C++复习

目录

一、C++语言概述

- C++语言的特点
- C++语言对C语言扩充和增强的具体体现
- C++的优点

C++语言的特点

- 全面兼容**C**

  –它保持了**C**的简洁、高效和接近汇编语言等特点；

  –对**C**的类型系统进行了改革和扩充；

  –**C++**也支持面向过程的程序设计，不是一个纯正的面向对象的语言；
- **支持面向对象的方法**

C++语言对C语言扩充和增强的具体体现

- 注释

  在C语言块注释的形式/*Explanation Sentence*/的基础上，C + + 语言提供了一种新的单行注释形式：//Explanation Sentence，即用“//”表示注释开始，从该位置直到当前行结束的所有字符都被作为注释。

- 更加灵活的变量说明

  在传统的C语言中，局部变量的说明必须集中放在执行代码的前面，数据说明语句和执行语句的混合将引起编译错误。而在C++中，可以在程序代码块的任何地方进行局部变量的说明。比如下面的代码在C语言中是不正确的，在C++语言中却可以正常运行。for(int i = 1; i < = 100; i++);

C++语言对C语言扩充和增强的具体体现

- 更加严格的函数原型说明

  C++摒弃了C语言对函数原型随意简化的方式，C++语言要求编程者为函数提供完整的原型，包括全部参数的类型和返回值得说明。

  例如，有字符型和双精度类型两个参数、返回整型值的函数f，原型应该写为：int f(char, double);

  而C语言中允许将这个原型写成"f( )；"。在函数原型说明中，参数名可有可无，并且可以和函数定义中的参数名不一致。

C++语言对C语言扩充和增强的具体体现

- 增加了函数重载机制

  重载是程序语言领域的重要概念。常规语言中最典型的例子是"+、－、×、/"等各种算术运算符的重载，这些符号可以同时用来表示多种类型数据之间的运算，这种对一个名字或一个符号赋予多重意义的情况就叫重载。

  C++语言增加了C语言所没有的函数重载机制。对一个函数名可以给出多个函数定义，只要这些定义可以通过参数个数或类型的不同区别开来即可。

  C++还允许对系统中预先定义的运算符号进行重载，增加新的定义。这样做的优点是在今后对新定义类型的变量进行运算时，计算公式写起来方便自然。

C++语言对C语言扩充和增强的具体体现

- 函数缺省参数

  C++中允许函数有缺省参数。所谓缺省，是指函数调用时可以不给出实际的参数值。有缺省参数的函数定义的实例：

```
int f(int a, int b=1)
{
    return a*b;
}
```

  此后，函数调用f(3,1)和f(3)将返回同样的结果。

- 更加方便的动态存储分配

  C++为了提高内存管理上的灵活性，提供了动态内存分配合释放的操作符new和delete，用来增强C语言中原有的函数malloc()和free();

C++语言对C语言扩充和增强的具体体现

- 增加了内联函数（Inline Function）

  C++提供了内联函数，用以代替C语言中的宏。宏的处理机构是预处理器而不是编译器，它虽然可以提高效率，但是却不能实现函数调用所拥有的参数类型检查等机制。内联函数不但能够象宏那样节约函数调用时保存现场所需的系统开销，提高程序执行效率，还保留了函数进行参数类型检查的机制；并且C++语言中的宏是不能够存取对象私有成员变量的，但是使用内联函数，则没有这一限制。

C++语言对C语言扩充和增强的具体体现

- 输入/输出流机制

  C + + 保留了C语言标准库中各种输入/输出函数，而且提供了一套新的输入/输出机制——流机制。

  比如向标准输出输出一个字符串：

  ```
  cout<<"C++ is beautiful!";
  ```

  或者由标准输入读一个整数，赋给变量a

  ```
  int a;
  cin>>a;
  ```

  流式输入/输出运算符能够根据变量类型自动确定数据交换过程中的转换方式，还可以定义"<<、>>"的重载，方便了编程者自定义类型的数据的输入/输出。

C++语言对C语言扩充和增强的具体体现

- 作用域限定运算符：：
  作用域限定运算符::，用于对当前作用域之外的同名变量进行访问。例如在下面的例子中，我们可以利用::实现在局部变量a的作用域范围内对全局变量a的访问。

```
#include <iostream.h>
int a;
void main()
{
    float a;
    a = 3.14;
    ::a = 6;
    cout<<"local variable a = "<<a<<endl;
    cout<<"global variable a ="<<::a<<endl;
}
```

程序执行结果如下：

```
local variable a = 3.14
global variable a = 6
```

C++的优点

- 与C语言兼容，既支持面向对象的程序设计，也支持结构化的程序设计。同时，熟悉C语言的程序员，能够迅速掌握C++语言。
- 修补了C语言中的一些漏洞，提供更好的类型检查和编译时的分析。使得程序员在C++环境下继续写C代码，也能得到直接的好处。
- 生成目标程序质量高，程序执行效率高。一般来说，用面向对象的C++编写的程序执行速度与C语言程序不相上下。

