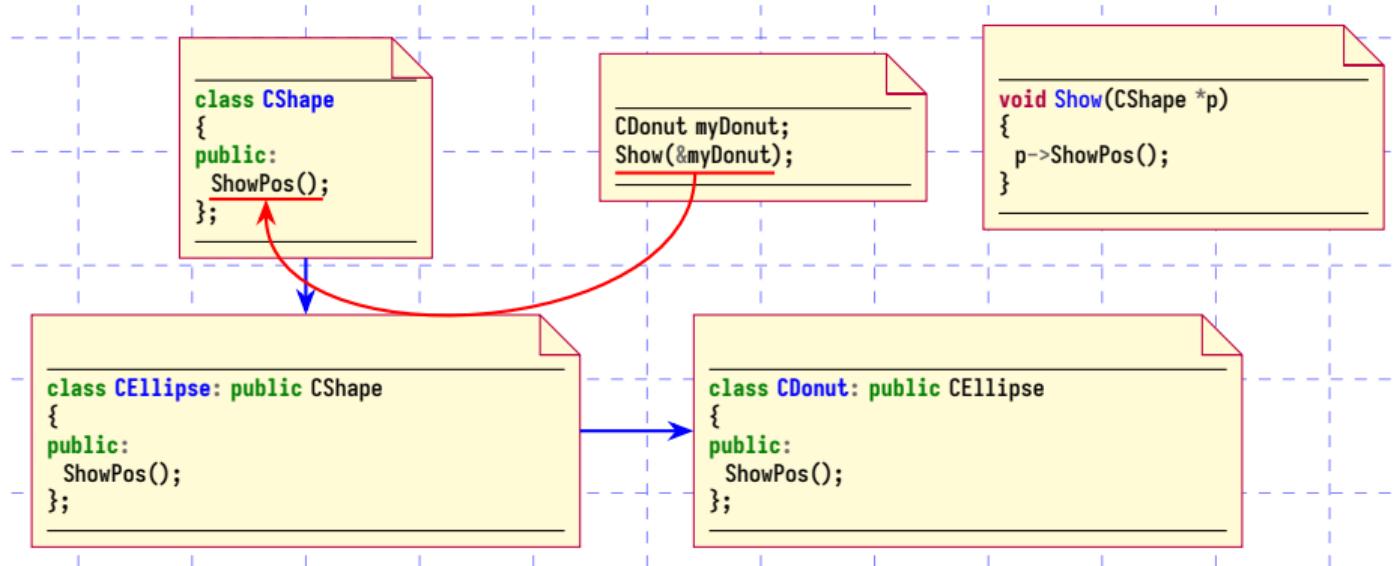


### ▶ 如何通过派生类对象调用基类中被覆盖的成员函数



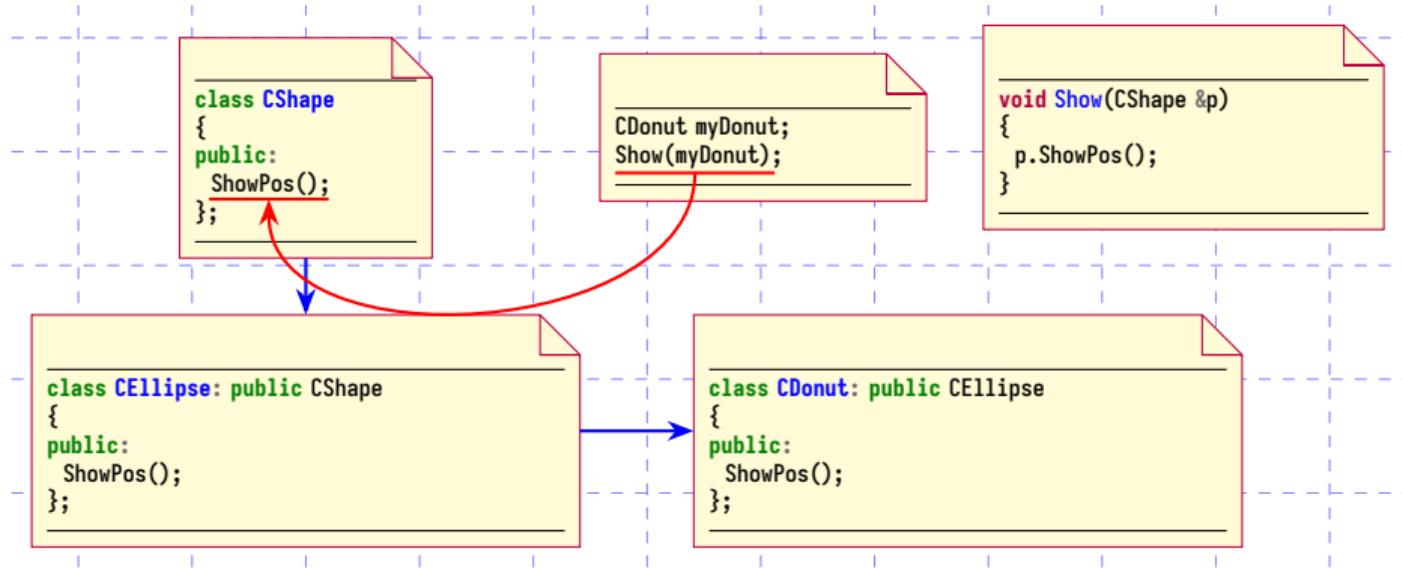


### ▶ 如何通过派生类对象调用基类中被覆盖的成员函数



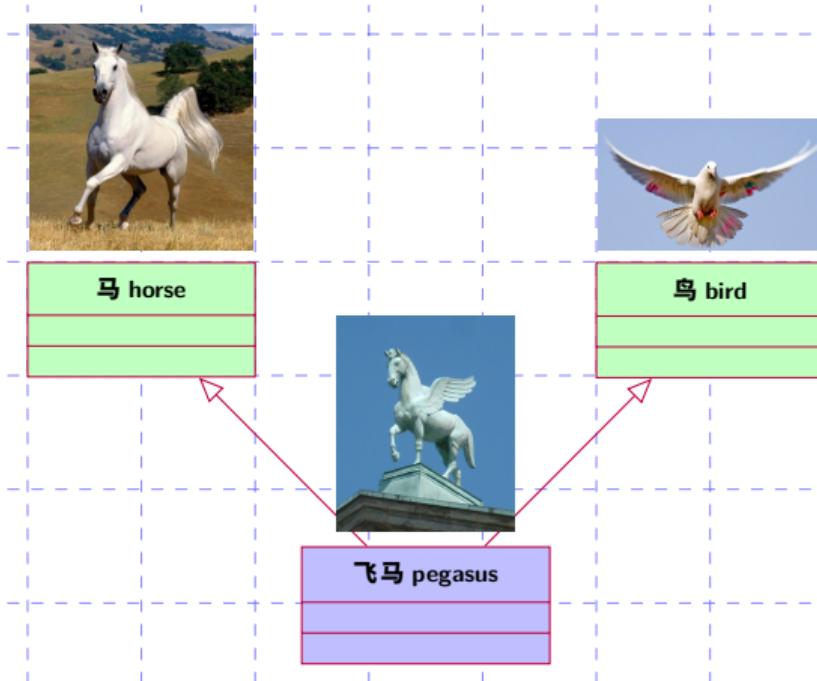


### ▶ 如何通过派生类对象调用基类中被覆盖的成员函数



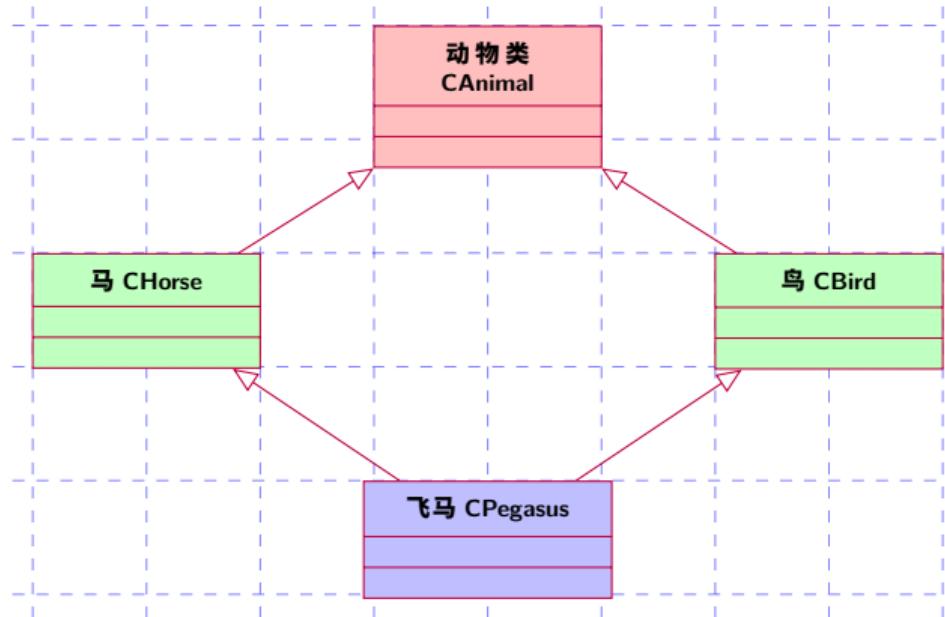


## ► 飞马 (Pegasus)





## ► 飞马 (Pegasus)





### ▶ 代码示例

```
// 例 05-09: ex05-09.cpp
// 类 protect 属性 和
// public 方法 的演示
#include <iostream>

using namespace std;
class base1
{
protected:
    int x;
public:
    void showx()
    {
        cout << x << "\n";
    }
};
class base2
{
protected:
    int y;
public:
    void showy()
    {
        cout << y << "\n";
    }
};
```

```
// 例 05-09: ex05-09.cpp
// 定义类 derived, 该类继承了 base1 和 base2

class derived: public base1, public base2
{
public:
    void set(int i, int j)
    {
        x=i;
        y=j;
    }
};
```

```
// 例 05-09: ex05-09.cpp

int main()
{
    derived ob;
    ob.set(10, 20);
    ob.showx();
    ob.showy();
}
```



### ▶ 多继承的构造与析构

- ▶ 调用各基类构造函数：调用顺序按基类被继承时声明的顺序，从左向右依次进行
- ▶ 调用派生类成员对象构造函数：调用顺序按其在类中定义的顺序依次执行
- ▶ 调用派生类构造函数





### ▶ 代码示例

```
// 例 05-10: ex05-10.cpp
// 构造函数和析构函数的演示

#include <iostream>

using namespace std;

class base1{
public:
    base1() {
        cout << "Constructing base1\n";
    }
    ~base1() {
        cout << "Destructing base1\n";
    }
};

class base2{
public:
    base2() {
        cout << "Constructing base2\n";
    }
    ~base2() {
        cout << "Destructing base2\n";
    }
};
```

```
// 例 05-10: ex05-10.cpp
// 定义类 derived, 该类继承了 base1 和 base2

class derived: public base1, public base2{
public:
    derived() {
        cout << "Constructing derived\n";
    }
    ~derived() {
        cout << "Destructing derived\n";
    }
};

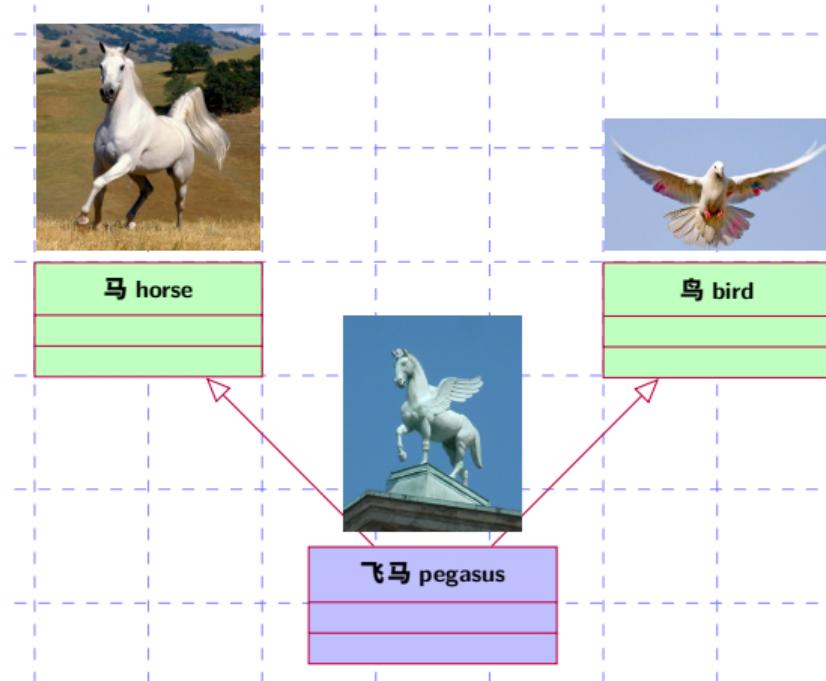
int main(){
    derived ob;
}
```

```
testcpp
Constructing base1
Constructing base2
Constructing derived
Destructing derived
Destructing base2
Destructing base1
Process returned 0 (0x0)   execution time : 0.002 s
Press ENTER to continue.
```



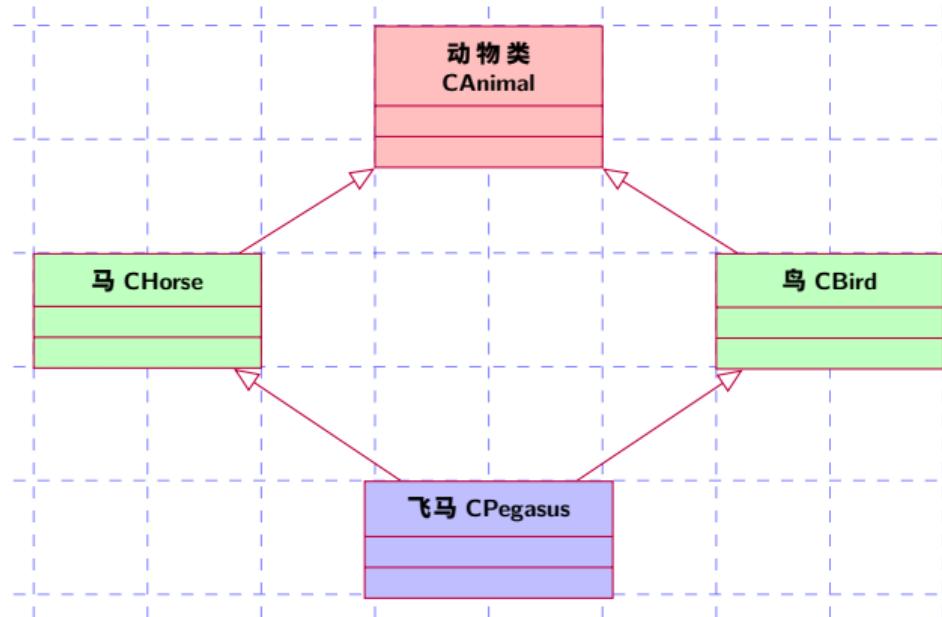


## ► 飞马 (Pegasus)



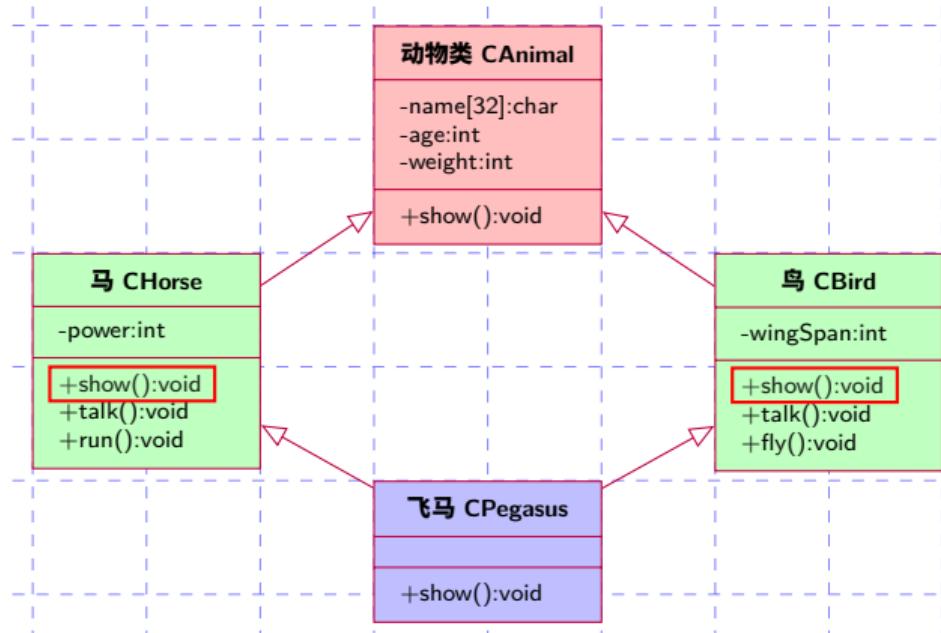


## ► 飞马 (Pegasus)





### ► 飞马 (Pegasus)





## ► 动物类

```
// 例 05-11: ex05-11.cpp
// 定义一个类 CAnimal

#include <iostream>
#include <cstring>

using namespace std;

class CAnimal
{
    char name[32];
    int age;
    int weight;
public:
    CAnimal(const char *strName="", int a=0, int w=0){
        strcpy(name, strName);
        age = a;
        weight = w;
        cout << "Animal constructor " << name << endl;
    }
    void Show(){
        cout << name << " " << age << " " << weight << endl;
    }
    ~CAnimal(){
        cout << "Animal destructor " << name << endl;
    }
};
```





## ► 鸟类

```
// 例 05-11: ex05-11.cpp
// 定义一个类 CBird，该类继承 CAnimal 类，并且增加了 wingSpan 属性和一些方法

class CBird: public CAnimal
{
    int wingSpan;
public:
    CBird(int ws=0, const char *strName="", int a=0, int w=0):
        CAnimal(strName, a, w)
    {
        wingSpan = ws;
        cout << "Bird constructor " << endl;
    }
    void Show(){
        CAnimal::Show();
        cout << "Wingspan:" << wingSpan << endl;
    }
    void Fly(){
        cout << "I can fly! I can fly!!" << endl;
    }
    void Talk(){
        cout << "Chirp..." << endl;
    }
    ~CBird(){
        cout << "Bird destructor " << endl;
    }
};
```





## ► 马类

```
// 例 05-11: ex05-11.cpp
// 定义一个类 CHorse，该类继承 CAnimal 类，并且增加了 power 属性和一些方法

class CHorse: public CAnimal
{
    int power;
public:
    CHorse(int pow=0, const char *strName="", int a=0, int w=0):
        CAnimal(strName, a, w)
    {
        power = pow;
        cout << "Horse constructor " << endl;
    }
    void Show(){
        CAnimal::Show();
        cout << "Power:" << power << endl;
    }
    void Run(){
        cout << "I can run! I run because I love to!!" << endl;
    }
    void Talk(){
        cout << "Whinny!..." << endl;
    }
    ~CHorse(){
        cout << "Horse destructor " << endl;
    }
};
```





### ► 飞马类

```
// 例 05-11: ex05-11.cpp
// 定义类 CPegasus , 该类继承 CHorse 和 CBird

class CPegasus : public CHorse, public CBird
{
public:
    CPegasus(const char *strName="", int a=0, int w=0, int ws=0, int pow=0):
        CHorse(pow, strName, a, w), CBird(ws, strName, a, w)
    {
        cout << "Pegasus constructor" << endl;
    }
    void Talk(){
        CHorse::Talk();
    }
    ~CPegasus(){
        cout << "Pegasus destructor" << endl;
    }
};

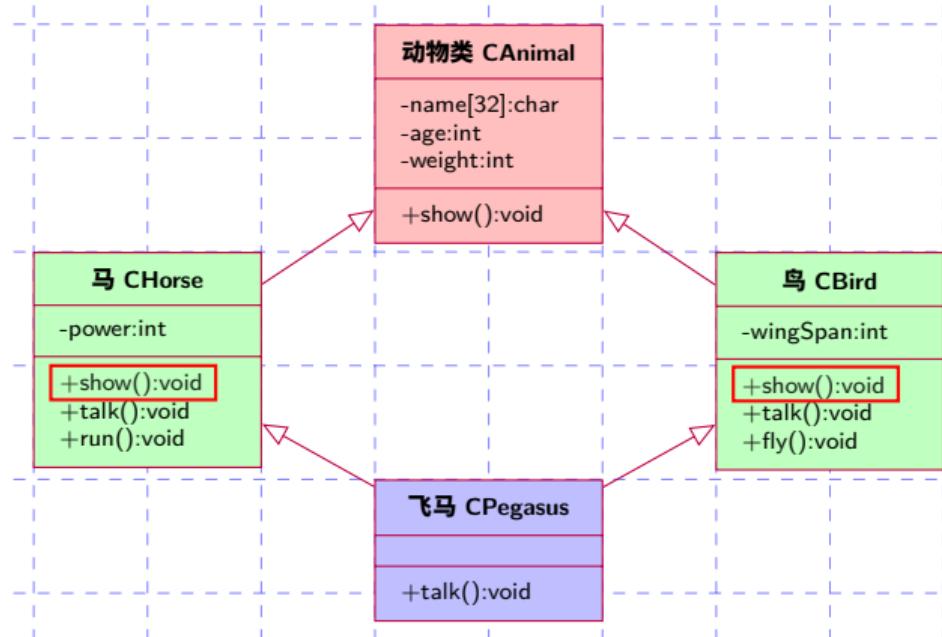
int main(){
    CPegasus pegObj("Eagle", 5, 100, 2, 500);

    return 0;
}
```





## ► 多继承的二义性问题





- ▶ 派生类的多个基类中拥有同名成员时，继承后通过对象调用同名成员将出现二义性

```
Cegasus pegObj("Pegasus", 5, 800, 2, 10000);
pegObj.Show(); pzd56

F:\CPP\example... 112      error: request for member 'Show' is ambiguous
F:\CPP\example... 19       note: candidates are: void CAnimal::Show()
F:\CPP\example... 39       note:                      void CBird::Show()
F:\CPP\example... 67       note:                      void CHorse::Show()
                                === Build failed: 1 error(s), 8 warning(s) (0
```





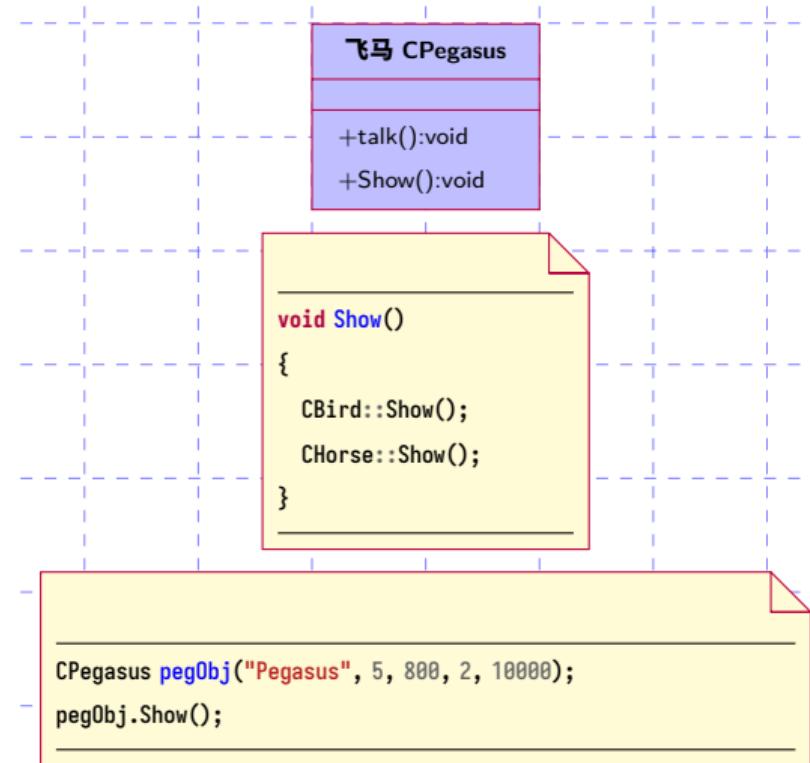
### ► 解决方法 1：类型兼容

```
CPegasus pegObj("Pegasus", 5, 800, 2, 10000);
CBird birdObj = pegObj;
birdObj.Show();
CHorse horObj = pegObj;
horObj.Show();
```





