C++程序设计（part 2）

OOP

- Why
  - non-OO Solution

```
#include    <stdio.h>
#define     STACK_SIZE   100
struct Stack
{   int   top;
    int   buffer[STACK_SIZE];
};
void main()
{   Stack   st1, st2;
    st1.top = -1;
    st2.top = -1;       安全隐患
    int  x;
    push(st1,12);
    pop(st1,x);       不符合数据类型定义
    st1.buffer[2] = -1;
    st2.buffer[2] ++;
}
```

```
bool  push(Stack &s, int i)
{   if  (s.top == STACK_SIZE-1)
    {   printf("Stack is overflow.\n");
        return false; }
    else
    {    s.top++;  s.buffer[s.top] = i;
        return true;  }
}
```

```
bool pop(Stack &s, int &i)
{   if  (s.top == -1)
    {   printf("Stack is empty.\n");
        return false;   }
    else
    {   i = s.buffer[s.top];
        s.top--;
        return true;  }
}
```

OOP

- OO Solution

```
bool  push(Stack &s, int i)
{   if  (s.top == STACK_SIZE-1)
    {   printf("Stack is overflow.\n");
        return false; }
    else
    {    s.top++;  s.buffer[s.top] = i;
         return true;  }
}
```

```
bool pop(Stack &s, int &i)
{   if  (s.top == -1)
    {   printf("Stack is empty.\n");
        return false;   }
    else
    {   i = s.buffer[s.top];
        s.top--;
        return true;  }
}
```

```
#include <iostream.h>
#define   STACK_SIZE 100
struct Stack
{   int   top;
    int   buffer[STACK_SIZE];
};
void main()
{   Stack   st1, st2;
     st1.top = -1;
     st2.top = -1;
     int  x;
     push(&st1, 12);
     pop(&st1, x);
}
```

```
struct Stack
{   int   top;
    int   buffer[STACK_SIZE];
};
void main()
{   Stack   st1, st2;
    st1.top = -1;  st2.top = -1;
    int  x;      push(st1,12);      pop(st1,x);
}
```

```
#include <iostream.h>
#define   STACK_SIZE 100
class Stack
{   private:
       int   top;
      int   buffer[STACK_SIZE];
    public:
        Stack()  { top = -1; }
        bool push(int i);
        bool pop(int& i);
};
void main()
{   Stack st1,st2;
     int x;
     st1.push(12);
     st1.pop(x);

     st1.buffer[2] = -1;   X
}
```

Cfront

```
bool  push(Stack * const this, int i)
{     if  ( this-> top == STACK_SIZE-1)
      {   cout << "Stack is overflow.\n";
          return false;  }
      else
      {  this-> top++;   this->buffer[this-> top] = i;
          return true;  }
}
```

```
bool pop(Stack * const this,   int &i)
{    if  (  this-> top == -1)
     {    cout << "Stack is empty.\n";
          return   false;  }
   else
     {    i =  this-> buffer[  this->top];
        this->top--;
          return   true;  }
}
```

Encapsulation

Information Hidding

OOP

- Concepts
  - Program＝Object1 + Object2 +...... + Objectn

  - Object: Data + Operation
  - Message: function call

  - Class

- Classify
  - Object-Oriented
  - Object-Based Ada
    - Without Inheritance

OOP

需求
架构
构建模式
代码
测试用例
项目组织

- Why
  - 评价标准
    - Efficency of Development
    - Quality
      - External

        Correctness、Efficiency、Robustness、Reliability
        Usability、Reusability
      - Internal

        Readability、Maintainability、Portability

产品在规定的条件下和规定的时间内完成规定功能的能力

ENCAPSULATION

类 成员变量 成员函数

a.h

```
class  TDate
{  public:
       void SetDate(int y, int m, int d);
       int IsLeapYear();
   private:
      int  year, month, day;
};
```

```
class  TDate
{ public:
       void SetDate(int y, int m, int d)
       {   year = y;  month = m; day = d; }
       int  IsLeapYear()
          { return (year%4 == 0 && year%100 != 0)
                          || (year%400==0); }

 private:
       int  year, month, day;
};      int  year=2000, ...
```

inline

X

ADT

a.cpp

```
void TDate::SetDate(int y,  int m,  int d)
{   year = y;
    month = m;
    day = d;
}
 int TDate:: IsLeapYear()
{ return (year%4 == 0 && year%100 != 0) || (year%400==0); }
```

```
Tdate g;

int  main()
{    g.SetDate(2000,1,1);

     Tdate t;
     t.SetDate(2015,11,17);

     Tdate  *p = new Tdate;
     p->SetDate(2015,11,17);
}
```

Value

构造函数

- 对象的初始化
- 描述
  - 与类同名、无返回类型
  - 自动调用，不可直接调用
  - 可重载

  - **默认构造函数**　　*无参数*　　Why？
    - **当类中未提供构造函数时，编译系统提供**

  - *public*
    - 可定义为*private*

      接管对象创建

构造函数

- 调用
  - 自动调用

```
 class A
{   …
  public:
         A();
         A(int i);
         A(char *p);
};
A   a1=A(1);  ⇔ A a1(1);  ⇔ A a1=1;           //调A(int i)
A   a2=A(); ⇔ A a2;         //调A()，注意：不能写成：A a2();
A   a3=A("abcd"); ⇔ A a3("abcd"); ⇔ A a3="abcd";  //调A(char *)
A   a[4];           //调用a[0]、a[1]、a[2]、a[3]的A()
A   b[5]={ A(), A(1), A("abcd"), 2, "xyz" };
```

成员初始化表

- 成员初始化**表**
  - 构造函数的补充
  - 执行
    - 先于构造函数体
    - 按类数据成员申明次序

```
class CString
{     char   *p;
      int   size;
public:
   CString(int x):size(x),p(new char[size]){}
};
```

?

