CS 1502H 程序设计思想与方法 (C++)
Chapter 11. 组合与继承

陈东尧
上海交通大学
2025年秋季

上节课回顾

const 对象

const对象

- 定义后值不能被修改的对象

const对象的定义

- const MyClass obj（参数表）；

常量成员

const数据成员

在定义时赋初值，然后在整个对象生存期中值不能被修改

常量成员的声明

在该成员声明前加上const。如

```
class  People {
        const  int birthday ;
         …
};
```

同一类的不同对象的const数据成员的值可以不同

（即：一个人的生日不会变，每个人的生日很可能不同）

静态数据成员

- **声明格式**
- 在rate前面加上static

```
class SavingAccount
{
   char  name[20];          //存户姓名
   char  addr[60];          //存户地址
   double  total;           //存款额
    static double  rate;    //利率
   …
}
```

静态数据成员注意事项

静态数据成员不属于对象的一部分，而是**类中所有对象**共享；

定义对象时，不分配静态成员的空间

静态数据成员必须单独分配空间，称静态数据成员的定义

静态成员函数

•专门处理静态数据成员、不能处理其他数据成员的函数

•静态成员函数的声明

•在类定义中的函数原型前加上保留词static

- 如在SavingAccount类中，当利率发生变化时，必须修改这个静态数据成员的值。为此可以设置一个静态的成员函数
- static void SetRate(double newRate)　{ rate = newRate; }

•特点

•没有this指针

因为静态成员函数不为对象服务，不需指明相关的对象

•无法处理类中的非静态成员

友元

- 友元是一条通往私有成员的通道

友元分类

- 全局函数（友元函数）
- 另一个类的成员函数（友元成员）
- 整个类（友元类）

友元用途

- 为某个类量身定做的一些工具
- 如电视机的遥控器：遥控器可以控制电视的内部开关，光电控制单元等等

组合与继承

- 组合
- 继承 （Inheritance）
- 多态（Polymorphism）
- 抽象类 （Abstract class）

组合

- 代码重用的一种方法
- 将对象作为类的数据成员
- 表示一种聚集关系，是一种部分和整体（is a part of）的关系
- 通常需要用初始化列表去初始化对象成员

组合实例

- 定义一个复数类，而复数的**虚部和实部都用有理数**表示，实现复数的加法和输出

  方案：

  1. 使用四个整数作为数据成员

组合实例

- 定义一个复数类，而复数的**虚部和实部都用有理数**表示，实现复数的加法和输出

  方案：

  1. 使用四个整数作为数据成员
  2. 利用已有的有理数类，数据成员是两个有理数类对象

组合实例

•定义一个复数类，而复数的**虚部和实部都用有理数**表示，实现复数的加法和输出

方案：利用已有的有理数类，数据成员是两个有理数类对象

```
class Rational {
        int num;
        int den;
        void ReductFraction();
public:
        Rational(int a = 0, int b = 1);
        void add(const Rational &r1, const Rational &r2);
        void multi(const Rational &r1, const Rational &r2);
        void display() ;
};
```

类定义

```
class Complex{
  private:
            Rational real;      //实部
            Rational imag;      //虚部

   public:
      Complex(int r1 = 0, int r2 = 1, int i1= 0, int i2 = 1) :  real(r1, r2), imag(i1, i2) { }
      Complex( Rational r1,  Rational r2 ) :  real(r1), imag(r2) { }
      void add(const  Complex  &c1,  const  Complex  &c2)
      {
          real.add(c1.real, c2.real);
          imag.add(c1.imag, c2.imag);
       }
      void display() {    real.display() ; cout << '+';   imag.display() ; cout << 'i'; }
};
```

类定义

```
class Complex{
  private:
            Rational real;      //实部
            Rational imag;      //虚部

   public:
      Complex(int r1 = 0, int r2 = 1, int i1= 0, int i2 = 1) :  real(r1, r2), imag(i1, i2) { }
      Complex( Rational r1,  Rational r2 ) :  real(r1), imag(r2) { }
      void add(const  Complex  &c1,  const  Complex  &c2)
      {
          real.add(c1.real, c2.real);
          imag.add(c1.imag, c2.imag);
       }
      void display() {    real.display() ; cout << '+';   imag.display() ; cout << 'i'; }
};
```

重用了有理数类的代码

复数类的使用

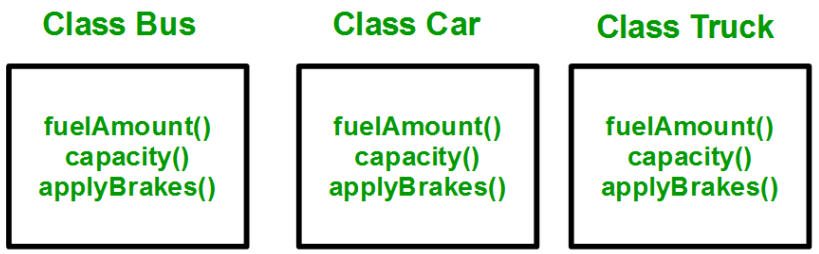
```
int main()
{
    Rational r1(1, 2),  r2(3,4);
    Complex x1(2,3,4,5)  , x2(r1, r2),   x3;

    x3.add( x1 ,  x2);
    x3.dispaly();

    return 0;
}
```