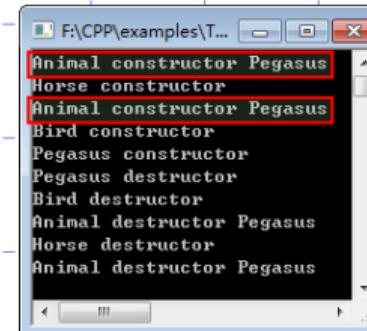
### ► 解决方法 2：成员重定义





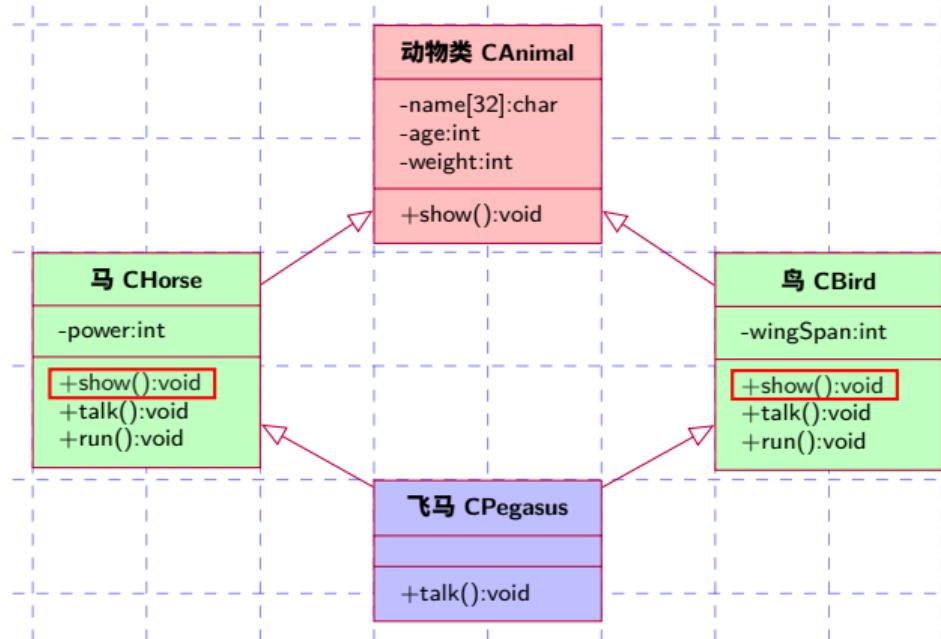
### ► 间接二义性：基类构造函数两次被调用

```
CPegasus(char *strName="", int a=0, int w=0, int ws=0,intpow=0):  
    CHorse(pow, strName, a,w), CBird(ws, strName, a, w)  
{  
    cout << "Pegasus constructor" << endl;  
}  
:  
:  
int main()  
{  
    CPegasus pegObj("Pegasus", 5, 100, 2, 500);  
    return 0;  
}
```





► 同名数据成员在内存中同时拥有多个拷贝





## ► 间接二义性

- 如何解决从不同途径继承来的同名的数据成员在内存中有不同的拷贝问题（**调用一次构造函数**）？





### ▶ 定义

```
class 派生类名:virtual 继承方式 基类名
class CHorse: virtual public CAnimal
class CBird: virtual public CAnimal
```

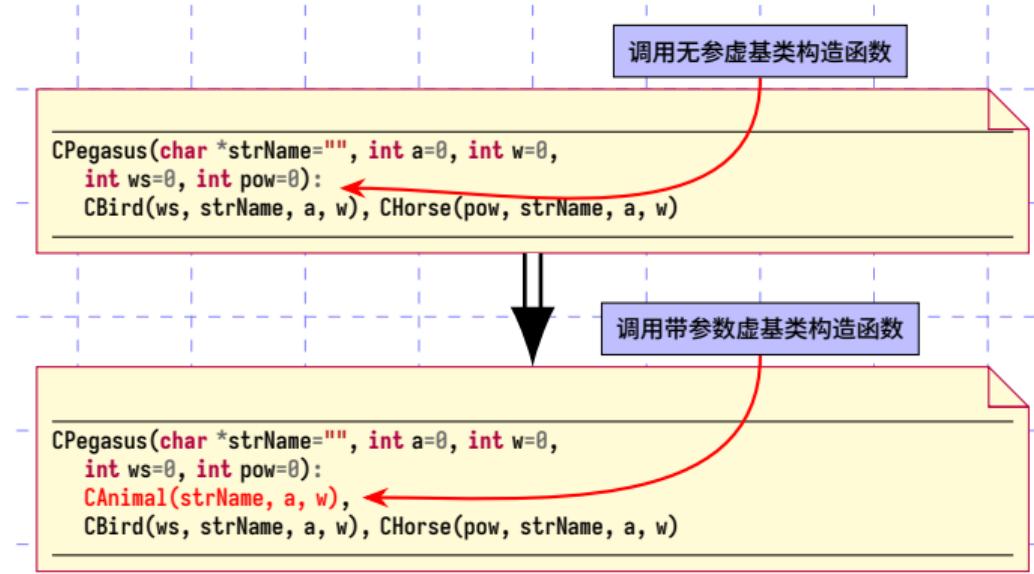
### ▶ 作用

▶ 虚基类构造函数只被调用一次





### ► 间接二义性：基类构造函数两次被调用





### ► 构造函数调用

```
CPegasus pegObj("Pegasus", 5, 800, 2, 10000);
```

```
F:\CPP\examples\T...
Animal constructor Pegasus
Horse constructor
Bird constructor
Pegasus constructor
Pegasus destructor
Bird destructor
Horse destructor
Animal destructor Pegasus

Process returned 0 (0x0)   e
Press any key to continue.
```





- ▶ 接口的多继承有一定价值，但应避免实现多继承。在决定使用多继承之前，先仔细考虑其他替代方案。
  
- ▶ 继承是面向对象提供的另外一种复用代码的重要机制，继承使得派生类与基类之间具有接口的相似性。派生类可看作是基类的特定子类型，派生类对象可替代基类对象。
  
- ▶ 与包含相比，继承需要更多的技巧，而且易出错，包含是面向对象编程中的主要技术之一。





## ► 何时使用继承，何时使用包含？

- ▶ 如果多个类共享数据而非行为，应该创建这些类可以包含的共用对象。
- ▶ 如果多个类共享行为而非数据，应该让它们从共同的基类继承而来，并在基类里定义共用的操作。
- ▶ 如果多个类既共享数据也共享行为，应该让它们从一个共同的基类继承而来，并在基类里定义共用的数据和操作。
- ▶ 如果想由基类控制接口，使用继承；如果想自己控制接口，使用包含。



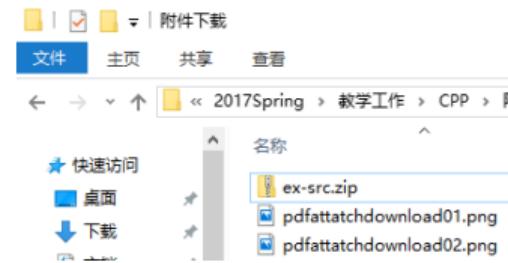
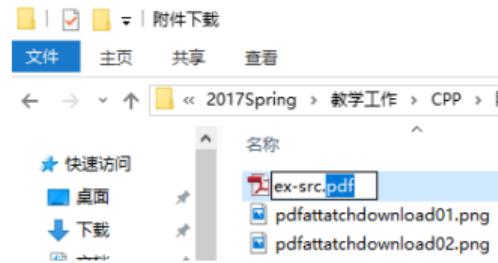
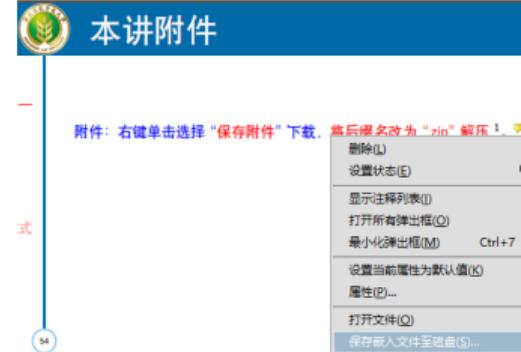
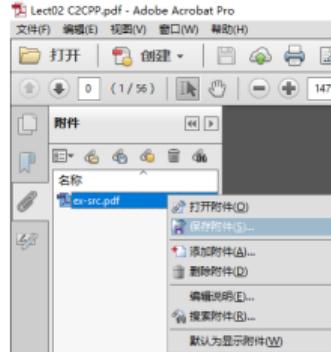


# 本讲附件

| 附件

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP  
  
Nine, G.

附件：右键单击该链接，选择“保存附件”下载，将后缀名改为“.zip”解压<sup>14 15</sup>。



<sup>14</sup>请退出全屏模式后点击该链接。

<sup>15</sup>以 Adobe Acrobat Reader 为例。



- ▶ 为派生类提供一致的接口 (Uniform Interface)
  
- ▶ **多态性**: 同一消息发送给不同对象执行不同操作





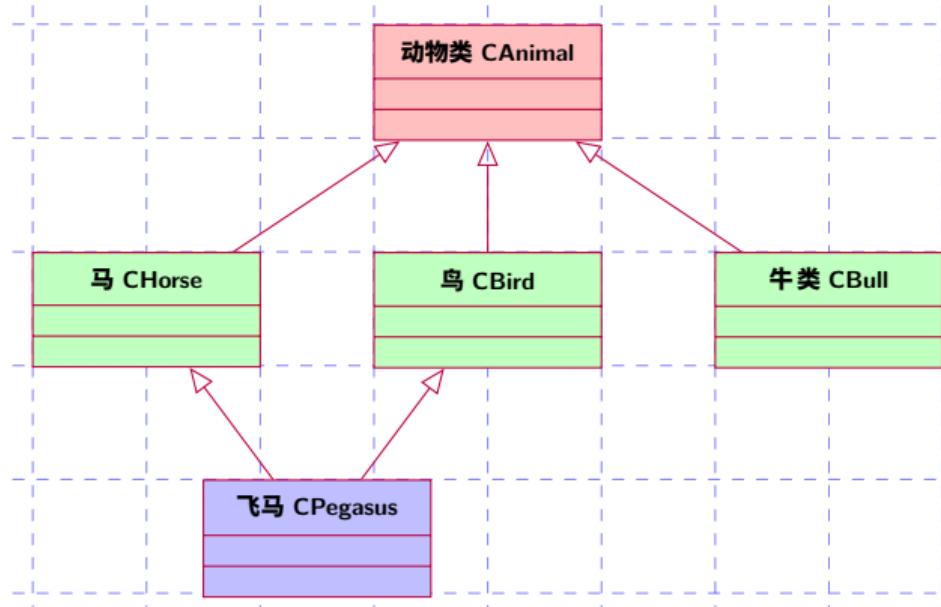
## ► 设计一个动物园类

- 包含大量不同种类的动物
  
- 可以显示各种动物的属性（如名字、年龄等）
  
- 可以输出各种动物的叫声





## ▶ 设计一个动物园类



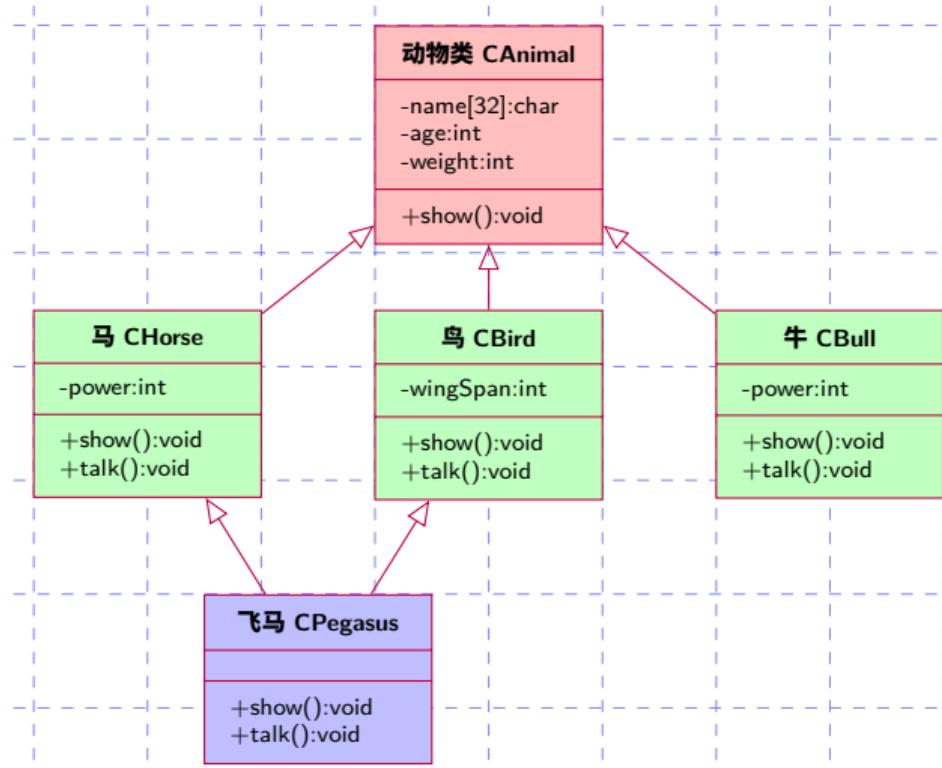


## ► 设计一个动物园类





## ▶ 设计一个动物园类





## ► 设计一个动物园类

```
// 例 01-zoo-CAnimal: ex01-zoo-CAnimal.cpp

class CAnimal
{
    char name[32];
    int age;
    int weight;
public:
    CAnimal(const char *strName = "", int a = 0, int w = 0)
    {
        strcpy(name, strName);
        age = a;
        weight = w;
    }
    void Show()
    {
        cout << name << " " << age << " " << weight << " ";
    }
};
```





## ▶ 设计一个动物园类

```
// 例 01-zoo-CBird: ex01-zoo-CBird.cpp
// 虚继承的演示

class CBird: virtual public CAnimal
{
protected:
    int wingSpan;
public:
    CBird(const char *strName = "", int a = 0, int w = 0, int ws = 0):
        CAnimal(strName, a, w)
    {
        wingSpan = ws;
    }
    void Show()
    {
        CAnimal::Show();
        cout << wingSpan << endl;
    }
    void Talk()
    {
        cout << "Chirp..." << endl;
        PlaySound("Sound\\eagle.wav", NULL, SND_FILENAME | SND_SYNC);
    }
};
```





## ▶ 设计一个动物园类

```
// 例 01-zoo-CHorse: ex01-zoo-CHorse.cpp
// 虚继承的演示

class CHorse: virtual public CAnimal
{
protected:
    int power;
public:
    CHorse(const char *strName = "", int a = 0, int w = 0, int pow = 0):
        CAnimal(strName, a, w)
    {
        power = pow;
    }
    void Show()
    {
        CAnimal::Show();
        cout << power << endl;
    }
    void Talk()
    {
        cout << "Whinny..." << endl;
        PlaySound("Sound\\horse.wav", NULL, SND_FILENAME | SND_SYNC);
    }
};
```





## ▶ 设计一个动物园类

```
// 例 01-zoo-CPegasus: ex01-zoo-CPegasus.cpp
// 定义一个类 CPegasus, 多重继承

class CPegasus : public CAnimal, public CBird
{
public:
    CPegasus(const char *strName = "", int a = 0, int w = 0, int ws = 0, int pow = 0):
        CAnimal(strName, a, w),
        CAnimal(strName, a, w, pow),
        CBird(strName, a, w, ws)

    {
    }
    void Show()
    {
        CAnimal::Show();
        cout << wingSpan << " " << power << endl;
    }
    void Talk()
    {
        CAnimal::Talk();
    }
};
```





## ▶ 设计一个动物园类

```
// 例 01-zoo-CBull: ex01-zoo-CBull.cpp
// 继承的演示

class CBull: public CAnimal
{
    int power;
public:
    CBull(const char *strName = "", int a = 0, int w = 0, int pow = 0):
        CAnimal(strName, a, w)
    {
        power = pow;
    }
    void Show()
    {
        CAnimal::Show();
        cout << power << endl;
    }
    void Talk()
    {
        cout << "Moo..." << endl;
        PlaySound("Sound\bull.wav", NULL, SND_FILENAME | SND_SYNC);
    }
};
```





## ▶ 设计一个动物园类

```
// 例 01-zoo-main: ex01-zoo-main.cpp
// 实例化 CBird, CHorse, CBull, CPegasus 的对象，并调用相应的方法

int main()
{
    CBird birdObj("Eagle", 5, 50, 2);
    CHorse horObj("Mogolia horse", 5, 1000, 10000);
    CBull bullObj("Africa ox", 3, 2000, 20000);
    CPegasus pegObj("Pegasus", 5, 5000, 4, 100000);

    birdObj.Show();
    birdObj.Talk();

    horObj.Show();
    horObj.Talk();

    bullObj.Show();
    bullObj.Talk();

    pegObj.Show();
    pegObj.Talk();

    return 0;
}
```





- ▶ 如果有大量的不同种类的动物，如何进行管理？
- ▶ 设计一个动物园类

```
// 例 02-zoo: ex02-zoo.cpp

class CZoo
{
    int sizeBird, sizeHorse, sizeBull, sizePeg;
    CBird *m_bird;
    CHorse *m_horse;
    CBull *m_bull;
    CPegasus *m_peg;
public:
    CZoo()
    {
        sizeBird = sizeBull = sizeHorse = sizePeg = 0;
    }
    void AddBird(CBird &bird) {}
    void AddBull(CBull &bull) {}
    void AddHorse(CHorse &hor) {}
    void AddPegasus(CPegasus &peg) {}
    void Show();
    void Talk();
};
```





## ▶ 多种动物

```
// 例 02-zoo-show: ex02-zoo-show.cpp
// 实现动物园的 Show 方法

void CZoo::Show()
{
    for (int i = 0; i < sizeBird; i++)
        m_bird[i].Show();
    for (int i = 0; i < sizeHorse; i++)
        m_horse[i].Show();
    for (int i = 0; i < sizeBull; i++)
        m_bull[i].Show();
    for (int i = 0; i < sizePeg; i++)
        m_peg[i].Show();
}
```

```
// 例 02-zoo-talk: ex02-zoo-talk.cpp
// 实现动物园的 Talk 方法

void CZoo::Talk()
{
    for (int i = 0; i < sizeBird; i++)
        m_bird[i].Talk();
    for (int i = 0; i < sizeHorse; i++)
        m_horse[i].Talk();
    for (int i = 0; i < sizeBull; i++)
        m_bull[i].Talk();
    for (int i = 0; i < sizePeg; i++)
        m_peg[i].Talk();
}
```





- ▶ 缺点：对具有相同基类的派生类对象需要分别设计
- ▶ 若派生类数量增加，管理复杂度上升

```
// 例 02-zoo: ex02-zoo.cpp

class CZoo
{
    int sizeBird, sizeHorse, sizeBull, sizePeg;
    CBird *m_bird;
    CHorse *m_horse;
    CBull *m_bull;
    CPegasus *m_peg;
public:
    CZoo()
    {
        sizeBird = sizeBull = sizeHorse = sizePeg = 0;
    }
    void AddBird(CBird &bird) {}
    void AddBull(CBull &bull) {}
    void AddHorse(CHorse &hor) {}
    void AddPegasus(CPegasus &peg) {}
    void Show();
    void Talk();
};
```





## ▶ 另外一种考虑方式：类型兼容

```
// 例 03-zoo-class: ex03-zoo-class.cpp

const int MAX_ANIM_NUM = 100;

class CZoo
{
    int size;
    CAnimal *m_animal[MAX_ANIM_NUM];
public:
    CZoo()
    {
        size = 0;
    }
    void Add(CAnimal *anim)
    {
        if(size < MAX_ANIM_NUM)
        {
            m_animal[size] = anim;
            size++;
        }
    }
    void Show();
    void Talk();
};
```

```
// 例 03-zoo-show-talk:
→ ex03-zoo-show-talk.cpp
// 实现类中的成员函数

void CZoo::Show()
{
    for (int i = 0; i < size; i++)
        m_animal[i]->Show();
}

void CZoo::Talk()
{
    for (int i = 0; i < size; i++)
        m_animal[i]->Talk();
}
```

In member function 'void CZoo::Talk()':  
143 error: 'class CAnimal' has no member named 'Talk'



- ▶ 问题 1：基类不能调用派生类成员函数 Talk
- ▶ 解决办法：虚函数

```
// 例 04-zoo-virtfun-show.cpp
// 实现 CAnimal 中的虚函数 Show

virtual void CAnimal::Show()
{
    cout << name << " " << age <<
    " " << weight << " ";
}
```

```
// 例 04-zoo-virtfun-talk.cpp
// 实现 CAnimal 中的虚函数 Talk

virtual void CAnimal::Talk()
{
    cout << "Do nothing!" << endl;
}
```





- ▶ 问题 1：基类不能调用派生类成员函数 Talk
- ▶ 解决办法：虚函数

```
// 例 04-zoo-main: ex04-zoo-main.cpp

int main()
{
    CBird birdObj("Eagle", 5, 50, 2);
    CHorse horObj("Mogolia horse", 5, 1000, 10000);
    CBull bullObj("Africa ox", 3, 2000, 20000);
    CPegasus pegObj("Pegasus", 5, 5000, 4, 100000);

    CZoo zoo;

    zoo.Add(&birdObj);
    zoo.Add(&horObj);
    zoo.Add(&bullObj);
    zoo.Add(&pegObj);

    zoo.Show();
    zoo.Talk();

    return 0;
}
```





### ▶ 定义

---

`virtual` 函数类型 函数表 (形参表)

```
{  
    函数体;  
}
```

---

- ▶ 只有类的成员函数才能声明为虚函数
- ▶ 虚函数不能是静态成员函数，也不能是友元函数
- ▶ 若在基类中定义虚函数，在派生类中还需重新定义
- ▶ 构造函数不能是虚函数，析构函数可以是虚函数





### ► 定义

---

```
virtual ~类名 ()  
{  
    函数体;  
}
```

---





## ▶ 虚析构函数

// 例 05-dtor-base: ex05-dtor-base.cpp  
// 构造函数和析构函数的演示

```
class A
{
public:
    ~A()
    {
        cout << "A::~A() is called." << endl;
    }
    A()
    {
        cout << "A::A() is called." << endl;
    }
};
```

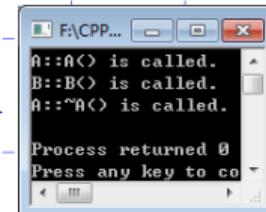
① pzd56

// 例 05-dtor-main: ex05-dtor-main.cpp

```
int main()
{
    A *b = new B(10);
    delete b;
    return 0;
}
```

