数据结构

第二、三、四讲 C++重点回顾

胡浩栋

信息管理与工程学院

2018 - 2019 第一学期

课堂内容

- 类class和对象object

- 重载（Overload）
  - 函数重载 (function overload)
  - 运算符重载 (operator overload)

- 对象Object关系
- 类继承Ineritance
- 动态绑定和虚函数 Virtual function

面向对象编程

- 传统编程
  - 数据（变量)
  - 函数（方法、行为）
  - **Study**(you， math)， **Study**(AI， you)

- 面向对象编程
  - Object Oriented Programming（OOP）
  - 建立对象（包括数据和行为）
    - 独立性（既有记忆，又有行为）
  - 可读性增强，更符合逻辑
    - you.**Study**(programming) （主谓宾）

- 推荐www.learncpp.com
- Book： Thinking in C++

类和对象

class and object

Struct

- 可以让你把不同数据放到一个自定义数据类型里
- 在C语言里， 函数不能放进struct
- C++语言里，函数允许放入struct
- 一般来说，struct只放数据和简单的函数

```
struct Lunchable
{
    string meat;
    string dessert;
    int numCrackers;
    bool hasCheese;

    int CountCalories();
};
```

C++类（class）

- 在面向对象编程里，
  - 定义对象的数据
  - 定义对象的行为
- class是C++里用来自定义数据类型
  - 即包括了数据，
  - 也包括了方法

类和结构体

下面两个数据类型完全等价
区别是类里加了关键词public

- 类class和结构体struct非常类似
  - 结构体看作是只包括数据的类
  - 都是类型声明，都不占内存
  - 只有定义了变量后，才分配内存

```
struct Lunchable
{
    string meat;
    string dessert;
    int numCrackers;
    bool hasCheese;
};
```

⇔

```
class Lunchable
{
public:
    string meat;
    string dessert;
    int numCrackers;
    bool hasCheese;
};
```

```
Lunchable box{"fish", "cake", 3}; // default 0 for hasCheese
```

- 类看作类型概念，定义的“变量”是对类的实例化，也叫做对象object（既有数据又有方法）

- 类提供更多的功能
  - 继承，封装，抽象，多态

类的方法

- 成员函数（member function）

```
class Lunchable
{
public:
    string m_meat;
    string m_dessert;
    int m_numCrackers;
    bool m_hasCheese;

    void PrintLunchItem()
    {
        cout << "Lunch: " << m_meat << ", " << m_dessert << endl;
    }
};
```

```
int main()
{
    Lunchable box{"fish", "cake", 3};
    box.m_hasCheese = true;

    box.PrintLunchItem();
    return 0;
}
```

```
Lunch: fish, cake
```

- 成员函数调用要通过object访问
- 成员函数可以使用类里的成员数据
- 注意每个对象保存自己的数据，类比int变量。
- 成员数据用“m_”作为前缀，来区分形参，局部变量

类的例子

- std::string
- std::array
- std:: vector
- std::cout, std::cin

- 比如std::string str = “Hello” ;是对string类的一个实例化
- 可以通过str这个对象，调用它的成员函数str.length()

类成员（数据和方法）访问权限

- public：不是成员也可以访问
- private：只有成员可以访问
- protected：类似private权限，后面讲类继承时涉及
- struct默认的是public；class默认是private

```
struct Lunchable
{
    string meat;
    string dessert;
    int numCrackers;
    bool hasCheese;
};

int main()
{
    Lunchable box {"chicken", "icecream", 2, false};
    box.numCrackers = 5;  // OK to access in main()
    box.hasCheese = true; // OK to access in main()
}
```

```
class Lunchable
{
    string meat;
    string dessert;
    int numCrackers;
    bool hasCheese;
};

int main()
{
    Lunchable box;
    box.numCrackers = 5;  // Error to access in main()
    box.hasCheese = true; // Error to access in main()
}
```

```
error: 'int Lunchable::numCrackers' is private
```

类成员访问权限

- 同一类里可以设置不同的访问权限
  - 所有的成员变量都应该设为private
  - 能在外部访问的成员函数设为public，通常称为接口interface
  - 好处是只要接口方法不变，内部数据怎么改变，不会影响用户使用

```
int main()
{
    Lunchable box;
    box.setMeat("chicken");
    box.PrintLunchItem();
}
```

虽然m_meat是private，通过public函数可以设置值

```
class Lunchable
{
public:
    void setMeat(string meat)
    {
        m_meat = meat;
    }
    void PrintLunchItem()
    {
        cout << "Lunch: " << m_meat << ", " << m_dessert << endl;
    }
private:
    string m_meat;
    string m_dessert;
    int m_numCrackers;
    bool m_hasCheese;
};
```

类成员访问权限

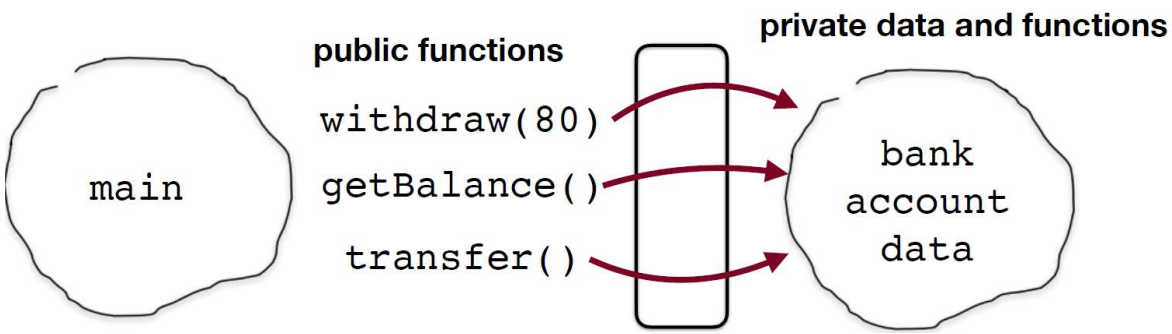
- 访问权限是作用在类自身上，不是实例化的对象身上
  - 下面CopyFrom不但可以读取m_meat，而且可以读取其它对象的m_meat

```
int main()
{
    Lunchable box;
    box.setMeat("chicken");

    Lunchable bowl;
    bowl.CopyFrom(box);
    bowl.PrintLunchItem();
}
```

```
class Lunchable
{
public:
    void setMeat(string meat)
    {
        m_meat = meat;
    }
    void PrintLunchItem()
    {
        cout << "Lunch: " << m_meat << ", " << m_dessert << endl;
    }

    void CopyFrom(const Lunchable &other)
    {
        m_meat = other.m_meat;
        m_dessert = other.m_dessert;
        m_numCrackers = other.m_numCrackers;
        m_hasCheese = other.m_hasCheese;
    }
private:
    string m_meat;
    string m_dessert;
    int m_numCrackers;
    bool m_hasCheese;
};
```

类：封装

- 封装就是把内部的数据和外部的接口分割开来
- 用户不关心内部实现。比如用户只关心开收音机，不关心电台信号怎么传递
- 一般来说成员变量都是private的，成员函数可以public
- 和外部的接口interface越少越好（不要把setBalance给用户)




```
int main()
{
    BankAccount checking;
    checking.withdraw(80);
    cout << checking.getBalance() << endl;

    // Do not access balance by client,
    // even by public method
    checking.balance = 812.345;
    checking.setBalance(812.345);
}
```

类：封装

- 封装使得更容易使用，比如std::string
  - 不用再担心是不是有\0结尾
  - 不用担心退化成指针（函数形参传值）后，长度信息丢失
- 封装可以防止内部数据被误用，比如

```
int main()
{
    IntArray arr;
    arr.m_array[10] = 1; // Error: Out of boundary
}
```

```
class IntArray
{
public:
    int m_array[10];
};
```

- 把数据私有化，只通过public函数

```
class IntArray
{
public:
    void setValue(int index, int value)
    {
        if (index < 0 || index >= 10)
        {
            return;
        }
        m_array[index] = value;
    }
private:
    int m_array[10];
};
```

类：封装

- 封装使得修改更容易
  - 如果把成员数据公开，以后修改就得把所有访问的地方都修改
  - 如果有很多人在使用这个类（std类），就要修改的代价更大

```
class Something
{
private:
    int m_value1;
    int m_value2;
    int m_value3;

public:
    void setValue1(int value) { m_value1 = value; }
    int getValue1() { return m_value1; }
};

int main()
{
    Something something;
    something.setValue1(5);
    std::cout << something.getValue1() << '\n';
}
```

```
class Something
{
private:
    int m_value[3];

public:
    void setValue1(int value) { m_value[0] = value; }
    int getValue1() { return m_value[0]; }
};
```

类：封装

- 封装使得debug更容易
  - 如果把成员数据公开，debug时候需要在所有访问的地方设断点
  - 如果封装，只要在成员函数内设置断点：setValue1（）
- 有时候类里的数据需要被频繁使用，逻辑上用户直接读写是合理的
  - 读取模式函数Getter，返回形式一般是传值，const reference
  - 设置模式函数Setter

```
class Date
{
private:
    int m_month;
    int m_day;
    int m_year;

public:
    int getMonth() { return m_month; } // getter for month
    void setMonth(int month) { m_month = month; } // setter for month

    int getDay() { return m_day; } // getter for day
    void setDay(int day) { m_day = day; } // setter for day

    int getYear() { return m_year; } // getter for year
    void setYear(int year) { m_year = year; } // setter for year
};
```

类的初始化

- 如果所有的成员变量都是public（类似struct）

```
int main()
{
    Point pt{3, 5};
}
```

```
class Point
{
public:
    int m_x;
    int m_y;
};
```

- 如果封装数据（私有成员变量不能在类外面设置）

```
int main()
{
    Point pt{3, 5}; // Error
}
```

```
class Point
{
private:
    int m_x;
    int m_y;
};
```

类的初始化

- 构造函数（constrcutor）：特殊的成员函数
  - 当定义一个类的对象（实例化），构造函数自动被调用
  - 一般被用来初始化成员变量，或者类使用前的准备工作（打开文件等）
  - 函数名和类名一样
  - 没有返回值（void都不能加）
  - 不要自己去调用构造函数
- 这个函数对类非常重要，即使你不写，编译器自动帮你添加一个缺省的空的构造函数
  - 没有返回值，没有参数，函数名和类名一样的函数

缺省的构造函数

- 没有返回值，没有参数
- 声明类对象时候自动调用缺省构造函数

```
class Fraction
{
public:
    Fraction() // default constructor
    {
        m_numerator = 0;
        m_denominator = 1;
    }

    int getNumerator() { return m_numerator; }
    int getDenominator() { return m_denominator; }
    double getValue() { return static_cast<double>(m_numerator) / m_denominator; }
private:
    int m_numerator;
    int m_denominator;
};

int main()
{
    Fraction frac; // Since no arguments, calls Fraction() default constructor
    std::cout << frac.getNumerator() << "/" << frac.getDenominator() << '\n';

    return 0;
}
```

自定义构造函数

- 在构造函数里加入参数
- 和初始化内置数据类型（int）变量类似
- 一旦有了自定义构造函数，编译器不会再自动添加缺省的构造函数
- C++11后推荐用{}

```
int main()
{
    int x(5);
    Fraction fiveThirds(5, 3);

    int y{4};
    Fraction fourThirds{4, 3};

    Fraction six{6};

    return 0;
}
```

```
class Fraction
{
public:
    Fraction() // default constructor
    {
        m_numerator = 0;
        m_denominator = 1;
    }

    // Constructor with two parameters, one parameter having a default value
    Fraction(int numerator, int denominator = 1)
    {
        assert(denominator != 0);
        m_numerator = numerator;
        m_denominator = denominator;
    }

    int getNumerator() { return m_numerator; }
    int getDenominator() { return m_denominator; }
    double getValue() { return static_cast<double>(m_numerator) / m_denominator; }
private:
    int m_numerator;
    int m_denominator;
};
```

赋值初始化

- 用赋值的方式初始化：
  - 类似于内置数据类型的复制初始化
  - six初始化时候，先建立一个临时的类，然后复制给six
  - seven初始化时候，先把7转换到Fraction，调用Fraction(int, int =1)，再复制给seven

```
int main()
{
    int x = 5;

    Fraction six = Fraction(6);
    Fraction seven = 7;

    return 0;
}
```

- 避免对自定义数据类型（包括类）进行赋值初始化

隐含的缺省构造函数

- 如果没有定义任何构造函数，编译器帮你生成一个

```
int main()
{
    // Call default constructor
    Date today;
    return 0;
}
```

```
class Date
{
public:
    // Implicit default constructor
    //Date(){}
private:
    int m_year = 1900;
    int m_month = 1;
    int m_day = 1;
};
```

- 如果定义一个带参数的构造函数，缺省构造函数不会自动生成
- 构造函数里设置优先于成员变量初始值

```
int main()
{
    Date date; // Error: Can't instantiate object
    Date today(2017); // today is initialized to 2017/1/1
    return 0;
}
```

```
class Date
{
public:
    // normal non-default constructor
    Date(int year)
    {
        m_year = year;
    }
private:
    int m_year = 1900;
    int m_month = 1;
    int m_day = 1;
};
```

类嵌套

- 在类里可以包含别的类

```
int main()
{
    B b;
    return 0;
}
```

```
Inside A
Inside B
```

```
class A
{
public:
    A() { cout << "Inside A\n"; }
};
```

```
class B
{
public:
    B() { cout << "Inside B\n"; }
private:
    // B contains A as a member variable
    A m_a;
};
```