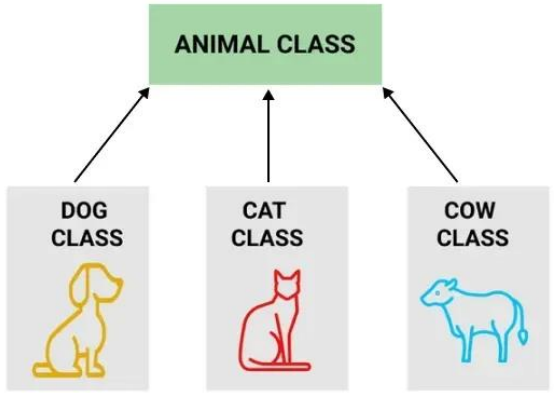
第12章 组合与继承

- 组合
- 继承 （Inheritance）
- 多态（Polymorphism）
- 抽象类 （Abstract class）

继承（Inheritance）

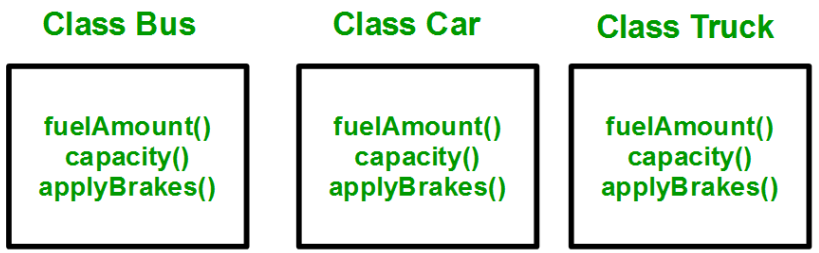
为车辆相关的类



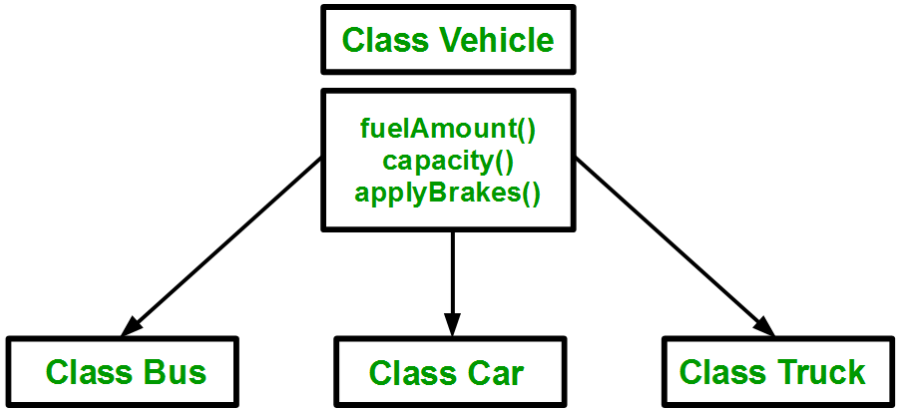


继承（Inheritance）

为车辆相关的类

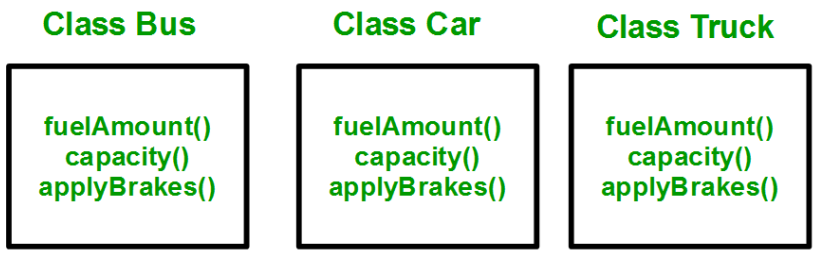


一个明显的更优方案

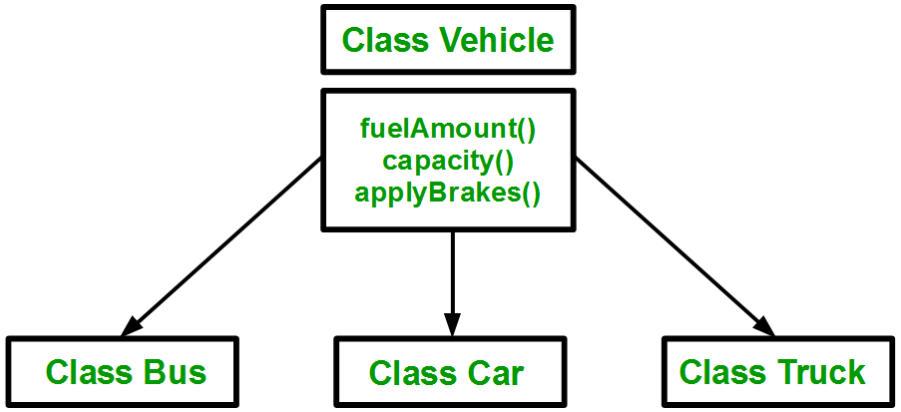


继承（Inheritance）

为车辆相关的类



一个明显的更优方案



- **基类（super class 或 base class）**：被继承的类，涵盖了基础属性
- **派生类（sub class或derived class）**：在已有类的基础上扩充属性或行为创建新的类

继承的意义

- 支持软件的**重用**，基类代码被派生类重用
- 可以反映事物之间的层次关系，基类和派生类是“一般和特殊”的关系
- 支持软件的增量开发，通过类的功能扩展整个软件的功能

派生类的定义

类定义格式

class 派生类名：**派生方法** 基类名

```
{
        //派生类新增的数据成员和成员函数
};
```

派生类的定义

类定义格式

```
class 派生类名：派生方法  基类名
{
        //派生类新增的数据成员和成员函数
};
```

派生方法：决定基类成员在派生类中的访问特性

- 派生类成员函数不能访问基类的私有成员

公有派生

public

基类的公有成员在派生类也是公有的

私有派生

private

基类的公有成员在派生类是私有的

派生实例

```
class base {
        int x;
   public:
        void setx(int k);
} ;

class derived1:public base {
        int y;
   public:
        void sety(int k);
} ;
```