C++的优点

- 提供了异常处理机制，简化了程序的出错处理。利用throw、try和catch关键字，出错处理程序不必与正常的代码紧密结合，提高了程序的可靠性和可读性。
- 函数可以重载及可以使用缺省参数。重载允许相同的函数名具有不同参数表，系统根据参数的个数和类型匹配相应的函数。缺省参数可以使得程序员能够以不同的方法调用同一个函数，并自动对某些缺省参数提供缺省值。

C++的优点

- 提供了模板机制。模板包括类模板和函数模板两种，它们将数据类型作为参数。对于具体数据类型，编译器自动生成模板类或模板函数，它提供了源代码复用的一种手段。

二、类和对象

- 类与对象概述
- 构造函数与析构函数
- 拷贝构造函数
- 类作用域
- const成员函数
- 静态成员

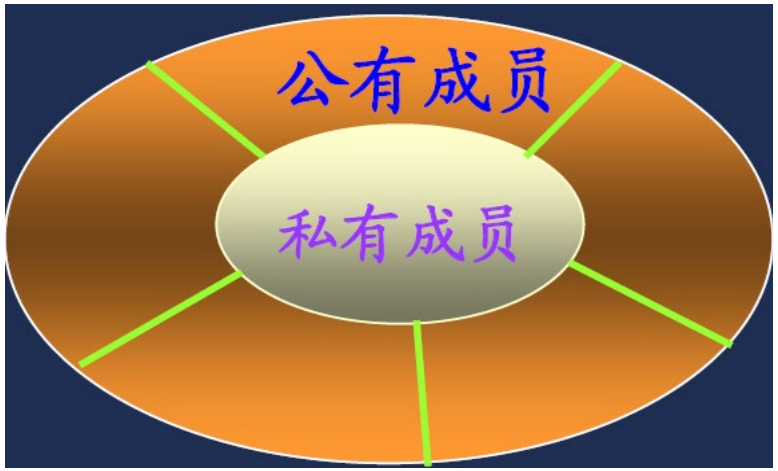
类与对象概述

- 类是具有相同属性和行为的一组对象的集合，它为属于该类的全部对象提供了统一的抽象描述，其内部包括属性和行为两个部分。
- 利用类可以实现数据的封装、隐藏、继承与派生。
- 利用类易于编写大型复杂程序，其模块化程度比**C**中采用函数更高。

类与对象概述

- 类的声明形式

```
class 类名称
{
    public:公有成员（外部接口）
    private:私有成员
    protected:保护型成员
};
```



程序模型则是若干个对象的相互作用——一堆“鸡蛋”

类与对象概述

- 公有属性成员

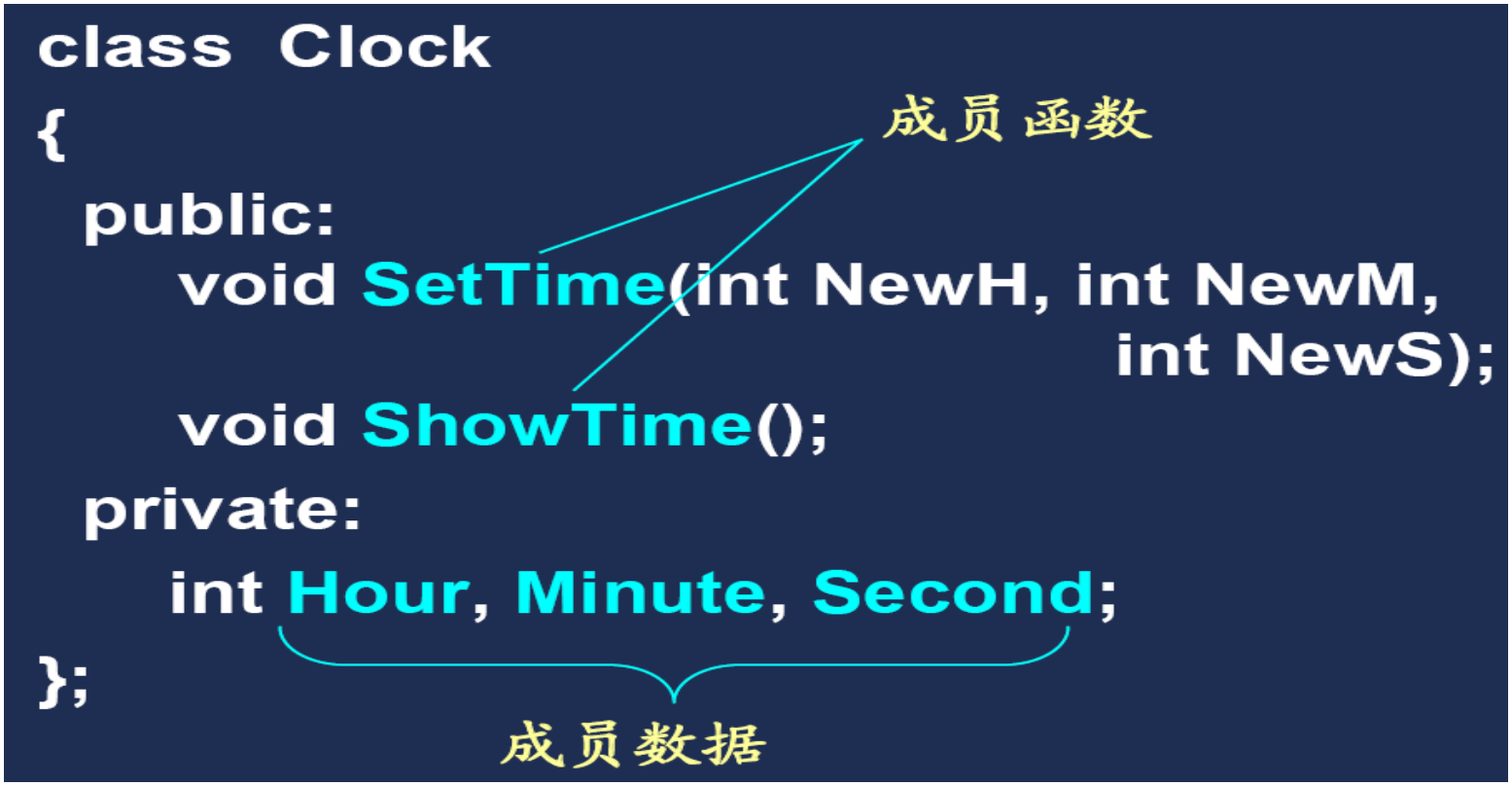
  在关键字**public**后面声明，它们是类与外部的接口，任何外部函数都可以访问公有属性的数据和函数。通俗讲是“谁都可以使用”。
- 私有属性成员