- 在定义b时，会调用B的构造函数，之前先进行成员m_a初始化
- 这个合理，因为你可能在B的构造函数里需要用到m_a

成员变量的初始化顺序

- 按类里面声明的顺序，相当于在main函数按这个顺序声明

```
int main()
{
    int m_value1;
    double m_value2;
    char m_value3;

    m_value1 = 1;
    m_value2 = 2.2;
    m_value3 = 'c';
}
```

```
class Something
{
public:
    Something()
    {
        // These are all assignments, not initializations
        m_value1 = 1;
        m_value2 = 2.2;
        m_value3 = 'c';
    }
private:
    int m_value1;
    double m_value2;
    char m_value3;
};
```

- 问题是有些变量声明时候必须初始化，比如const， reference

```
int main()
{
    //Error: const vars must be initialized with a value
    const int value;
}
```

```
class Something
{
private:
    const int m_value; // Error

public:
    Something()
    {
        //Error: const vars cannot be assigned to
        m_value = 1;
    }
};
```

成员变量的初始化列表

- C++允许在构造函数里对成员变量使用初始化列表

```cpp
class Something
{
private:
    const int m_value;
public:
    Something() : m_value(5)
    {
    }
};
```

```cpp
class Something
{
private:
    const int m_value;
public:
    Something() : m_value{5}
    {
    }
};
```

- 成员变量是数组时的初始化（C++11后可以初始化非零）

```cpp
class Something
{
private:
    const int m_array[5];
public:
    // Initialize to zero
    Something() : m_array{}
    {
    }
};
```

```cpp
class Something
{
private:
    const int m_array[5];
public:
    // C++11: Initialize to non-zero
    Something() : m_array{1, 2, 3, 4, 5}
    {
    }
};
```

成员类变量的初始化

- 如果类B里包含另外一个类A的对象，而且A的构造函数不是缺省

```
class A
{
public:
    A(int x) { cout << "Inside A\n"; }
};

class B
{
public:
    // Error: cannot instantiate m_a
    B() { cout << "Inside B\n"; }
private:
    // B contains A as a member variable
    A m_a;
};
```

```
class A
{
public:
    A(int x) { cout << "Inside A\n"; }
};

class B
{
public:
    // OK to initialize m_a by calling A(int x)
    B() : m_a(0) { cout << "Inside B\n"; }
private:
    // B contains A as a member variable
    A m_a;
};
```

初始化注意点

- 初始化顺序是按变量声明，不是按初始化列表

```
int main()
{
    C c;
}
```

```
Inside A
Inside B
Inside C
```

```
class A
{
public:
    A(int x) { cout << "Inside A\n"; }
};

class B
{
public:
    B(int x) { cout << "Inside B\n"; }
};

class C
{
public:
    C() : m_b(1), m_a(0) { cout << "Inside C\n"; }
private:
    A m_a;
    B m_b;
};
```

- 最好不要在初始化时有依赖关系，容易犯错
- 初始化列表中的顺序最好和变量声明顺序一致

非static成员变量的初始化

- C++11后，可以对普通成员变量直接初始化

```
class Point
{
public:
    Point() {}
    void Print() { cout << m_x << ", " << m_y; }
private:
    int m_x = 1;
    int m_y = 1;
};
```

- 如果构造函数也初始化，那构造函数的优先

```
class Point
{
public:
    Point(int x, int y) : m_x(x), m_y(y) {}
    void Print() { cout << m_x << ", " << m_y; }
private:
    int m_x = 1;
    int m_y = 1;
};
```

析构函数destructor

- 析构函数（destructor）：特殊的成员函数，和构造函数成对
  - 当一个类的对象out of scope，在销毁前，析构函数自动被调用
  - 用来释放类占用的资源（动态内存）
  - 函数名和类名一样，前面要加一个“~”
  - 没有返回值（void都不能加），没有参数
  - 所以析构函数是唯一的（不能重载）
  - 不要调用析构函数，但析构函数可以调用别的成员函数
- 这个函数对类非常重要，即使你不写，编译器自动帮你添加一个空的析构函数
- 析构函数里不要加assert，或者说不要出现exception

例子

- 类里申请了动态内存，需要在析构函数里释放

```
class IntArray
{
private:
    int *m_array;
    int m_length;

public:
    IntArray(int length) // constructor
    {
        assert(length > 0);
        m_array = new int[length];
        m_length = length;
    }

    ~IntArray() // destructor
    {
        // Dynamically delete the array we allocated earlier
        delete[] m_array ;
    }
    int getLength() { return m_length; }
};


int main()
{
    IntArray arr(10); // allocate 10 integers
    return 0;
}//arr is destroyed and destructor ~IntArray() is called here
```

RAII

- RAII (Resource Acquisition Is Initialization)是一种编程技巧
  - 用来防止资源泄漏，能自动释放资源
  - 比如std::vector，动态数组，不需要担心内存泄漏
- C++利用了构造函数和析构函数被自动调用的特性，使得像动态内存这样的资源可以自动被释放掉
- 一般来说占用资源的对象不应该被动态分配，因为
  - 如果没有delete对象，析构函数不会被调用
  - 占用的资源还是不会释放

```
int main()
{
    IntArray *pArr = new IntArray{10};
    return 0;
}// memory leaked since ~IntArray() is never called.
```

```
int main()
{
    IntArray *pArr = new IntArray{10};
    delete pArr;
    return 0;
}// OK now.
```

- C++11后，增加了smart pointer： std::shared_ptr<IntArray *>
  - 其实是一个管理指针的类，轻量级的
  - 使得指向的动态内存自动被释放

类的内存大小

- 类的成员函数不占对象空间

```
int main()
{
    cout << "Size of Point is: " << sizeof(Point) << endl;
    return 0;
}
```

```
Size of Point is: 8
```

```
class Point
{
public:
    Point(){}
    void setX(int x) {m_x = x;}
    void Print() { cout << m_x << ", " << m_y << endl; }
private:
    int m_x = 1;
    int m_y = 1;
};
```

- 成员函数和普通函数一样存在代码区，不在类对象里
- 成员函数调用时和动态对象关联（和普通函数调用方式不同？）

```
int main()
{
    Point a, b;
    b.setX(11);
    a.Print();
    b.Print();
}
```

```
1, 1
11, 1
```

“this” 指针

- 成员函数只有一份，怎么知道自己被哪个对象调用？
- C++在调用成员函数时候，在参数里隐含了const “this” 指针

```
void setX(int x) {m_x = x;}
b.setX(11);
```

→

```
void setX(Point * const this, int x) { this->m_x = x;};
b.setX(&b, 11);
```

- 在成员函数里可以直接使用this

```
class Something
{
public:
    Something(int data)
    {
        this->data = data;
    }
private:
    int data;
};
```

推荐加上“m_”避免混淆

“this” 指针

- std::cout的链式输出
  - std::cout << “xxx” << “yyyy”;
  - <<是运算符，可以看作函数（下一章）
  - （ std::cout << “xxx”）运算符（函数）返回值还是std::cout
- 类里可以通过this指针实现类似的链式调用

```
int main()
{
    Calc x;
    x.Add(10).Add(9);
}
```

```
class Calc
{
public:
    Calc() { m_value = 0; }
    Calc& Add(int value)
    {
        m_value += value;
        return *this;
    }
private:
    int m_value;
};
```

成员函数定义（实现）

- 成员函数可以在类的外面定义（不属于类对象）
  - 类似普通函数，需要加上类scope

```
class Point
{
public:
    void setX(int x) {m_x = x;}
    void Print();
private:
    int m_x = 1;
    int m_y = 1;
};

void Point::Print()
{
    cout << m_x << ", " << m_y << endl;
}
```

- 很简单的成员函数，比如Setter，Getter，也可定义在类里面
  - 意味着inline

类接口interface

类实现后很方便被别的地方调用，比如引用std::vector, std::string

- .h头文件
  - 类声明Declaration，包括成员变量和成员函数
  - 可以在别的地方（函数，类）引用（include），
- .cpp代码文件
  - 成员函数实现
  - 定义Definition
- 一个cpp和h对应一个类，文件名和类名一样
  - 要不然会出现循环依赖的问题
- Std标准模板库，就是把成员函数实现预编译成库，头文件被引用

头文件

- C++不允许重复定义，需要避免头问题重复引用
  - #pragma once 使用方便，在windows系统用比较好
  - #ifndef对于跨平台和兼容性比较好

```cpp
// classname.h
#pragma once

class ClassName {
    // class definition
};
```

```cpp
// classname.h
#ifndef _CLASSNAME_H
#define _CLASSNAME_H

class ClassName {
    // class definition
};

#endif
```

头文件和代码文件

```cpp
// point.h
#ifndef _POINT_H
#define _POINT_H
class Point
{
public:
    Point(int x, int y);
    void setX(int x) {m_x = x;}
    void Print();
private:
    int m_x;
    int m_y;
};
#endif // _POINT_H
```

```cpp
// point.cpp
#include "point.h"
Point::Point(int x, int y)
{
    m_x = x;
    m_y = y;
}
void Point::Print()
{
    cout << m_x << ", " << m_y << endl;
}
```

```cpp
#include "point.h"
int main()
{
    Point a{1, 2};
    a.Print();
}
```

小结

- 如果类只用一次，类的声明和定义可以都在一个cpp里
  - 比如作业（不推荐）
- 如果类要在别的cpp调用，那么把
  - 类声明放在头文件
  - 成员函数的实现放在相应的cpp代码文件
- 一些简单的成员函数，比如简单的构造/析构函数，Getter/Setter，可以定义在类里面（inline）
- 一般的成员函数，都应该在cpp文件里定义
  - 除了template函数，必须定义在头文件

类和const

- 普通常量变量，必须在声明的同一行初始化
  - 赋值初始化 const int a = 1;
  - 直接初始化 const int b(2);
  - C++11初始化const int c{3};
- 类常量对象（自动初始化）：不但写不行，getValue也不行

```
int main()
{
    const A a;
    a.m_value = 3;
    a.setValue(3);
    cout << a.getValue();
}
```

```
class A
{
public:
    int m_value;

    A(): m_value(0) { }

    void setValue(int value) { m_value = value; }
    int getValue() { return m_value ; }
};
```

```
error: assignment of member 'A::m_value' in read-only object
error: passing 'const A' as 'this' argument of 'void A::setValue(int)' disc...
error: passing 'const A' as 'this' argument of 'int A::getValue()' discards...
```

类和const

- 如何允许读取操作
  - 在getValue后面加上const

```
int main()
{
    const A a;
    cout << a.getValue();
}
```

```
class A
{
public:
    int m_value;

    A(): m_value(0) { }

    void setValue(int value) { m_value = value; }
    int getValue() const { return m_value ; }
};
```

  - 不过加了const后，在这个函数里不能修改成员变量

```
In member function 'void A::setValue(int) const':
error: assignment of member 'A::m_value' in read-only object
```

```
class A
{
public:
    int m_value;

    A(): m_value(0) { }

    void setValue(int value) const { m_value = value; }
    int getValue() const { return m_value ; }
};
```

类和const

- 类作为形参传递给函数
  - 常常是const reference
  - 非常量对象会被强制转换到常量对象

```
void Print(const A &a)
{
    cout << a.getValue();
}

int main()
{
    A a;
    Print(a);
}
```

```
class A
{
public:
    int m_value;

    A(): m_value(0) { }

    void setValue(int value) { m_value = value; }
    int getValue() { return m_value ; }
};
```

```
error: passing 'const A' as 'this' argument of 'int A::getValue()' discards.
```

  - 解决的办法就是定义int getValue const ();
- 所以如果成员函数不修改成员变量，最好在函数名后加上const

静态成员变量

- 函数里的静态变量不会自动销毁，每次调用都是同一个
- 类里也可以定义静态成员变量
  - 不同对象的普通成员变量是私有的，即分配不同内存
  - 不同对象公用静态成员变量，只有全局静态变量区有一个
  - 静态成员变量不能在类声明处初始化，除非是static const
- 静态成员变量不属于类对象，像全局变量，要在类外面定义

```
int main()
{
    A a;
    cout << sizeof(A) << endl;
}
```

```
4
```

```
class A
{
public:
    A(){ }
private:
    static int m_value;
    int m_size;
};

int A::m_value = 1;
```

例子

- 用静态成员变量自动产生类对象的ID

```
int main()
{
    Something first;
    Something second;
    Something third;

    cout << first.getID() << endl;
    cout << second.getID() << endl;
    cout << third.getID() << endl;
    return 0;
}
```

```
1
2
3
```

```
class Something
{
public:
    // grab the next value from the id generator
    Something() { m_id = s_idGenerator++; }
    int getID() const { return m_id; }
private:
    static int s_idGenerator;
    int m_id;
};

// Definition of static member variable
// OK even if it is private
int Something::s_idGenerator = 1;
```

静态成员函数

- 静态成员函数更不属于类对象，和普通成员函数相比
  - 没有隐含this指针
  - 不能读取非静态的成员变量
  - 没有类对象，也可以调用，需要加上类scope
  - C++不支持static构造函数

```
class Something
{
public:
    // Constructor cannot be static
    Something() { m_id = getNextID(); }
    static int getNextID();
private:
    static int s_nextID;
    int m_id;
};
```

```
int Something::s_nextID = 1;

/*static*/ int Something::getNextID()
{
    // Error: cannot access non-static member variable
    // m_id = s_nextID;
    return s_nextID; // OK
}

int main()
{
    cout << Something::getNextID() << endl;
    return 0;
}
```

- 谨慎使用静态成员变量/函数，只真正需要才使用

友元(friend)函数

- 类的数据变量一般是private，但有的时候别的函数需要读取内部数据，比如void Display(const A &obj)
- 在类的声明中可使用关键字friend定义友元函数，
  - 友元函数可以访问类的私有成员和保护成员
  - 友元函数实际上并不是这个类的成员函数，
    - 可以是一个常规函数，
    - 也可以是另一个类的成员函数。
  - 友元函数可以在private或者public，都一样

```
class Value
{
private:
    int m_value;
public:
    Value(int value) { m_value = value; }
    friend void Display(const Value &obj);
};

void Display(const Value &obj)
{
    cout << obj.m_value;
}
```

不需要加类的scope，
即Value::

友元(friend)类

- 用friend也可以定义友元类
  - 在Rectangle里用Square做为参数时，Square还没定义
  - 需要先声明一下存在class Square
  - 在Square里把Rectangle定义为友元类
  - Square不是Rectangle友元类
  - 尽量不使用友元类，会破坏类的封装

```
// forward declare for Square
class Square;
class Rectangle
{
public:
    Rectangle(): m_width(3), m_height(4) {}
    void convert(const Square &sqr);
private:
    int m_width;
    int m_height;
};

class Square
{
  public:
    Square (int a) : m_side(a) {}
  private:
    int m_side;

    friend class Rectangle;
};


void Rectangle::convert(const Square &sqr)
{
    m_width = sqr.m_side;
    m_height = sqr.m_side;
}
```

友元(friend)类函数

- 可以设置另一个类的成员函数为友元
- 这种次序才能编译

- 为了避免依赖定义，一个类一个头文件/代码文件

```
// forward declare for Square
class Square;
class Rectangle
{
public:
    Rectangle(): m_width(3), m_height(4) {}
    // Must declare with forward declare of Square
    void convert(const Square &sqr);
private:
    int m_width;
    int m_height;
};

class Square
{
  public:
    Square (int a) : m_side(a) {}
  private:
    int m_side;

    // Must declare after full Rectangle
    friend void Rectangle::convert(const Square &sqr);
};

// Must define after Square
void Rectangle::convert(const Square &sqr)
{
    m_width = sqr.m_side;
    m_height = sqr.m_side;
}
```

Q&A

Thanks!