派生实例

```
class base {
        int x;
   public:
        void setx(int k);
} ;

class derived1:public base {
        int y;
   public:
        void sety(int k);
} ;
```

derived1数据成员

x

y

两个成员函数

setx

sety

派生实例

```
class base {
        int x;
   public:
        void setx(int k);
} ;

class derived1:public base {
        int y;
   public:
        void sety(int k);
} ;
```

derived1数据成员

x

y

两个成员函数

setx

sety

derived1 中的访问权限

不可访问	**int x**
private	**int y**
public	**setx()** **sety()**

保护成员和保护派生

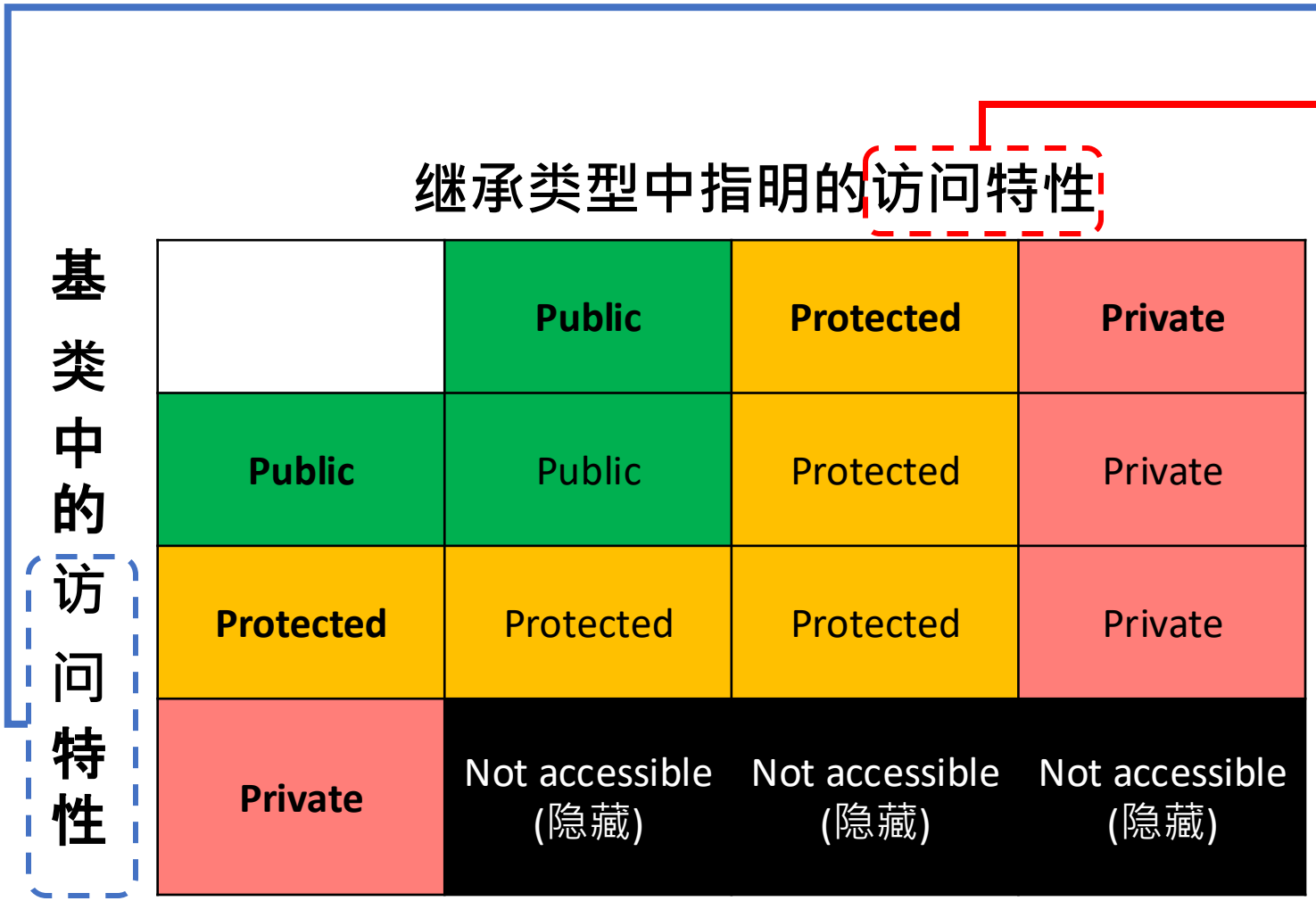
保护成员

- protected
- **一类特殊的私有成员，** 可以被**派生类的成员函数访问**

保护派生

- 基类的公有成员和保护成员在派生类中都是保护成**员**

基类中成员在继承后的访问特性




```
class base {
        int x;
    public:
        void setx(int k);
} ;

class derived1:public base {
        int y;
    public:
        void sety(int k);
} ;
```

基类中成员在继承后的访问特性

继承类型中指明的访问特性

基类中的访问特性	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (隐藏)	Not accessible (隐藏)	Not accessible (隐藏)

```
class A
{
        public:
        int x;
        protected:
        int y;
        private:
        int z;
};
class B : public A
{
// x is public
// y is protected
// z is not accessible from B
};
class C : protected A
{
// x is protected
// y is protected
// z is not accessible from C
};
class D : private A
{
// x is private
// y is private
// z is not accessible from D
};
```

派生类对象的构造和析构

派生类对象的构造

- 基类的构造函数初始化基类数据成员
- 派生类构造函数初始化派生类新增加的数据成员
- 派生类构造函数自动调用基类的构造函数

基类**无法知道自己是否被派生**或**被那个类所继承** 。
因此，继承的过程中的权限控制**总是越来越严格的**。

派生类对象的构造和析构

派生类对象的构造

- 基类的构造函数**初始化基类**数据成员
- 派生类构造函数**初始化派生类新增加的数据成员**
- 派生类构造函数**自动调用**基类的构造函数

派生类对象的析构

- 基类的析构函数**析构基类**数据成员
- 派生类析构函数析构派生类**新增加的数据成员**
- 派生类的析构函数**自动调用**基类的析构函数
- 派生类对象析构时，**先执行派生类**的析构函数，**再执行基类**的析构函数