  在关键字**private**后面声明，只允许本类中的函数访问，而类外部的任何函数都不能访问。通俗讲是“谁都不可以使用，只供我自己用”。
- 保护属性成员

  在关键字**protected**后面声明，允许本类及子类中的函数访问，通俗讲是“我的子孙可以使用，外人不可用”。

类与对象概述

- 类的成员


class Clock
{
public:
void SetTime(int NewH, int NewM,
int NewS);
void ShowTime();
private:
int Hour, Minute, Second;
};
成员函数
成员数据

类与对象概述

- 成员数据
- 与一般的变量声明相同，但需要将它放在类的声明体中。
- 可放置在类中的任何位置；
- 不可声明时初始化；
- 不可声明为**extern**、**register,**但可以是**static** 或**const**。

类与对象概述

- 成员函数
- 可以仅在类中说明原型，而在类外给出函数体实现，并在函数名前使用类名加以限定。
- 也可以直接在类中给出函数体，形成成员函数的隐含内联。
- 允许成员函数为重载函数和带默认形参值的函数。
- 对成员数据的使用不再遵循“先声明后使用”的原则，即可以放置在类中任意位置。
- 凡被调用的成员函数一定要有函数实现。

类与对象概述

- 内联成员函数
- 为了提高运行时的效率，对于较简单的函数可以声明为内联形式。
- 内联函数体中不要含有复杂结构（如循环语句和**switch**语句）。
- 在类中声明内联成员函数的方式:

  –将函数体放在类的声明中——隐式声明。

  –使用**inline**关键字——显式声明。

const成员函数

- const成员函数定义

```
class Point
{
public:
    int GetX() const;
    int GetY() const;
    void SetPt (int, int);
    void OffsetPt (int, int);
private:
    int xVal, yVal;
};
```

```
int Point::GetY() const
{
   return yVal;
}
```

const成员函数应该在函数原型说明和函数定义中都增加const限定 。

静态成员

- 静态数据成员

  要定义静态数据成员，只要在数据成员的定义前增加static关键字。静态数据成员不同于非静态的数据成员，一个类的静态数据成员仅创建和初始化一次，且在程序开始执行的时候创建，然后被该类的所有对象共享；而非静态的数据成员则随着对象的创建而多次创建和初始化。

类与对象概述

- 对象

  类的对象是该类的某一特定实体，即类类型的变量。

  定义对象时，系统会为每一个对象分配自己的存储空间，该空间只保存数据，函数代码是所有对象共享的。

  声明形式：类名 对象名；

  例：**Clock myClock;**

构造函数与析构函数

- 构造函数的作用是在对象被创建时使用特定的值构造对象，或者说将对象初始化为一个特定的状态。
- 初始化是指对象（变量）诞生的那一刻即得到原始值的过程。是动词。对象声明然后赋值，经历了状态由不确定到确定的过程。

构造函数与析构函数

- 构造函数的特点:
  - 与类同名;
  - 无返回类型;
  - 通常是公有成员;
  - 在对象创建时由系统自动调用;
  - 如果程序中未声明，则系统自动产生出一个默认形式的构造函数，它无参，函数体为空;
  - 允许为内联函数、重载函数、带默认形参值的函数。