课堂内容

- 类class和对象object

- 重载（Overload）
  - 函数重载 (function overload)
  - 运算符重载 (operator overload)

- 对象Object关系
- 类继承Ineritance
- 动态绑定和虚函数 Virtual function

重载（Overload）

- 函数重载 (function overload)
- 运算符重载 (operator overload)

函数重载

函数重载

- 同一个函数名，参数不同
- 编译器根据参数来识别调用

- 注意返回类型不能区别

- 注意同一个函数实现应该一致，
  不要一会儿a+b，一会儿a*b

问题：Calc（10， 20），调用哪个？

```
double Calc(int a, double b)
{
    return a + b;
}

double Calc(double a, double b)
{
    return a * b;
}

int main()
{
    // int, double
    cout << Calc(10, 20.1) << endl;
    // double, double
    cout << Calc(1.5, 2.5) << endl;
}
```

函数重载调用规则

1. C++首先找完全匹配
2. 兼容内置数据类型匹配
   - Char, unsigned char, short, unsigned short ⇒ int
   - float ⇒ double
   - enum ⇒ int
3. 别的内置数据类型转换
4. 自定义数据类型转换（未讲到）
5. 前面步骤不能转换（歧义），编译出错

```
void Print(unsigned int x)
{
    cout << x;
}

void Print(float x)
{
    cout << x;
}
int main()
{
    Print('a'); // Error: Ambiguity
    Print(0);   // Error: Ambiguity
    Print(3.1415);  //Error: Ambiguity
}
```

```
int main()
{
    Print(3.1415f);
    Print(static_cast<unsigned int>('a'));
    Print(static_cast<unsigned int>(0));
    Print(static_cast<float>(3.1415));
}
```

函数重载调用歧义

多个参数的函数重载

- 类型转换时发现有一种匹配更好的
- 如果有两种匹配需要同样的转换次数，意味着歧义

```
error: call of overloaded 'Calc(double, int)' is ambiguous
```

不过歧义问题不大，很容易解决

```
double Calc(int a, int b)
{
    return a+b;
}

double Calc(double a, double b)
{
    return a+b;
}

int main()
{
    // int, int
    cout << Calc(10, 20) << endl;
    // Ambiguity: int, double
    cout << Calc(10, 20.0) << endl;
    // Ambiguity: double, int
    cout << Calc(1.0, 20) << endl;
    // double, double
    cout << Calc(1.0, 1.5) << endl;

    return 0;
}
```

模板(template)简介

- 目的：实现软件重用

- 定义

　　适合多种数据类型的类定义或算法，在特定环境下通过简单地代换，变成针对具体某种数据类型的类定义或算法。

格式

- 定义template

```
template <class/typename identifier> function_declaration;
```

```
template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}
```

- 使用template

```
function_name <datatype> (parameters);
```

```
int x,y;
GetMax <int> (x,y);
```

Class template

```
// class templates
#include <iostream>
using namespace std;

template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}
```

```
mypair<int> myobject (115, 36);
```

```
mypair<double> myfloats (3.0, 2.18);
```

```
template <class T>
T mypair<T>::getmax ()
```

函数重载和模板

1. 函数重载目的是让代码使用简单
2. 模板同样可以达成这个目的
3. 如果重载的类型很多的话，模板更方便

```
template <class T>
T Calc(T a, T b)
{
    return a+b;
}

double Calc(double a, double b)
{
    return a+b;
}
int main()
{
    // int, int, call template
    cout << Calc(10, 20) << endl;
    // int, double, convert int to double
    cout << Calc(10, 2.5) << endl;
    // double, int, force to call template
    cout << Calc<double>(1.0, 20) << endl;
    // double, double, match normal function
    cout << Calc(1.5, 2.5) << endl;
}
```

注意这里没有歧义，因为内置类型转换匹配 优先于 自定义类型匹配

函数重载和模板

- 完全使用模板

```
template <class SomeType>
SomeType Calc(SomeType a, SomeType b)
{
    return -a-b;
}

int main()
{
    // int, int
    cout << Calc(10, 20) << endl;
    // int, double
    cout << Calc<double>(10, 20.0) << endl;
    // double, int
    cout << Calc<double>(1.0, 20) << endl;
    // double, double
    cout << Calc(1.0, 1.5) << endl;

    return 0;
}
```

```
template <class SomeType>
SomeType Calc(SomeType a, double b)
{
    return -a-b;
}

int main()
{
    // int, int
    cout << Calc(10, 20) << endl;
    // int, double
    cout << Calc(10, 20.0) << endl;
    // double, int
    cout << Calc(1.0, 20) << endl;
    // double, double
    cout << Calc(1.0, 1.5) << endl;

    return 0;
}
```

必须使用Calc<double>()，不能自动得出SomeType

运算符重载

运算符重载

- 运算符其实也是函数
  - datatype operator **sign** (parameters) { /*... body ...*/ }
  - 内置类型int：　　a + b ⇒ int operator**+** (a, b)

- C++看到运算符时，
  - 所有操作数都是内置类型，调用缺省操作，或者编译错误
  - 操作数至少一个是自定义数据类型（struct，class），
    - 看有没有运算符重载定义，
    - 然后可不可以类型转换，
    - 都不行就编译错误

运算符重载限制

必须的

- 不能重载的运算符：
  - conditional (?:)
  - Sizeof
  - scope (::)
  - member selector (.)
- 只能重载已经存在的算术符
- 重载算术符的优先级，左/右结合，操作数数量都不变
- 至少有一个参数是自定义数据类型，即不能重载operator+ (double, char)

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

推荐的

- 注意重载符号意义最好和之前类似，比如不要把异或运算符(^)定义为指数操作。
- 对于意义不清楚的运算操作，比如string - string，最好改用更容易理解的函数

运算符重载例子

- string + string 字符串相加
- Matrix A * double 矩阵乘以数值
- cout << Point << endl; 输出自定义类
- Faction A == Fraction B 判断分数相等

使得程序更直观，简单，符合人的思维逻辑

运算符重载方式

看使用场景哪个更适合

1. 用友元函数：算术运算符+-*/
2. 用正常函数：和友元差不多，不能直接访问私有成员，但更安全
3. 用类成员函数：第一个参数必须是类自己

1.友元函数重载

- 对算术运算符+ - * /重载
- 在类里声明友元函数，从而使得operator+可以直接读私有数据
- 注意友元函数不是成员函数

```
class Cents
{
private:
    int _cents;

public:
    Cents(int num)
    {
        _cents = num;
    }

    int getCents() const { return _cents; }

    // 声明友元函数operator+
    friend Cents operator+ (const Cents & c1, const Cents & c2);
};

// 注意这个不是Cents的成员函数
Cents operator+ (const Cents & c1, const Cents & c2)
{
    return Cents(c1._cents + c2._cents);
}

int main()
{
    Cents c1(6);
    Cents c2(8);
    Cents sum = c1 + c2;
    cout << "I have " << sum.getCents() << " cents." << endl;

    return 0;
}
```

1.友元函数重载

- 对算术运算符重载，如果操作数类型不一样，可以根据次序定义多个

- 算术运算符第一个参数不见得一定是类自己，用友元重载更灵活

```
class Cents
{
private:
    int _cents;

public:
    Cents(int num)
    {
        _cents = num;
    }

    int getCents() const { return _cents; }

    // 声明友元函数operator+
    friend Cents operator+ (const Cents & c1, int value);
    friend Cents operator+ (int value, const Cents & c1);
};

// 注意这个不是Cents的成员函数
Cents operator+ (const Cents & c1, int value)
{
    return Cents(c1._cents + value);
}

Cents operator+ (int value, const Cents & c1)
{
    return Cents(c1._cents + value);
}

int main()
{
    Cents c1(6);
    Cents sum = 2 + c1 + 9;
    cout << "I have " << sum.getCents() << " cents." << endl;

    return 0;
}
```

2.正常函数重载

- 几乎和友元重载一样
- 声明不在类里面，这样类更安全，不破坏封装性
- 重载函数体里不能访问私有成员

```
class Cents
{
private:
    int _cents;

public:
    Cents(int num)
    {
        _cents = num;
    }

    int getCents() const { return _cents; }
};

// 注意这个不是Cents的成员函数
Cents operator+ (const Cents & c1, const Cents & c2)
{
    return Cents(c1.getCents() + c2.getCents());
}

int main()
{
    Cents c1(6);
    Cents c2(8);
    Cents sum = c1 + c2;
    cout << "I have " << sum.getCents() << " cents." << endl;

    return 0;
}
```

输出运算符<<重载

- std::cout << pt，操作数是ostream和Point，不能直接输出
- 只能友元/正常重载，因为<<第一个参数规定必须是输出流自身
- 返回值是ostream引用，这样才可以链式调用

  std::cout << pt << endl

注意这里返回的ostream是输入参数自身，也就是实参自身

```
class Point
{
private:
    double _x;
    double _y;

public:
    Point(double x, double y) : _x(x), _y(y)
    {
    }

    // 声明友元函数
    friend ostream & operator<< (ostream & out, const Point & pt);
};

// 注意这个不是Point的成员函数
ostream & operator<< (ostream & out, const Point & pt)
{
    out << "Point(" << pt._x << "," << pt._y << ")" << endl;
    return out;
}

int main()
{
    Point pt(6.0, 2.5);
    cout << pt << endl;

    return 0;
}
```

输入>>重载

- std::cin >> pt，操作数是istream和Point
- 和输出<<几乎一样，但是第二个参数不能加const

```
class Point
{
private:
    double _x;
    double _y;

public:
    Point(double x = 0, double y = 0) : _x(x), _y(y)
    {
    }

    // 声明友元函数
    friend ostream & operator<< (ostream & out, const Point & pt);
    friend istream & operator>> (istream & in, Point & pt);
};

// 注意这个不是Point的成员函数
ostream & operator<< (ostream & out, const Point & pt)
{
    out << "Point(" << pt._x << "," << pt._y << ")" << endl;
    return out;
}

istream & operator>> (istream & in, Point & pt)
{
    in >> pt._x >> pt._y;
    return in;
}

int main()
{
    Point pt;
    cin >> pt;
    cout << pt << endl;

    return 0;
}
```

3.成员函数重载

- 成员运算符第一个参数略过，因为隐含this指针作为参数
- c1+8其实调用的是c1.operator+(2)，也就是operator+(&c1, 8)
- 重载运算符左边的参数必须是自定义类自身

```
class Cents
{
private:
    int _cents;

public:
    Cents(int num)
    {
        _cents = num;
    }

    int getCents() const { return _cents; }

    // 声明成员函数operator+
    Cents operator+ (int value);
  //friend Cents operator+ (const Cents & c1, int value);
};

// Cents的成员函数，函数里隐含了this指针
Cents Cents::operator+ (int value)
{
    return Cents(this->_cents + value);
}

int main()
{
    Cents c1(6);
    Cents c2 = c1 + 8;
    cout << "I have " << c2.getCents() << " cents." << endl;

    return 0;
}
```

一元运算符重载

- 一元运算符+，-，！
- 因为重载必须包含一个自定义数据类型，所以只能是类自己
- 所以通常是成员函数重载方式
- 这里和二元运算符-没有歧义，用参数数量识别

```
class Cents
{
private:
    int _cents;

public:
    Cents(int num)
    {
        _cents = num;
    }

    int getCents() const { return _cents; }

    // 声明成员函数，一元运算符operator-
    Cents operator- ();
  //Cents operator- (int value);
};

// Cents的成员函数，函数里隐含了this指针
Cents Cents::operator- ()
{
    return Cents(-this->_cents);
}

int main()
{
    Cents c1(6);
    cout << "I have " << (-c1).getCents() << " cents." << endl;

    return 0;
}
```