派生类构造函数

派生类构造函数的格式

- 派生类构造函数名（参数表）： 基类构造函数名（参数表） {……}

执行过程

- 先执行**基类的构造函数**，再构造派生类新增部分

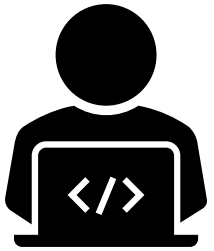
注意事项

- 基类构造函数中的参数表通常来源于派生类构造函数的参数表，也可以用常量
- 若基类使用缺省或不带参数的构造函数，则在派生类定义构造函数时可略去基类构造函数调用

派生类构造析构实例

•定义一个表示人的类People。每个人包含两个信息：姓名和年龄。在People类的基础上，派生出一个表示学生的类Student。每位学生有一个学号和班级号。观察学生类对象的构造和析构过程。

people类的定义与实现



代码演示

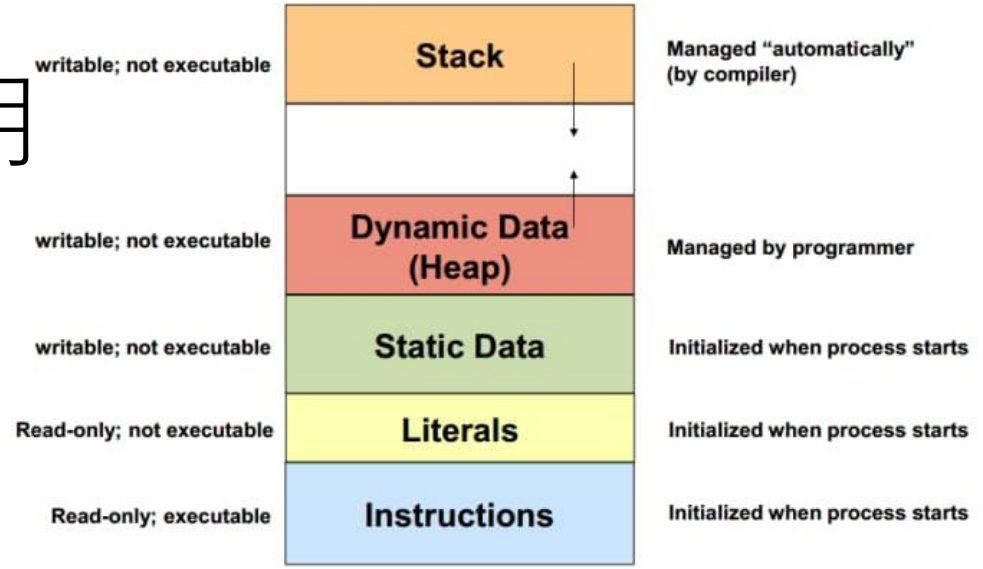
people-student.cpp

```
class People {
public:
    People( const char *s , int a)
    {
        name = new char[strlen(s) + 1];
        strcpy(name, s);
        age = a;
        cout << "People constructor:"  << '[' << name << "] age : “  << age << endl;
    }
    ~People()
    {
        cout << “People destructor: ” << ‘[’ << name << “]   age : “  << age << endl;
        delete  name;
    }
private:
      char *name;
     int age;
};
```

Student类的定义与实现

```
class Student : public People {
public:
   Student(const char *s,  int a,  int n,  char *cls):  People(s, a)
    {
         s_no = n;
         class_no = new char[strlen(cls) + 1];
        strcpy(class_no, cls);
         cout << "Student constructor: student number is " << s_no
                 << ", class number is " << class_no << endl;
   }
   ~Student( )
   {
         cout << "Student destructor:  student numberis " << s_no
                << "  ", class number is " << class_no  << endl;
      delete   class_no;
    }
private:       int s_no;
            char *class_no;
};
```

People类和Student类的使用




```
int main()
{
    {
        People p( "zhang", 100);
     }
    cout << endl;
    Student s1( "li", 10, 29, "F2403001");
    cout << endl;
    Student s2( "wang", 13, 30, "F2402008" );
    cout << endl;

    return 0;
}
```

People constructor:[zhang] age : 100
People destructor: [zhang] age : 100

People constructor:[li] age : 10
Student constructor: student number is 29, class number is F2403001

People constructor:[wang] age : 13
Student constructor: student number is 30, class number is F2402008

Student destructor: student numberis 30, class number is F2402008
People destructor: [wang] age : 13
Student destructor: student numberis 29, class number is F2403001
People destructor: [li] age : 10

重定义基类的函数

派生类扩展基类的某个功能

- 当派生类对基类的某个功能进行扩展时，他定义的成员函数名可能会和基类的成员函数名重复
- 如果只是函数名相同，而原型不同时，则派生类中有两个重载函数
- 如果原型完全相同，则派生类的函数会覆盖基类的函数
- 派生类对象看见的是重定义的函数

重定义基类的函数

某银行有两类账户

普通账户： 每个账户包括姓名、账号和余额

VIP账户：允许透支的VIP账户。还必须包括透支额度、已透支额和贷款利率。每个账户需有存款、取款、查看账户各个属性和显示账户信息的操作。VIP账户还必须有维护透支信息方面的操作。