构造函数与析构函数

- 构造函数举例

```
class Clock
{
    public:
    Clock (int NewH, int NewM, int NewS);
    //构造函数
    void SetTime(int NewH, int NewM, int NewS);
    void ShowTime();
    private:
        int Hour, Minute, Second;
};
```

构造函数与析构函数

- 构造函数的实现：

```
Clock::Clock(int NewH, int NewM, int NewS)
{
    Hour= NewH;
    Minute= NewM;
    Second= NewS;
}
void main()
{
    Clock c (0,0,0); //隐含调用构造函数，将初始值作为实参。
    c.ShowTime();
}
```

构造函数与析构函数

- 析构函数
- 完成对象被删除前的一些清理工作（恰与构造函数对称）。
- 在对象的生存期结束的时刻系统自动调用它，然后再释放此对象所属的空间。
- 如果程序中未声明析构函数，编译器将自动产生一个默认的析构函数。
- 不接受任何参数，亦无法重载。

构造函数与析构函数

- 析构函数的调用时机
- **}** 或文件尾；
- **delete (**有条件地**)**；
- **catch()**
- 当中途退出程序时**(exit()**、**abort())**不调用析构函数

构造函数与析构函数

- 构造函数和析构函数举例

```
#include<iostream>
using namespace std;
Class Point
{
public:
    Point(int xx, int yy);
    ~Point();
    //...其它函数原型
private:
    int X, int Y;
};
```

构造函数与析构函数

- 构造函数和析构函数举例

```
Point::Point(int xx, int yy)
{
    X=xx;
    Y=yy;
}
Point::~Point()
{
}
//...其它函数的实现略
```

拷贝构造函数

拷贝构造函数是一种特殊的构造函数，其形参为本类的对象引用

```
class 类名
{
public :
    类名（形参）；//构造函数
    类名(类名&对象名)；//拷贝构造函数原型
    ...
};
类名:: 类名（类名&对象名）//函数的实现
{
    函数体
}
```

拷贝构造函数

- 拷贝构造函数定义

```
class Point
{
public:
    Point(int xx=0,int yy=0){X=xx; Y=yy;}
    Point(Point& p);
    int GetX() {return X;}
    int GetY() {return Y;}
private:
    int X, Y;
};
```

拷贝构造函数

- 拷贝构造函数定义

```
Point::Point(Point& p)
{
    X = p.X;
    Y = p.Y;
    cout << “拷贝构造函数被调用” << endl;
}
```

拷贝构造函数

- 拷贝构造函数调用

当用类的一个对象去初始化该类的另一个对象时系统自动调用拷贝构造函数实现拷贝赋值。

```
void main(void)
{
    Point A(1,2);
    Point B(A); //拷贝构造函数被调用
    cout << B.GetX() << endl;
}
```

拷贝构造函数

- 拷贝构造函数调用

若函数的形参为类对象，调用函数时，实参赋值给形参，系统自动调用拷贝构造函数。例如：

```
void fun1(Point p)
{
    cout<<p.GetX()<<endl;
}
void main()
{
    Point A(1,2);
    fun1(A); //调用拷贝构造函数
}
```

拷贝构造函数

- 拷贝构造函数调用

当函数的返回值是类对象时，系统自动调用拷贝构造函数。例如：

```
Point fun2()
{
    Point A(1,2);//调用拷贝构造函数
    return A;
}
void main()
{
    Point B=fun2();
}
```