小结

- 有些不能用成员函数重载，因为最左边的参数不是自定义类
  <<，>>，必须用友元/正常函数重载
  Operator+(int, Cents)，必须用友元/正常函数重载
- 有些不能用友元函数重载
  =，[]，()，->，必须用成员函数重载
- 二元运算符
  - 修改了左边参数，比如+=，用成员函数重载
  - 不修改了左边参数，比如+，用友元/正常函数重载
- 一元运算符
  - 一般用成员函数重载

常用运算符重载

比较运算符重载

- <, >, ==, !=
- 因为不改变左边参数，所以用友元函数重载
- 这里定义>不是很有意义

```
class Student
{
private:
    string _lastName;
    string _firstName;

public:
    Student(string last, string first) : _lastName(last), _firstName(first)
    {
    }

    friend bool operator!= (const Student & s1, const Student & s2);
};

bool operator!= (const Student & s1, const Student & s2)
{
    return !(s1._lastName == s2._lastName) || !(s1._firstName == s2._firstName);
}

int main()
{
    Student s1("John", "Mike");
    Student s2("Ben", "Mary");

    if (s1 != s2)
    {
        cout << "Different student names" << endl;
    }
    return 0;
}
```

自增减运算符重载

前缀++， --

首先自加/减一，

然后再进行表达式计算

返回的是自身的引用，不进行额外的复制

```
class Digit
{
private:
    int _count;

public:
    Digit(int count = 0) : _count(count)
    {
    }

    // prefix increment
    Digit& operator++ ();

    friend ostream& operator<< (ostream& out, const Digit & digit);
};

Digit& Digit::operator++ ()
{
    _count++;
    if (_count > 9)
    {
        _count = 0;
    }
    return *this;
}

ostream& operator<< (ostream& out, const Digit & digit)
{
    out << digit._count << endl;
    return out;
}

int main()
{
    Digit digit(8);
    cout << ++digit;
    cout << ++digit;
    return 0;
}
```

自增减运算符重载

后缀++， --

首先进行表达式计算，

然后自加/减一，

需要加dummy int来区别前缀

返回的是临时变量，需要复制类自身，开销比前缀的大

```
class Digit
{
private:
    int _count;

public:
    Digit(int count = 0) : _count(count)
    {
    }

    // prefix increment
    Digit& operator++ ();
    // postfix decrement
    Digit operator-- (int);

    friend ostream& operator<< (ostream& out, const Digit & digit);
};

Digit& Digit::operator++ ()
{
    _count++;
    if (_count > 9)
    {
        _count = 0;
    }
    return *this;
}

Digit Digit::operator-- (int)
{
    Digit tmp(_count);
    _count--;
    if (_count < 0)
    {
        _count = 9;
    }
    return tmp;
}

ostream& operator<< (ostream& out, const Digit & digit)
{
    out << digit._count << endl;
    return out;
}

int main()
{
    Digit digit(8);
    cout << ++digit;
    cout << ++digit;
    cout << digit--;
    cout << digit--;
    return 0;
}
```

下标运算符[]重载

假如我们封装了一个静态数组
怎么读取数组元素？

```
class IntArray
{
private:
    int _data[10];

public:
};

int main()
{
    IntArray mylist;

    // 怎么读取数组元素？

    return 0;
}
```

下标运算符[]重载

1. 定义一个函数setItem

```
class IntArray
{
private:
    int _data[10];

public:
    void setItem(int index, int value)
    {
        _data[index] = value;
    }
};

int main()
{
    IntArray mylist;

    // 怎么读取数组元素？
    mylist.setItem(2, 3);

    return 0;
}
```

下标运算符[]重载

1. 定义一个函数setItem
2. 返回内部数组地址

```
class IntArray
{
private:
    int _data[10];

public:
    void setItem(int index, int value)
    {
        _data[index] = value;
    }

    int * getList() { return _data; }
};

int main()
{
    IntArray mylist;

    // 怎么读取数组元素?
    mylist.setItem(2, 3);
    mylist.getList()[2] = 3;

    return 0;
}
```

下标运算符[]重载

1. 定义一个函数setItem
2. 返回内部数组地址
3. 重载[]

```
class IntArray
{
private:
    int _data[10];

public:
    void setItem(int index, int value)
    {
        _data[index] = value;
    }

    int * getList() { return _data; }

    int & operator[] (const int index);
};

int & IntArray::operator[](const int index)
{
    return _data[index];
}

int main()
{
    IntArray mylist;

    // 怎么读取数组元素?
    mylist.setItem(2, 3);
    mylist.getList()[2] = 3;
    mylist[2] = 3;

    return 0;
}
```

下标运算符[]重载

1. 必须用成员函数重载
2. operator[]必须返回引用
   或者说必须是左值（有实际地址）
3. const对象问题
   注意成员函数加const，也是重载（除了参数区分）
4. 重载[]后可以检查数组越界

```
class IntArray
{
private:
    int _data[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

public:

    int & operator[] (const int index);
    const int & operator[] (const int index) const;
};

int & IntArray::operator[](const int index)
{
    return _data[index];
}

const int & IntArray::operator[](const int index) const
{
    return _data[index];
}

int main()
{
    IntArray oldlist;
    // 调用不带const版本
    oldlist[2] = 3;

    const IntArray mylist;
    // Error，调用const版本
    // mylist[2] = 3;
    cout << mylist[2];

    return 0;
}
```

下标运算符[]重载

1. 注意指针和[]混淆
   pMylist[2]可以使用，不过不是IntArray的[]，这里其实已经越界。
2. 重载[]的参数不限于int，可以是别的，比如string
3. 需要delete pMylist

```
class IntArray
{
private:
    int _data[10];

public:

    int & operator[] (const int index);
};

int & IntArray::operator[](const int index)
{
    // assert(index >=0 && index < 10);
    return _data[index];
}

int main()
{
    IntArray * pMylist = new IntArray;

    // 假定以IntArray为元素的数组
    pMylist[2] = 3;
    // 单个IntArray，读取内部的int
    (*pMylist)[2] = 3;

    return 0;
}
```

特殊运算符重载

函数运算符()重载

- 必须是成员函数
- 重载()的参数的类型和数目都可以变动
- 可以用来读取二维数组
- 可以用来读取一维数组一批元素
- 谨慎使用，容易出错

```
class Matrix
{
private:
    double _data[4][4];

public:

    double & operator() (int row, int col);
    double operator() ();
};

double & Matrix::operator() (int row, int col)
{
    return _data[row][col];
}

double Matrix::operator() ()
{
    return _data[0][0];
}

int main()
{
    Matrix matrix;
    matrix(0, 0) = 1.1;
    cout << matrix() << endl;

    return 0;
}
```

Function object(functor)

- 使用时像函数，其实是class，保存了数据
- 类似于函数里定义了static局部变量

```
class Accumulator
{
private:
    int _count = 0;

public:

    int operator() (int count) { return (_count += count); }
};

int main()
{
    Accumulator acc;
    cout << acc(9) << endl;
    cout << acc(8) << endl;

    return 0;
}
```

重载typecast

- 比如operator int()
- 没有返回类型
- 可以重载自定义类型cast

```
class Cents
{
private:
    int _cents;

public:
    Cents(int cents = 0) { _cents = cents; }

    operator int() { return _cents; }
};

void PrintInt(int value)
{
    cout << value;
}

int main()
{
    Cents cents(8);
    PrintInt(cents);

    return 0;
}
```

复制构造函数和
赋值运算符重载

复制构造函数copy constructor

- C++自动加缺省的构造/析构函数，如果你没加
  - 如果你加了自定义构造函数，缺省构造函数不再自动加
- 复制构造函数
  - C++还会自动加
  - 会把成员变量逐个按值复制

```
class Fraction
{
public:
    // constructor
    Fraction(int numerator=0, int denominator=1)
        :
        m_numerator(numerator),
        m_denominator(denominator)
    {
        assert(denominator != 0);
    }
private:
    int m_numerator;
    int m_denominator;
};

int main()
{
    Fraction fiveThirds(5, 3); // call constructor(int, int)
    Fraction fCopy(fiveThirds); // still compiled, why?
}
```

复制构造函数copy constructor

- 调用场景
  - 做为函数形参传值
  - 用已有对象来初始化新的对象
- 很多情况下自动建的复制构造函数够用
  - 比如成员变量都是int这样的普通变量

```
class Fraction
{
public:
    // constructor
    Fraction(int numerator=0, int denominator=1)
        :
        m_numerator(numerator),
        m_denominator(denominator)
    {
        assert(denominator != 0);
    }
    // Copy constructor
    Fraction(const Fraction &frac)
        :
        m_numerator(frac.m_numerator),
        m_denominator(frac.m_denominator)
    {
        cout << "Copy constructor called\n";
    }
private:
    int m_numerator;
    int m_denominator;
};

int main()
{
    Fraction fiveThirds(5, 3); // call constructor(int, int)
    Fraction fCopy(fiveThirds); // call copy constructor
}
```

注意点

- 如果把复制构造函数放入private，
- 意味着类不允许复制

- 注意如果所有的构造函数都是private，那么意味着不能创建对象。有一种场景需要这样，比如singleton

```
class Fraction
{
public:
    // constructor
    Fraction(int numerator=0, int denominator=1)
        :
        m_numerator(numerator),
        m_denominator(denominator)
    {
        assert(denominator != 0);
    }
private:
    // copy is not allowed
    Fraction(const Fraction &frac);
    int m_numerator;
    int m_denominator;
};

int main()
{
    // Call constructor(int, int)
    Fraction fiveThirds(5, 3);
    // Error: 'Fraction::Fraction(const Fraction&)' is private
    Fraction fCopy(fiveThirds);
}
```

赋值运算符重载

- 必须做为成员函数重载
- C++自动加，如果你没加
- 和复制构造函数类似，调用的区别是
  - 当左边对象已存在，调用赋值运算符
  - 当需要建立一个新对象，调用复制构造函数

```cpp
int main()
{
    // Call constructor(int, int)
    Fraction fiveThirds(5, 3);
    Fraction fCopy = fiveThirds;  ✗ ERROR: call private copy constructor
    fCopy = fiveThirds;           // OK: call overloading assignment
}
```

```cpp
class Fraction
{
public:
    // constructor
    Fraction(int numerator=0, int denominator=1)
        :
        m_numerator(numerator),
        m_denominator(denominator)
    {
        assert(denominator != 0);
    }

    // Overloaded assignment
    Fraction& operator= (const Fraction &fraction);

private:
    // copy is not allowed
    Fraction(const Fraction &frac);
    int m_numerator;
    int m_denominator;
};

Fraction& Fraction::operator= (const Fraction &fraction)
{
    m_numerator = fraction.m_numerator;
    m_denominator = fraction.m_denominator;

    // return the existing object so we can chain this operator
    return *this;
}
```

赋值运算符重载

- 注意self assignment

```
class IntArray
{
public:
    IntArray(int size) : m_size(0), m_data(nullptr)
    {
        if (size <= 0)
        {
            return;
        }

        // 申请大小为size的动态数组内存
        m_data = new (nothrow) int[size]{0};

        if (m_data != nullptr)
        {
            m_size = size;
        }
    }

    IntArray & operator= (const IntArray &rhs);
private:
    IntArray(const IntArray &other);
    int * m_data;
    int m_size;
};
```

```
IntArray & IntArray::operator= (const IntArray &rhs)
{
    // self check
    if (this == &rhs)
    {
        return *this;
    }

    // release memory if existing.
    if (m_data)
    {
        delete [] m_data;
        m_data = nullptr;
        m_size = 0;
    }

    m_size = rhs.m_size;
    m_data = new (nothrow) int[rhs.m_size]{0};

    // copy data from rhs to this
    assert(m_data != nullptr);
    for (int i = 0; i < m_size; i++)
    {
        m_data[i] = rhs[i];
    }
    // using this pointer to return self reference
    return *this;
}


int main()
{
    IntArray a(5);
    IntArray b(10);
    a = b;   // call overloading assignment
    b = b;   // The case of self assignment
}
```

Shallow copy vs Deep copy

- 缺省的复制构造函数和重载赋值运算符都是shallow copy
  - 把成员变量直接复制
- 如果成员变量是动态内存等资源，shallow copy会出问题
- 正确的做法，应该自己实现复制构造函数和重载赋值运算符
  - Deep copy
  - 把指针指向的内存块复制

```
IntArray(const IntArray &other);
IntArray & operator= (const IntArray &rhs);
```

```
class IntArray
{
public:
    IntArray(int size) : m_size(0), m_data(nullptr)
    {
        // 申请大小为size的动态数组内存
        m_data = new (nothrow) int[size]{0};

        if (m_data != nullptr)
        {
            m_size = size;
        }
    }

    ~IntArray()
    {
        delete [] m_data;
        m_size = 0;
        m_data = nullptr;
    }

    friend ostream & operator<< (ostream & out, const IntArray & arr)
    {
        for (int i = 0; i < arr.m_size; i++)
        {
            cout << arr[i] << " ";
        }
        return out;
    }
private:
    int * m_data;
    int m_size;
};

int main()
{
    IntArray a(7);
    if (true)
    {
        IntArray b = a; //call default copy constructor
        cout << b << endl;
    } // b out of scope, call destructor
    cout << a << endl; // Error here!!!
}
```

Rule of three

- 析构函数destructor
- 复制构造函数copy constructor
- 重载赋值运算符overloading assignment

- 每个都是C++自动加，如果你不加
- **Rule of three**：你如果需要自己实现其中一个（自动加的不够），那么剩下两个也必须自己实现（才能避免内存泄漏）
- 注意后两者可以变成private，如果想偷懒暂时不实现

类.特殊函数

Member function	Typical form for class C:
Default constructor	C::C();
Destructor	C::~C();
Copy constructor	C::C (const C&);
Copy assignment	C& operator= (const C&);
Move constructor （C++11）	C::C (C&&);
Move assignment （C++11）	C& operator= (C&&);