普通账户类定义

属性

客户姓名

账号

当前结余

行为

创建账户

存款

取款

显示账户信息

```
class Account
{
private:
    char name[20];              //  账户名
    long acctNum;               //  账号
    double balance;             //  余额
public:
    Account(const char *s = "", long an = -1,  double bal = 0.0)
          : acctNum(an), balance(bal) {   strcpy(name, s);   }
    void Deposit(double amt)  { balance  += amt; }
    void Withdraw(double amt);
    double getBalance() const { return  balance; }
    const char *getName() const { return  name; }
    int getAcctNum() const { return  acctNum; }
    void ViewAcct() const;
};
```

普通账户类成员函数的实现

```
void Account::Withdraw(double amt)
{
    if (amt <= balance)
        balance -= amt;
    else
        cout << “余额不够¥n";
}

void Account::ViewAcct() const
{
    cout << "账户姓名: " << name << ‘¥t’;
    cout << "账号: " << acctNum << ‘¥t’;
    cout << "余额: " << balance << endl;
}
```

VIP账户类定义

重要账户的额外信息

透支上限

透支贷款利率

当前的透支总额

新增操作

设置透支额度

设置透支利率

清空透支总额

不同操作过程的行为

取款

显示账户信息

```
class AccountVIP : public Account
{
private:
    double maxLoan;      //  贷款额度
    double rate;         //  贷款利率
    double owesBank;     //  欠款额
public:
    AccountVIP (const char *s = "", long an = -1,double bal = 0.0
          , double ml = 500,   double r = 0.11125) : Account(s, an, bal)
     {
          maxLoan  = ml;
          rate = r;
          owesBank = 0;
     }
```

VIP账户类定义

```
    AccountVIP (const Account & ba, double ml = 500,   double r = 0.11125) :Account(ba)
    {
        maxLoan  = ml;
        rate = r;
        owesBank = 0;
    }
    void ViewAcct()const;
    void Withdraw(double amt);
    void ResetMax(double m) { maxLoan = m; }     //  修改额度
    void ResetRate(double r) { rate = r; };      // 修改利率
    void ResetOwes() { owesBank = 0; }           //  还款
};
```

VIP账户类成员函数的实现

```
void AccountVIP::ViewAcct() const
{
    cout << "账户姓名: " << getName() << '¥t';
    cout << "账号: " << getAcctNum() << '¥t';
    cout << "余额: " << getBalance() << '¥t';
    cout << "透支额度: " << maxLoan << '¥t';
    cout << "欠款总额: " << owesBank << '¥t';
    cout << "透支利率: " << 100 * rate << "%¥n";
}
```

调用基类的同名函数

- 派生类重定义的函数功能可能涵盖了基类的功能
- 是否需要重写实现基类功能的代码？

调用基类的同名函数

- 派生类重定义的函数功能可能涵盖了基类的功能
- 是否需要重写实现基类功能的代码？
- 派生类函数**可以调用基类的同名函数**
- 格式：基类::函数名

调用基类的同名函数

- 派生类重定义的函数功能可能涵盖了基类的功能
- 是否需要重写实现基类功能的代码？
- 派生类函数可以调用基类的同名函数
- 格式：基类::函数名

```
void AccountVIP::ViewAcct() const
{
   Account::ViewAcct();
   cout << "透支额度: " << maxLoan << '\t';
   cout << "欠款总额: " << owesBank << '\t';
   cout << "透支利率: " << 100 * rate << "%\n";
}
```

VIP账户类成员函数的实现

```
void AccountVIP ::Withdraw(double amt)
{
    double bal = Account::getBalance();
    if (amt <= bal) Account::Withdraw(amt);
    else if ( amt <= bal + maxLoan – owesBank)    {
        double advance = amt - bal;
        owesBank += advance * (1.0 + rate);
        cout <<   “需要透支: " << advance <<    ‘\t’  ;
        cout <<   “利息: " << advance * rate << endl;
        Deposit(advance);
        Account::Withdraw(amt);
    }
    else
        cout << "Credit limit exceeded”  ;
}
```

有没有觉得这些地方很累赘?

巧用保护成员

```
class Account
{
protected:
    char name[20];
    long acctNum;
    double balance;
public:
    Account(const char *s = "", long an = -1,  double bal = 0.0)
        : acctNum(an), balance(bal) { strcpy(name, s); }
    void Deposit(double amt)  { balance  += amt; }
    void Withdraw(double amt);
    void ViewAcct() const;
};
```

巧用保护成员

```
void AccountVIP ::Withdraw(double amt)
{
    double bal = Account::getBalance();
    if (amt <= bal)
        Account::Withdraw(amt);
    else if ( amt <= balance + maxLoan – owesBank )
        double advance = amt – bal;
        owesBank += advance * (1.0 + rate);
        cout << “需要透支: ” << advance << ‘\t’ ;
        cout << “利息: " << advance * rate << endl;
        Deposit(advance);
        Account::Withdraw(amt);
    }
    else
        cout << "Credit limit exceeded” ;
}
```

使用保护成员之前

```
void AccountVIP ::Withdraw(double amt)
{
    double bal = Account::getBalance();
    if (amt <= bal)
            Account::Withdraw(amt);
    else if ( amt <= bal + maxLoan – owesBank)    {
        double advance = amt - bal;
        owesBank += advance * (1.0 + rate);
        cout <<  “需要透支: ”  << advance <<  ‘\t’ ;
        cout <<  “利息: " << advance * rate << endl;
        Deposit(advance);
        Account::Withdraw(amt);
    }
    else
        cout << "Credit limit exceeded”;
}
```

使用保护成员之后

```
void AccountVIP ::Withdraw(double amt)
{

    if (amt <= balance)
            balance -= amt;
    else if ( amt <= balance + maxLoan – owesBank)   {
        double advance = amt – balance;
        owesBank += advance * (1.0 + rate);
        cout <<  “需要透支: ”  << advance <<  ‘\t’ ;
        cout <<  “利息: " << advance * rate << endl;
        balance += advance;
         balance -= amt;
    }
    else
        cout << "Credit limit exceeded” ;
}
```