// 例 05-dtor-derived: ex05-dtor-derived.cpp  
// 定义类 B, 该类继承类 A

```
class B: public A
{
private:
    int *ip;
public:
    B(int size = 0)
    {
        ip = new int[size];
        cout << "B::B() is called." << endl;
    }
    ~B()
    {
        cout << "B::~B() is called." << endl;
        delete []ip;
    }
};
```





## ▶ 虚析构函数

// 例 06-vdtor-base: ex06-vdtor-base.cpp  
// 类虚析构函数的演示

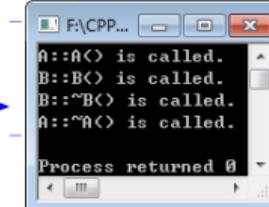
```
class A
{
public:
    virtual ~A() //虚析构函数
    {
        cout << "A::~A() is called." << endl;
    }
    A()
    {
        cout << "A::A() is called." << endl;
    }
};
```

// 例 05-dtor-main: ex05-dtor-main.cpp

```
int main()
{
    A *b = new B(10);
    delete b;
    return 0;
}
```

// 例 05-dtor-derived: ex05-dtor-derived.cpp  
// 定义类 B, 该类继承类 A

```
class B: public A
{
private:
    int *ip;
public:
    B(int size = 0)
    {
        ip = new int[size];
        cout << "B::B() is called." << endl;
    }
    ~B() //①
    {
        cout << "B::~B() is called." << endl;
        delete []ip;
    }
};
```





- ▶ 定义基类的析构函数是虚析构函数，当程序运行结束时，通过基类指针删除派生类对象时，先调用派生类析构函数，然后调用基类析构函数





### ► 纯虚函数定义

---

`virtual 函数类型 函数名 (参数表)=0;`

---

### ► 抽象类定义

- 类中含有至少一个纯虚函数





► 回顾动物类中的虚函数 Talk→ 函数体无意义

---

```
virtual void CAnimal::Talk()  
{  
    cout << "Do nothing!" << endl;  
}
```

---

► 解决办法：纯虚函数

---

```
virtual void CAnimal::Talk() = 0;
```

---





### ► 不能实例化（不能创建类的对象）

// 例 07-abstract-main.cpp

```
int main()
{
    CAnimal anim("God"); pzd56

    anim.Show();
    anim.Talk();

    return 0;
}
```

In function 'int main()':

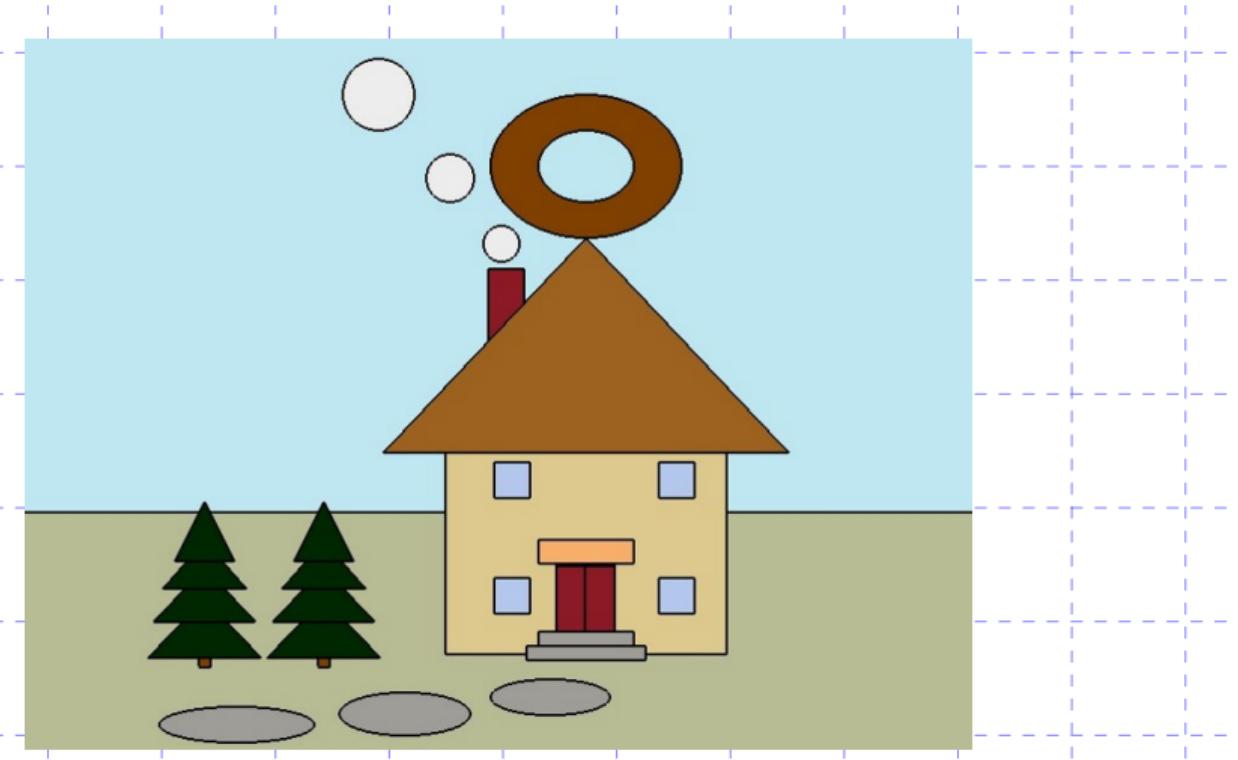
```
142 error: cannot declare variable 'anim' to be of abstract type 'CAnimal'
8 note: because the following virtual functions are pure within 'CAnimal':
24 note:     virtual void CAnimal::Talk()

==== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```



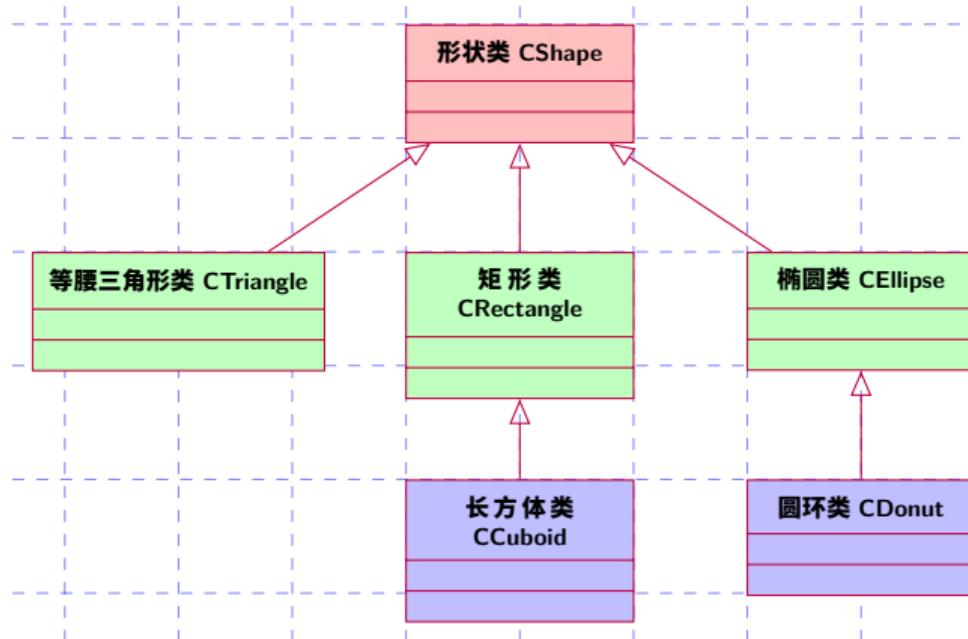


## ► 例子：甜甜圈小屋分析





### ► 例子：甜甜圈小屋类设计





### ▶ 例子：甜甜圈小屋代码设计

```
// 例 08-abstract-Shape: ex08-abstract-Shape.cpp
// 纯虚函数的定义

class CShape{
protected:
    ULONG textColor;
    string strText;
    CPoint2D wPos;
    ULONG objColor;
public:
    CShape():wPos(CPoint2D(400,300)){
        textColor = 0x000000;
        objColor = 0xBBE0E3;
        strText = "";
    }
    CShape(CPoint2D w, string strText="",
           ULONG objColor = 0xBBE0E3,
           ULONG textColor = 0):wPos(w){
        this->textColor = textColor;
        this->objColor = objColor;
        this->strText = strText;
    }
    void Translate(float x, float y);
    void DrawText();
    virtual void Draw() = 0;
    virtual void ShowPos() = 0;
};
```



### ▶ 例子：甜甜圈小屋代码设计

```
// 例 08-abstract-ShapeArr: ex08-abstract-ShapeArr.cpp

#ifndef SHAPEARR_H
#define SHAPEARR_H

#include "Donut.h"
#include "Triangle.h"
#include "Rectangle.h"
// 定义类 CShapeArr，该类具有类 CShape 的成员变量
const int MAX_SHAPE_NUM = 1000;

class CShapeArr
{
    CShape *m_shape[MAX_SHAPE_NUM];
    int size;
public:
    CShapeArr();
    void Add(CShape *node);
    bool GetInput();
    void ShowPos();
    void Translate(int x, int y);
    void Draw();
    ~CShapeArr();
};

#endif // SHAPEARR_H
```

基类类型指针数组

基类类型指针形参





### ► 例子：甜甜圈小屋代码设计

```
// 例 08-abstrace-ShapeArr: ex08-abstract-ShapeArr.cpp
// 类成员函数的实现
```

```
void CShapeArr::Draw()
{
    for (int i=0; i<size; i++)
    {
        m_shape[i]->Draw();
    }
}
```

基类类型指针指向派生类对象，调用派生类中的函数

```
void CShapeArr::ShowPos()
{
    for (int i=0; i<size; i++)
        m_shape[i]->ShowPos();
}
```

```
void CShapeArr::Translate(int x, int y)
{
    for (int i=0; i<size; i++)
    {
        m_shape[i]->Translate(x,y);
    }
}
```





### ► 例子：甜甜圈小屋代码设计

```
// 例 08-abstract-dtor: ex04-abstract-dtor.cpp
// 实现类 CShapeArr 的析构函数
```

```
CShapeArr::~CShapeArr()
{
    //dtor
    if(size != 0)
    {
        for(int i=0; i< size; i++)
        {
            if(m_shape[i]!=NULL)
                delete m_shape[i];
        }
    }
}
```

基类类型指针指向派生类对象，调用派生类中的函数





## ► 例子：甜甜圈小屋-测试数据

```
/*
Rectangle 800 400 400 400 sky 0xC0E8F2 0x000000
Rectangle 800 200 400 100 earth 0xB8BD95 0x000000
Donut 0.5 80 60 474 490 rooftop 0x814000 0x000000
Rectangle 30 100 407 354 chimney 0x8C1926 0x000000
Ellipse 15 15 403 425 smoke1 0xFFEEEE 0x000000
Ellipse 20 20 360 480 smoke2 0xFFEEEE 0x000000
Ellipse 30 30 380 550 smoke3 0xFFEEEE 0x000000
Rectangle 236 200 474 181 wall 0xDECA8F 0x000000
Triangle 340 180 474 340 roof 0x9E6120 0x000000
Rectangle 30 30 412 227 window11 0xB5C9EE 0x000000
Rectangle 30 30 550 227 windowr1 0xB5C9EE 0x000000
Rectangle 30 30 412 130 window12 0xB5C9EE 0x000000
Rectangle 30 30 550 130 windowr2 0xB5C9EE 0x000000
Rectangle 80 20 474 167 eave 0xF9B86B 0x000000
Rectangle 25 55 462 128 door1 0x8C1926 0x000000
Rectangle 25 55 486 128 door2 0x8C1926 0x000000
Rectangle 80 12 474 94 step1 0x9F9E99 0x000000
Rectangle 100 12 474 82 step2 0x9F9E99 0x000000
Ellipse 50 15 444 45 stone1 0x9F9E99 0x000000
Ellipse 55 18 322 31 stone2 0x9F9E99 0x000000
Ellipse 65 15 181 22 stone3 0x9F9E99 0x000000
Rectangle 10 15 154 78 trunk11 0x814000 0x000000
Triangle 95 50 154 183 crown11 0x002800 0x000000
Triangle 85 45 154 131 crown12 0x002800 0x000000
Triangle 70 40 154 156 crown13 0x002800 0x000000
Triangle 50 50 154 184 crown14 0x002800 0x000000
Exit
*/
```



### ▶ 练习：形状类的设计

#### ▶ 设计要求

- ▶ 设计一个 CShape 抽象类，类中包含纯虚函数
- ▶ 从 CShape 类派生出三角形类 CTriangle、矩形类 CRectangle 和椭圆类 CEllipse
- ▶ 使用一个公共接口计算三角形对象、矩形对象和椭圆对象面积之和
- ▶ 重载运算符 > 用于判断两个形状面积的大小，返回 true 或 false





- ▶ 运行时刻类型识别 (RunTime Type Identification): 允许使用基类指针或引用操纵对象的程序获得这些指针或引用实际所指对象的类型。
  
- ▶ **dynamic\_cast**运算符
  
- ▶ **typeid**运算符





► **static\_cast**

► **dynamic\_cast**

► 向下类型转换

► **reinterpret\_cast**

► **const\_cast**





## ► 向下类型转换

```
// 例 09-rtti-class: ex09-rtti-class.cpp

class CBase
{
    // Since RTTI is included in the virtual method table
    // there should be at least one virtual function.
public:
    virtual void foo() {}
    void methodBase()
    {
        cout << "method Base" << endl;
    }
};

class CDerived : public CBase
{
public:
    void methodDerived()
    {
        cout << "method Derived" << endl;
    }
};
```

```
// 例 09-rtti-fun: ex09-rtti-fun.cpp
// 动态类型转换的演示

void my_function(CBase* my_a)
{
    CDerived *my_b = dynamic_cast<CDerived*>(my_a);
    if(my_b != NULL)
        my_b->methodDerived();
    else
        my_a->methodBase();
}
```





## ▶ 向下类型转换

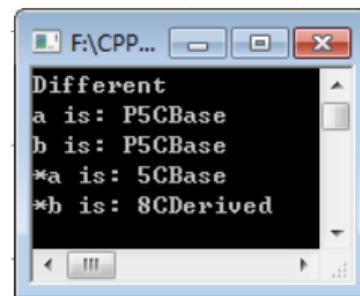
```
// 例 09-rtti-main: ex09-rtti-main.cpp
// 实例化 CBase 和 CDerived 的对象，并测试他们的类型

#include <iostream>
#include <typeinfo>

using namespace std;

int main( void )
{
    CBase* a = new CBase;
    CBase* b = new CDerived;
    if(typeid(* a) == typeid(* b))
        cout << "Identical" << endl;
    else
        cout << "Different" << endl;
    cout << "a is: " << typeid( a ).name() << endl;
    cout << "b is: " << typeid( b ).name() << endl;
    cout << "*a is: " << typeid( *a ).name() << endl;
    cout << "*b is: " << typeid( *b ).name() << endl;
    return 0;
}
```

编译器不同，输出可能不同！





### ► 静态联编 (static binding or early binding)

► 在编译时确定同名函数的具体操作对象

- 重载函数
- 函数模板
- 运算符重载

► 特点：执行效率高，但灵活性差

### ► 动态联编 (dynamic binding or late binding)

► 在程序运行时 (run time) 确定对象所调用的函数

- 虚函数
- 纯虚函数

► 特点：灵活性强，但效率可能会降低





- ▶ **多态性**是面向对象程序设计的重要特性之一，多态是指同样的消息被不同类型的对象接收时导致完全不同的行为。
- ▶ **虚函数**是在基类中定义的以 `virtual` 关键字作为开头的成员函数，需要在派生类中重新定义。
- ▶ **纯虚函数**是一个在基类中声明的没有定义具体实现的虚函数，带有纯虚函数的类称为抽象类。
- ▶ **抽象类**为一个类族提供统一的操作界面。抽象类处于类层次的上层，自身无法实例化，只能通过继承机制，生成抽象类的具体派生类，然后再实例化。通过抽象类的指针和引用，可以指向并访问各派生类成员，实现多态性。
- ▶ 联编可以在编译和连接时进行，称为**静态联编**；联编也可以在运行时进行，称为**动态联编**。



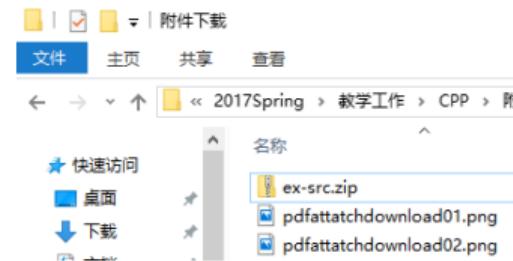
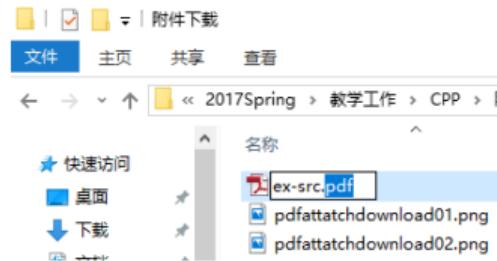
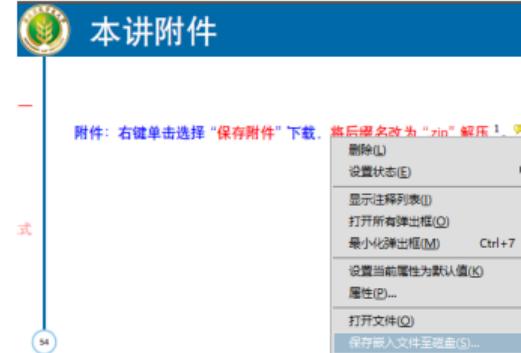
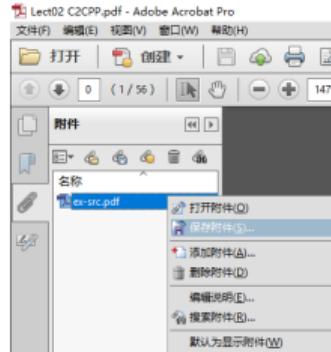


# 本讲附件

| 附件

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP  
Nine, G.

附件：右键单击该链接，选择“保存附件”下载，将后缀名改为“.zip”解压<sup>16 17</sup>。



<sup>16</sup>请退出全屏模式后点击该链接。

<sup>17</sup>以 Adobe Acrobat Reader 为例。



### ► 函数模板 (function templates or generic functions)

```
template <class 类型名1, class 类型名2, ...>
返回类型 函数名 (形参表)
{
    ...
}
```





### ▶ 利用函数模板进行求和运算

```
//例 07-01; ex07-01.cpp
//求不同类型的两个数之和，演示 c++ 的简单函数模版的定义与使用
#include <iostream>

using namespace std;

template <class T1, class T2>
T1 add(T1 x, T2 y)
{
    return x + y;
}

int main()
{
    cout << add(9, 'A') << endl;           //隐式实例化
    cout << add<int, char>(9, 'A') << endl; //显式实例化
}
```





### ► 类模板 (class templates or generic classes) 定义

或**typename** ...， 或**typename** ...

```
template <class 参数化类型1, class 参数化类型2, ...,
          普通类型1, 普通类型2, ...>
```

```
class 类名
```

```
{
```

```
    成员名;
```

```
}
```

### ► 关于**class**和**typename**，请参阅：

<http://dev.yesky.com/13/2221013.shtml>

<http://blogs.msdn.com/b/slippman/archive/2004/08/11/212768.aspx>





### ► 安全数组类模板的定义

```
//例 07-02; ex07-02.cpp
//模板类的写法, 演示 c++ 的简单模板类的写法
template <class T, int size>
class CSafeArray
{
    T a[size];
public:
    CSafeArray()
    {
        for(int i = 0; i < size; i++) a[i] = i;
    }
    T &operator[](int i)
    {
        if(i < 0 || i > size - 1)
        {
            cout << "Index value of " << i << " is out-of-bounds.\n";
            exit(1);
        }
        return a[i];
    }
};
```

参数化类型





### ► 安全数组类模板的定义

```
//例 07-02; ex07-02.cpp
//模板类的写法, 演示 c++ 的简单模板类的写法
template <class T, int size>
class CSafeArray
{
    T a[size];
public:
    CSafeArray()
    {
        for(int i = 0; i < size; i++) a[i] = i;
    }
    T &operator[](int i)
    {
        if(i < 0 || i > size - 1)
        {
            cout << "Index value of " << i << " is out-of-bounds.\n";
            exit(1);
        }
        return a[i];
    }
};
```

普通类型





### ▶ 类模板中的成员函数 → 函数模板

```
//例 07-02; ex07-02.cpp
//模板类的写法, 演示 c++ 的简单模板类的写法
template <class T, int size>
class CSafeArray
{
    T a[size];
public:
    CSafeArray()
    {
        for(int i = 0; i < size; i++) a[i] = i;
    }
    T& operator[](int i)
    {
        if(i < 0 || i > size - 1)
        {
            cout << "Index value of " << i << " is out-of-bounds.\n";
            exit(1);
        }
        return a[i];
    }
};
```

函数模板





### ▶ 类外定义类模板成员函数

```
//例 07-03; ex07-03.cpp
//在模板类的基础上写符号重载函数，演示 c++ 的符号重载函数的书写方法
template <class T, int size>
class CSafeArray
{
    T a[size];
public:
    CSafeArray()
    {
        for(int i = 0; i < size; i++)
            a[i] = i;
    }
    T &operator[](int i);
};
```

模板参数表

```
//例 07-04; ex07-04.cpp
//符号重载，演示 c++ 的符号简单重载方法
template <class T, int size>
T &CSafeArray<T, size>::operator[](int i)
{
    if(i < 0 || i > size - 1)
    {
        cout << "Index value of ";
        cout << i << " is out-of-bounds.\n";
        exit(1);
    }
    return a[i];
}
```



- ▶ 类模板 → 具体类 → 对象
- ▶ 语法

---

类模板名<基本数据类型名或构造数据类型名或常数表达式> 对象1, 对象2, … , 对象 n;

---

- ▶ 示例:

---

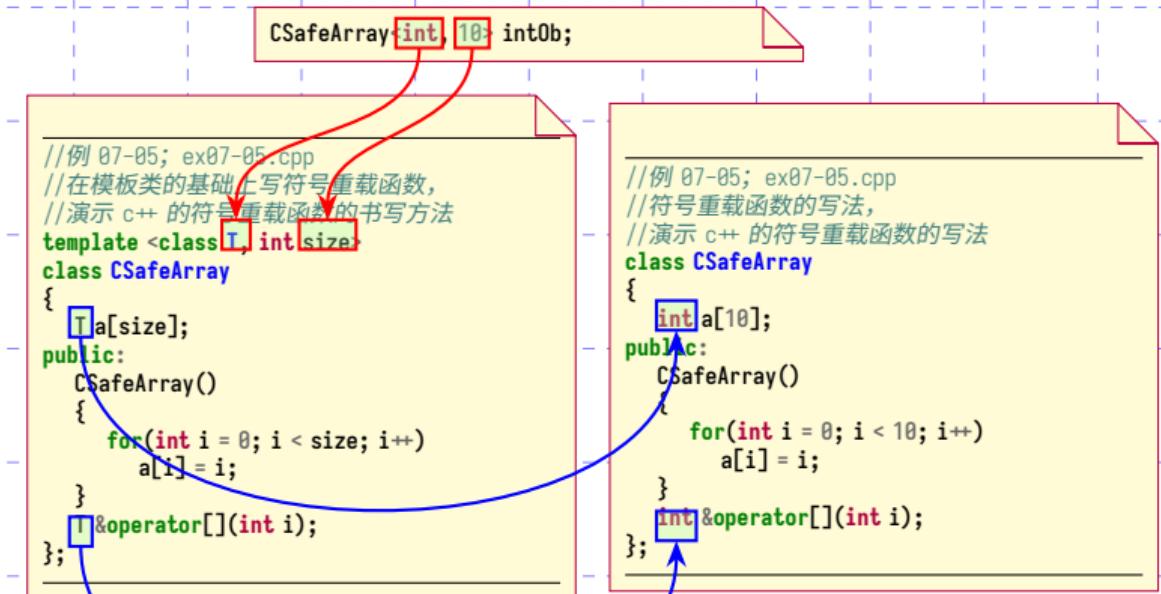
```
CSafeArray <int, 10> int0b;  
CSafeArray <double, 10> double0b;
```

---





### ▶ 类模板的实例化和函数的实参与形参的结合类似





### ▶ 类模板的实例化和函数的实参与形参的结合类似