三、继承与派生

- 类的继承与派生
- 类成员的访问控制
- 单继承与多继承
- 派生类的构造、析构函数
- 类成员的标识与访问

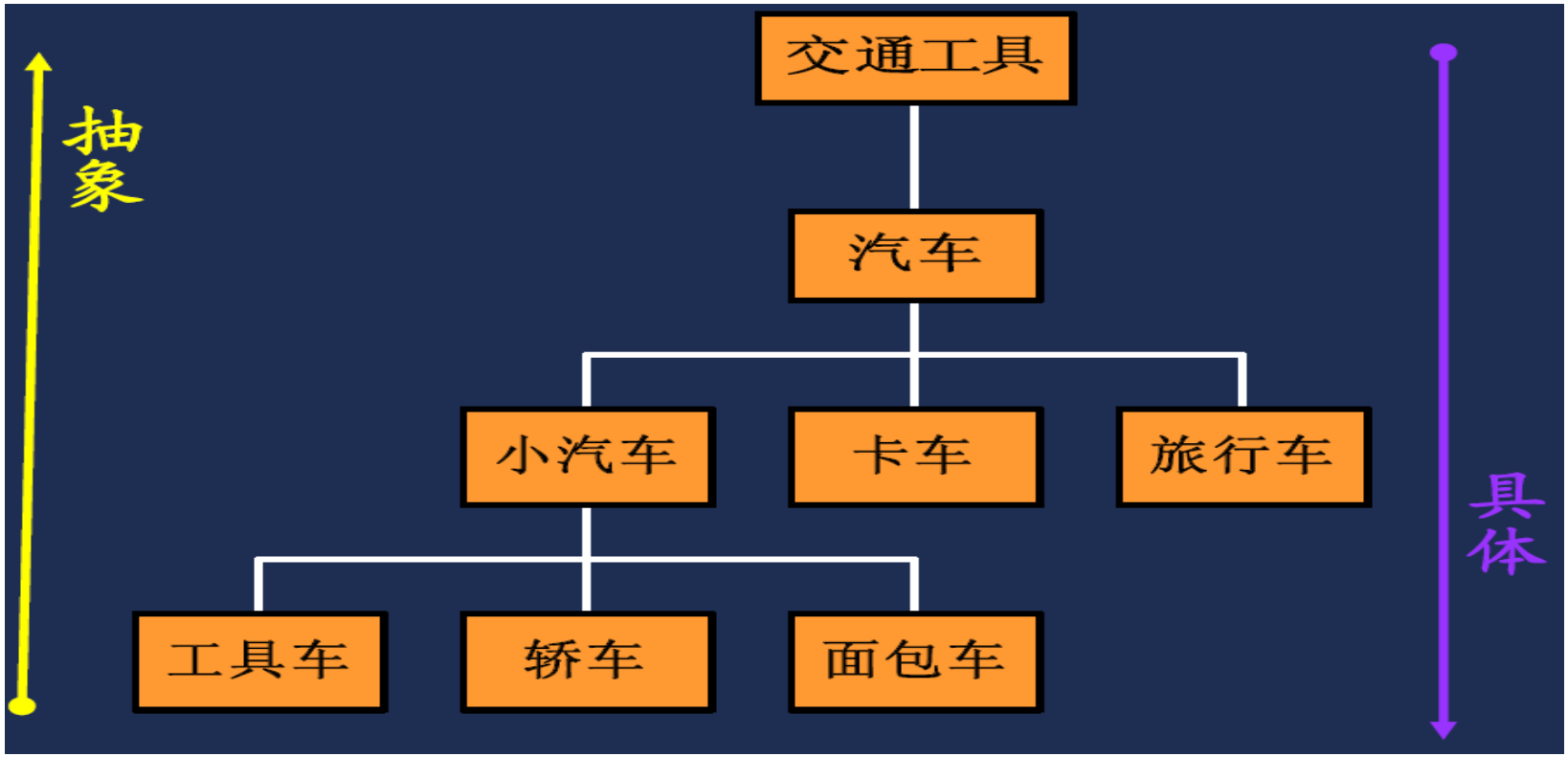
类的继承与派生

- 类的继承与派生
- 保持已有类的特性而构造新类的过程称为继承。
- 在已有类的基础上新增一些特性而产生新类的过程称为派生。

  继承和派生叫法不同，是一件事物的两个方面。
- 被继承的已有类称为基类（或父类）。
- 派生出的新类称为派生类（或子类）。

类的继承与派生

- 继承与派生问题举例



类的继承与派生

- 继承与派生的目的
- 继承的目的：实现代码重用。
- 派生的目的：当新的问题出现，原有程序无法解决（或不能完全解决）时，需要对原有程序进行改造。
- 继承为软件的层次化开发提供了保证。

类的继承与派生

- 派生类的声明

class 派生类名：继承方式 基类名
{
　　新增成员声明;
};

- 继承方式

➢ 三种继承方式
　–公有继承**public** （原封不动）
　–保护继承**protected** （折中）
　–私有继承**private** （化公为私）

➢ 继承方式影响子类的访问权限
　–派生类成员对基类成员的访问权限
　–通过派生类对象对基类成员的访问权限

类成员的访问控制

- 公有继承**(public)**
- 基类的**public**和**protected**成员的访问属性在派生类中保持不变，但基类的**private**成员不可直接访问。
- 派生类中的成员函数可以直接访问基类中的**public**和**protected**成员，但不能直接访问基类的**private**成员。
- 通过派生类的对象只能访问基类的**public**成员。

类成员的访问控制

- 私有继承**(private)**
- 基类的**public**和**protected**成员都以**private**身份出现在派生类中，但基类的**private**成员不可直接访问。
- 派生类中的成员函数可以直接访问基类中的**public**和**protected**成员，但不能直接访问基类的**private**成员。
- 通过派生类的对象不能直接访问基类中的任何成员。

类成员的访问控制

- 保护继承**(protected)**
- 基类的**public**和**protected**成员都以**protected**身份出现在派生类中，但基类的**private**成员不可直接访问。
- 派生类中的成员函数可以直接访问基类中的**public**和**protected**成员，但不能直接访问基类的**private**成员。
- 通过派生类的对象不能直接访问基类中的任何成员

类成员的访问控制

- 三种继承方式的选择
- 若想完全保留基类的操作功能，只是扩展新功能，则用公有继承。这叫“类型继承”。
- 若想完全改变基类的功能，将其改头换面，做成基于原来类，但隐藏、伪装了原功能，则用私有继承。只能算“实现继承”。
- 若想既隐藏基类的操作功能，又能方便的传给后代，不至于难以访问，则用保护继承。也是“实现继承”。