Member function	implicitly defined:	default definition:
Default constructor	if no other constructors	does nothing
Destructor	if no destructor	does nothing
Copy constructor	if no move constructor and no move assignment	copies all members
Copy assignment	if no move constructor and no move assignment	copies all members
Move constructor	if no destructor, no copy constructor and no copy nor move assignment	moves all members
Move assignment	if no destructor, no copy constructor and no copy nor move assignment	moves all members

设计IntArray类

- 要求内部动态分配数组内存
- 注意释放资源（rule of three）
- 支持常用操作（Length(), [], <<, etc)
- 注意边界检查
- 注意常量const对象使用

步骤一：决定数据元素

1. 指向**int**数组的指针
   - 因为我们要动态决定数组的大小，不希望必须编译阶段的固定长度
   - **private int * _pData;**

2. 数组的长度
   - 用来告诉用户数组大小
   - 用来检查数组元素读取时候的越界问题
   - **private int _size;**

步骤二：如何分配动态内存和释放

- 配对进行
  - **pData = new int[sz]**
  - **delete [] pData**
  - 分配应该在构造函数，释放应该在析构函数里

- 别忘了**rule of three**
  - 写了析构函数释放资源 **~IntArray()**
  - 必须写复制构造函数 **IntArray(const IntArray &other)**
  - 和赋值重载函数 **IntArray & operator= (const IntArray &rhs)**

步骤三：数组下标操作符[]

- 对自定义类型IntArray重载[]
  - **int & operator[] (int index)**
  - 返回数组内部元素的引用
  - 用assert做越界检查

- Const对象也要能用下表操作符[]
  - **const int & operator[] (int index) const**
  - 加const重载[]，实现内容一样

步骤四：别的补充

- 返回长度
  - **int Length() const**
  - 这样const对象也能调用

- 重新申请数组
  - **void Resize(int size);**
  - 同样需要释放之前的内存，然后申请新的数组内存

- 添加功能过程中，有些重复用到的代码块，可以考虑变成私有函数调用，比如分配动态数组内存getArray()，释放内存Reset()

IntArray

- 公开方法
- 尤其是
  - 析构函数，
  - 复制构造函数，
  - 赋值运算符重载

```
public:
    IntArray(int size)
        :
        _pData(nullptr),
        _size(0)
    {
        getArray(size);
        for (int i = 0; i < _size; i++)
        {
            _pData[i] = i;
        }
    }

    ~IntArray()
    {
        Reset();
    }

    IntArray(const IntArray & other)
    {
        getArray(other.Length());

        for (int i = 0; i < _size; i++)
        {
            _pData[i] = other[i];
        }
    }

    IntArray & operator= (const IntArray & rhs)
    {
        if (this == &rhs)
        {
            return *this;
        }

        getArray(rhs.Length());
        for (int i = 0; i < _size; i++)
        {
            _pData[i] = rhs[i];
        }

        return *this;
    }
```

IntArray

- 其余公开方法

```
public:
    int Length() const
    {
        return _size;
    }

    void Resize(int size)
    {
        getArray(size);
    }

    int & operator[] (int index)
    {
        assert(index >= 0 && index < _size);
        return _pData[index];
    }

    const int & operator[] (int index) const
    {
        assert(index >= 0 && index < _size);
        return _pData[index];
    }

    friend ostream & operator<< (ostream & out, const IntArray & list);
};

ostream & operator<< (ostream & out, const IntArray & list)
{
    for (int i = 0; i < list._size; i++)
    {
        out << list[i] << endl;
    }
    return out;
}
```

IntArray

- 私有数据和方法

```
class IntArray
{
private:
    int * _pData;
    int _size;

    void getArray(int size)
    {
        // 避免分配负的内存引起的exception
        if (size < 0)
        {
            return;
        }

        // 重置内部数组，释放资源
        Reset();

        // 申请大小为size的数组内存
        if (size > 0)
        {
            _size = size;
            _pData = new (nothrow) int[size] {0};
        }

        // 如果内存分配失败，确保_size还是0
        if (_pData == nullptr)
        {
            _size = 0;
        }
    }

    void Reset()
    {
        if (_pData != nullptr)
        {
            delete [] _pData;
            _pData = nullptr;
            _size = 0;
        }
    }
```

应用IntArray

- 在mergesort里使用IntArray
- 保证mergesort可以对大于百万数据元素排序
- IntArray的作用是对动态内存封装，不需要用户去释放，即使出现exception也保证内存释放

Q&A

Thanks!

课堂内容

- 类class和对象object

- 重载（Overload）
  - 函数重载 (function overload)
  - 运算符重载 (operator overload)

- 对象Object关系
- 类继承Ineritance
- 动态绑定和虚函数 Virtual function

对象之间关系

- 抽象出问题的本质
- 把复杂事物划分成相对独立的小对象
- 可以重用代码 (引用次数)
- 使得逻辑分明，易于扩展和维护

一种对象之间关系（组合）

对象具体实例	组合Composition	聚合Aggregation	联合Association
关系类型	部分--整体	部分--整体	没有从属关系
成员可以属于多个类	No （心脏不能共享）	Yes （老师可以共享）	Yes （病人可以看多个医生）
成员由类建立/销毁	Yes （心脏不能单独存在）	No （老师可以单独存在）	No （病人和医生不共生）
关系描述语	Part-of	Has-a	Uses-a
例子	人有一颗心脏	系里有几个老师	医生和病人
使用	一般使用正常成员定义(指针也可以) 负责建立和销毁	使用指针或者引用定义成员 不负责建立和销毁	使用指针或者引用定义成员 不负责建立和销毁

还有一种更弱的依赖关系（dependency），只是在类要执行某个任务时候才有的关系。
比如你只有腿扭了才需要拐杖，Point class输出到屏幕时候才需要ostream。

组合例子 -- part of

- UML用填充的钻石形状表示组合关系
- Car负责销毁Carburetor
- 如果使用指针的话，在Car类内部创建和销毁

```
// Composition
class Car
{
private:

  // Car is the owner of carburetor.
  // carburetor is created when Car is created,
  // it is destroyed when Car is destroyed.
  Carburetor carburetor;

};
```

聚合例子 -- has a

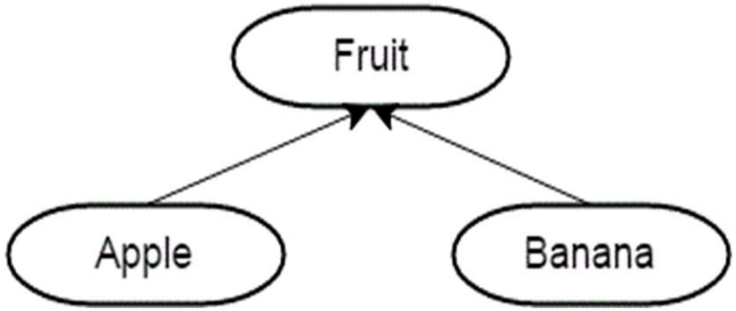
- UML用空心的钻石形状表示聚合关系
- Duck还活着，即使Pond被填了
- 一般使用指针或者引用

```
// Aggregation
class Pond
{
private:

  // Pond is not the owner of ducks,
  // it has references on other ducks managed somewhere else
  std::vector<Duck*> ducks;
};
```

类的继承

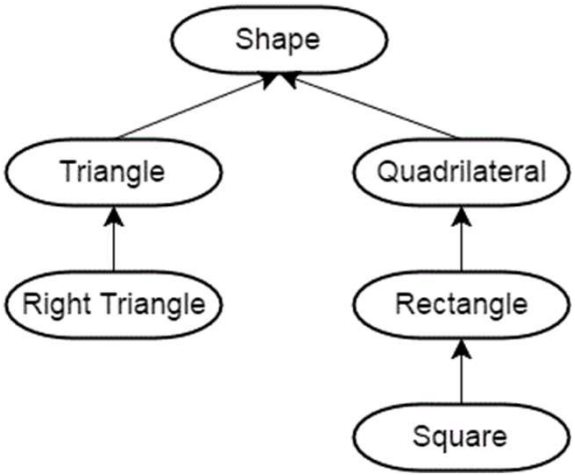
另一种对象之间关系（继承）

- 组合和聚合适合于松散的“has-a”关系
- 另外一种主要内因关系：“is-a”关系，用**继承**来描述
- 比如苹果和香蕉都属于（is-a）水果。



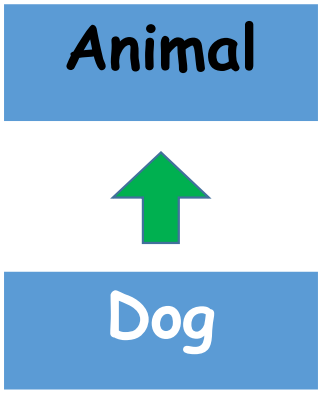
继承Inheritance

- 基类/衍生类，父类/子类
- 子类自动包含父类的数据和函数
- 子类不能直接使用父类private数据和函数




```
Class Shape(...);                          // Base class Shape
Class Triangle : public Shape(...);        // Triangle derived from Shape
Class RightTriangle : public Triangle(...); // Right-Triangle derived from Triangle
```

继承例子 Re-use Code




```
class Animal {
        public:
                int legs = 4;
};
class Dog : public Animal {
        public:
                int tail = 1;
};
int main() {
        Dog d;
        cout <<"Legs are: "<<d.legs<<endl;
        cout << "Tail is: "<<d.tail;
}
```

子类和父类的构造函数和析构函数

- 构造函数和析构函数不能被继承
- 子类的构造函数执行时，父类的构造函数先被执行
- 子类的析构函数先被执行，然后是父类的析构函数

构造函数和析构函数顺序

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

class B : public A
{
public:
    B() { cout << " B" << endl; }
    ~B() { cout << "~B" << endl; }
};

class C : public B
{
public:
    C() { cout << " C" << endl; }
    ~C() { cout << "~C" << endl; }
};

int main()
{
    cout << "Construct C object" << endl;
    C object;
    cout << "Destroy C object" << endl;
    return 0;
}
```

```
Construct C object
 A
 B
 C
Destroy C object
~C
~B
~A
```

基类构建过程

- 分配内存
- 调用基类的构造函数
- 初始化列表
- 构造函数体
- 返回caller

```
class Base
{
public:
    int _id;
    Base(int id) : _id(id) {}
    int getId() const { return _id; }
};

class Derived : public Base
{
public:
    double _cost;
    Derived(double cost = 0.0) : _cost(cost) {}
    double getCost() const { return _cost; }
};

int main()
{
    Base base(5);
    return 0;
}
```

子类构建过程

- 分配内存（子类和父类）
- 调用子类的构造函数
- 先调用基类构造函数，如果不指定就调用基类缺省构造函数
- 初始化子类列表
- 构造函数体
- 返回caller

```
class Base
{
public:
    int _id;
    Base(int id = 0) : _id(id) {}
    int getId() const { return _id; }
};

class Derived : public Base
{
public:
    double _cost;
    Derived(double cost = 0.0) : _cost(cost) {}
    double getCost() const { return _cost; }
};

int main()
{
    Derived obj(1.3);
    return 0;
}
```

怎么设定基类的成员数据

- 尝试在初始化列表里进行
- 错误，因为已经在基类构造函数里初始化了，不能初始化两次

```
class Derived : public Base
{
public:
    double _cost;
    Derived(double cost = 0.0, int id = 0)
     :
      _cost(cost),
      _id(id)      // ERROR: cannot initialize here
     {}
    double getCost() const { return _cost; }
};
```

- 可以改成在构造函数体里赋值
- 但如果_id是const，或者是私有成员，也不行

```
class Derived : public Base
{
public:
    double _cost;
    Derived(double cost = 0.0, int id = 0)
     :
      _cost(cost)
     {
        _id = id; // OK, 赋值可以
     }
    double getCost() const { return _cost; }
};
```

调用基类构造函数

衍生类构造函数　　基类构造函数

```
Derived:: Derived(cost, id): Base(id)
```

衍生类构造函数参数　　基类构造函数参数

- 如果基类没有缺省构造函数的话，必须显式调用别的构造函数

设定基类的成员数据

- 分配内存（子类和父类）
- 调用子类的构造函数
- 首先调用基类构造函数，就是指定的Base(id)
- 初始化基类数据
- 返回到子类构造函数
- 初始化子类列表
- 构造函数体
- 返回caller