```
//例 07-06; ex07-06.cpp
//模版类的使用，演示 c++ 的简单模板类的使用
int main()
{
    const int SIZE = 10;
    CSafeArray<int, SIZE> intOb;
    CSafeArray<double, SIZE> doubleOb;

    cout << "Integer array: ";
    for(int i = 0; i < SIZE; i++) intOb[i] = i;
    for(int i = 0; i < SIZE; i++)
        cout << intOb[i] << " "; 整型安全数组
    cout << '\n';

    cout << "Double array: ";
    for(int i = 0; i < SIZE; i++) doubleOb[i] = (double) i / 3;
    for(int i = 0; i < SIZE; i++)
        cout << doubleOb[i] << " ";
    cout << '\n'; 浮点型安全数组

    intOb[SIZE + 1] = 100;

    return 0;
}
```



## ► 对模板参数表中的类型赋默认的数据类型或数值（默认类属形参值）

```
//例 07-07; ex07-07.cpp
//在模板类的基础上写符号重载函数，演示 c++ 的符号重载函数的书写方法
template <class T = int, int size = 10>
class CSafeArray
{
    T a[size];
public:
    CSafeArray()
    {
        int i;
        for(i = 0; i < size; i++)
            a[i] = i;
    }
    T &operator[](int i);
};
```

模板参数默认值

```
//例 07-07; ex07-07.cpp
//定义模板类的对象，演示 c++ 定义模板类对象的方法
int main()
{
    CSafeArray <>int0b;
    CSafeArray <double> double0b1;
    CSafeArray<double, 100> double0b2;
}
```





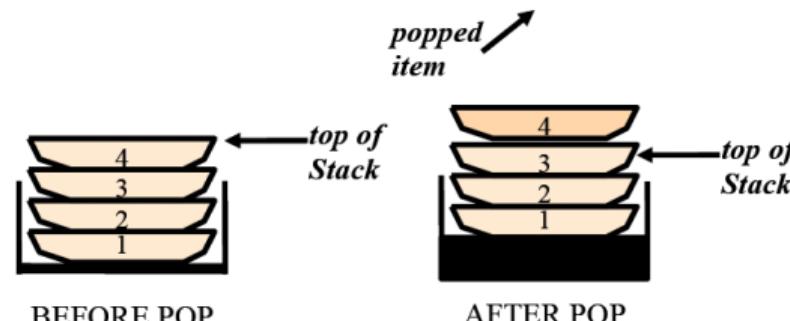
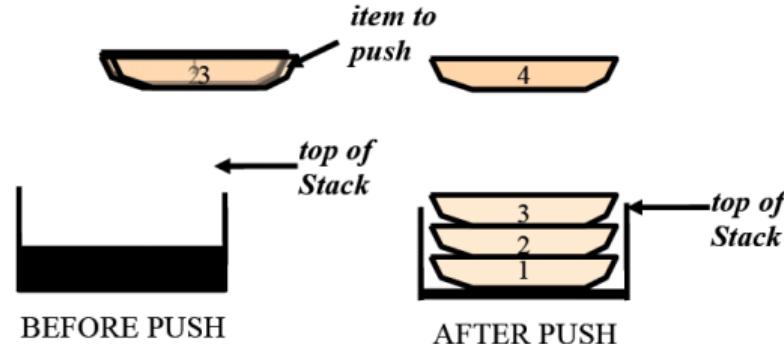
### ► 顺序堆栈类 (FILO)

- ▶ 数据成员：栈顶索引 (top)，数据存储空间 (s)
- ▶ 成员函数：(Push) 和 (Pop)





## ► 顺序堆栈类的基本操作 (FILO)





## ▶ 类模板的定义

```
//例 07-08; ex07-08.cpp
//定义一个栈类
const int SIZE = 10;

template <class ST>
class stack
{
    ST s[SIZE]; // holds the stack
    int top; // index of top-of-stack
public:
    stack()
    {
        top = 0; // initialize stack
    }

    void Push(ST ob); // push object on stack
    ST Pop(); // pop object from stack
};
```





## ▶ 类模板的函数模板

```
//例 07-08; ex07-08.cpp
//定义一个入栈函数
// Push an object.
template <class ST>
void stack<ST>::Push(ST ob)
{
    if(top == SIZE)
    {
        cout << "Stack is full.\n";
        return;
    }
    s[top] = ob;
    top++;
}
```

```
//例 07-08; ex07-08.cpp
//定义一个出栈函数
// Pop an object.
template <class ST>
ST stack<ST>::Pop()
{
    if(top == 0)
    {
        cout << "Stack is empty.\n";
        return 0;
    }
    top--;
    return s[top];
}
```





## ▶ 类模板的使用

```
//例 07-08; ex07-08.cpp
//使用定义的栈类模板来完成不同类型栈的入栈出栈操作
int main()
{
    int i;

    stack<char> cs;
    cs.Push('a');
    cs.Push('b');
    cs.Push('c');
    for(i = 0; i < 3; i++)
        cout << "Pop cs: " << cs.Pop() << "\n";

    stack<double> ds;
    ds.Push(1.1);
    ds.Push(2.2);
    ds.Push(3.3);
    for(i = 0; i < 3; i++)
        cout << "Pop ds: " << ds.Pop() << "\n";

    return 0;
}
```

字符栈

浮点数栈



- ▶ 类模板成员函数声明与定义(实现)需处于同一文件。
- ▶ 分离编译模式: **export**关键字
  - ▶ VC 与 GCC 编译器不支持
- ▶ 其它解决方案

[http://www.codeproject.com/KB/cpp/Template\\_implementaion.aspx](http://www.codeproject.com/KB/cpp/Template_implementaion.aspx)





### ► 优势

- ▶ 通过数据类型参数化实现代码重用（设计一个类模板，可用于多种数据类型场合）
- ▶ STL 的基础

### ► 缺点

- ▶ 类层次上再次抽象，对初学者来说可读性较差，不易使用





- ▶ 使用函数模板和类模板实现（类型参数化）
- ▶ 提供标准的数据结构和算法
- ▶ 可提高程序开发效率

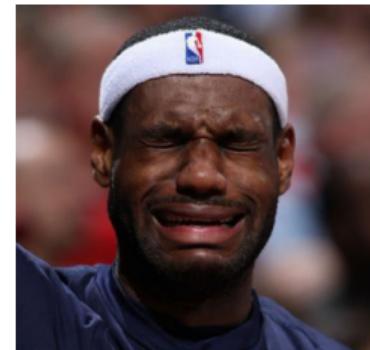




- ▶ 对初学者来说，语法晦涩难懂
- ▶ STL 函数较多，不易记忆
- ▶ 建议：耐心学习参考例程
- ▶ 多看多练：

<http://msdn.microsoft.com/en-us/library/c191tb28.aspx>

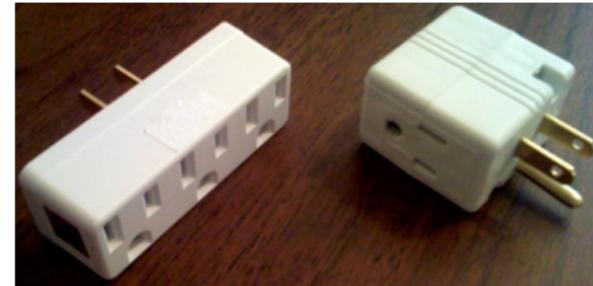
<http://www.cplusplus.com/reference/stl/>





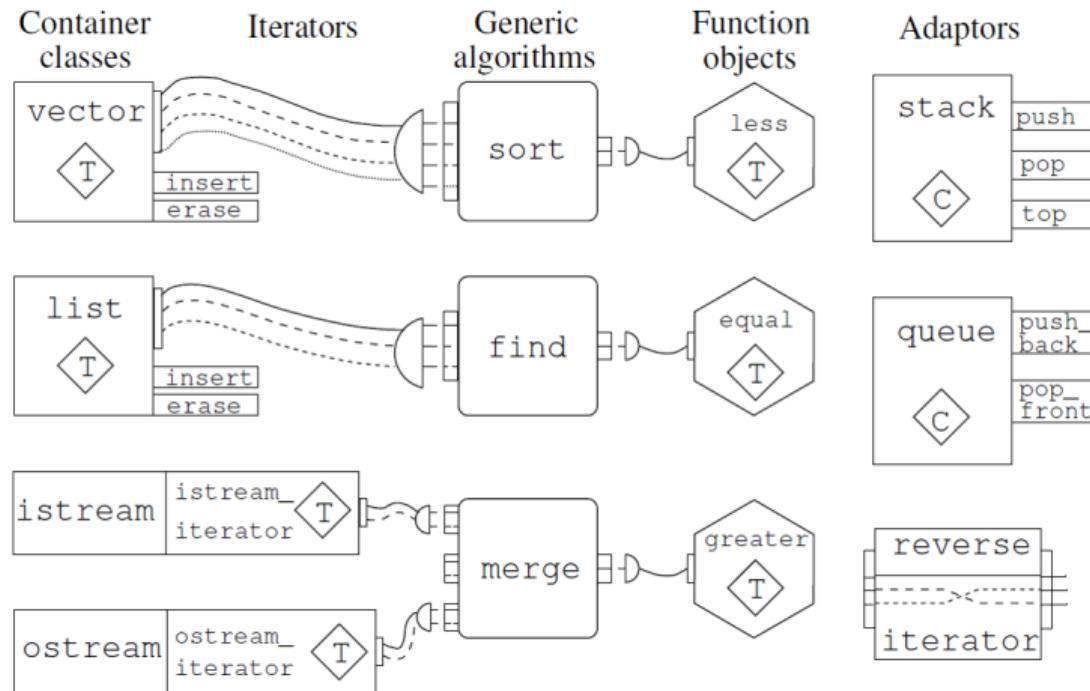
## ► 组成 (六部分)

- ▶ 容器 (Containers)
- ▶ 迭代器 (Iterators)
- ▶ 算法 (Algorithms)
- ▶ 适配器 (Adapters)
- ▶ 函数对象 (Function Objects)
- ▶ 分配器 (Allocators)





## ► 组成 (六部分)





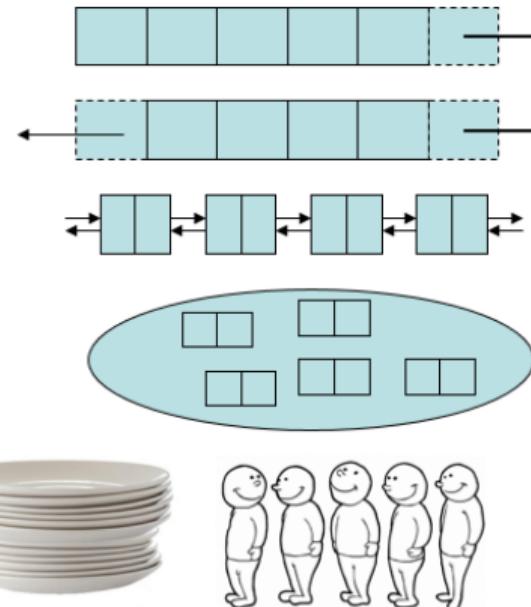
- ▶ 一个通用的数据结构
  - ▶ 可以处理不同数据类型
  - ▶ 包含基本的数据结构，如链表、堆栈、队列等





## ▶ 分类

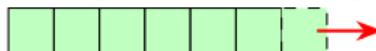
- ▶ 顺序容器 (sequence)
- ▶ 关联容器 (sorted associative)
- ▶ 容器适配器 (adaptors)
- ▶ 特殊容器 (special)





► 特点：添加或插入位置与元素值无关（无自动排序）

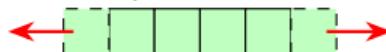
► 向量 (动态数组 vector)



► 列表 (双向链表 list)



► 双端队列 (deque: double-ended queue)





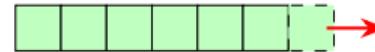
- ▶ 注意：不同容器的插入、删除和存取特性不同，需根据任务选择合适容器
- ▶ 评价指标之一：时间复杂度

Typical Values of Complexity		
Type	Notation	Meaning
Constant	$O(1)$	The runtime is independent of the number of elements.
Logarithmic	$O(\log(n))$	The runtime grows logarithmically with respect to the number of elements.
Linear	$O(n)$	The runtime grows linearly (with the same factor) as the number of elements grows.
n-log-n	$O(n \log(n))$	The runtime grows as a product of linear and logarithmic complexity.
Quadratic	$O(n^2)$	The runtime grows quadratically with respect to the number of elements.





## ► 特点



- ▶ 在内存中占有一块连续的空间 (动态数组)
- ▶ 可自动扩充，且提供越界检查
- ▶ 适合在向量末尾插入或删除数据
- ▶ 可用 [] 运算符直接存取数据 (随机访问 random access)





## ► 例 1

```
//例 07-09; ex07-09.cpp
//容器 vector 的使用，演示 c++ 中容器 vector 的简单使用
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> v;

    for (int i = 0; i < 6; ++i)
        v.push_back(i);

    for (int i = 0; i < 3; ++i)
        v.pop_back();

    for (int i = 0; i < (int)v.size(); ++i)
        cout << v[i] << ' ';
    cout << endl;

    return 0;
}
```





## ► 例 2

```
//例 07-09; ex07-09.cpp
//容器 vector 的使用, 演示 c++ 中容器 vector 的简单使用
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> v(6,1);           箭头指向此行          容量为 6, 用 1 初始化

    for (int i=0; i<v.size( ); ++i)
        cout << v[i] << ' ';
    cout << endl;

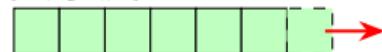
    for (int i=0; i<v.size( ); ++i)
        v[i]=i;

    for (int i=0; i<v.size( ); ++i)
        cout << v[i] << ' ';
    cout << endl;
}
```





## ▶ 复杂度



- ▶ 随机存取:  $O(1)$
- ▶ 向量末尾插入或删除:  $O(1)$

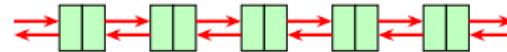
## ▶ 使用

- ▶ `#include <vector>`
- ▶ 适于快速存取数据但非尾部频繁插入删除的场合





► 特点



- 双向链表（前驱和后继）
- 可在任意位置插入或删除数据
- 不能用 [] 运算符直接存取数据 (不支持随机访问 random access)





## ► 例 3

```
//例 07-10; ex07-10.cpp
//容器 list 的插入函数、迭代器的使用，演示 c++ 中容器 list 的简单使用
#include <iostream>
#include <list>
#include <vector>
using namespace std;
int main(){
    list<int> mylist;

    list<int>::iterator it;
    for (int i = 1; i <= 5; i++) mylist.push_back(i);

    it = mylist.begin();
    ++it;

    mylist.insert (it, 10);
    mylist.insert (it, 2, 20);

    --it;

    vector<int> myvector (2, 30);
    mylist.insert (it, myvector.begin(), myvector.end());

    for (it = mylist.begin(); it != mylist.end(); it++)
        cout << *it << " ";
    cout << endl;

    return 0;
}
```

迭代器：类似于指针





## ▶ 例 4

```
//例 07-11; ex07-11.cpp
//容器 list 中的排序、合并等函数的使用，演示 c++ 中容器 list 的高级函数使用
#include <iostream>
#include <list>
using namespace std;
bool mycomparison (double first, double second){
    return ( int(first) < int(second) );
}
int main (){
    list<double> first, second;
    first.push_back (3.1);
    first.push_back (2.2);
    first.push_back (2.9);

    second.push_back (3.7);
    second.push_back (7.1);
    second.push_back (1.4);

    first.sort();
    second.sort();

    first.merge(second);
    second.push_back (2.1);

    first.merge(second, mycomparison);

    for (list<double>::iterator it = first.begin(); it != first.end(); ++it)
        cout << *it << " ";
    cout << endl;

    return 0;
}
```

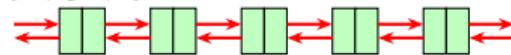
声明对象

排序

合并



## ▶ 复杂度



- ▶ 访问前驱或后继:  $O(1)$
- ▶ 根据索引值访问节点数据:  $O(n)$
- ▶ 任意位置插入或删除:  $O(1)$

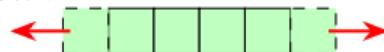
## ▶ 使用

- ▶ `#include <list>`
- ▶ 适于数据频繁插入删除而不在意查找速度的场合
- ▶ 排序、合并操作效率高





## ► 特点



- 以多内存块 (chunks of storage) 形式存储数据
- 可自动扩充，且提供越界检查
- 适合在向量头尾插入或删除数据
- 可用 [] 运算符直接存取数据 (随机访问 random access)





## ► 例 5

```
//例 07-12; ex07-12.cpp
//容器 deque 的使用，演示 c++ 中容器 deque 的简单函数使用
#include <iostream>
#include <deque>
using namespace std;

int main()
{
    deque<float> dv;
    for (int i = 0; i < 6; ++i)
    {
        dv.push_front(i * 1.1);
    }

    for (int i = 0; i < dv.size(); ++i)
        cout << dv[i] << " ";
    cout << endl;
}

return 0;
}
```





# 双端队列容器 (deque)

| 容器

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP  
Nine, G.

## ▶ 例 6

```
//例 07-13; ex07-13.cpp
//容器 deque 的 assign 函数的使用, 演示 c++ 中容器 deque 的高级函数使用
#include <iostream>
#include <deque>
#include <string>

using namespace std;

int main()
{
    deque<string> ds;
    ds.assign(3, string("Hello")); → 声明对象
    ds.push_back("] last");
    ds.push_front("[ first");

    for (int i = 0; i < ds.size(); ++i)
        cout << ds[i] << " ";
    cout << endl;

    ds.pop_front();
    ds.pop_back();

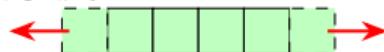
    for (int i = 1; i < ds.size(); ++i)
        ds[i] = "another " + ds[i]; → 下标访问
    ds.resize(4, "Hello C++"); → 扩展为 4 个元素, 扩展后的元素赋值为 Hello C++

    for (int i = 0; i < ds.size(); ++i)
        cout << ds[i] << " ";
    cout << endl;

    return 0;
}
```



## ▶ 复杂度



- ▶ 随机存取:  $O(1)$
- ▶ 队列头尾插入或删除:  $O(1)$

## ▶ 使用

- ▶ `#include <deque>`
- ▶ 适于快速在队列头尾存取数据但非频繁插入删除的场合
- ▶ 若非头尾插入删除数据，效率比 list 低





► 特点：元素的添加或插入位置与元素的值相关

► 集合 (set) 和多集 (multiset, 允许重复元素)

► 映射 (map) 和多映射 (multimap, 允许重复元素键值)

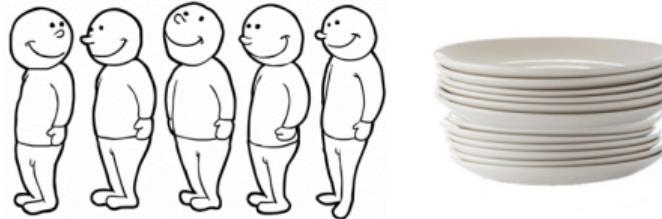
► 复杂度：插入、删除和查找  $O(\log(n))$





▶ 由顺序容器转换出的新容器 ( $\text{list} \rightarrow \text{queue}$ ,  $\text{vector} \rightarrow \text{stack}$ ), 不支持迭代器

- ▶ 队列 (queue)
- ▶ 优先队列 (priority queue)
- ▶ 堆栈 (stack)





- ▶ 位集 `bitset`: 灵活对二进制位进行操作

- ▶ 整型到二进制的转换

- ▶ 位的直接访问和设置等



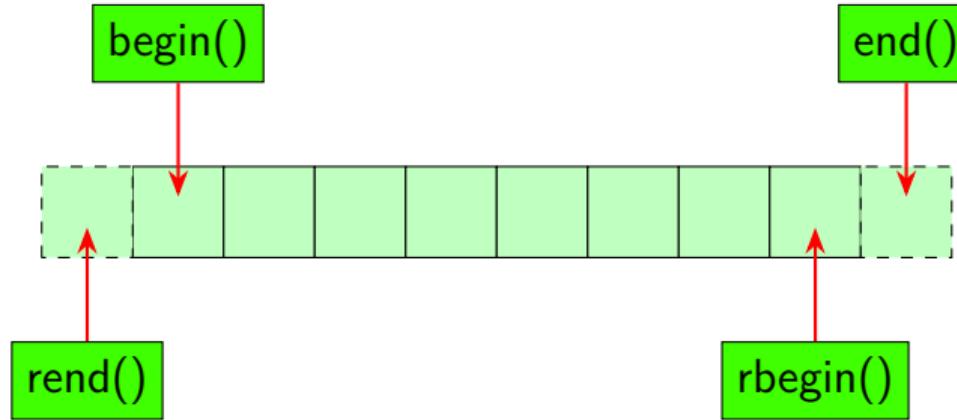


- ▶ 能对顺序容器或关联容器中的每个元素进行连续存取的对象 (一个特殊的指针)
- ▶ 容器名<数据类型>::iterator 迭代器名
- ▶ 非标准迭代器
  - ▶ const\_iterator
  - ▶ reverse\_iterator
  - ▶ const\_reverse\_iterator





- ▶ 容器名.begin(), 容器名.end()
- ▶ 容器名.size()





### ▶ 例 7

```
//例 07-14; ex07-14.cpp
//容器 vector 的反向迭代器的使用，演示 c++ 中容器 vector 中反向迭代器的使用
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector <int>v;
    vector <int>::reverse_iterator p;

    for(int i = 0; i < 10; i++)
        v.push_back(i);

    for(p = v.rbegin(); p != v.rend(); p++)
        cout << *p << " ";
    cout << endl;
}
```

非 p-



## ▶ 分类

category		characteristic		valid expressions
all categories		Can be copied and copy-constructed		X b(a); b = a;
		Can be incremented		++a; a++; *a++
Bidirectional	Input	Accepts equality/inequality comparisons		a == b; a != b
		Can be dereferenced as an <i>rvalue</i>		*a; a->m
	Forward	Can be dereferenced to be the left side of an assignment operation		*a = t; *a++ = t
		Can be default-constructed		X a; X()
	Random Access	Can be decremented		-a; a--; *a--
		Supports arithmetic operators + and -		a + n; n + a; n - a; a - b
		Supports inequality comparisons (< and >) between iterators		a < b; a > b
		Supports compound assignment operations +=, -=, <= and >=		a += n; a -= n a <= b; a >= b
		Supports offset dereference operator ([])		a[n]





► 在 vector 和 deque 中实现跳跃式访问