类的继承与派生

- 继承方式与访问权限的区别
- 继承方式是表示类间关系，即从大的方面控制类的所有成员，表现了类的继承性。
- 访问权限仅是在类内控制其成员，表现了类的封装性。

单继承与多继承

- 基类与派生类的关系
- 单继承
  –派生类只从一个基类派生。
- 多继承
  –派生类从多个基类派生。
- 多重派生
  –由一个基类派生出多个不同的派生类。
- 多层派生
  –派生类又作为基类，继续派生新的类。

单继承与多继承

- 单继承举例

```
class B
{
public:
    ...
private:
    int b;
};
```

```
class D :public B
{
public:
        ...
private:
        int d;
};
D objd;
B * pb= &objd;
D * pd = &objd;
```

单继承与多继承

- 多继承时派生类的声明

class 派生类名：继承方式**1** 基类名**1**，
　　继承方式**2** 基类名**2**，**...**
{
　　新增成员声明;
};

注意：每一个“继承方式”，只用于控制紧随其后之基类的继承。

默认的继承方式是私有继承。

单继承与多继承

- 多继承举例

```
class A
{
public:
    void setA(int);
    void showA( );
private:
    int a;
};
class B{
public:
    void setB(int);
    void showB( );
private:
     int b;
};
class C : public A, private B
{
public:
    void setC(int,  int, int);
    void showC();
private:
    int c;
};
```

单继承与多继承

- 多继承举例

```
void A::setA(int x)
{ a=x; }
void B::setB(int x)
{ b=x; }
void C::setC(int x, int y, int z)
{ //派生类成员直接访问基类
    的公有成员
    setA(x);
    setB(y);
    c=z;
}
//其它函数实现略
```

```
void main()
{
      C obj;
      obj.setA(5);
      obj.showA();
      obj.setC(6,7,9);
      obj.showC();
      // obj.setB(6); 错误
      // obj.showB(); 错误
}
```

派生类的构造、析构函数

- 继承时的构造函数
- 基类的构造函数不被继承，派生类中需要重新定义自己的构造函数。
- 定义构造函数时，只需要对本类中新增成员进行初始化，继承来的基类成员的初始化工作由基类构造函数完成，基类构造函数是自动被调用的，但对有参的要用初始化表给出实参。
- 派生类的构造函数要用参数总表备齐实参，以便于传递给基类的构造函数所用。

派生类的构造、析构函数

- 单继承时构造函数的形式

派生类名::派生类名(基类所需的形参，本类成员所需的形参): 基类名(参数表)

{

本类成员赋初值语句;

}

派生类的构造、析构函数

- 单继承时的构造函数举例

```
#include<iostream>
using namespace std;
class B{ //声明父类
public:
    B();
    B(int i);
    ~B();
    void Print() const;
private:
    int b;
};
```

```
//父类成员函数的实现，也叫“类实现”。
B::B()
{    b=0;
     cout<<"B's default
        constructor called."<<endl;
}
B::B(inti)
{    b=i;
     cout<<"B's constructor
called." <<endl;
}
B::~B()
{    cout<<"B's destructor
called."<<endl; }
void B::Print() const
{
     cout<<b<<endl;
}
```

派生类的构造、析构函数

- 单继承时的构造函数举例

```
//声明子类
class C:public B
{
public:
    C();
    C(int i,int j);
    ~C();
    void Print() const;
private:
    int c;
};
```

派生类的构造、析构函数

- 单继承时的构造函数举例

```
C::C()
{
    c=0;
    cout<<"C's default
    constructor
    called."<<endl;
}
C::C(int i,int j) : B(i)
{
    c=j;
    cout<<"C's constructor
    called."<<endl;
}
```

```
C::~C()
{
    cout<<“C‘s destructor
        called.”<<endl;
}
void C::Print() const
{
    B::Print();
    cout<<c<<endl;
}
void main()
{
    C obj1(5,6);
    obj1.Print();
    C obj2;
    obj2.Print();
}
```

派生类的构造、析构函数

- 派生类与基类构造函数的关系
- 当基类中未声明任何构造函数或声明了无参构造函数时，派生类构造函数可以不向基类构造函数传递参数。
- 若基类中未声明构造函数，派生类中也可以不声明，全采用缺省形式构造函数。
- 当基类声明有带参构造函数时，派生类也必须声明带参构造函数，并将参数传递给基类构造函数。

派生类的构造、析构函数

- 构造函数的调用次序
- 首先调用基类构造函数，调用顺序按照它们被继承时声明的顺序（从左向右）。
- 然后调用成员对象的构造函数，调用顺序按照它们在类中声明的顺序。
- 最后，执行派生类的构造函数体中的语句。