```
class Derived : public Base
{
public:
    double _cost;
    Derived(double cost = 0.0, int id = 0)
     :
      Base(id),
      _cost(cost)
     {
     }
    double getCost() const { return _cost; }
};
```

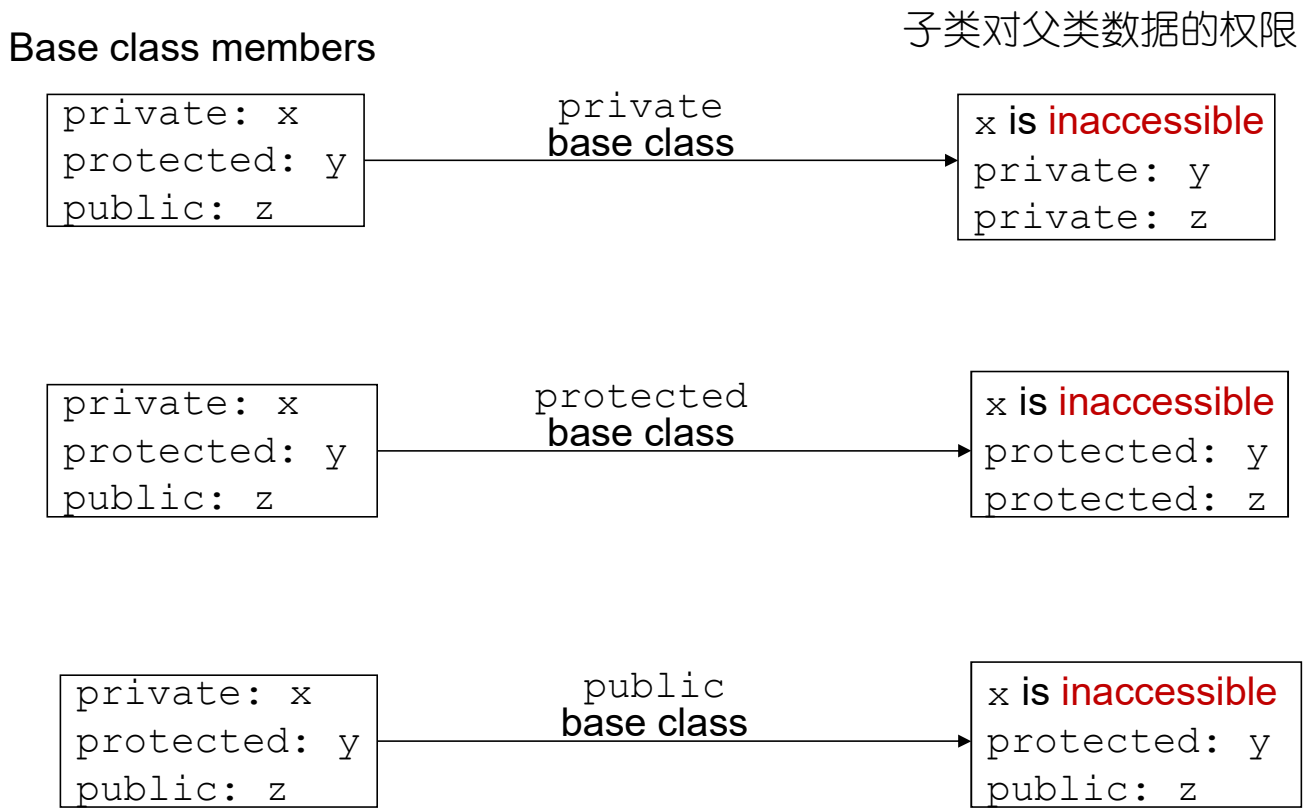
把数据变成private

```
class Base
{
private:
    int _id;
public:
    Base(int id = 0) : _id(id) {}
    int getId() const { return _id; }
};

class Derived : public Base
{
private:
    double _cost;
public:
    Derived(double cost = 0.0, int id = 0)
     :
      Base(id),
      _cost(cost)
     {
     }
    double getCost() const { return _cost; }
};
```

类继承方式

- 公有继承
  - Class Triangle : public Shape(...);
- 保护继承
  - Class Triangle : protected Shape(...);
- 私有继承
  - Class Triangle : private Shape(...);
  - 如果子类和父类关系不大，不想子类公开父类方法

继承方式和权限


Base class members
子类对父类数据的权限
private: x
protected: y
public: z
private
base class
x is inaccessible
private: y
private: z
private: x
protected: y
public: z
protected
base class
x is inaccessible
protected: y
protected: z
private: x
protected: y
public: z
public
base class
x is inaccessible
protected: y
public: z

继承方式和权限

class Grade

private members:
```
char letter;
float score;
void calcGrade();
```
public members:
```
void setScore(float);
float getScore();
char getLetter();
```

class Test : **public** Grade

private members:
```
int numQuestions;
float pointsEach;
int numMissed;
```
public members:
```
Test(int, int);
```

`Test` class 共有继承 `Grade` class 后: →

private members:
```
int numQuestions:
float pointsEach;
int numMissed;
```
public members:
```
Test(int, int);
void setScore(float);
float getScore();
char getLetter();
```

继承方式和权限

class Grade

private members:
```
char letter;
float score;
void calcGrade();
```
public members:
```
void setScore(float);
float getScore();
char getLetter();
```

class Test : protected Grade

private members:
```
int numQuestions;
float pointsEach;
int numMissed;
```
public members:
```
Test(int, int);
```

Test class 保护继承 Grade class 后:

private members:
```
int numQuestions:
float pointsEach;
int numMissed;
```
public members:
```
Test(int, int);
```
protected members:
```
void setScore(float);
float getScore();
float getLetter();
```

继承方式和权限

class Grade

private members:
```
char letter;
float score;
void calcGrade();
```
public members:
```
void setScore(float);
float getScore();
char getLetter();
```

class Test : **private** Grade

private members:
```
int numQuestions;
float pointsEach;
int numMissed;
```
public members:
```
Test(int, int);
```

Test class 私有继承 Grade class 后:

private members:
```
int numQuestions:
float pointsEach;
int numMissed;
void setScore(float);
float getScore();
float getLetter();
```
public members:
```
Test(int, int);
```

问题

- 子类继承了父类public函数
- 子类对于父类的有些行为表现不一样
  - 比如动物都会叫，但是叫法不一样
  - 车辆都能跑，速度和方式不一样

```
class Base
{
private:
    int _id;
public:
    Base(int id) : _id(id) {}
    int identity() const { cout << "Base " << _id << endl; }
};

class Derived : public Base
{
public:
    Derived(int id = 0)
     :
      Base(id)
     {
     }
};

int main()
{
    Base base(5);
    base.identity();

    Derived derived(7);
    derived.identity();
    return 0;
}
```

重定义基类函数

- 子类的函数可以用一样的函数名和参数列表（数据类型，数目，顺序）来重定义
- 一般是用来替换基类的函数
- 不是重载，因为参数列表一样
- 子类对象用子类函数；父类对象用父类函数
- 子类对象调用父类重定义函数，必须加scope，Base::Identity()

重定义基类函数

```
class Base
{
private:
    int _id;
public:
    Base(int id) : _id(id) {}
    int identity() const { cout << "Base " << _id << endl; }
};

class Derived : public Base
{
public:
    Derived(int id = 0)
     :
      Base(id)
     {
     }

    int identity() const { cout << "Derived " << endl; }
};

int main()
{
    Base base(5);
    base.identity();

    Derived derived(7);
    derived.identity();
    return 0;
}
```

```
int identity() const
{
    cout << "Derived " << endl;
    Base::identity();
}
```

重定义问题

- 考虑:
  - Class Base 定义函数 identity() 和 ShowMe().
    - ShowMe() 调用 identity().
  - Class Derived 继承 Base 并重定义函数 identity().
  - 建立一个Derived 对象，并调用函数 ShowMe().
  - 调用 ShowMe() 时， identity() 也被调用了

- 问题:
  - 是基类还是子类里的identity() 被调用了?

```
class Base
{
private:
    int _id;
public:
    Base(int id) : _id(id) {}
    void Identity() const { cout << "Base " << _id << endl; }
    void ShowMe() const { Identity(); }
};

class Derived : public Base
{
public:
    Derived(int id = 0)
     :
      Base(id)
     {
     }

     void Identity() const
     {
          cout << "Derived " << endl;
          Base::Identity();
     }
};

int main()
{
    Derived derived(7);
    derived.ShowMe();

    Base * pDerived = new Derived(2);
    pDerived->ShowMe();
    return 0;
}
```

重定义问题

- 基类的identity()被调用了。
- 因为函数调用是在编译阶段决定的，叫静态绑定

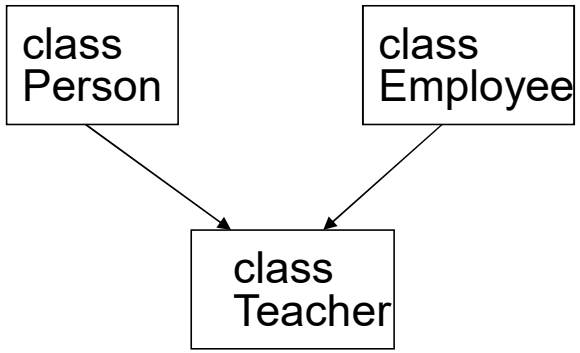
- pDerived->Identity()调用哪个？

```
class Base
{
private:
    int _id;
public:
    Base(int id) : _id(id) {}
    void Identity() const { cout << "Base " << _id << endl; }
    void ShowMe() const { Identity(); }
};

class Derived : public Base
{
public:
    Derived(int id = 0)
     :
      Base(id)
     {
     }

     void Identity() const
     {
          cout << "Derived " << endl;
          Base::Identity();
     }
};

int main()
{
    Derived derived(7);
    derived.ShowMe();

    Base * pDerived = new Derived(2);
    pDerived->ShowMe();
    return 0;
}
```

多继承

- 子类可以有不止一个父类

```
class Teacher : public Person,
                public Employee;
```



多继承问题

- 多个基类包含相同的函数，名字冲突
- 如果是菱形继承的话，问题尤其严重
- 解决方法：
  - 子类重定义相同的函数（重定义问题）
  - 子类在调用时加上scope 运算符 ::
- 例子：iostream里的cin，cout
- 一般来说避免多继承

动态绑定

基类指针

- 可以把子类的指针cast到基类
- 基类指针只知道基类的数据和函数
- 子类的重定义函数（静态绑定）会被忽略

```
Base * pObject = new Derived(2);
```

虚函数和多态

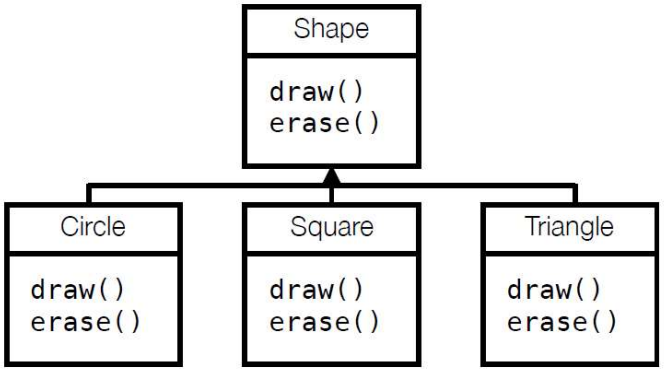
- 虚函数:是指通过基类访问派生类定义的函数
- 需要在基类里加上关键词virtual:

```
virtual void func() {...}
```

- 虚函数支持动态绑定，就是同一段代码在执行中根据输入决定用哪个版本，这种行为就是多态
- 不用virtual重定义就是在编译阶段静态绑定

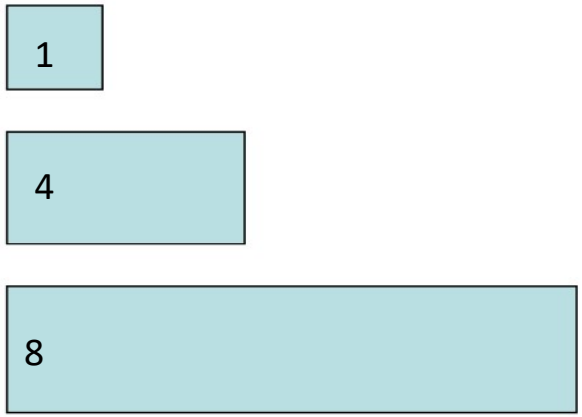
多态

- 多态就是不同类型的对象对于同一个事情行为不一样
- 模板和重载都是编译阶段（静态）多态
  - void DoSomething(const Shape& obj)
  - void DoSomething(const Circle& obj)
  - void DoSomething(const Square& obj)
- 继承提供了动态绑定，多态
  - void DoSomething(Base * pObj)
- **关键:**如何通过基类访问派生类定义的函数



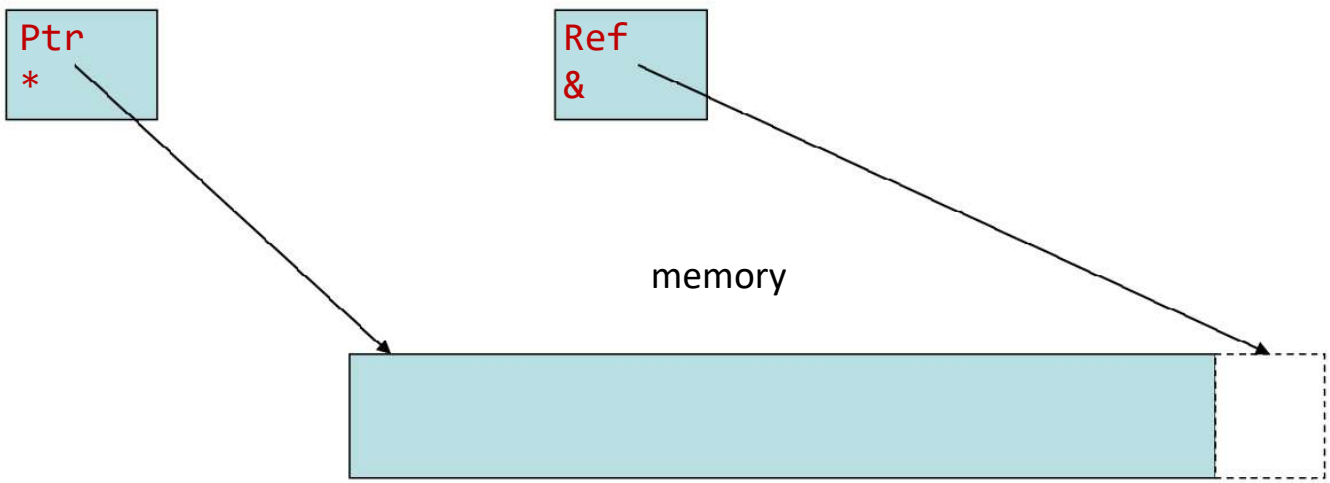
内存模型（内置类型）

- char
- short
- int
- long
- long long
- float
- double
- long double



内存模型

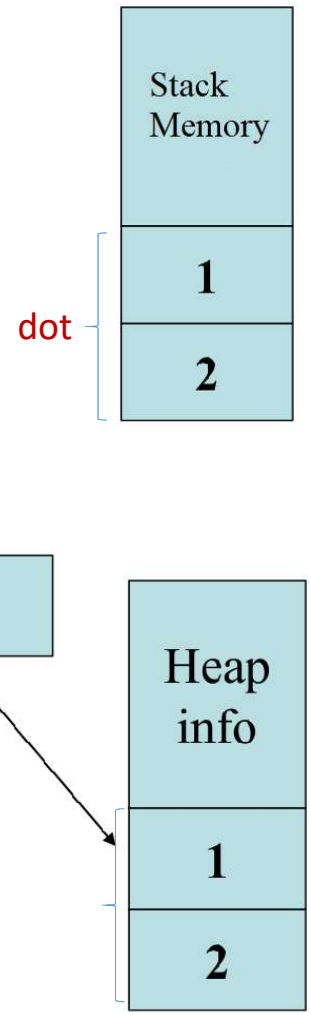
- 指针
- T* (pointer)
- T& (implemented as pointer)



内存模型（正常Class）

- class **Point**