第12章 组合与继承

- 组合
- 继承 （Inheritance）
- 多态（Polymorphism）
- 抽象类 （Abstract class）

多态性

多态性：不同对象收到相同的消息时产生不同的动作

多态性的作用：便于系统功能的扩展

多态性的实现

静态联编（编译时的多态性， Compile-time polymorphism ）

编译时已决定用哪一个函数实现某一动作

动态联编（运行时的多态性，Runtime polymorphism ）

直到运行时才决定用哪一个函数来实现动作

虚函数

虚函数

基类中的**一类成员函数**，允许函数调用与函数体之间的联系**在运行时才建立**

通过基类指针或基类引用调用基类的虚函数时，会判断指针指向的对象

- 如指向的基类对象，则执行虚函数。
- 如指向派生类对象，则检查派生类是否重定义了此函数。
- 如重定义，则执行派生类的重定义函数，否则执行基类的虚函数

虚函数的定义

声明时函数头前加关键词**virtual**

```
class base {
public:
   virtual void print() { cout << "print base class\n"; }

   void show() { cout << "show base class\n"; }
};
class derived : public base {
public:
   void print() { cout << "print derived class\n"; }
   void show() { cout << "show derived class\n"; }
};

int main()
{
   base* bptr;
   derived d;
   bptr = &d;

   // Virtual function, binded at runtime
   bptr->print();
   // Non-virtual function, binded at compile time
   bptr->show();
   return 0;
}
```

print derived class
show base class

运行时的多态性

编译时的多态性

解释：**基类 pointer ‘bptr’ 指向派生类对象 ‘d’**
在这里，**运行时的多态通过指针指向的内容**决定；编译时的多态由指针的类型决定

Reference: https://www.geeksforgeeks.org/virtual-function-cpp/

使用虚函数的注意事项

- 在派生类中重新定义虚函数时，它的原型**必须**与基类中的虚函数完全相同。否则编译器会把它认为是**重载函数**，而不是虚函数的重定义。
- 派生类在对基类的虚函数重定义时，关键字virtual可以写也可以不写。不管virtual写或者不写，该函数都被认为是虚函数。

例子

正方形是一类特殊的矩形，因此，可以从rectangle类派生一个square类

在这两个类中，都要求有一个显示形状的函数

```
class rectangle {
    int w, h;
public:
    rectangle(int ww, int hh): w(ww), h(hh) { }
    virtual void display()  { cout <<  “this is a rectangle\n” ; }
};

class square:public rectangle {
public:
    square(int ss): rectangle(ss, ss) { }
    void display()  {  cout <<  “this is a square\n” ;  }          //虚函数
};
```

显式覆盖：override

用途

让编译器**检查派生类覆盖基类的虚函数的函数原型是否相同**

方法

将关键字override放在派生类函数原型的最后。如

```
class A {
public :
    virtual  void  f1(int);
    virtual void  f2();
};
class B :public A {
public:
   void f1(int) override;      // 正确，f1覆盖了基类的f1
   void f2(int) override;     // 错误，编译器会报错
   int f3(int) override;       // 错误，编译器会报错
};
```

不允许覆盖：final

作用

将某个虚函数指定为final，表示派生类中**不允许覆盖**此函数。如

```
class A
{
    protected:
            int a;
        public:
            A(int x) {a = x;}
            virtual int f() { return a;}
};
```

```
class B : public A
{
  public:
      B(int x):A(x) {}
      int f() final {  return 2 *a;    }
};

class C : public B{
   public:
       C(int x) : B(x) {}
       int f()  {  return 3 *a;   }   //编译器报错
};
```

多态性实例：账户类和VIP账户类

```
class Account
{
    ......
public:
    virtual void Withdraw(double amt);
    virtual void ViewAcct() const;
     ......
};
```

```
class AccountVIP:  public  Account
{
    ......
public:
    void Withdraw(double amt);
    void ViewAcct() const;
     ......
};
```

```
void createAccount(Account *data[] )
{
    const char *name[5] = { "aaa", "bbb", "ccc", "ddd", "eee" };

    for ( int i = 0; i < 5; ++i){
        if ( i %2 )
            data[i] = new Account(name[i], i+10, i * 100);
        else
            data[i] = new AccountVIP(name[i],i + 10, i *100, i * 100 + 500, i * 0.01);
    }
}
```

利用多态性显示各类账户信息以及取款

```
int main()
{
    Account *data[5];
    createAccount(data);

    for ( int i = 0; i < 5; ++i)      data[i]->ViewAcct();
    cout << endl;
   for ( int i = 0; i < 5; ++i)   {
            data[i]->Withdraw(150);
            data[i]->ViewAcct();
            delete data[i];
    }
   return 0;
}
```

利用多态性显示各类账户信息以及取款

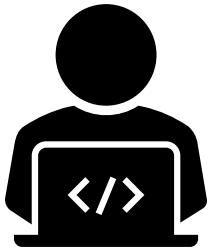
账户姓名: aaa 账号: 10 余额: 0 透支额度: 500 欠款总额: 0 透支利率: 0%
账户姓名: bbb 账号: 11 余额: 100
账户姓名: ccc 账号: 12 余额: 200 透支额度: 700 欠款总额: 0 透支利率: 2%
账户姓名: ddd 账号: 13 余额: 300
账户姓名: eee 账号: 14 余额: 400 透支额度: 900 欠款总额: 0 透支利率: 4%