```
vector <int>::iterator p;  
for(p=v.begin(); p!=v.end(); p+=2)
```

► list 中使用 “+” 无法实现

► 使用函数 advance

```
list <int>::iterator p;  
for(p=v.begin(); p!=v.end(); advance(p,2))
```

► 提供一种一般化方法 (generic method) 对不同类型容器中的元素进行访问





### ▶ 迭代器示例

```
//例 07-15; ex07-15.cpp
//在容器 vector 上定义寻找函数 FIND 来
//获得数 value 在容器 vector 中的位置
template <class ITER, class T>
ITER Find(ITER first, ITER last, T value)
{
    while(first != last && *first != value)
        ++first;
    return first;
}
```

```
//例 07-15; ex07-15.cpp
//容器 vector 上定义输出函数
template <class ITER>
void Print(ITER first, ITER last)
{
    while(first != last)
    {
        cout << *first << " ";
        ++first;
    }
    cout << endl;
}
```





## ▶ 迭代器示例

```
//例 07-15; ex07-15.cpp
//容器 vector 的使用实例，演示 c++ 中容器 vector 的简单使用
typedef vector <int> container;
typedef vector <int>::iterator iterCon;

int main()
{
    container v;
    iterCon where;
    int key = 10;

    for(int i = 0; i < 10; i++)
        v.push_back(i);

    where = Find(v.begin(), v.end(), key);

    if(where != v.end())
        cout << *where << endl;
    else
        cout << "Fail to find the value!" << endl;

    Print(v.begin(), v.end());

    return 0;
}
```





- ▶ 算法实现机理
- ▶ 非修改操作 Non-modifying sequence operations
- ▶ 修改操作 Modifying sequence operations
- ▶ 排序 Sorting
- ▶ 堆 Heap
- ▶ 二分查找 Binary search
- ▶ 合并 Merge
- ▶ 最小最大值 Min/max





- ▶ 基于迭代器和函数模板
- ▶ 算法 <algorithm> 是用来处理一个数据序列区间 (range) 的函数集合
- ▶ 区间中的元素通过迭代器进行访问
- ▶ 独立于所操作的容器





<b>for_each</b>	Apply function to range
<b>find</b>	Find value in range
<b>find_if</b>	Find element in range
<b>find_end</b>	Find last subsequence in range
<b>find_first_of</b>	Find element from set in range
<b>adjacent_find</b>	Find equal adjacent elements in range
<b>count</b>	Count appearances of value in range
<b>count_if</b>	Return number of elements in range satisfying condition
<b>mismatch</b>	Return first position where two ranges differ
<b>equal</b>	Test whether the elements in two ranges are equal
<b>search</b>	Find subsequence in range
<b>search_n</b>	Find succession of equal values in range





## ► find 函数模板

```
//例 07-16; ex07-16.cpp
//定义寻找函数 find 来获得数 value 所在的位置
template<class InputIterator, class T>
InputIterator find( InputIterator first, InputIterator last, const T& value )
{
    for ( ; first != last; first++ ) if ( *first == value ) break;
    return first;
}
```





## ▶ find 函数模板

```
//例 07-15; ex07-15.cpp
//容器 vector 的综合使用实例，演示 c++ 中容器 vector 的使用
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main ()
{
    int myints[] = { 10, 20, 30 , 40 };
    int * p;

    // pointer to array element:
    p = find(myints, myints + 4, 30);
    ++p;
    cout << "The element following 30 is " << *p << endl;

    vector<int> v (myints, myints + 4);
    vector<int>::iterator it;

    // iterator to vector element:
    it = find (v.begin(), v.end(), 30);
    ++it;
    cout << "The element following 30 is " << *it << endl;

    return 0;
}
```



## ► for\_each 函数模板

```
//例 07-15; ex07-15.cpp
//定义 for_each 函数, 使得在 first 与 last 之间的元素都做 f 处理
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f)
{
    for ( ; first != last; ++first) f(*first);
    return f;
}
```

函数指针

函数对象





## ► for\_each 函数模板

```
//例 07-17; ex07-17.cpp
//容器 vector 的综合使用，演示 c++ 中容器 vector 的综合使用实例
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

void myfunction (int i)
{
    cout << " " << i;
}

struct myclass
{
    void operator() (int i)
    {
        cout << " " << i;
    }
} myobject;

int main ()
{
    vector<int> v;
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);

    cout << "v contains:";
    for_each (v.begin(), v.end(), myfunction);

    // or:
    cout << "\nv contains:";
    for_each (v.begin(), v.end(), myobject);

    cout << endl;
    return 0;
}
```





## ▶ count\_if 函数模板

```
//例 07-18; ex07-18.cpp
//返回指定区间内的奇数个数，演示 c++ 中 count_if 的使用
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool IsOdd (int i)
{
    return ((i % 2) = 1);

}

int main ()
{
    int mycount;

    vector<int> v;
    for (int i = 1; i < 10; i++) v.push_back(i); // v: 1 2 3 4 5 6 7 8 9

    mycount = (int)count_if (v.begin(), v.end(), IsOdd);
    cout << "v contains " << mycount << " odd values.\n";

    return 0;
}
```





<b>copy</b>	Copy range of elements
<b>copy_backward</b>	Copy range of elements backwards
<b>swap</b>	Exchange values of two objects
<b>swap_ranges</b>	Exchange values of two ranges
<b>iter_swap</b>	Exchange values of objects pointed by two iterators
<b>transform</b>	Apply function to range
<b>replace</b>	Replace value in range
<b>replace_if</b>	Replace values in range
<b>replace_copy</b>	Copy range replacing value
<b>replace_copy_if</b>	Copy range replacing value
<b>fill</b>	Fill range with value
<b>fill_n</b>	Fill sequence with value
<b>generate</b>	Generate values for range with function





<code>generate_n</code>	Generate values for sequence with function
<code>remove</code>	Remove value from range
<code>remove_if</code>	Remove elements from range
<code>remove_copy</code>	Copy range removing value
<code>remove_copy_if</code>	Copy range removing values
<code>unique</code>	Remove consecutive duplicates in range
<code>unique_copy</code>	Copy range removing duplicates
<code>reverse</code>	Reverse range
<code>reverse_copy</code>	Copy range reversed
<code>rotate</code>	Rotate elements in range
<code>rotate_copy</code>	Copy rotated range
<code>random_shuffle</code>	Rearrange elements in range randomly
<code>partition</code>	Partition range in two
<code>stable_partition</code>	Partition range in two - stable ordering





## ► 替换函数 replace

```
//例 07-19; ex07-19.cpp
//容器 vector 中 replace 函数的使用，演示 c++ 中容器 vector 的高级函数使用
// replace algorithm example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main ()
{
    int myints[] = { 10, 20, 30, 30, 20, 10, 10, 20 };
    // 10 20 30 30 20 10 10 20
    vector<int> v (myints, myints + 8);

    // 10 99 30 30 99 10 10 99
    replace (v.begin(), v.end(), 20, 99);

    cout << "v contains:";
    for (vector<int>::iterator it = v.begin(); it != v.end(); ++it)
        cout << " " << *it;

    cout << endl;

    return 0;
}
```



▶ 删除函数 `remove`

```
//例 07-20; ex07-20.cpp
//容器 vector 中 remove 函数的使用，演示 c++ 中容器 vector 的高级函数使用
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main ()
{
    int myints[] = {10, 20, 30, 30, 20, 10, 10, 20};
    // 10 20 30 30 20 10 10 20
    vector <int> v(myints, myints + 8);
    vector <int>::iterator p, pEnd;

    pEnd = remove (v.begin(), v.end(), 20);

    for (p = v.begin(); p != pEnd; ++p)
        cout << " " << *p;
    cout << endl;

    return 0;
}
```





### ► 排序算法

**sort**

Sort elements in range

**stable\_sort**

Sort elements preserving order  
of equivalents

**partial\_sort**

Partially Sort elements in range

**partial\_sort\_copy**

Copy and partially sort range

**nth\_element**

Sort element in range





# 修改操作

算法 II

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP

Nine, G.

```
//例 07-21.cpp
//对各类容器的排序进行比较，演示 c++ 中各类容器的使用
#include <iostream>
#include <list>
#include <vector>
#include <set>
#include <map>
#include <algorithm>
#include <ctime>
#include <iomanip>

using namespace std;

bool compare(int i, int j)
{
    return (i > j);
}

int main()
{
    const int sz = 10000000;
    const int top100 = 100;

    vector<int> origin;
    for(int i = 0; i < sz; i++)
        origin.push_back(i);

    random_shuffle(origin.begin(), origin.end());
    //copy(origin.begin(), origin.end(), ostream_iterator<int>(cout, "\t"));

    list<int> ln;
    for(int i = 0; i < sz; i++)
        ln.insert(ln.begin(), origin[i]);
    clock_t ticks = clock();
    ln.sort();
    cout << "list sort:" << clock() - ticks << "ms" << endl;

    multiset<int> ss;
    ticks = clock();
    for(int i = 0; i < sz; i++)
        ss.insert(origin[i]);
    //copy(ss.begin(), ss.end(), ostream_iterator<int>(cout, "\t"));
    cout << "set sort:" << clock() - ticks << "ms" << endl;

    vector<int> vo;
    for(int i = 0; i < sz; i++)
        vo.push_back(origin[i]);
    ticks = clock();
    sort(vo.begin(), vo.end());
    //copy(vo.begin(), vo.end(), ostream_iterator<int>(cout, "\t"));
    cout << "vector sort:" << clock() - ticks << "ms" << endl;

    for(int i = 0; i < sz; i++) {
        vo[i] = origin[i];
    }

    for(int i = 0; i < sz; i++) {
        vo[i] = origin[i];
    }
    ticks = clock();
    partial_sort(vo.begin(), vo.begin() + top100, vo.end());
    copy(vo.begin(), vo.begin() + top100, ostream_iterator<int>(cout, "\t"));
    cout << "partial sort top100:" << clock() - ticks << "ms" << endl;
}
```





## ► 排序算法

Name	Best	Average	Worst	Memory	Stable	Method
<u>Quicksort</u>	$n \log n$	$n \log n$	$n^2$	$\log n$	Depends	Partitioning
<u>Merge sort</u>	$n \log n$	$n \log n$	$n \log n$	Depends	Yes	Merging
<u>Heapsort</u>	$n \log n$	$n \log n$	$n \log n$	1	No	Selection
<u>Insertion sort</u>	$n$	$n^2$	$n^2$	1	Yes	Insertion
<u>Selection sort</u>	$n^2$	$n^2$	$n^2$	1	Depends	Selection
<u>Shell sort</u>	$n$ <i>or</i> $n^{3/2}$	$n(\log n)^2$ <i>or</i> $O(n \log^2 n)$		1	No	Insertion
<u>Bubble sort</u>	$n$	$n^2$	$n^2$	1	Yes	Exchanging
<u>Bucket sort</u>		$n+k$			Yes	





## ▶ 排序算法

```
//例 07-22; ex07-22.cpp
//容器 vector 中排序函数的使用，演示 c++ 中容器 vector 中排序函数的高级使用
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool myfunction (int i, int j)
{
    return (i < j);
}

struct myclass
{
    bool operator() (int i, int j)
    {
        return (i < j);
    }
} myobject;

int main ()
{
    int myints[] = {32, 71, 12, 45, 26, 88, 53, 33};
    vector<int> v (myints, myints + 8);           // 32 71 12 45 26 88 53 33
    vector<int>::iterator it;

    // using default comparison (operator <):
    sort (v.begin(), v.begin() + 4);             // (12 32 45 71)26 88 53 33

    // using function as comp
    sort (v.begin() + 4, v.end(), myfunction); // 12 32 45 71(26 33 53 88)

    // using object as comp
    sort (v.begin(), v.end(), myobject);         // (12 26 32 33 45 53 71 88)

    // print out content:
    cout << "v contains:";
    for (it = v.begin(); it != v.end(); ++it)
        cout << " " << *it;

    cout << endl;
    return 0;
}
```

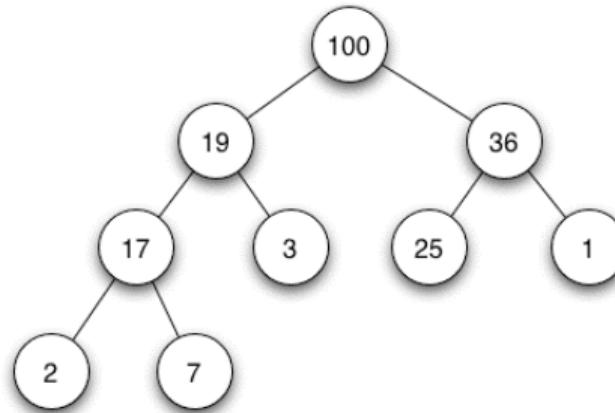




## ▶ 堆算法

**push\_heap**  
**pop\_heap**  
**make\_heap**  
**sort\_heap**

Push element into heap range  
Pop element from heap range  
Make heap from range  
Sort elements of heap





## ► 堆排序 sort\_heap

```
//例 07-23; ex07-23.cpp
//堆的建立, 插入元素到堆, 从堆中移出元素, 演示 c++ 中关于堆的函数使用
// range heap example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main ()
{
    int myints[] = {10, 20, 30, 5, 15};
    vector<int> v(myints, myints + 5);
    vector<int>::iterator it;

    make_heap (v.begin(), v.end());
    cout << "initial max heap : " << v.front() << endl;

    pop_heap (v.begin(), v.end());
    v.pop_back();
    cout << "max heap after pop : " << v.front() << endl;

    v.push_back(99);
    push_heap (v.begin(), v.end());
    cout << "max heap after push: " << v.front() << endl;

    sort_heap (v.begin(), v.end());

    cout << "final sorted range : ";
    for (unsigned i = 0; i < v.size(); i++) cout << " " << v[i];

    cout << endl;

    return 0;
}
```





## ► 二分查找

`lower_bound`  
`upper_bound`  
`equal_range`  
`binary_search`

Return iterator to lower bound  
Return iterator to upper bound  
Get subrange of equal elements  
Test if value exists in sorted array

前提: 操作的对象序列已经排序





## ► 二分查找 binary\_search

```
//例 07-24; ex07-24.cpp
//二分查找函数的使用，演示 c++ 中 binary_search 函数的使用
// binary_search example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool myfunction (int i, int j)
{
    return (i < j);
}

int main ()
{
    int myints[] = {1, 2, 3, 4, 5, 4, 3, 2, 1};
    vector<int> v(myints, myints + 9);           // 1 2 3 4 5 4 3 2 1

    // using default comparison:
    sort (v.begin(), v.end());

    cout << "looking for a 3... ";
    if (binary_search (v.begin(), v.end(), 3))
        cout << "found!\n";
    else cout << "not found.\n";

    // using myfunction as comp:
    sort (v.begin(), v.end(), myfunction);

    cout << "looking for a 6... ";
    if (binary_search (v.begin(), v.end(), 6, myfunction))
        cout << "found!\n";
    else cout << "not found.\n";

    return 0;
}
```





## ▶ 合并

<code>merge</code>	Merge sorted ranges
<code>inplace_merge</code>	Merge consecutive sorted ranges
<code>includes</code>	Whether sorted range includes another range
<code>set_union</code>	Union of two sorted ranges
<code>set_intersection</code>	Intersection of two sorted ranges
<code>set_difference</code>	Difference of two sorted ranges
<code>set_symmetric_difference</code>	Symmetric difference of two sorted ranges

前提: 操作的对象序列已经排序





## ► 求交集 set\_intersection

```
//例 07-25; ex07-25.cpp
//求两个集合的交集，演示 c++ 中 set_intersection 函数的使用
// set_intersection example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main ()
{
    int first[] = {5, 10, 15, 20, 25};
    int second[] = {50, 40, 30, 20, 10};
    vector<int> v(10);           // 0 0 0 0 0 0 0 0 0 0
    vector<int>::iterator it;

    sort (first, first + 5);   // 5 10 15 20 25
    sort (second, second + 5); // 10 20 30 40 50

    it = set_intersection (first, first + 5, second, second + 5, v.begin());
    // 10 20 0 0 0 0 0 0 0 0

    cout << "intersection has " << int(it - v.begin()) << " elements.\n";

    return 0;
}
```



## ► 最小最大值

<b>min</b>	Return the lesser of two arguments
<b>max</b>	Return the greater of two arguments
<b>min_element</b>	Return smallest element in range
<b>max_element</b>	Return largest element in range
<b>lexicographical_comparator</b>	Lexicographical less-than comparison
<b>next_permutation</b>	Transform range to next permutation
<b>prev_permutation</b>	Transform range to previous permutation





## ► 求最大值 max\_element

```
//例 07-26; ex07-26.cpp
//求数组中的最大最小值，演示 c++ 中 min_element、max_element 函数的使用
// min_element/max_element
#include <iostream>
#include <algorithm>
using namespace std;

bool myfn(int i, int j)
{
    return i < j;
}

struct myclass
{
    bool operator() (int i, int j)
    {
        return i < j;
    }
} myobj;

int main ()
{
    int myints[] = {3, 7, 2, 5, 6, 4, 9};

    // using default comparison:
    cout << "The smallest element is " << *min_element(myints, myints + 7) << endl;
    cout << "The largest element is " << *max_element(myints, myints + 7) << endl;

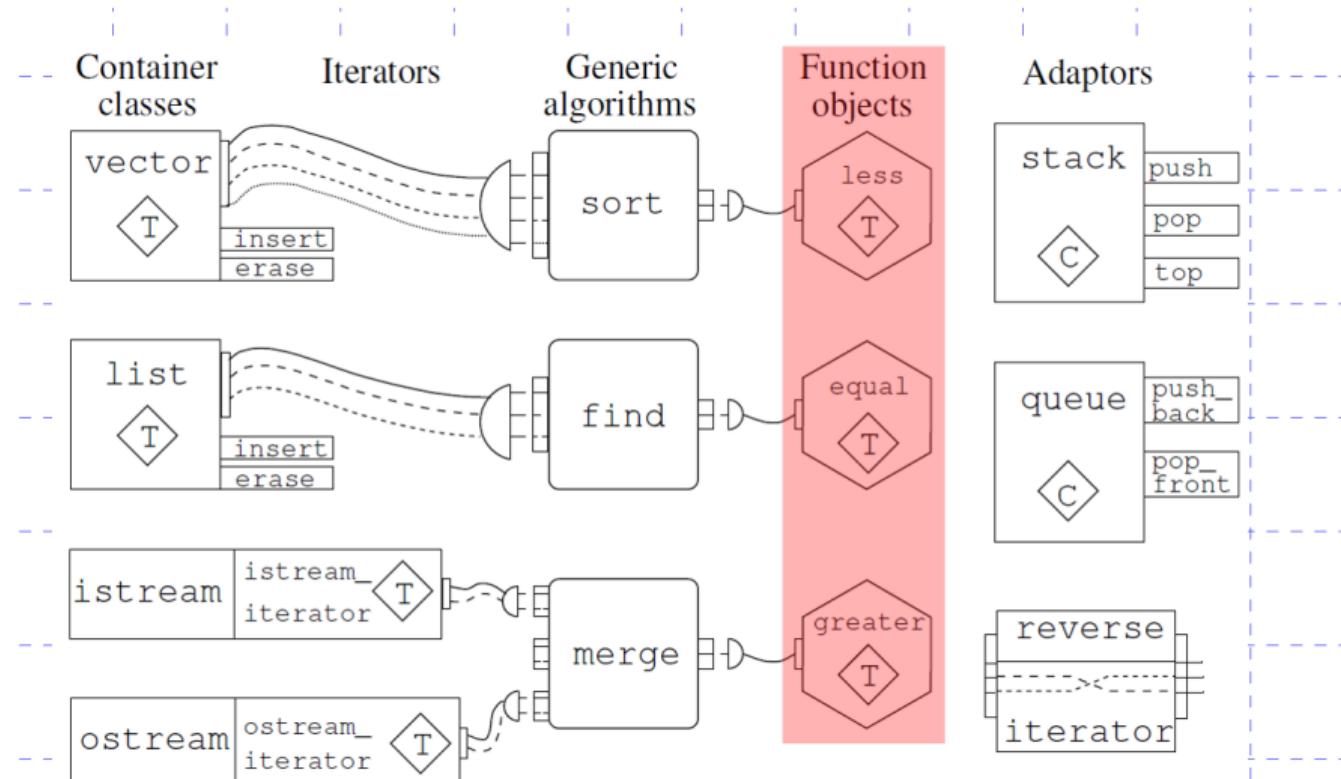
    // using function myfn as comp:
    cout << "The smallest element is " << *min_element(myints, myints + 7, myfn) << endl;
    cout << "The largest element is " << *max_element(myints, myints + 7, myfn) << endl;

    // using object myobj as comp:
    cout << "The smallest element is " << *min_element(myints, myints + 7, myobj) << endl;
    cout << "The largest element is " << *max_element(myints, myints + 7, myobj) << endl;

    return 0;
}
```



## ▶ 函数对象





### ▶ 定义

- ▶ 函数调用运算符 ()
- ▶ 如果某个类重载了 ()，该类的实例就是一个函数对象

---

```
struct MyClass
{
    返回值 operator()(参数列表) {}
};

MyClass myObj;
myObj(实参);
```

---

### ▶ 优点

- ▶ 使用灵活，可包含状态信息
- ▶ 可当作数据类型作为模板参数传递





### ► 例 1

```
//例 07-27; ex07-27.cpp
//模板类的定义
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

template <class T>
class AddClass
{
private:
    T theValue;
public:
    AddClass (const T& v) : theValue(v) {}
    void operator() (T& elem) const
    {
        elem += theValue;
    }
};

template <class T, int theValue>
void AddFun(T& elem)
{
    elem += theValue;
}
```





## ► 例 1

```
//例 07-27; ex07-27.cpp
//输出 AddFun 处理过后的容器中的值，演示 c++ 中 for_each 的使用
int main()
{
    vector<int> v(10, 0);
    const int n = 10;

    for_each (v.begin(), v.end(), AddFun<int, 10>);
    //for_each (v.begin(), v.end(), AddClass<int>(10));
    for_each (v.begin(), v.end(), Print);
    cout << endl;
    for_each (v.begin(), v.end(), AddFun < int, 11 > );
    //for_each (v.begin(), v.end(), AddClass<int>(11));
    for_each (v.begin(), v.end(), Print);
    cout << endl;
}
```





### ▶ 例 2

```
//例 07-27; ex07-27.cpp
//输出 AddFun 处理过后的容器中的值，演示 c++ 中 for_each 的使用
int main()
{
    vector<int> v(10, 0);
    const int n = 10;

    for_each (v.begin(), v.end(), AddFun<int, n>);
    //for_each (v.begin(), v.end(), AddClass<int>(n));
    for_each (v.begin(), v.end(), Print);
    cout << endl;
    for_each (v.begin(), v.end(), AddFun < int, n + 1 > );
    //for_each (v.begin(), v.end(), AddClass<int>(n+1));
    for_each (v.begin(), v.end(), Print);
    cout << endl;
}
```





### ► 例 2