```
 {
      int x, y;
     // int getX();
 };
Point dot(1,2);
Point* pDot = new Point(1,2);
```



内存模型（Derived Class）

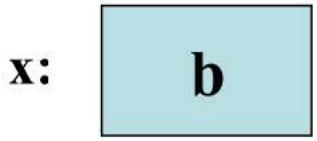
- class **Base**

```
{
    int b;
};
```

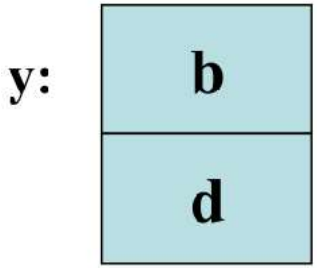
- class **Derived** : public Base

```
{
    int d:
};
```

Base x;



Derived y;



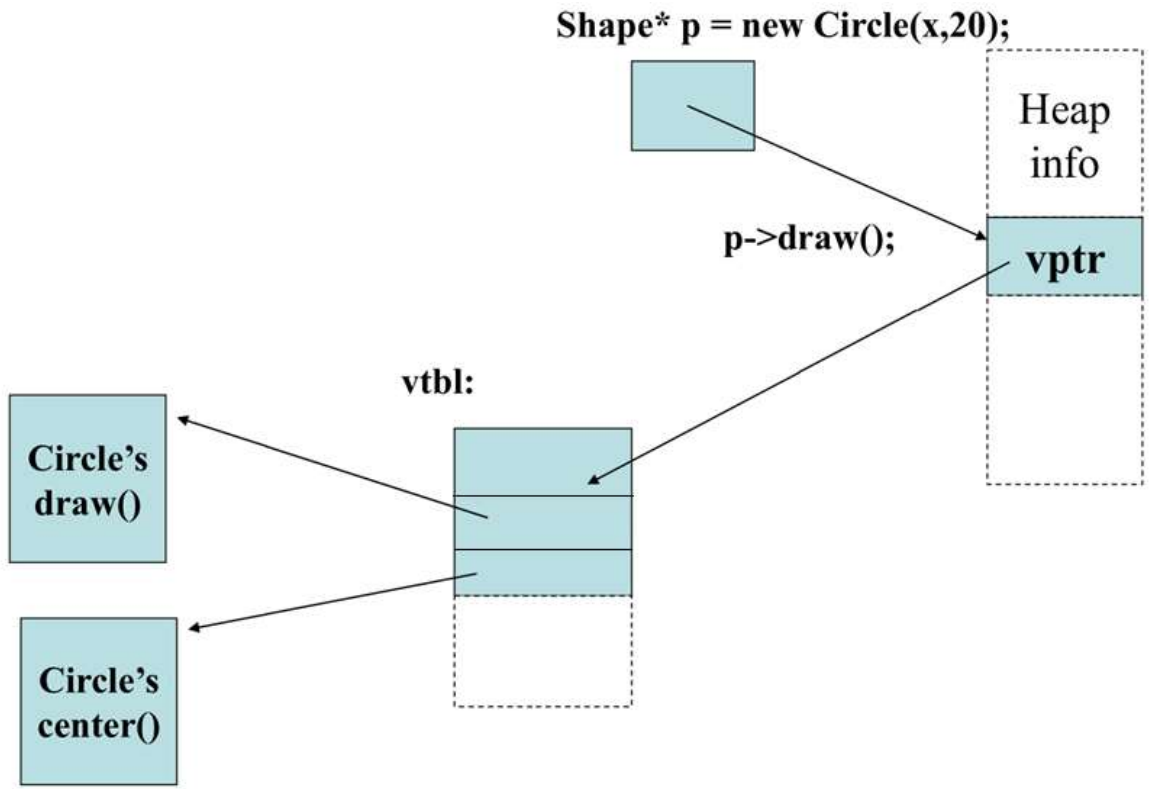
內存模型（Virtual function）

- class **Shape**

```
{
    virtual void draw();
    virtual Point center();
};
```

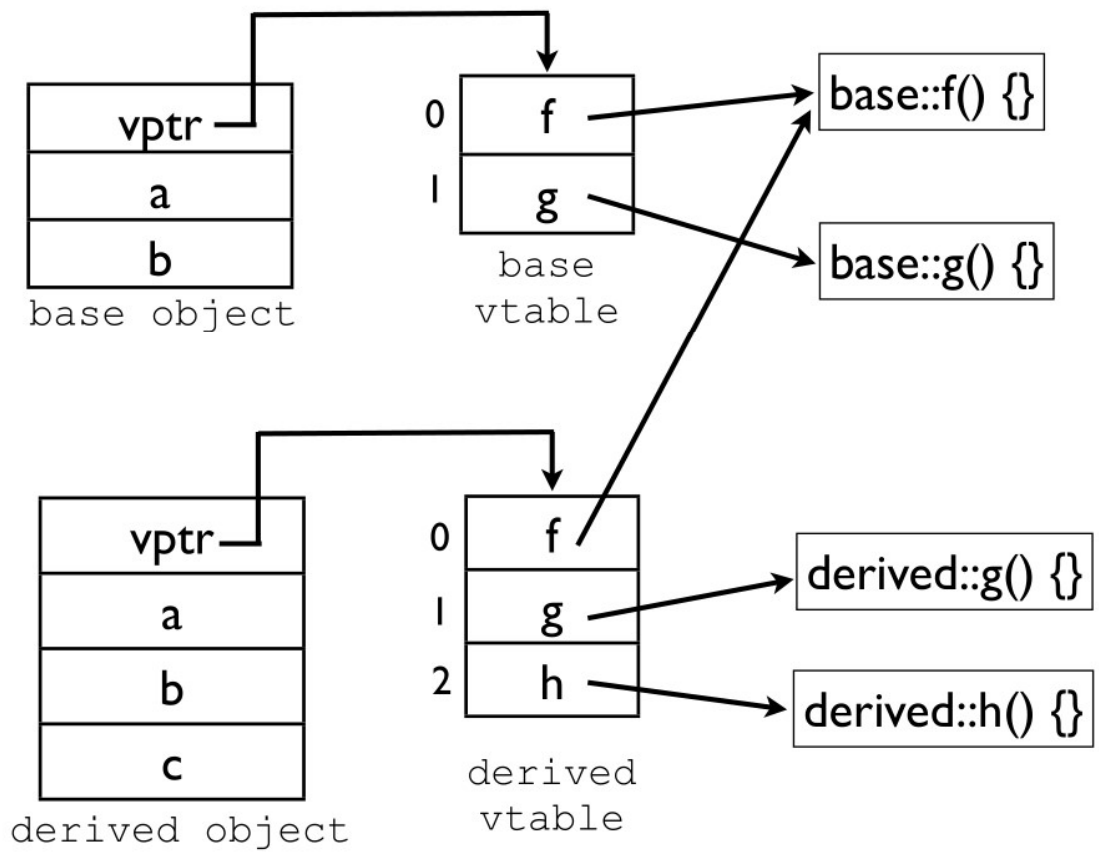
- class **Circle** : public Shape

```
{
    virtual void draw();
    virtual Point center();
};
```



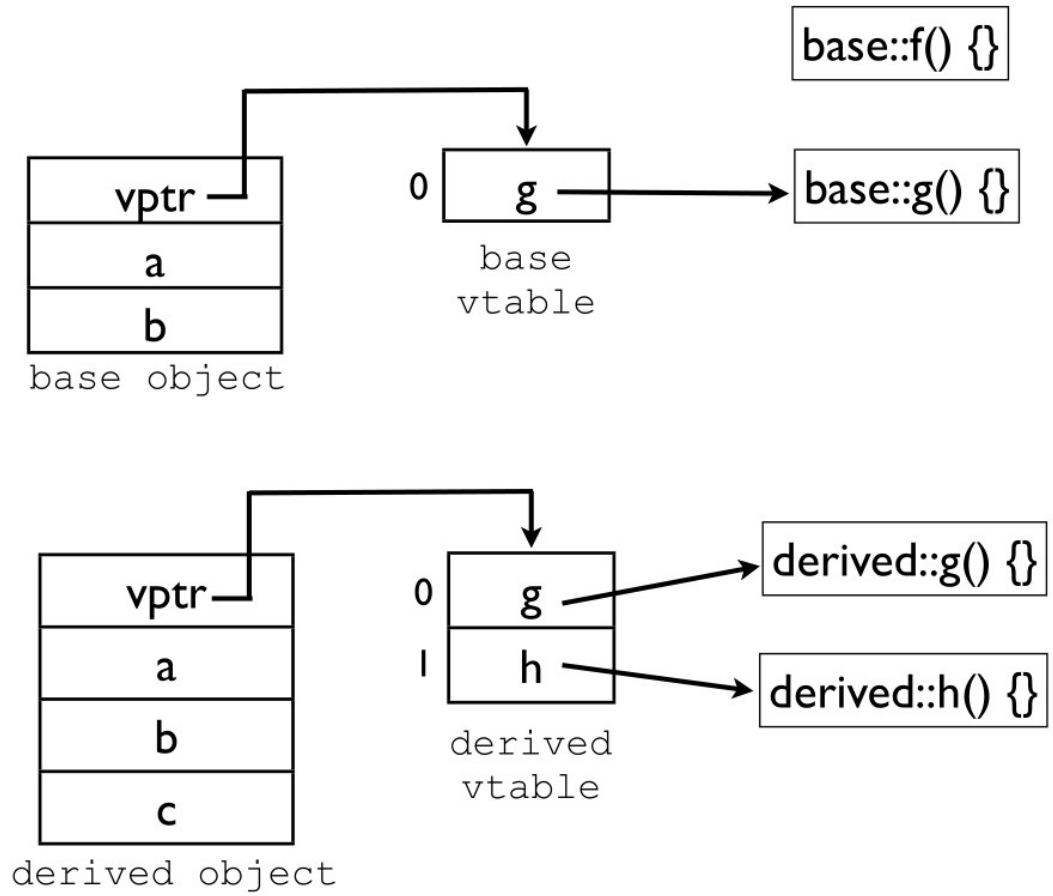
The vtable

```
struct base
{
    virtual void f();
    virtual void g();
    int a,b;
};

struct derived : base
{
    virtual void g();
    virtual void h();
    int c;
};

void poly(base * ptr)
{
    ptr->f();
    ptr->g();
}

int main()
{
    poly(&base());
    poly(&derived());
}
```



The vtable

```
struct base
{
            void f();
    virtual void g();
    int a,b;
};

struct derived : base
{
    virtual void g();
    virtual void h();
    int c;
};

void poly(base * ptr)
{
    ptr->f();
    ptr->g();
}

int main()
{
    poly(&base());
    poly(&derived());
}
```



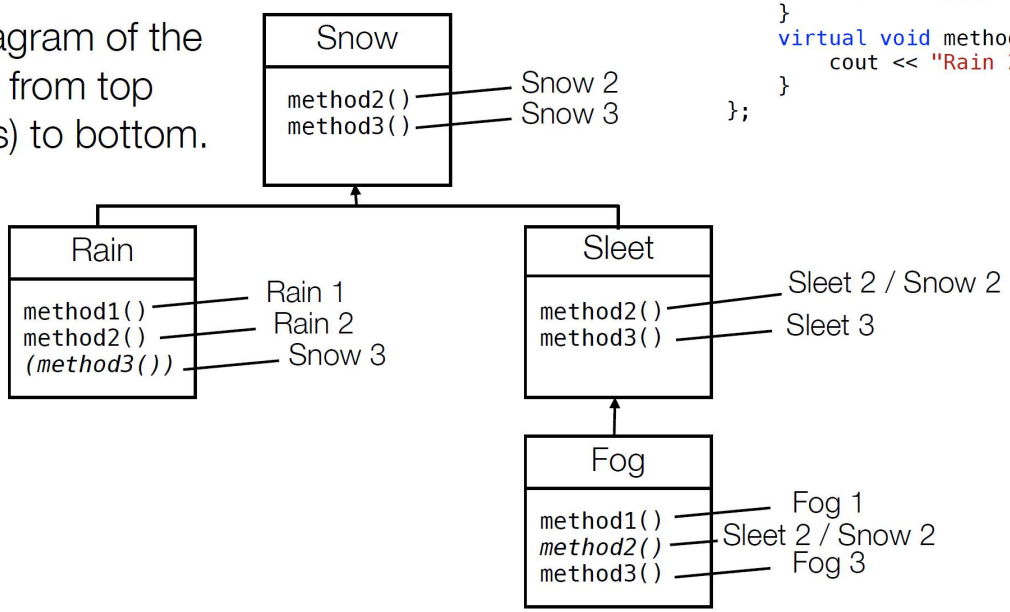
多态例子