派生类的构造、析构函数

- 继承时的析构函数
- 析构函数也不被继承，派生类自行声明
- 声明方法与一般（无继承关系时）类的析构函数相同。

- 不需要显式地调用基类的析构函数，系统会自动隐式调用
- 析构函数的调用次序与构造函数相反。

派生类的构造、析构函数

- 派生类析构函数举例

```
#include <iostream>
using namespace std;
class B1//基类B1声明
{
public:
    B1(int i) {cout<<"constructing B1 "<<i<<endl;}
    ~B1() {cout<<"destructing B1 "<<endl;}
};
class B2//基类B2声明
{
public:
    B2(int j) {cout<<"constructing B2 "<<j<<endl;}
    ~B2() {cout<<"destructing B2 "<<endl;}
};
class B3//基类B3声明
{
public:
    B3(){cout<<"constructing B3 *"<<endl;}
    ~B3() {cout<<"destructing B3 "<<endl;}
};
```

派生类的构造、析构函数

- 派生类析构函数举例

```
class C: public B2, public B1, public B3
{
public:
    C(int a, int b, int c, int d):
    B2(a), memberB2(d), memberB1(c), B1(b) { }
private:
    B1 memberB1;
    B2 memberB2;
    B3 memberB3;
};
void main()
{
    C obj(1, 2, 3, 4);
}
```

派生类的构造、析构函数

- 派生类析构函数举例

constructing B2 1
constructing B1 2
constructing B3 *
constructing B1 3
constructing B2 4
constructing B3 *
destructing B3
destructing B2
destructing B1
destructing B3
destructing B1
destructing B2

类成员的标识与访问

- 二义性问题
- 在继承时，基类之间、或基类与派生类之间发生成员同名时，将出现对成员访问的不确定性——同名二义性。
- 当派生类从多个基类派生，而这些基类又从同一个基类派生，则在访问此共同基类中的成员时，将产生另一种不确定性——路径二义性。

类成员的标识与访问

- 同名隐藏的原理

  尽管基类与派生类在声明时是并列的：可逻辑语义上是嵌套的：大概念包含了小概念。

  自然地，在子类的地盘上，子类的标识符会屏蔽父类的同名标识符。

  若想启用（“捞出来”），用::

类成员的标识与访问

- 解决同名二义的方法

  当派生类与基类有同名成员时,派生类中的成员将屏蔽基类中的同名成员。

  若未特别指明，则通过派生类对象使用的都是派生类中的同名成员;

  如要通过派生类对象访问基类中被屏蔽的同名成员，应使用基类名限定**(::)**。

类成员的标识与访问

- 二义性问题举例

```
class A
{
public:
    void f();
};
class B
{
public:
    void f();
    void g()
};
```

```
class C: public A, public B
{
public:
    void g();
    void h();
};
```

如果声明：**C c1;**

则**c1.f();** 具有二义性

而**c1.g();** 无二义性（同名覆盖）

类成员的标识与访问

- 解决同名二义的方法
- 解决方法一：用类名来限定**c1.A::f()** 或**c1.B::f()**
- 解决方法二：同名覆盖，再造接口在**C** 中再声明一个同名成员函数**f()**，该函数根据需要调用**A::f()** 或**B::f()**

四、多态性

- 多态性
- 虚函数
- 纯虚函数与抽象类

多态性

- 多态性的概念

（这里讲的多态性是狭义的，仅指动态多态。广义的多态应包括静态多态。）

- 动态多态性是指发出同样的消息被不同类型的对象接收时有可能导致完全不同的行为。即，在用户不作任何干预的环境下，类的成员函数的行为能根据调用它的对象类型自动作出调整。
- 多态性是面向对象程序设计的重要特征之一。
- 是扩充性在“继承”之后的又一大表现。

多态性

- 多态性的实现
- 多态的实现可分为编译时多态和运行时多态，它们分别对应静态联编和动态联编。
- 联编又称为绑定（**binding**），是指计算机程序中的语法元素（标识符、函数等）彼此相关联的过程。
- 从绑定的时机看，在编译时就完成的绑定叫静态绑定；直到运行时才能确定并完成的绑定叫动态绑定。
- 静态绑定消耗编译时间，动态绑定消耗运行时间。
- 静态绑定的程序到了运行阶段其功能就固定了，即使情况发生了变化，功能无法改变。
- 动态绑定的程序由于绑定发生在运行阶段，其功能是未定的，当情况变化了，功能也跟着变。于是表现出会聪明的判断及具有灵活的行为。

多态性

- 静态绑定与动态绑定

➢ 绑定

–程序自身彼此关联的过程，确定程序中的操作调用与执行该操作的代码间的关系。

➢ 静态绑定（静态联编）

–联编工作出现在编译阶段，用对象名或者类名来限定要调用的函数。

➢ 动态绑定

–联编工作在程序运行时执行，在程序运行时才确定将要调用的函数。

虚函数

- 虚函数
- 虚函数是动态绑定的技术基础。
- 是非静态的成员函数。
- 在类的声明中，在函数原型之前写**virtual**
- **virtual**只用来说明类声明中的原型，不能用在函数实现时。
- 具有继承性，基类中声明了虚函数，派生类中无论是否说明，同原型函数都自动为虚函数。
- 本质：不是重载**(overload)**而是覆盖**(override)**。
- 调用方式：通过基类指针或引用，执行时会根据指针指向的对象的类型，决定调用哪个类的函数。