```
//例 07-28; ex07-28.cpp
//定义 PersonSortClass 类
#include <iostream>
#include <string>
#include <set>

using namespace std;
struct Person
{
    string fName;
    string lName;
    Person(string gName = "", string fName = "") :
        fName(gName), lName(fName) {}
};

class PersonSortClass
{
public:
    bool operator() (const Person& p1, const Person& p2)
    {
        if (p1.lName < p2.lName)
            return true;
        else if(p1.lName == p2.lName)
            return (p1.fName < p2.fName);
        else
            return false;
    };
    bool PersonSortFun(const Person& p1, const Person& p2)
    {
        if (p1.lName < p2.lName)
            return true;
        else if(p1.lName == p2.lName)
            return (p1.fName < p2.fName);
        else
            return false;
    }
}
```





### ► 例 2

```
//例 07-28; ex07-28.cpp
//在容器 set 中插入元素，演示 c++ 中 set 容器的高级用法
int main()
{
    typedef set<Person, PersonSortClass> PersonSet;
    PersonSet s;
    PersonSet::iterator p;

    s.insert(Person("Thomas ", "Edison"));
    s.insert(Person("Helen   ", "Keller"));
    s.insert(Person("James   ", "Taylor"));
    s.insert(Person("Elizabeth", "Taylor"));
    s.insert(Person("Isaac   ", "Newton"));
    s.insert(Person("Lewis   ", "Carrol"));

    for (p = s.begin(); p != s.end(); ++p)
        cout << (*p).fName << "\t" << (*p).lName << endl;
}
```





- ▶ 算术操作 Arithmetic operations
- ▶ 比较操作 Comparison operations
- ▶ 逻辑操作 Logical operations
- ▶ 其它





### ► 算术操作

<b>plus</b>	Addition function object class
<b>minus</b>	Subtraction function object class
<b>multiplies</b>	Multiplication function object class
<b>divides</b>	Division function object class
<b>modulus</b>	Modulus function object class
<b>negate</b>	Negative function object class





### ► 算术操作

```
template <class T>
struct plus : binary_function <T,T,T>
{
    T operator() (const T& x, const T& y) const
    {
        return x+y;
    }
};
```





## ▶ plus 类模板

```
//例 07-29; ex07-29.cpp
//first、second 数组中的对应元素相加，演示 c++ 中 transform 函数的使用
#include <iostream>
#include <functional>
#include <algorithm>
using namespace std;

int main ()
{
    int first[] = {1, 2, 3, 4, 5};
    int second[] = {10, 20, 30, 40, 50};
    int results[5];
    transform(first, first + 5, second, results, plus<int>());
    for (int i = 0; i < 5; i++)
        cout << results[i] << " ";
    cout << endl;
    return 0;
}
```





### ► 比较操作

<code>equal_to</code>	Function object class for equality comparison
<code>not_equal_to</code>	Non-equality comparison
<code>greater</code>	Greater-than inequality comparison
<code>less</code>	Less-than inequality comparison
<code>greater_equal</code>	Greater-than-or-equal-to comparison
<code>less_equal</code>	Less-than-or-equal-to comparison





## ► 大于操作 greater

```
//例 07-30; ex07-30.cpp
//数组排序，演示 c++ 中 sort 函数的使用
#include <iostream>
#include <functional>
#include <algorithm>
using namespace std;

int main ()
{
    int numbers[] = {20, 40, 50, 10, 30};
    sort (numbers, numbers + 5, greater<int>());
    for (int i = 0; i < 5; i++)
        cout << numbers[i] << " ";
    cout << endl;
    return 0;
}
```





### ► 逻辑操作

<code>logical_and</code>	Logical AND function object class
<code>logical_or</code>	Logical OR function object class
<code>logical_not</code>	Logical NOT function object class





### ► 逻辑非操作 logical\_not

```
template <class T>
struct logical_not : unary_function <T,bool>
{
    bool operator() (const T& x) const
    {
        return !x;
    }
};
```





## ▶ 逻辑非操作 logical\_not

```
//例 07-31; ex07-31.cpp
//对数组 value 中的布尔值取反，演示 c++ 中 transform 函数的使用
#include <functional>
#include <algorithm>

int main ()
{
    bool values[] = {true, false};
    bool result[2];

    transform (values, values + 2, result, logical_not<bool>());
    cout << boolalpha << "Logical NOT:\n";

    for (int i = 0; i < 2; i++)
        cout << "NOT " << values[i] << " = " << result[i] << "\n";

    return 0;
}
```





- ▶ 树
- ▶ 图
- ▶ 其它
  - ▶ 矩阵运算
  - ▶ 正则表达式
  - ▶ 智能指针
  - ▶ 高斯分布随机数
  - ▶ 多线程操作等

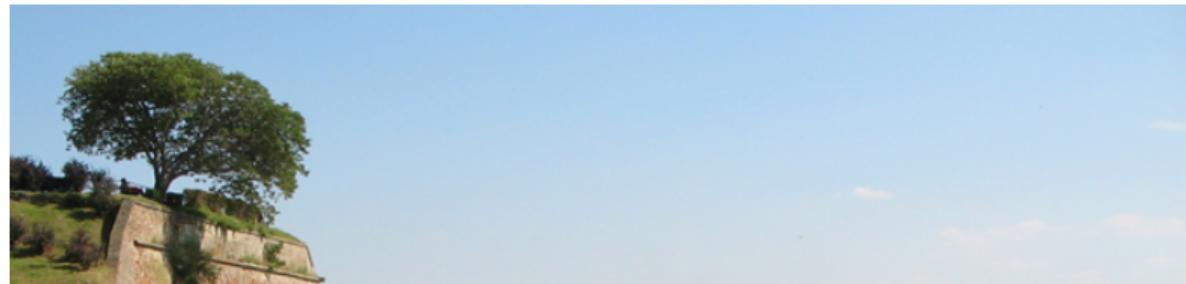




- ▶ Core::tree(适合在 VS 下编译)
- ▶ Tree.hh: an STL-like C++ tree class

<http://archive.gamedev.net/archive/reference/programming/features/coretree2/page5.html>

<http://tree.phi-sci.com/documentation.html>





- ▶ 一组扩充 C++ 功能性的经过同行评审（Peer-reviewed）且开放源代码程序库。
- ▶ 开源代码。 <http://www.boost.org/>
- ▶ 许多 Boost 的开发人员来自 C++ 标准委员会，而部份的 Boost 库成为 C++ 的 TR1 标准之一。



[http://mail.ustc.edu.cn/~jxw95216/boost\\_doc/libs/libraries.htm](http://mail.ustc.edu.cn/~jxw95216/boost_doc/libs/libraries.htm) <http://www.kmonos.net/aling/boost/>





## ▶ 分类库列表

- ▶ String and text processing 字符串与文本处理
- ▶ Containers 容器
- ▶ Iterators 迭代器
- ▶ Algorithms 算法
- ▶ Function Objects and higher-order programming 函数对象与高阶编程
- ▶ Generic Programming 泛型编程
- ▶ Template Metaprogramming 模板元编程
- ▶ Preprocessor Metaprogramming 预处理元编程
- ▶ Concurrent Programming 并发编程
- ▶ Math and numerics 数学与数字
- ▶ Correctness and testing 正确性与测试
- ▶ Data structures 数据结构
- ▶ Image processing 图像处理
- ▶ Input/Output 输入/输出
- ▶ Inter-language support 交叉语言支持
- ▶ Memory 内存
- ▶ Parsing 语法分析
- ▶ Programming Interfaces 编程接口
- ▶ Miscellaneous 杂项
- ▶ Broken compiler workarounds 不合标准的编译器支持





## ▶ 安装与编译

- ▶ `date_time`, `regex`, `thread`, `python`, `signals`, `test`,  
`filesystem`, `serialization`, `program_options` 需要 `bjam` 进行编译生  
成 `*.lib` 文件;  
VS: <http://www.boostpro.com/download/>
- ▶ 其余组件只需包含相应头文件即可。



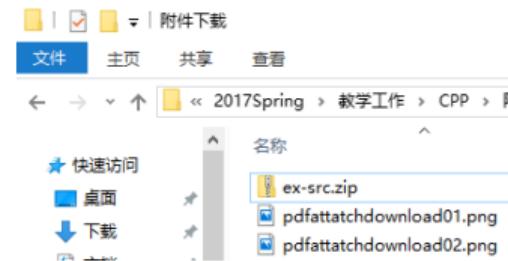
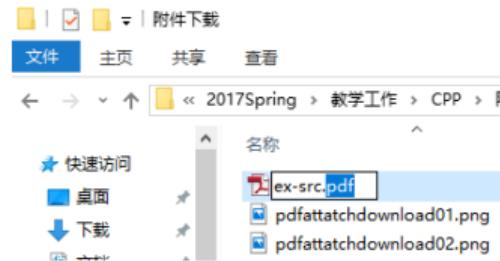
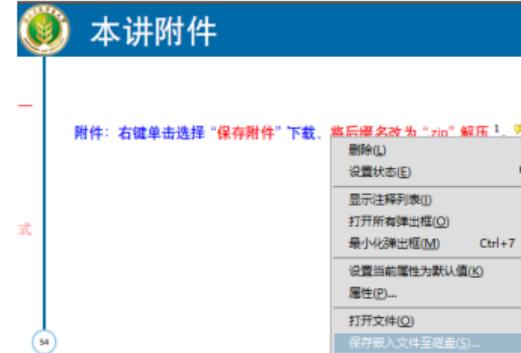
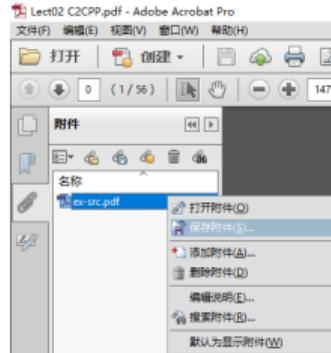


# 本讲附件

| 附件

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP  
  
Nine, G.

附件：右键单击该链接，选择“保存附件”下载，将后缀名改为“.zip”解压。<sup>18 19</sup>

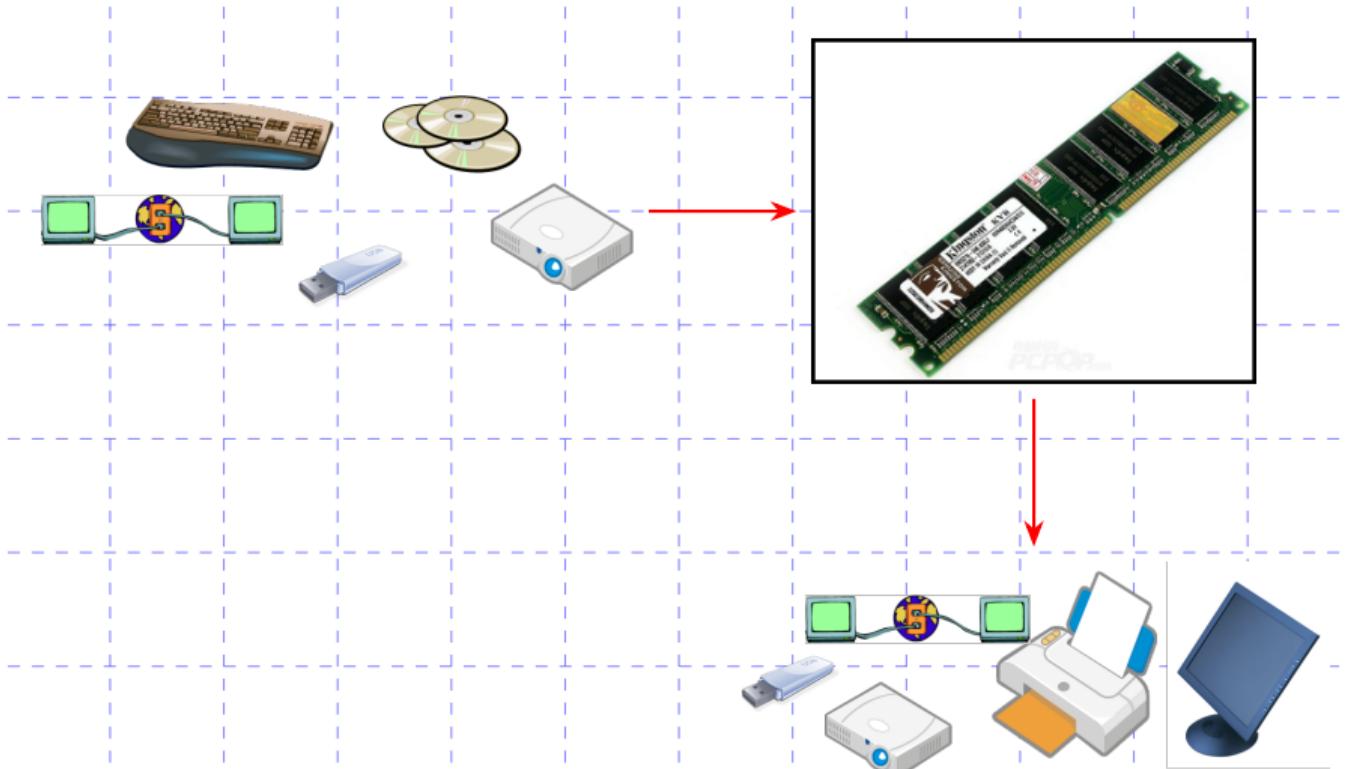


<sup>18</sup>请退出全屏模式后点击该链接。

<sup>19</sup>以 Adobe Acrobat Reader 为例。

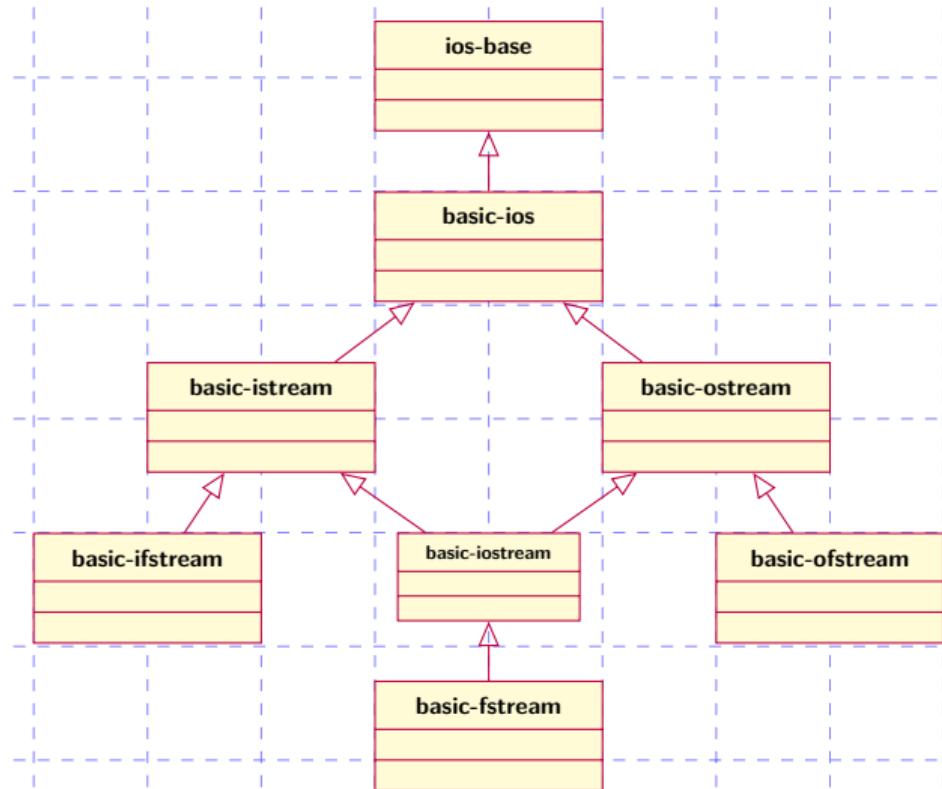


- ▶ 什么是流 (stream)
  - ▶ 与物理输入输出设备无关的数据序列





## ▶ 输入输出流类模板层次结构图





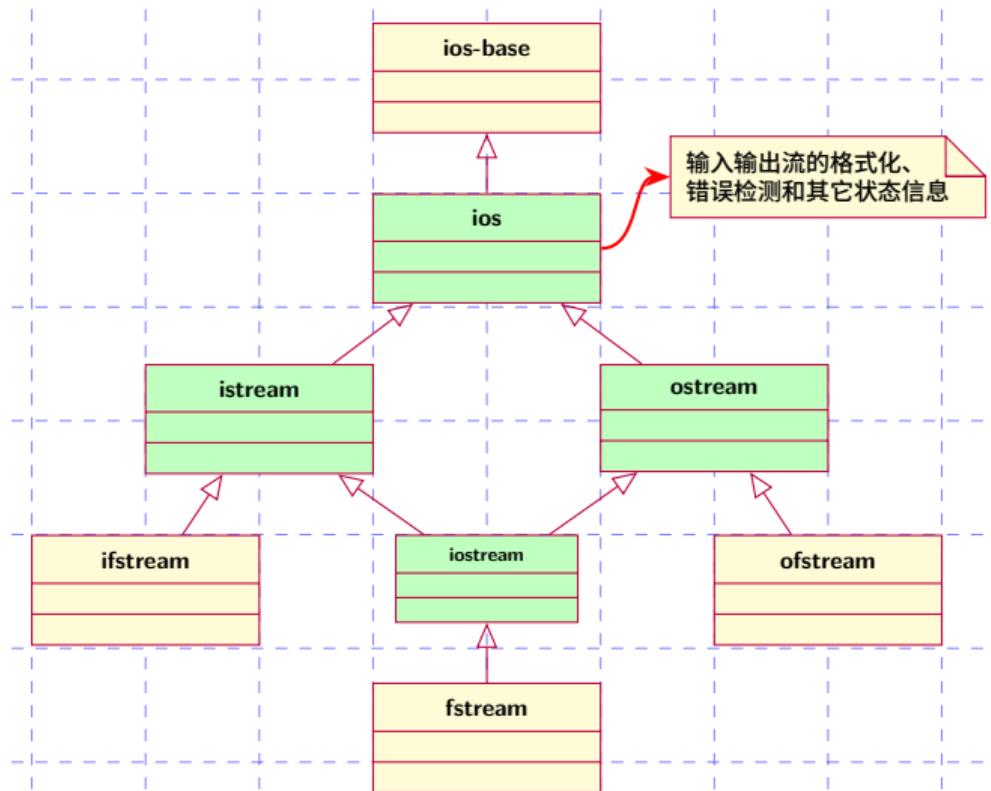
- 类模板可实例化为窄字符类 (`char`) 和宽字符类 (`wchar_t`)

<b>Template Class</b>	<b>Character-based Class</b>	<b>Wide-Character-based Class</b>
<code>basic_ios</code>	<code>ios</code>	<code>wios</code>
<code>basic_istream</code>	<code>istream</code>	<code>wistream</code>
<code>basic_ofstream</code>	<code>ostream</code>	<code>wostream</code>
<code>basic_iostream</code>	<code>iostream</code>	<code>wiostream</code>
<code>basic_fstream</code>	<code>fstream</code>	<code>wfstream</code>
<code>basic_ifstream</code>	<code>ifstream</code>	<code>wifstream</code>
<code>basic_ofstream</code>	<code>ofstream</code>	<code>wofstream</code>



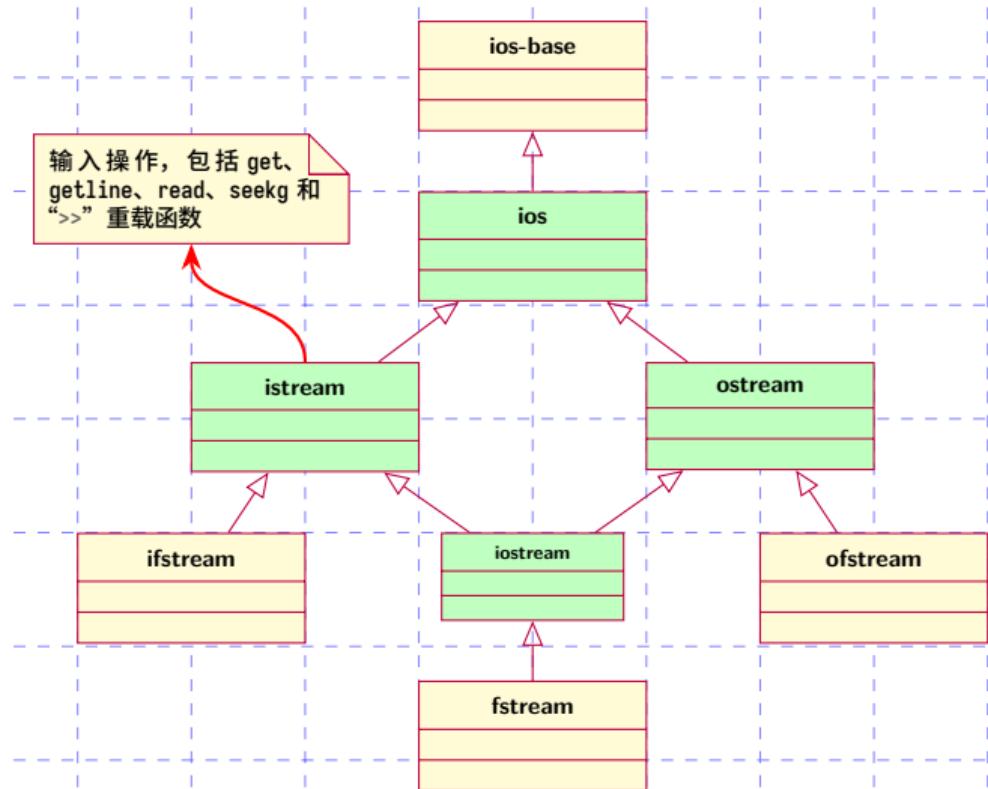


## ▶ 输入输出流类模板层次结构图



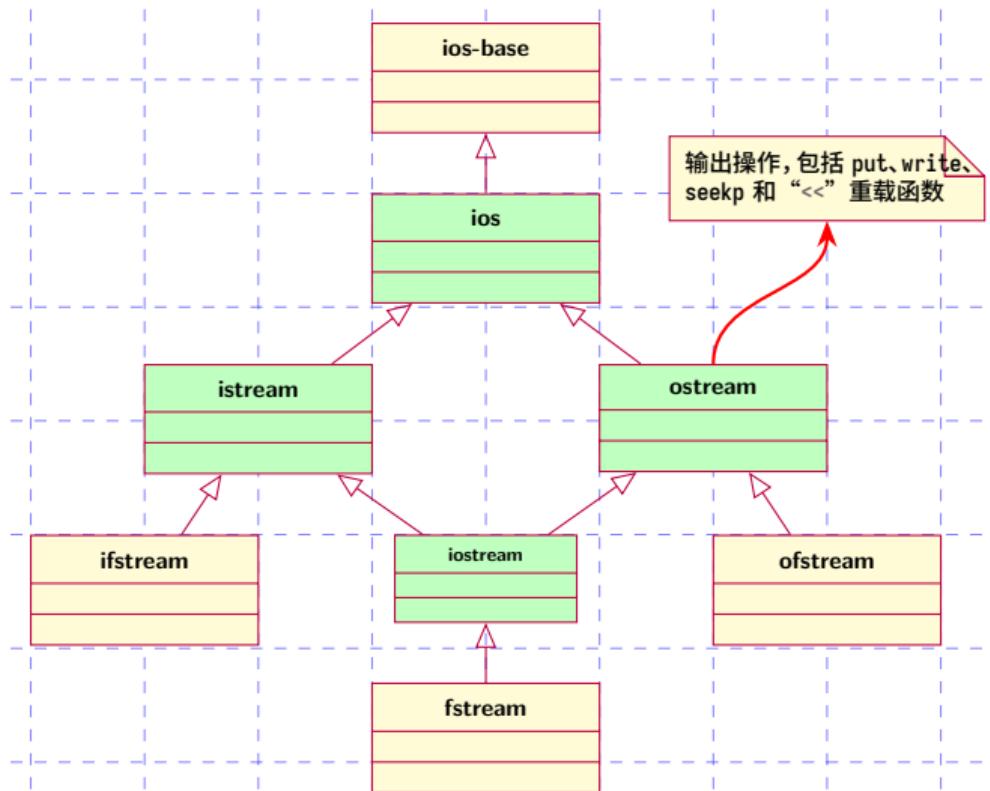


## ▶ 输入输出流类模板层次结构图



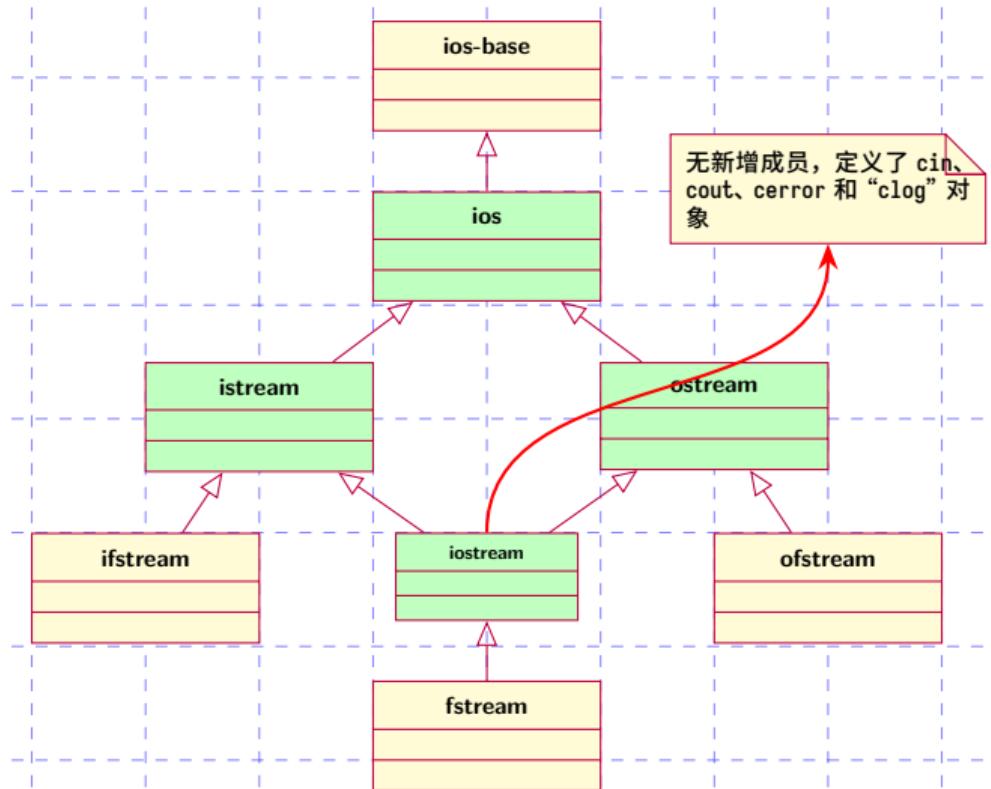


## ▶ 输入输出流类模板层次结构图



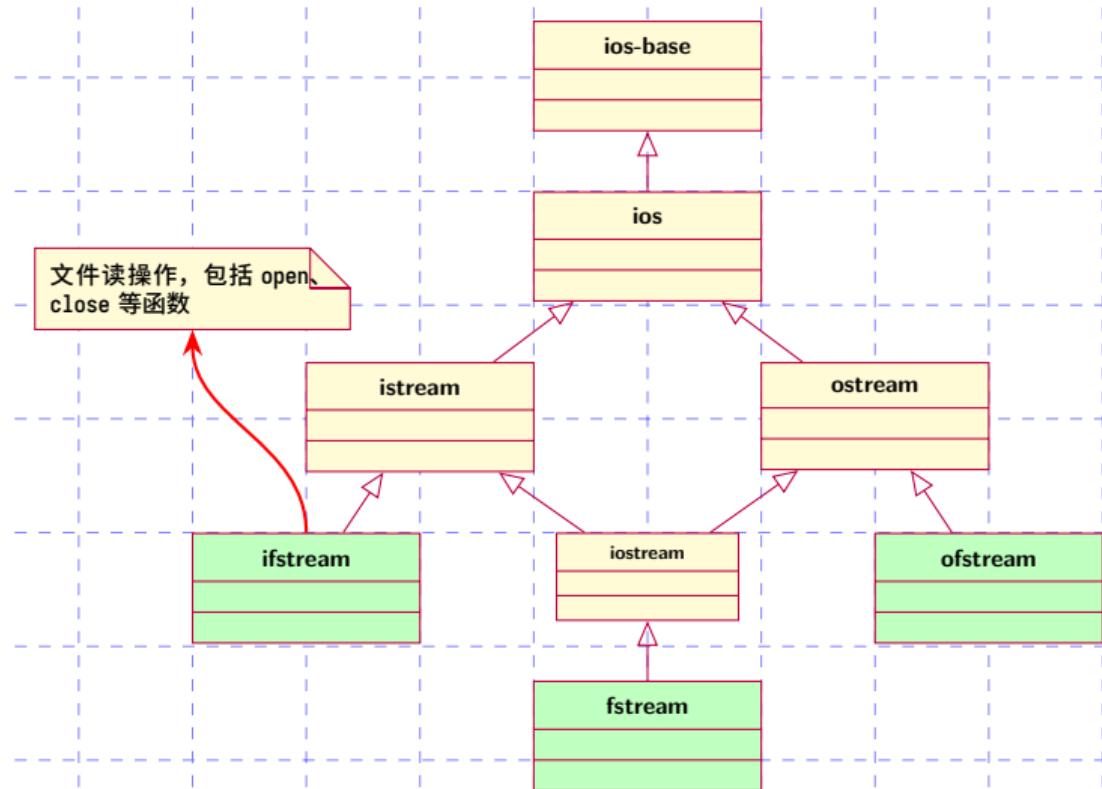


## ▶ 输入输出流类模板层次结构图



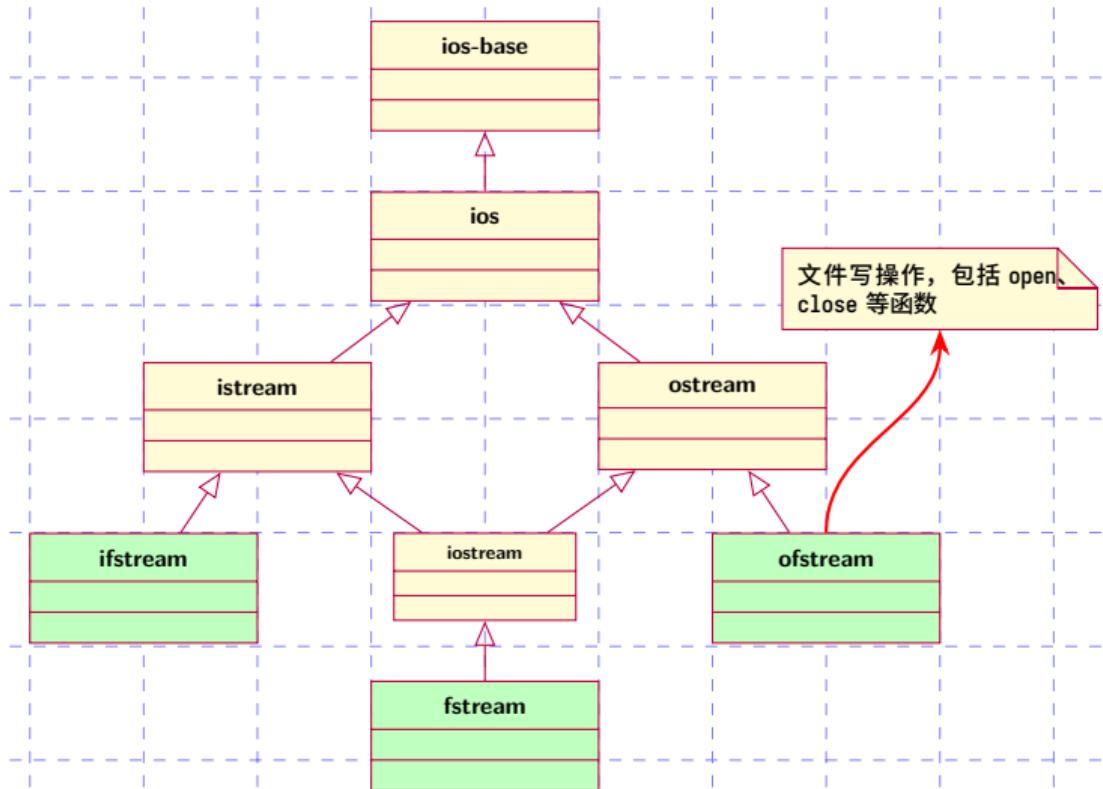


## ▶ 输入输出流类模板层次结构图



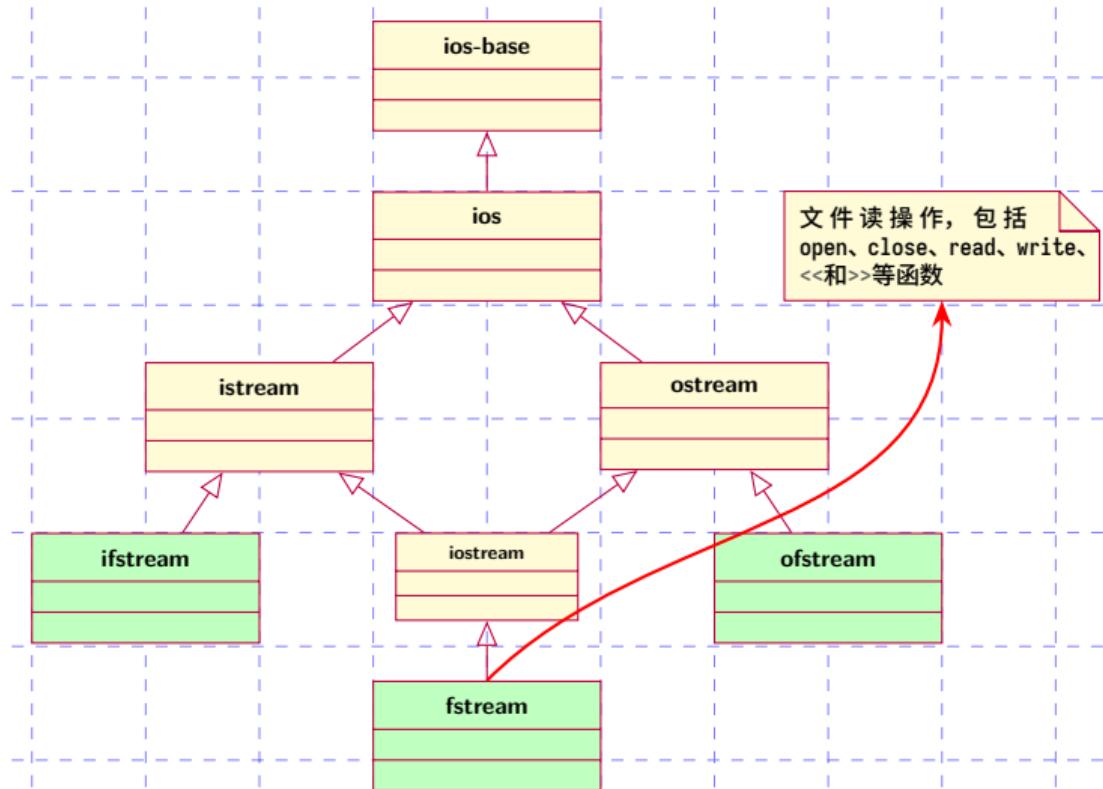


## ▶ 输入输出流类模板层次结构图





## ▶ 输入输出流类模板层次结构图





### ▶ 输出流常用函数 (ostream)

- ▶ 运算符重载函数 “<<”
- ▶ 输出单个字符到屏幕或文件  
`ostream &put(char ch)`
- ▶ 输出字符块信息到屏幕或文件  
`ostream &write(const char* pch, int nCount)`





### ▶ 输出流常用函数 (ostream)