```
class Snow {
public:
    virtual void method2() {
        cout << "Snow 2" << endl;
    }
    virtual void method3() {
        cout << "Snow 3" << endl;
    }
};

class Rain : public Snow {
public:
    virtual void method1() {
        cout << "Rain 1" << endl;
    }
    virtual void method2() {
        cout << "Rain 2" << endl;
    }
};
```

```
class Sleet : public Snow {
public:
    virtual void method2() {
        cout << "Sleet 2" << endl;
        Snow::method2();
    }
    virtual void method3() {
        cout << "Sleet 3" << endl;
    }
};

class Fog : public Sleet {
public:
    virtual void method1()   {
        cout << "Fog 1" <<   endl;
    }
    virtual void method3()   {
        cout << "Fog 3" <<   endl;
    }
};
```

类关系图

Draw a diagram of the classes from top (superclass) to bottom.

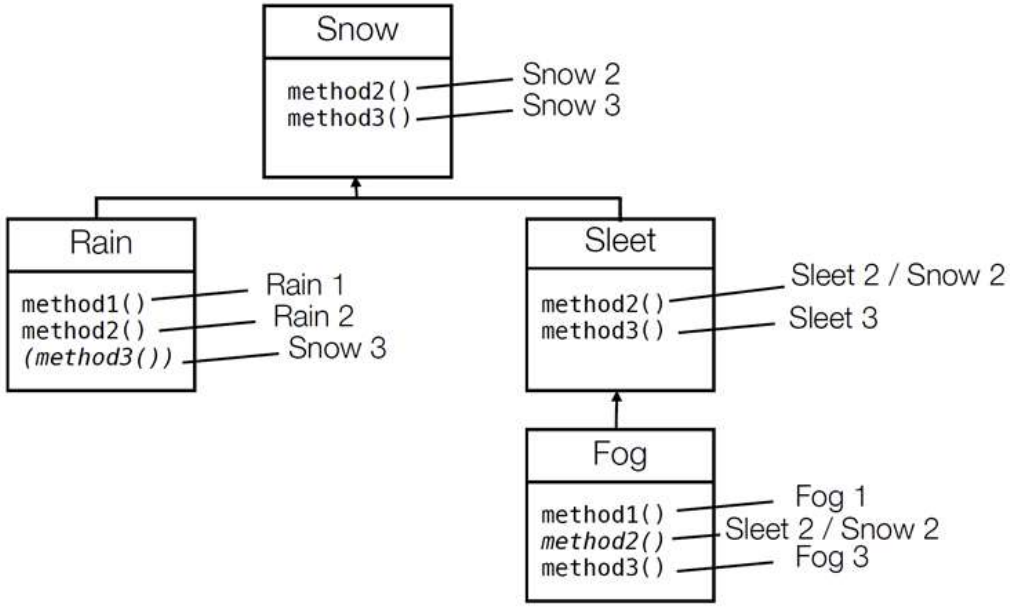



```
class Snow {
public:
    virtual void method2() {
        cout << "Snow 2" << endl;
    }
    virtual void method3() {
        cout << "Snow 3" << endl;
    }
};

class Rain : public Snow {
public:
    virtual void method1() {
        cout << "Rain 1" << endl;
    }
    virtual void method2() {
        cout << "Rain 2" << endl;
    }
};
```

```
class Sleet : public Snow {
public:
    virtual void method2() {
        cout << "Sleet 2" << endl;
        Snow::method2();
    }
    virtual void method3() {
        cout << "Sleet 3" << endl;
    }
};

class Fog : public Sleet {
public:
    virtual void method1()   {
        cout << "Fog 1" <<   endl;
    }
    virtual void method3()   {
        cout << "Fog 3" <<   endl;
    }
};
```

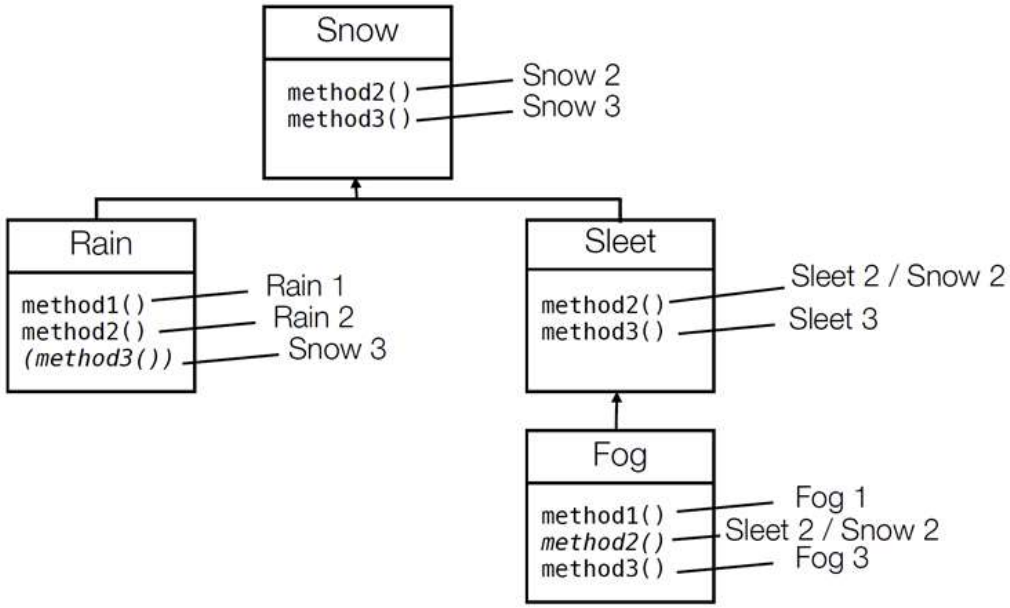
例子

```
Snow* var1 = new Sleet();
var1->method2();

Snow* var2 = new Rain();
var2->method1();

Snow* var3 = new Rain();
var3->method2();
```



例子

```
Snow* var4 = new Rain();
((Rain *) var4)->method1();

Snow* var5 = new Fog();
((Sleet *) var5)->method1();

Snow* var7 = new Sleet();
((Rain*) var7)->method1();
```



注意点

- 不要再构造函数里调用虚函数
  - 在基类的构造函数里调用虚函数时，子类还没建立
- 不要再析构函数里调用虚函数
  - 在基类的析构函数里调用虚函数时，子类已经销毁
- 虚函数有额外开销，不必要所有的函数都成为虚函数

虚析构函数(virual destructor)

- 对于基类的析构函数，最好用virtual
- 要不然编译器会用静态绑定，容易造成内存泄漏

```
Snow* var4 = new Rain();

Delete var4;
```

- 赋值运算符重载，可以变成虚函数，不过最好不要。

抽象基类和纯虚函数

- 纯虚函数是指子类必须实现基类的虚函数
- 抽象基类包括至少一个纯虚函数:

```
virtual void Y() = 0;
```

- 这里 = 0 表示了纯虚函数
- 基类不能对纯虚函数定义
- 基类可以没有对象
- 抽象基类一般用来作为接口(interface)

```
class Shape {
 public:
    virtual void draw() = 0;
    virtual Point center() = 0;
    // …
};
```

Override标识符

```
class A
{
public:
    virtual const char* getName1(int x) { return "A"; }
    virtual const char* getName2(int x) { return "A"; }
};

class B : public A
{
public:
    // note: parameter is a short int
    virtual const char* getName1(short int x) { return "B"; }
    // note: function is const
    virtual const char* getName2(int x) const { return "B"; }
};

int main()
{
    B b;
    A &rBase = b;
    std::cout << rBase.getName1(1) << '\n';
    std::cout << rBase.getName2(2) << '\n';

    return 0;
}
```

编译通过

问题是如果本意是想动态绑定，变成隐蔽错误

```
class A
{
public:
    virtual const char* getName1(int x) { return "A"; }
    virtual const char* getName2(int x) { return "A"; }
    virtual const char* getName3(int x) { return "A"; }
};

class B : public A
{
public:
    // compile error, function is not an override
    virtual const char* getName1(short int x) override { return "B"; }
    // compile error, function is not an override
    virtual const char* getName2(int x) const override { return "B"; }
    // okay, function is an override of A::getName3(int)
    virtual const char* getName3(int x) override { return "B"; }

};
```

对于动态绑定，最好都加上override

Object Slicing

```
int main()
{
    Derived derived(5);
    std::cout << "derived is a " << derived.getName() << '\n';

    Base &ref = derived;
    std::cout << "ref is a " << ref.getName() << '\n';

    Base *ptr = &derived;
    std::cout << "ptr is a " << ptr->getName() << 'n';

    return 0;
}
```

```
class Base
{
protected:
    int m_value;

public:
    Base(int value)
        : m_value(value)
    {
    }

    virtual const char* getName() const { return "Base"; }
    int getValue() const { return m_value; }
};

class Derived: public Base
{
public:
    Derived(int value)
        : Base(value)
    {
    }

    virtual const char* getName() const { return "Derived"; }
};
```

Object Slicing （1）

```cpp
int main()
{
    Derived derived(5);
    std::cout << "derived is a " << derived.getName() << '\n';

    Base &ref = derived;
    std::cout << "ref is a " << ref.getName() << '\n';

    Base *ptr = &derived;
    std::cout << "ptr is a " << ptr->getName() << 'n';

    // what happens here?
    Base base = derived;
    std::cout << "base is a " << base.getName();

    return 0;
}
```

```cpp
class Base
{
protected:
    int m_value;

public:
    Base(int value)
        : m_value(value)
    {
    }

    virtual const char* getName() const { return "Base"; }
    int getValue() const { return m_value; }
};

class Derived: public Base
{
public:
    Derived(int value)
        : Base(value)
    {
    }

    virtual const char* getName() const { return "Derived"; }
};
```

- 调用复制构造函数，生成Base对象；
- 变量base不再是derived class；
- 这称为**Object slicing**

Object Slicing （2）

```
// note: base passed by reference now
void printName(const Base & base)
{
    std::cout << "I am a " << base.getName() << '\n';
}

int main()
{
    Derived d(5);

    // oops, didn't realize this was pass by value
    printName(d);

    return 0;
}
```

```
class Base
{
protected:
    int m_value;

public:
    Base(int value)
        : m_value(value)
    {
    }

    virtual const char* getName() const { return "Base"; }
    int getValue() const { return m_value; }
};

class Derived: public Base
{
public:
    Derived(int value)
        : Base(value)
    {
    }

    virtual const char* getName() const { return "Derived"; }
};
```

- 同样，形参base不是derived class，即使实参d是derived class；
- 这称为**Object slicing**

Slicing vector （3）

```
#include <vector>
int main()
{
    std::vector<Base> v;
    // add a Base object to our vector
    v.push_back(Base(5));
    // add a Derived object to our vector
    v.push_back(Derived(6));

    // Print out all of the elements in our vector
    for (int i = 0; i < v.size(); ++i)
    {
        std::cout << "I am a " << v[i].getName() << "\n";
    }
    return 0;
}
```

```
#include <vector>
int main()
{
    std::vector<Base *> v;
    // add a Base object to our vector
    v.push_back(new Base(5));
    // add a Derived object to our vector
    v.push_back(new Derived(6));

    // Print out all of the elements in our vector
    for (int i = 0; i < v.size(); ++i)
    {
        std::cout << "I am a " << v[i].getName() << "\n";
    }

    for (int i = 0; i < v.size(); ++i)
    {
        delete v[i];
    }
    return 0;
}
```

- 同样，容器元素v[i]都是base class，即使有的元素输入时是derived class；
- 这称为**Object slicing**

小结（重复利用代码的手段）

- 类class本身目的是为了更好地重利用代码
  - 对象有公共点（关系），也有自己的特色
  - 可以组合，也可以继承
- 函数和算术符重载**overload**
  - 重载是函数名一样，但是参数列表不一样，静态绑定
- 基类函数重定义**re-definition**
  - 是子类定义了和父类一样的函数(包括函数名和参数列表），静态绑定
- 基类函数覆盖**override**
  - 是指基类带有**virtual**的函数会被覆盖，属于动态绑定
  - 实现多态polymorphism的关键

改善下面代码

```
class B;
class A
{
public:
    A(int sz) { sz_ = sz; v = new B[sz_]; }
    ~A() { delete v; }
    // ...
private:
    // ...
    B * v;
    int sz_;
};
```

```
class B;
class A
{
public:
    A(int sz) { sz_ = sz; v = new B[sz_]; }
    ~A() { delete [] v; }
    // ...
private:
    // ...
    B * v;
    int sz_;
};
```

基类才需要改成虚函数

```
class B;
class A
{
public:
    A(int sz) { sz_ = sz; v = new B[sz_]; }
    virtual ~A() { delete [] v; }
    // ...
private:
    // ...
    B * v;
    int sz_;
};
```

改善下面代码

```
class B;
class A
{
public:
    A(int sz) { sz_ = sz; v = new B[sz_]; }
    ~A() { delete [] v; }
    // ...
private:
    // ...
    B * v;
    int sz_;
};
```

```
class B;
class A
{
public:
    A(int sz) : sz_(sz), v(new B[sz_]) { }
    ~A() { delete [] v; }
    // ...
private:
    // ...
    B * v;
    int sz_;
};
```

```
class B;
class A
{
public:
    A(int sz) : sz_(sz), v(new B[sz]) { }
    ~A() { delete [] v; }
    // ...
private:
    // ...
    B * v;
    int sz_;
};
```

```
class B;
class A
{
public:
    A(int sz) : v(new B[sz]), sz_(sz) { }
    ~A() { delete [] v; }
    // ...
private:
    // ...
    B * v;
    int sz_;
};
```

Q&A

Thanks!