虚函数

- 虚函数的实现机制
- 编译器发现某类含有虚函数，则对其生成的对象悄悄地加入一个**void** 型的指向指针的指针**vptr**，并让其指向一个虚函数表**vtable**（其实是个指针数组），每个表项是一个虚函数名，排列次序按虚函数声明的次序排列。

- 在类族中，无论是基类还是派生类，都拥有各自的**vptr**和**vtable**。相同类型所生成的对象拥有相同的**vtable**。

- 派生类新增的虚函数依次排在后面。当然，派生类的**vtable**表项中放的是新的覆盖函数的首址。

虚函数

- 虚函数举例

```
#include <iostream>
using namespace std;
class B0//基类B0声明
{
public://外部接口
    virtual void display() //
    虚成员函数
    {
        cout<<"B0::displ
        ay()"<<endl;
    }
};
class B1: public B0//公有派生
{
public:
    void display()
    {
    cout<<"B1::display()"<<endl; }
};
class D1: public B1//公有派生
{
public:
    void display() {
    cout<<"D1::display()"<<endl;}
};
```

虚函数

- 虚函数举例

```
void fun(B0 *ptr)//普通函数
{
    ptr->display();
}
void main()//主函数
{
    B0 b0, *p;//声明基类对象和指针
    B1 b1;//声明派生类对象
    D1 d1;//声明派生类对象
    p = &b0;
    fun(p);//调用基类B0函数成员
    p = &b1;
    fun(p);//调用派生类B1函数成员
    p = &d1;
    fun(p);//调用派生类D1函数成员
}
```

运行结果：

```
B0::display()

B1::display()

D1::display()
```

虚函数

- 动态多态的前提
  - 必须有继承产生的类族；
  - 必须是公有继承；
  - 派生类的成员函数要重写该虚函数；
  - 基类的某成员函数使用了**virtual;**
  - 派生类的对象要使用指针或引用来调用该虚函数；

虚函数

- 虚析构函数
- 为何需要虚析构函数?

  避免析构对象不彻底。
- 何时需要虚析构函数?

  当一个类含有虚函数时。
- 因为，当你打算用基类指针（或引用）删除子类对象，由于切割现象，会只释放基类部分，这会遗留内存垃圾。让基类的析构函数成为虚函数，则会彻底释放内存。

纯虚函数与抽象类

- 抽象类

  带有纯虚函数的类称为抽象类:

```
class 类名
{
public:
    virtual 类型函数名(参数表) = 0; //纯虚函数
    ...
};
```

纯虚函数与抽象类

- 抽象类

➢ 作用

–抽象类为抽象的设计目的服务。将有关的数据和行为组织在一个类中，保证其继承层次结构的派生类具有要求的行为。

–对于暂时无法实现的函数，可以声明为纯虚函数，留给派生类去实现。

➢ 注意

–抽象类通常只作为基类来使用,不能作子类。

–不能定义抽象类的对象，不能作转换的目标类型。

–构造函数不能是虚函数，析构函数可以是虚函数。

纯虚函数与抽象类

- 纯虚函数与抽象类举例

```
#include <iostream>
using namespace std;
class B0 //抽象基类B0声明
{
public: //外部接口
    virtual void display( ) = 0; //纯虚函数成员
};
class B1: public B0 //公有派生
{
public:
    void display() { cout<<“B1::display()”<<endl; } //虚成员函数
};
class D1: public B1 //公有派生
{
public:
    void display() { cout<<“D1::display()”<<endl; } //虚成员函数
};
```

纯虚函数与抽象类

- 纯虚函数与抽象类举例

```
void fun(B0 *ptr)//普通函数
{
    ptr->display();
}
void main()//主函数
{
    B0 *p;//声明抽象基类指针
    B1 b1;//声明派生类对象
    D1 d1;//声明派生类对象
    p=&b1;
    fun(p);//调用派生类B1函数成员
    p=&d1;
    fun(p);//调用派生类D1函数成员
}
```

运行结果：

```
B1::display()
D1::display()
```

五、总结

- 良好的编程习惯
- 代码的书写规范

良好的编程习惯

- 坚持对代码加适当的注释；主要集中于类、函数和算法解释；
- 始终让代码的各模块具有高内聚性，模块间具有低耦合性；一定要切断模块间的联系；
- 只让方法作有限的、简单的事，切忌大而全；
- 采用渐进式开发——设计一个功能，马上测试它，逐渐添加功能，边添边测，不要一股脑写完然后算总账；
- 假设使用者是白痴，千万不要想当然的认为他应该知道些什么！

代码的书写规范

- 不要在引用运算符(**. -> [ ] ::**)的前后加空格;
- 不要在单目运算符与操作数间加空格;
- 不要在函数名和形参表的左括号间加空格;
- 函数原型的形参表中变量名通常省略不写，若写了，则要与函数实现一致;
- 不要再使用宏,包括无参宏和有参宏;
- **C++**对结构体类型、共用体类型、枚举类型，定义变量时不再带关键字**struct union enum;**
- 不要命名以下划线为前导的标识符。