需要透支: 150 利息: 0
账户姓名: aaa 账号: 10 余额: 0 透支额度: 500 欠款总额: 150 透支利率: 0%
余额不够
账户姓名: bbb 账号: 11 余额: 100
账户姓名: ccc 账号: 12 余额: 50 透支额度: 700 欠款总额: 0 透支利率: 2%
账户姓名: ddd 账号: 13 余额: 150
账户姓名: eee 账号: 14 余额: 250 透支额度: 900 欠款总额: 0 透支利率: 4%

利用多态性显示各类账户信息以及取款

```
账户姓名: aaa  账号: 10     余额: 0 透支额度: 500  欠款总额: 0    透支利率: 0%
账户姓名: bbb  账号: 11     余额: 100
账户姓名: ccc  账号: 12     余额: 200     透支额度: 700  欠款总额: 0    透支利率: 2%
账户姓名: ddd  账号: 13     余额: 300
账户姓名: eee  账号: 14     余额: 400     透支额度: 900  欠款总额: 0    透支利率: 4%

需要透支: 150  利息: 0
账户姓名: aaa  账号: 10     余额: 0 透支额度: 500  欠款总额: 150  透支利率: 0%
余额不够
账户姓名: bbb  账号: 11     余额: 100
账户姓名: ccc  账号: 12     余额: 50      透支额度: 700  欠款总额: 0    透支利率: 2%
账户姓名: ddd  账号: 13     余额: 150
账户姓名: eee  账号: 14     余额: 250     透支额度: 900  欠款总额: 0    透支利率: 4%
```

```
账户姓名: aaa  账号: 10     余额: 0
账户姓名: bbb  账号: 11     余额: 100
账户姓名: ccc  账号: 12     余额: 200
账户姓名: ddd  账号: 13     余额: 300
账户姓名: eee  账号: 14     余额: 400

余额不够
账户姓名: aaa  账号: 10     余额: 0
余额不够
账户姓名: bbb  账号: 11     余额: 100
账户姓名: ccc  账号: 12     余额: 50
账户姓名: ddd  账号: 13     余额: 150
账户姓名: eee  账号: 14     余额: 250
```



代码演示

polymorphism_example_accountVIP.cpp

```
void Deposit(double amt) { balance
virtual void Withdraw(double amt);
virtual void ViewAcct() const;
```

← **把那两个virtual去掉，这是没有运行时多态的结果**

利用多态性扩充软件功能：新增一个子女账户

```
class AccountChild : public Account
{
private:
        double maxWithdraw;
public:
        AccountChild(const char *s = "", long an = -1,double bal = 0.0, double md = 100)
            : Account(s, an, bal) , maxWithdraw(md) {}
        AccountChild(const Account & ba, double md = 100) :Account(ba), maxWithdraw(md) {}
        void ViewAcct()const;
        void Withdraw(double amt);
};
```

关键点：取款时有上限限额

子女账户成员函数的实现

```
void AccountChild::ViewAcct() const
{
	cout << "账户姓名: " << getName() << '\t';
	cout << "账号: " << getAcctNum() << '\t';
	cout << "余额: " << getBalance() << '\t';
	cout << "取现额度: " << maxWithdraw << '\n';
}

void AccountChild::Withdraw(double amt)
{
	if (amt > Account::getBalance())
	cout << "余额不够\n";
	else if (amt > maxWithdraw)
	cout << "超出取现额度\n";
	else
	Account::Withdraw(amt);
}
```

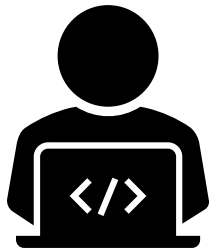
创建账户函数

```
void createAccount(Account *data[])
{
	const char *name[5] = { "aaa", "bbb", "ccc", "ddd", "eee" };

	for ( int i = 0; i < 5; ++i)
		switch ( i % 3 ) {
			case 0: data[i] = new AccountChild(name[i], i+10, 200, (i+1) * 50); break;
			case 1: data[i] = new Account(name[i], i+10, 200); break;
			default: data[i] = new AccountVIP(name[i],i + 10, 100, (i+1)*100, i * 0.01);
		}
}
```

利用多态性显示各类账户信息以及取款

```
int main()
{
    Account *data[5];
    createAccount(data);

    for ( int i = 0; i < 5; ++i)    data[i]->ViewAcct();
    cout << endl;
   for ( int i = 0; i < 5; ++i)   {
        data[i]->Withdraw(150);
     data[i]->ViewAcct();
     delete data[i];
    }

   return 0;
}
```

```
账户姓名: aaa   账号: 10      余额: 200      取现额度: 50
账户姓名: bbb   账号: 11      余额: 200
账户姓名: ccc   账号: 12      余额: 100      透支额度: 300   欠款总额: 0    透支利率: 2%
账户姓名: ddd   账号: 13      余额: 200      取现额度: 200
账户姓名: eee   账号: 14      余额: 200

超出取现额度
账户姓名: aaa   账号: 10      余额: 200      取现额度: 50
账户姓名: bbb   账号: 11      余额: 50
需要透支: 50    利息: 1
账户姓名: ccc   账号: 12      余额: 0 透支额度: 300   欠款总额: 51    透支利率: 2%
账户姓名: ddd   账号: 13      余额: 50       取现额度: 200
账户姓名: eee   账号: 14      余额: 50
```



代码演示

polymorphism_example_accountChild.cpp

```
账户姓名: aaa   账号: 10      余额: 200
账户姓名: bbb   账号: 11      余额: 200
账户姓名: ccc   账号: 12      余额: 100
账户姓名: ddd   账号: 13      余额: 200
账户姓名: eee   账号: 14      余额: 200

账户姓名: aaa   账号: 10      余额: 50
账户姓名: bbb   账号: 11      余额: 50
余额不够
账户姓名: ccc   账号: 12      余额: 100
账户姓名: ddd   账号: 13      余额: 50
账户姓名: eee   账号: 14      余额: 50
```

```
void Deposit(double amt) { balance
virtual void Withdraw(double amt);
virtual void ViewAcct() const;
```