```
#include <iostream>

using namespace std;

int main()
{
    char buff[11] = "0123456789";

    cout << buff << endl;

    for (int i=0; i<10; i++)
        cout.put(buff[i]);
    cout << endl;

    cout.write(buff,10);
    cout << endl;
    return 0;
}
```





### ▶ 格式控制

- ▶ ios 类提供直接设置标志字的控制格式函数
- ▶ iostream 和 iomanip 库还提供一批操纵符简化 I/O 格式化操作

### ▶ 格式控制标志位 (16 位二进制数)

- ▶ 进制 (进制组合 ios::basefield)
- ▶ 对齐 (对齐组合 ios::adjustfield)
- ▶ 浮点数 (浮点组合 ios::floatfield)
- ▶ 忽略输入流空格 (ios::skipws)
- ▶ 显示正号
- ▶ 显示小数点
- ▶ ...

0	0	0	1	0	0	0	0	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---





## ▶ 格式控制

field	member constant	effect when set
independent flags	boolalpha	read/write bool elements as alphabetic strings (true and false).
	showbase	write integral values preceded by their corresponding numeric base prefix.
	showpoint	write floating-point values including always the decimal point.
	showpos	write non-negative numerical values preceded by a plus sign (+).
	skipws	skip leading whitespaces on certain input operations.
	unitbuf	flush output after each inserting operation.
	uppercase	write uppercase letters replacing lowercase letters in certain insertion operations.
numerical base (basefield)	dec	read/write integral values using decimal base format.
	hex	read/write integral values using hexadecimal base format.
	oct	read/write integral values using octal base format.
float format (floatfield)	fixed	write floating point values in fixed-point notation.
	scientific	write floating-point values in scientific notation.
adjustment (adjustfield)	internal	the output is padded to the <i>field width</i> by inserting <i>fill characters</i> at a specified internal point.
	left	the output is padded to the <i>field width</i> appending <i>fill characters</i> at the end.
	right	the output is padded to the <i>field width</i> by inserting <i>fill characters</i> at the beginning.

Three additional bitmask constants made of the combination of the values of each of the three groups of selective flags can also be used:

flag value	equivalent to
adjustfield	left   right   internal
basefield	dec   oct   hex
floatfield	scientific   fixed



### ► 状态字设置函数

函数	功能
<code>long flags( long lFlags );</code> <code>long flags() const;</code>	用参数 lFlags 更新标志字 返回标志字
<code>long setf( long lFlags );</code> <code>long setf( long lFlags, long lMask );</code>	设置 lFlags 指定的标志位 将 lMask 指定的位清 0，然后设置 lFlags 指定位
<code>long unsetf( long lFlags );</code>	将 lMask 指定的标志位清 0





## ▶ 输出流常用函数 (ostream)

```
#include <iostream>
using namespace std;

struct fntflags
{
    long flag;
    char flagname[12];
} flags[18] = {{ios::hex, "hex"}, {ios::dec, "dec"}, {ios::oct, "oct"}, {ios::basefield, "basefield"}, {ios::internal, "internal"}, {ios::left, "left"}, {ios::right, "right"}, {ios::adjustfield, "adjustfield"}, {ios::fixed, "fixed"}, {ios::scientific, "scientific"}, {ios::basefield, "basefield"}, {ios::showbase, "showbase"}, {ios::showpoint, "showpoint"}, {ios::showpos, "showpos"}, {ios::skipws, "skipws"}, {ios::uppercase, "uppercase"}, {ios::boolalpha, "boolalpha"}, {ios::unitbuf, "unitbuf"}};

int main()
{
    long IFlags, INewFlags;
    IFlags = cout.setf(ios::hex, ios::basefield);
    INewFlags = cout.flags();
    cout << "Default flag is:" << IFlags << endl;
    cout << "New flag is:" << INewFlags << endl;
    for(int i = 0; i < 18; i++)
        cout << flags[i].flag << '\t' << flags[i].flagname << endl;
    return 0;
}
```

dec	2	10
hex	8	100

oct	40	1000000
-----	----	---------

0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0



### ► 格式控制成员函数

- 设置域宽: `int width(int nw);`
- 填充: `fill(char c);`
- 设置浮点数精度: `int precision(int n);`
- 设置格式: `long flags(long lFlags);`





### ► 格式操纵符：简化 I/O 格式化操作 (iomanip)

- 进制 (dec、oct、hex 和 setbase())
- 对齐
- 浮点数显示
- 设置当前域宽 (setw)
- 填充 (setfill)
- 设置精度 (setprecision)
- ...





### ► 格式操纵符：简化 I/O 格式化操作

```
#include <iostream>
using namespace std;

int main()
{
    long IFlags, INewFlags;
    IFlags = cout.flags();
    cout << hex;
    INewFlags = cout.flags();
    cout << "Default flag is:" << IFlags << endl;
    cout << "New flag is:" << INewFlags << endl;
    for(int i = 0; i < 18; i++)
        cout << flags[i].flag << '\t' << flags[i].flagname << endl;
    return 0;
}
```





### ► 格式控制成员函数

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    double PI = 3.1415926535;
    int precision;
    cout << fixed;
    cout << PI << endl;
    cout.width(8);
    cout.fill('0');
    for(precision = 0; precision <= 9; precision++)
    {
        cout.precision(precision);
        cout << PI << endl;
    }
    cout << PI << endl;
    cout << setw(8) << setfill('0');
    for(precision = 0; precision <= 9; precision++)
    {
        cout << setprecision(precision);
        cout << PI << endl;
    }
    return 0;
}
```





### ▶ 输入流常用函数 (istream)

<code>read</code>	无格式输入指定字节数
<code>get</code>	从流中提取字符，包括空格
<code>getline</code>	从流中提取一行字符
<code>seekg</code>	移动输入流指针
<code>operator&gt;&gt;</code>	输入运算符





### ▶ 输入流常用函数 get()

```
#include<iostream>

using namespace std;
int main()
{
    char c;

    while ( (c = cin.get()) != '\n' )
        cout.put(c);

    cout << endl;

    char s[ 80 ];
    cin.get(s, 10);
    cout << s << endl;
}
```





### ▶ 输入流常用函数 getline

```
#include<iostream>

using namespace std;
int main()
{
    const int max_len = 256;
    char line[max_len];
    int i=0;

    while ( (cin.getline(line, max_len)))
    {
        cout << "[" << i << "]:" << line << endl;;
        i++;
    }
}
```





### ▶ 输入流常用函数 getline

```
#include<iostream>

using namespace std;
int main()
{
    const int max_len = 256;
    char line[max_len];
    int i=0;

    while ( (cin.getline(line, max_len, '\n')) )
    {
        cout << "[" << i << "]: " << line << endl;;
        i++;
    }
}
```





## ▶ 文件的创建

### ▶ 方法 1

- ▶ 使用 ifstream、 ofstream 或 fstream 类建立文件流对象（无参构造函数）
- ▶ 调用成员函数 open 建立文件

### ▶ 方法 2

- ▶ 使用 ifstream、 ofstream 或 fstream 类建立文件流对象（调用带参数构造函数）





### ▶ 文件的创建

```
fstream file;
file.open("d:\\data\\exe1.txt", ios::out);
file.close();
```

```
fstream file("d:\\data\\exe1.txt", ios::out);
file.close();
```





### ▶ 文件的创建

```
fstream file;  
file.open("d:\\data\\exe1.txt", ios::out);  
file.close();
```

```
ofstream file;  
file.open("d:\\data\\exe1.txt");  
file.close();
```





### ▶ 文件的创建

```
fstream file;  
file.open("d:\\data\\exe1.txt", ios::in);  
file.close();
```

```
ifstream file;  
file.open("d:\\data\\exe1.txt");  
file.close();
```





## ▶ 文本文件的写操作

```
#include <fstream>
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string name;
    float score;
    fstream myfile;
    myfile.open("record.txt", ios::out);
    if(!myfile)
    {
        cerr << "File open or create error!" << endl;
        return 0;
    }

    while( cin >> name && name != "exit")
    {
        cin >> score;
        myfile << name << ' ' << score << endl;
    }

    myfile.close();
    return 0;
}
```





## ▶ 文本文件的读操作

```
#include <fstream>
#include <iostream>
using namespace std;

int main()
{
    string name;
    float score;
    ifstream myFile;

    myFile.open("record.txt");
    if(!myFile)
    {
        cerr << " record.txt open error!" << endl;
        return 0;
    }

    while(!myFile.eof())
    {
        myFile >> name >> score;
        if(myFile) cout << name << '\t' << score << endl;
    }

    myFile.close();
    return 0;
}
```





## ▶ 文件合并排序操作

- ▶ 高考成绩
- ▶ 姓名、准考证号、所考大学、总成绩
- ▶ 准考证号是关键字
- ▶ 有重复记录
- ▶ 合并，去掉重复记录，并按准考证号升序排列

```
struct StuNode
{
    string name;
    int ID;
    string univ;
    int score;
    StuNode(string _name = "", int _ID = 0, string _univ = "", int _score = 0)
        : name(_name), ID(_ID), univ(_univ), score(_score) {}
    friend bool operator == (const StuNode &l, const StuNode &r)
    {
        return (l.ID == r.ID);
    }
    friend bool operator < (const StuNode &l, const StuNode &r)
    {
        return (l.ID < r.ID);
    }
};
```



## ▶ 文件合并排序操作

```
void ReadFile(char *fileName, list <StuNode> &buff)
{
    ifstream inFile(fileName);
    StuNode stu;

    if(!inFile)
    {
        cout << fileName << " read error!" << endl;
        exit(0);
    };

    while(!inFile.eof())
    {
        inFile >> stu.name;
        inFile >> stu.ID;
        inFile >> stu.univ;
        inFile >> stu.score;
        buff.push_back(stu);
    }
    inFile.close();
}
```





## ▶ 文件合并排序操作

```
void PrintFile(char *fileName, const list <StuNode> &buff)
{
    list <StuNode>::const_iterator it;
    ofstream outFile(fileName);

    if(!outFile) return;
    for (it = buff.begin(); it != buff.end(); it++)
    {
        outFile << it->ID << "\t" << it->name << "\t"
            << it->univ << "\t" << it->score << endl;
    }
    outFile.close();
}
```





## ▶ 文件合并排序操作

```
#include <iostream>
#include <string>
#include <list>
#include <fstream>
#include <cstdlib>

using namespace std;

int main()
{
    list<StuNode> first, second;

    ReadFile("first.txt", first);
    ReadFile("second.txt", second);

    first.sort();
    second.sort();

    first.merge(second);
    first.unique();

    PrintFile("result.txt", first);

    return 0;
}
```





## ▶ 二进制文件的输出

```
#include <fstream>
#include <iostream>
#include <cstring>

using namespace std;

int main()
{
    char name[8];
    float score;
    fstream myfile;
    myfile.open("record.bin", ios::out | ios::binary);
    if(!myfile)
    {
        cerr << "File open or create error!" << endl;
        return 0;
    }

    while( cin >> name && strcmp(name, "exit"))
    {
        cin >> score;
        myfile.write(name, 8 * sizeof(char));
        myfile.write((char *)&score, sizeof(float));
    }

    myfile.close();
    return 0;
}
```

ostream& write(const char\* s , streamsize n);





## ▶ 二进制文件的输入

```
#include <fstream>
#include <iostream>
#include <string>

using namespace std;

int main()
{
    char name[8];
    float score;
    fstream myfile;
    myfile.open("record.bin", ios::in | ios::binary);
    if(!myfile)
    {
        cerr << "File open or create error!" << endl;
        return 0;
    }

    while(!myfile.eof())
    {
        myfile.read((char *)name, 8 * sizeof(char));
        myfile.read((char *)&score, sizeof(float));
        if(myfile) cout << name << '\t' << score << endl;
    }

    myfile.close();
    return 0;
}
```

istream& read(char\* s, streamsize n);





## ▶ 二进制文件的输入





## ▶ 二进制文件的输入

```
#include <iostream>
#include <fstream>

using namespace std;

//下列代码处理图像宽度必须为 4 的倍数!
#define MIN3(x,y,z) ((y) <= (z) ? ((x) <= (y) ? (x) : (y)) : ((x) <= (z) ? (x) : (z)))
#define MAX3(x,y,z) ((y) >= (z) ? ((x) >= (y) ? (x) : (y)) : ((x) >= (z) ? (x) : (z)))

#pragma pack(1)
struct FILEHEADER
{
    char type[18];      // 标识"BM"
    int width;          // 图像宽
    int height;         // 图像高
    char dummy[28];     // 无用信息
};

struct RGB
{
    unsigned char b; // 蓝
    unsigned char g; // 绿
    unsigned char r; // 红
};
```





## ▶ 二进制文件的输入

```
int main()
{
    FILEHEADER imgHead1, imgHead2;
    RGB pixel1, pixel2, pixel;
    float blendRatio = 0.5;
    fstream imgFile1, imgFile2, imgFileResult;

    imgFile1.open("Lighthouse.bmp", ios::in | ios::binary);
    imgFile2.open("Penguins.bmp", ios::in | ios::binary);
    imgFileResult.open("Result.bmp", ios::out | ios::binary);

    if(!imgFile1 || !imgFile2 || !imgFileResult)
    {
        cerr << "File open or create error!" << endl;
        return 0;
    }

    imgFile1.read((char *)(&imgHead1), sizeof(FILEHEADER));
    if(!imgFile1)
        return 0;

    cout << imgHead1.type[0] << imgHead1.type[1] << endl;
    cout << imgHead1.width << "," << imgHead1.height << endl;

    imgFile2.read((char *)(&imgHead2), sizeof(FILEHEADER));
    if(!imgFile2)
        return 0;

    cout << imgHead2.type[0] << imgHead2.type[1] << endl;
    cout << imgHead2.width << "," << imgHead2.height << endl;
```





## ▶ 二进制文件的输入

```
if(imgHead1.width != imgHead2.width || imgHead1.height != imgHead2.height)
    return 0;

imgFileResult.write((char *)(&imgHead1), sizeof(FILEHEADER));

for(int i = 0; i < (imgHead1.width * imgHead1.height); i++)
{
    imgFile1.read((char *)&pixel1), sizeof(RGB));
    if(!imgFile1) return 0;
    imgFile2.read((char *)&pixel2), sizeof(RGB));
    if(!imgFile2) return 0;
    pixel = pixel1;

    pixel.b = pixel1.b * blendRatio + pixel2.b * (1 - blendRatio);
    pixel.g = pixel1.g * blendRatio + pixel2.g * (1 - blendRatio);
    pixel.r = pixel1.r * blendRatio + pixel2.r * (1 - blendRatio);

    pixel.r = pixel.g = pixel.b;
    imgFileResult.write((char *)(&pixel), sizeof(RGB));
}

imgFile1.close();
imgFile2.close();
imgFileResult.close();
return 0;
}
```



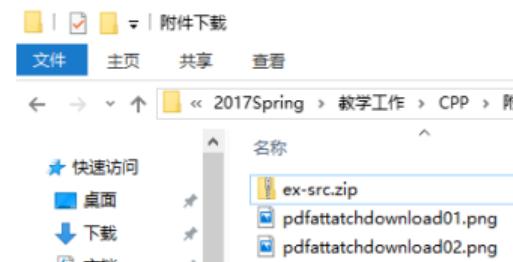
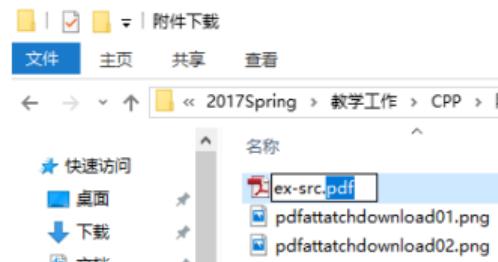
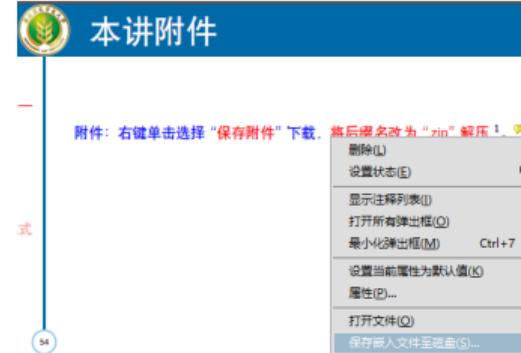
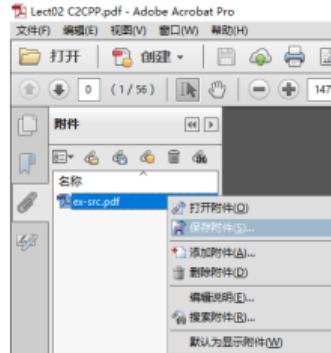


# 本讲附件

| 附件

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP  
Nine, G.

附件：右键单击该链接，选择“保存附件”下载，将后缀名改为“.zip”解压。<sup>20 21</sup>



<sup>20</sup>请退出全屏模式后点击该链接。

<sup>21</sup>以 Adobe Acrobat Reader 为例。



- ▶ #include <string>
- ▶ string 类构造函数原型
  - ▶ string()
  - ▶ string(**const** string& rhs)
  - ▶ string(**const** string& rhs, **unsigned** pos,  
**unsigned** n)
  - ▶ string(**const** char \*)
  - ▶ string(**const** char \*s, **unsigned** n)
  - ▶ string(**unsigned** n, **char** c)





### ► string 类构造函数

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    char * S1 = "The quick brown fox jumps over the lazy dog";
    string S2 = S1;
    string S3("lazy dog");           //用字符串初始化新串
    string S4(S3);                 //利用已存在的串 S3 初始化新串
    string S5(S2, 4, 15);          //利用已存在的串 S3 初始化新串
    string S6(S1, 19);             //利用已存在的字符数组初始化新串
    string S7(50, '-');
    cout << S7 << endl;
    cout << "S2=" << S2 << endl;
    cout << "S3=" << S3 << endl;
    cout << "S4=" << S4 << endl;
    cout << "S5=" << S5 << endl;
    cout << "S6=" << S6 << endl;
    cout << S7 << endl;
}
```





### ► string 类成员函数

- ▶ `length, size`: 返回字符串长度
- ▶ `append, push_back`: 添加新串到本字符串末尾
- ▶ `assign`: 字符串选择赋值
- ▶ `insert`: 字符串插入函数
- ▶ `substr`: 返回子字符串
- ▶ `find, rfind`: 字符串查找, 未找到返回 `string::npos`
- ▶ `replace`: 字符串替换
- ▶ `swap`: 交换两个字符串





### ▶ 使用 string 类成员函数查找字符串并替换

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string text("I like C++, I like to code in C++.");
    string newstr, sstr;
    int pos;
    cout << "Input string and new string:";
    cin >> sstr >> newstr;
    if((pos = text.find(sstr)) == string::npos)
        cout << sstr << " not found in \"\" " << text << "\"\" " << endl;
    else
    {
        cout << "old string: " << text << endl;
        text.replace(pos, sstr.length(), newstr);
        cout << "new string: " << text << endl;
    }
    return 0;
}
```





## ▶ 使用 string 类成员函数查找字符串并替换

```
#include <string>
#include <iostream>
using namespace std;

int main()
{
    string originText("I like C++, I like to code in C++.");           //文本
    string text = originText;
    string newstr, sstr;
    int pos;                                                               //存放查找到串的位置
    cout << "Input string and new string:";
    cin >> sstr >> newstr;
    if((pos = text.find(sstr)) == string::npos)                          //未查找到
        cout << sstr << " not found in \" " << text << "\" " << endl;
    else
        text.replace(pos, sstr.length(), newstr);

    while((pos = text.find(sstr, pos + 1)) != string::npos)
        text.replace(pos, sstr.length(), newstr);

    cout << "01234567890123456789012345678901234567890" << endl;
    cout << originText << endl;
    cout << text << endl;
    return 0;
}
```





## ► string 类操作符

- +
- =
- +=
- ==, !=, <, <=, >, >=
- []
- <<
- >>





## ► string 类操作符

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string str1("I like C++!");
    string str2("I like Java!");
    string str3;
    int pos;

    if(str1 < str2)
        str3 = str1 + str2;
    else
        str3 = str2 + str1;
    pos = str3.rfind("+");
    str3.replace(pos + 1, 7, " and");

    cout << str3 << endl;
    return 0;
}
```





### ► string 类迭代器

- `string::iterator`

- `begin()`, `end()`, `rbegin()`, `rend()`

### ► string 类泛型算法

- `copy`, `reverse`, `sort` 等





### ► string 类迭代器与泛型算法

```
#include <iostream>
#include <string>
#include <algorithm>

using namespace std;

void print(char c)
{
    cout << c;
}

int main()
{
    string s1 = "abcdefghijklmnopqrstuvwxyz";

    cout << s1 << endl;
    reverse(s1.begin(), s1.end());
    cout << s1 << endl;
    sort(s1.begin(), s1.end());
    for_each(s1.begin(), s1.end(), print);

    return 0;
}
```





## ► 操作

- ▶ `unsigned copy(char *s, unsigned pos=0) const;`
- ▶ `const char *c_str() const;`
- ▶ `const char *c_str() const;`

### 结尾符

string 类串不以 '\0' 结尾，使用 `c_str()` 函数时自动添加结尾符。





## ▶ 提取单词

```
#include <iostream>
#include <string>
#include <cstring>
using namespace std;

int main()
{
    string s1 = "The quick brown fox jumps over the lazy dog!";
    char *c_str = NULL;
    char *p;

    c_str = (char *)s1.c_str();
    p = strtok(c_str, " !");
    while(p != NULL)
    {
        cout << strlen(p) << '\t' << p << endl;
        p = strtok(NULL, " !");
    }
    return 0;
};
```

```
char* strtok(char* str, const char* delimiters);
```



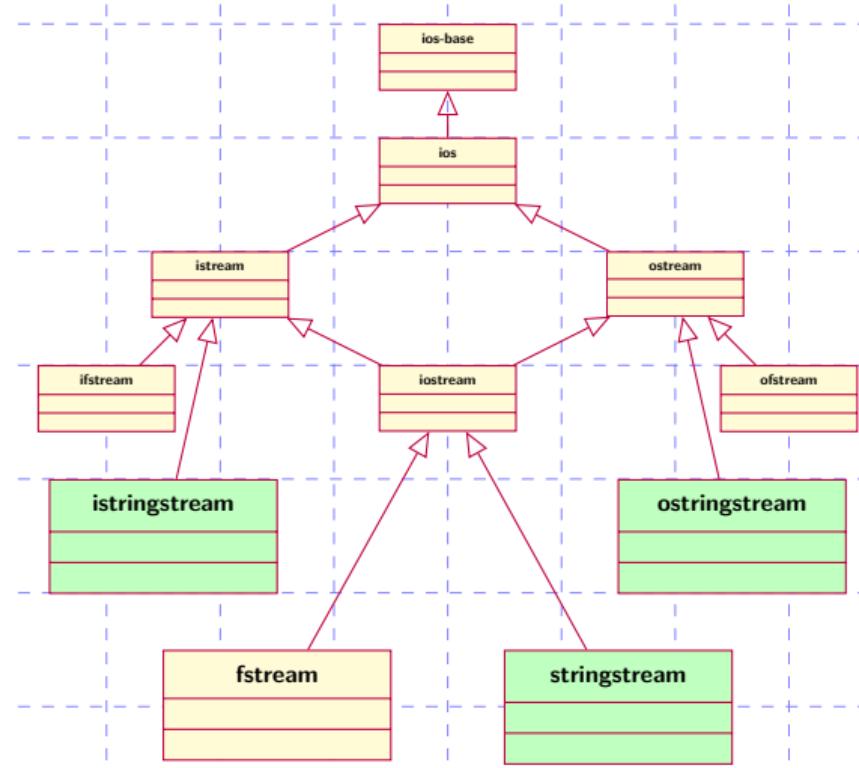


- ▶ `#include <sstream>`
- ▶ `istringstream`
- ▶ `ostringstream`
- ▶ `istream& getline ( istream& is, string& str,  
char delim );`





## ▶ 输入输出流类层次结构图





### ▶ 字符串到数字的转换 (=atoi)

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <sstream>
using namespace std;

int main()
{
    int year, month, day;
    string strDate = "2010 12 31";
    istringstream streamInfo(strDate);
    streamInfo >> year >> month >> day;
    cout << year << month << day;
    return 0;
};
```





### ► 数字到字符串的转换 (=itoa)

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <sstream>
using namespace std;

int main()
{
    int year = rand() % 2011;
    int month = rand() % 13;
    int day = rand() % 29;

    string strDate;
    ostringstream streamInfo;
    streamInfo << year << '/' << month << '/' << day;
    strDate = streamInfo.str();
    cout << strDate;
    return 0;
}
```



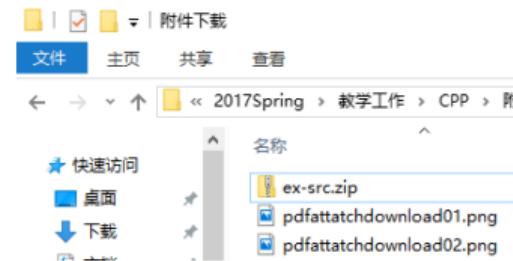
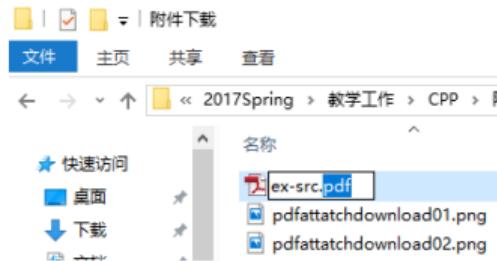
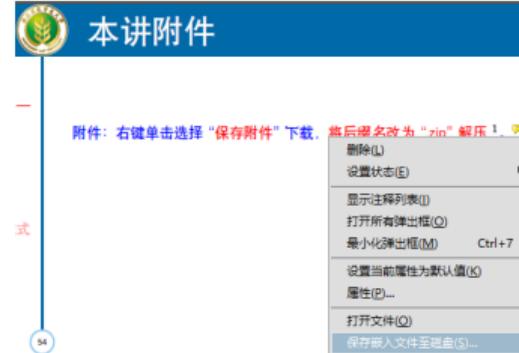
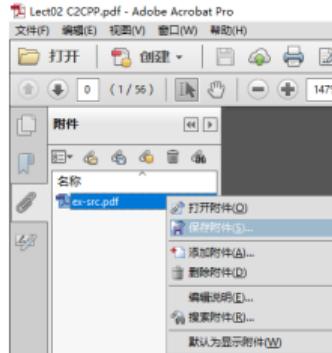


# 本讲附件

| 附件

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP  
Nine, G.

附件：右键单击该链接，选择“保存附件”下载，将后缀名改为“.zip”解压<sup>22 23</sup>。



22 请退出全屏模式后点击该链接。

23 以 Adobe Acrobat Reader 为例。



- ▶ 错误：语法错误，逻辑错误和运行错误。
- ▶ 常见异常错误：数组下标越界、运算溢出、除 0、动态分配内存失败和文件读写失败。
- ▶ 异常处理机制：**try-throw-catch**





- ▶ 能够抛出异常的语句被包围在 try 块中。
  
- ▶ try 块以关键字 try 开始，后面是花括号括起来的语句序列。try 块之后紧跟一组处理代码，称为 catch 子句。try 块将语句分成组，并将它们与处理这些语句可能抛出的异常的语句相关联。





- ▶ C++ 的异常处理代码是 catch 子句。当一个异常被 try 块中的语句抛出时，系统在 try 块后的 catch 子句列表中查找能够处理该异常的 catch 子句。





## ► 一元二次方程的根

```
struct Solution
{
    double x1, x2;
    Solution(double x1 = 0, double x2 = 0)
    {
        this->x1 = x1;
        this->x2 = x2;
    }
    friend ostream &operator<<(ostream &out, Solution s)
    {
        if(s.x1 == s.x2)
            out << "x1=x2=" << s.x1;
        else
            out << "x1=" << s.x1 << ", " << "x2=" << s.x2;
        return out;
    }
};

Solution FindRoots(double a, double b, double c)
{
    double x1, x2;

    if(abs(a) < 1.0e-8)
    {
        if(abs(b) < 1.0e-8)
            throw a;
        else
            return Solution(-c / b, -c / b);
    }
    if((b * b - 4 * a * c) < 0) throw (b * b - 4 * a * c));
    x1 = (-b + sqrt(b * b - 4 * a * c)) / (2 * a);
    x2 = (-b - sqrt(b * b - 4 * a * c)) / (2 * a);
    return Solution(x1, x2);
}
```





## ► 一元二次方程的根

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    try
    {
        cout << FindRoots(0, 2, 2) << endl;
        cout << FindRoots(1, 2, 1) << endl;
        cout << FindRoots(1, 4, 1) << endl;
        cout << FindRoots(0, 0, 1) << endl;
        cout << FindRoots(-1, 9, 3) << endl;
    }

    catch(double x)
    {
        if(abs(x) < 1.0e-8) cout << "Dividing zero!" << endl;
        if(x < 0) cout << "The roots are complex!" << endl;
    }

    cout << "Continue..." << endl;
    return 0;
}
```





## ▶ 文件异常

```
#include <iostream>
#include <fstream>
using namespace std;

int main ()
{
    ifstream file;
    file.exceptions ( ifstream::failbit | ifstream::badbit );
    try
    {
        file.open ("test.txt");
        while (!file.eof()) file.get();
    }
    catch (ifstream::failure e)
    {
        cout << "Exception opening/reading file";
    }

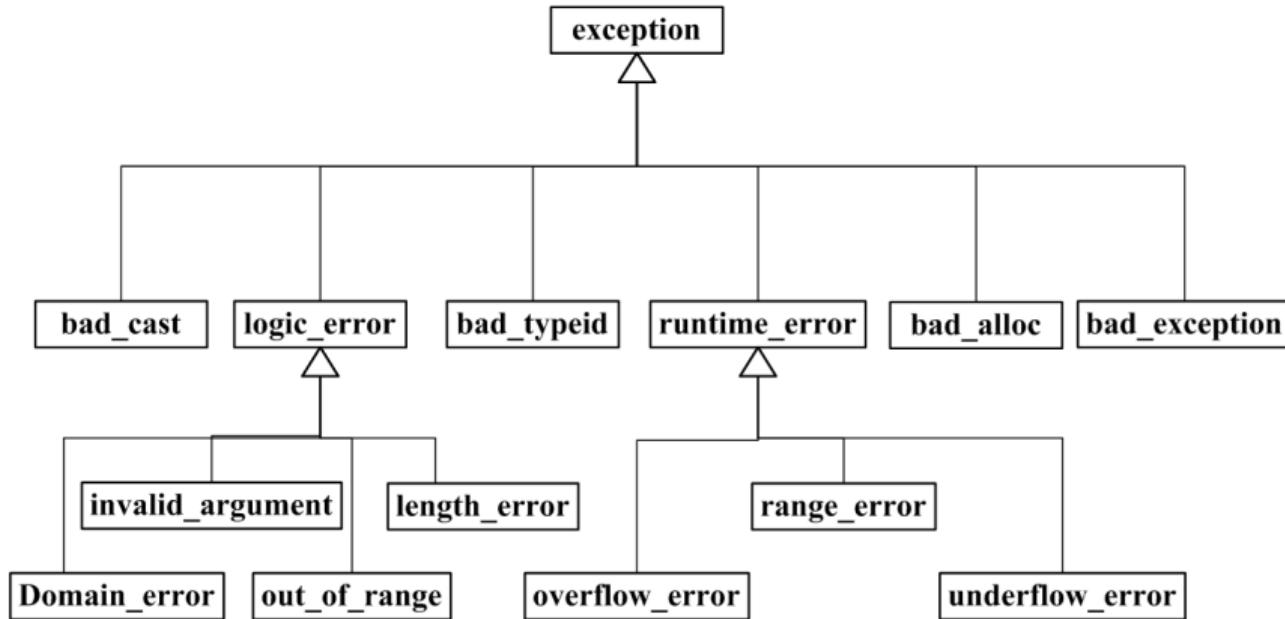
    file.close();

    return 0;
}
```





### ▶ 系统预定义异常





- ▶ 集中处理容易出问题的代码。
  
- ▶ 异常只是错误处理技术的一种，程序中还应该使用其他错误处理技术，如断言、返回错误代码等。
  
- ▶ 异常使程序的复杂性增加，可以参考一些指导原则，慎用异常机制。



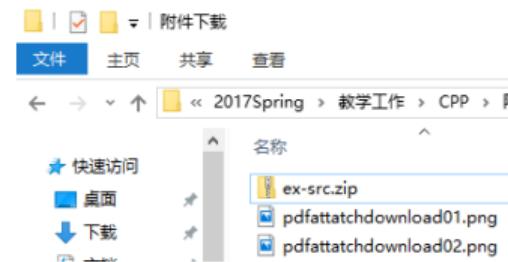
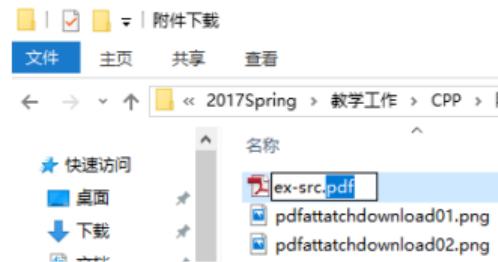
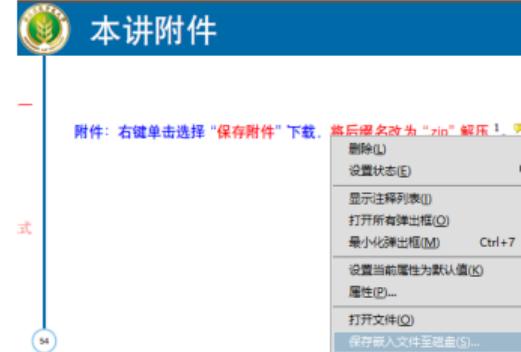
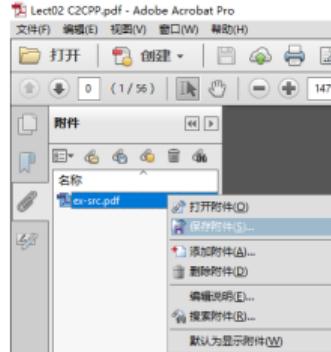


# 本讲附件

| 附件

OBJECT  
ORIENTED  
PROGRAMMING—  
OOP  
Nine, G.

附件：右键单击该链接，选择“保存附件”下载，将后缀名改为“.zip”解压<sup>24 25</sup>。



<sup>24</sup>请退出全屏模式后点击该链接。

<sup>25</sup>以 Adobe Acrobat Reader 为例。

本讲结束，谢谢！  
欢迎多提宝贵意见和建议