把那两个virtual去掉，这是没有运行时多态的结果

dynamic_cast

基类指针可以指向派生类对象，基类对象可以引用派生类对象，反之不行

```
class A {
public:
        virtual void printA() {cout << "A\n"; }
};
class B: public A {
public:
        void printB() {cout << "B\n"; }
        void printA() {cout << "AB\n"; }
};
class C: public B {
public:
        void printC() {cout << "C\n"; }
        void printB() {cout << "B\n"; }
        void printA() {cout << "AC\n"; }
};
```

```
A  *pa = new B;
```
正确赋值

```
C  *pc = pa;
```
错误

```
B  *pb = pa;
```
错误

```
B  *pb = (B *)pa;
```
正确赋值

dynamic_cast

编译器会检查转换是否合理

格式

dynamic_cast <类名 *> 指针

转换失败，返回空指针

dynamic_cast <类名 &> 变量名

转换失败，抛出异常

基类要有虚函数

```
class A {
public:
        virtual void printA() {cout << "A\n"; }
};
class B: public A {
public:
        void printB() {cout << "B\n"; }
        void printA() {cout << "AB\n"; }
};
class C: public B {
public:
        void printC() {cout << "C\n"; }
        void printB() {cout << "B\n"; }
        void printA() {cout << "AC\n"; }
};
```

```
A  *pa  = new A;
A *pb = new B;
A *pc = new C;
```

```
B *p1   = dynamic_cast  <B *>  pb;      正确赋值
B *p2   = dynamic_cast <B *>  pa;       返回空指针
B *p3 = dynamic_cast <B *>  pc;         正确赋值
```

```
void f(A *p)
{
    if  (dynamic_cast<C *> (p))
        dynamic_cast<C *> (p)->printC();
    else  if  (dynamic_cast<B *> (p))
            dynamic_cast<B *> (p)->printB();
    else cout << "NULL\n";
}
```

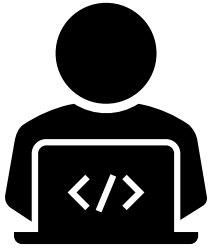
```
int main()
{
    B b;
    C c;
    A a;

    f(&a);
    f(&b);
    f(&c);

    return 0;
}
```

输出:

```
NULL
B
C
```

使用注意

- **dynamin_cast是用于多态性，基类一定要有虚函数**
- **会用到Run-Time Type Information (RTTI) 功能**



代码演示

dynamic_cast-example_wo_virtual.cpp ⟵ **观察报错结果，为何不运行**

dynamic_cast-example_w_virtual.cpp ⟵ **可以运行，区别在哪里?**

虚析构函数

- **构造函数不能是虚函数，析构函数可以是虚函数**
- **作用：**防止内存泄露

应用场景

- 如果派生类新增加的数据成员中含有动态变量，那么派生类**必须定义析构函数**释放这部分空间。但如果派生类的对象是通过**基类的指针操作的**，那么其对应的析构应利用多态来析构派生类的对象，如果delete基类指针指向的对象，则会造成内存泄漏。

虚析构函数的继承性

- 和其他的虚函数一样，析构函数的虚函数的性质将被继承
- 如果继承层次树中的根类的析构函数是虚函数的话，所有派生类的析构函数都将是虚函数
- 建议：设计继承层次时，将根结点类的析构函数设为虚函数

第12章 组合与继承

- 组合
- 继承 （Inheritance）
- 多态（Polymorphism）
- 抽象类 （Abstract class）

纯虚函数和抽象类

- 纯虚函数（pure virtual function）：没有函数体的函数
- 纯虚函数声明格式

  virtual 类型 函数名（参数表）= 0；

  例如：virtual void funtion1()=0;
- 抽象类（Abstract class）
  - 含有纯虚函数的类
  - 抽象类的对象不可以定义
- 抽象类的意义：**保证**进入继承层次的每个类都具有纯虚函数所要求的行为

抽象类实例

- 对所有的汽车都要求有一个显示车辆信息的功能
- 如何保证每个类都有这个功能？
- 可以定义一个抽象类，包含一个显示车辆信息的纯虚函数，各类汽车类都从他派生

```
class  vehicle {
public:
    virtual void display() const = 0;
};
```

```
class  car : public vehicle {
public:
    ……
    void  display() const  { ……};
};
```

```
class  bus : public vehicle {
public:
    ……
    void  display() const  { ……};
};
```

抽象类的意义

保证进入继承层次的每个类都具有纯虚函数所要求的行为

- 保证了围绕这个继承层次所建立起来的软件系统能正常运行，避免了这个继承层次的用户由于偶尔的失误（忘了为它所建立的派生类提供继承层次所要求的行为）而影响系统正常运行

作为基类，实现运行时的多态性

```
class shape{
  public:
     virtual void area()=0;
};
```

抽象类实例

- **下面程序用于计算各类形状的面积**

```
class shape{  public: virtual void area()=0;  };
class rectangle : public shape{  float w, h;
    public :  rectangle(float ww, float hh)  { w = ww;  h = hh; }
            void area( ) { cout << "¥narea is:“ << w*h; }  };
class circle : public shape{ float  r;
    public:  circle(float rr) { r = rr; }
            void area( ){ cout << "¥narea is:“ << 3.14*r*r;}  };
int main( )
{
    shape *ptr;  rectangle ob1(5,10);  circle ob2(10);
    ptr=&ob1; ptr->area();
    ptr=&ob2; ptr->area();
     return  0;
}
```

本章小结

继承是创建类的一种方法

优点:

- 代码重用
- 提高软件的可靠性
- 方便软件扩展
- 实现代码重用