减轻Compiler负担

```
class A
{     int    x;
      const  int   y;
      int& z;
   public:
      A(): y(1),z(x), x(0)  {  x = 100; }
};
```

成员初始化表

```
class A
{      int m;
    public:
      A() { m = 0; }
      A(int m1) { m = m1; }
};
```

```
class B
{        int x;
         A a;
      public:
         B(){ x = 0; }
         B(int x1) { x = x1; }
         B(int x1, int m1): a(m1) { x = x1; }
};
```

```
void main()
{      B b1;         //调用B::B()和A::A()
       B b2(1);   //调用B::B(int)和A::A()
       B b3(1,2); //调用B::B(int,int)和A::A(int)
       ...
}
```

成员初始化表

- 在构造函数中尽量使用成员初始化表取代赋值动作
  - const 成员/reference 成员/对象成员
  - 效率高
  - 数据成员太多时，不采用本条准则
    - 降低可维护性

GC

效率障碍

*存在不能用**GC**的场合*

需要时，程序员自行实现

析构函数

- 析构函数
  - ~<类名>()
  - 对象消亡时，系统自动调用

Java: finalize()

RAII vs GC

Resource Acquisition Is Initialization

释放对象持有的非内存资源

  - *public*
    - 可定义为*private*

```
class A
{   public:
        A();
     void destroy() {delete this;}
    private:
        ~A();
};
```

~~A a;~~

```
int main()
{  A aa;
};
```

(with `A aa;` struck through)

A *p = new A;

~~**delete p;**~~

p->destroy();

Better Solution:

```
static void free(A *p)
      { delete p; }
```

A::free(p);

强制自主控制对象存储分配

反之亦可

析构函数

```
class String
{           char *str;
  public:
            String()   {  str = NULL; }
            String(char *p)
            { str = new char[strlen(p)+1];
              strcpy(str,p); }

            ~String()  {  delete []str; }

            int length()  { return strlen(str); }
            char get_char(int i)  {  return str[i];
}
```

```
void set_char(int i, char value)
{  str[i] = value; }
char &char_at(int i)
{  return str[i]; }
char *get_str()  {   return str; }
char *strcpy(char *p)
{   delete []str;
    str = new char[strlen(p)+1];
   strcpy(str,p); return str;
}
String &strcpy(String &s)
{   delete []str;  str =
newchar[strlen(s.str)+1];
  strcpy(str,s.str);  return *this; }

char *strcat(char *p);
String &strcat(String &s); };
```

拷贝构造函数

- Copy Constructor
  - 创建对象时，用一同类的对象对其初始化
  - 自动调用

```
A  a;
A  b=a;
```

```
f(A a)
{   ....  }

A b;
f(b);
```

```
A  f()
{   A a;  ....
    return a;
}

f();
```

```
public:
    A(const A& a);
```

默认拷贝构造函数

➢逐个成员初始化(member-wise initialization)

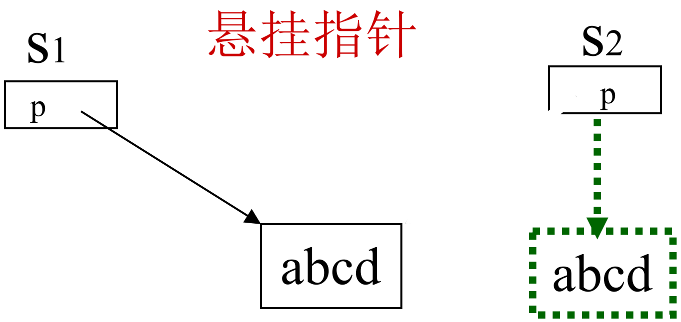
➢对于对象成员，该定义是递归的

何时需要copy constructor?

拷贝构造函数

```
class string
{   char *p;
public:
     string(char *str)
     {  p = new char[strlen(str)+1];
        strcpy(p, str);
     }
    ~string() { delete[] p; }
}

string  s1(“abcd”);
string  s2=s1;
```




```
string::string(const string& s)
{   p = new char[strlen(s.p)+1];
    strcpy(p,  s.p);
}
```

deep copy

拷贝构造函数

包含成员对象的类

➢默认拷贝构造函数

调用成员对象的**拷贝构造函数**

➢自定义拷贝构造函数

调用成员对象的**默认构造函数**

```
class A
{       int x,y;
    public:
        A() { x = y = 0; }
        void inc() { x++; y++; }
};
class B
{       int z;
        A a;
   public:
        B() { z = 0; }
        B(const B& b) : a(b.a) { z = b.z; }
        void inc() { z++; a.inc(); }
};
......
B b1;           //b1.z=b1.a.x=b1.a.y =0
b1.inc();       //b1.a.x=b1.a.y=b1.z=1
B b2(b1);       //b2.z=1，b2.a.x=0，b2.a.y=0
```

```
string generate()
{    ......
     return string("test");
}
string S=generate();
```

转移构造函数

move constructor

A(A&&)

动态内存

- Types of memory from Operating System
  - Stack – local variables and pass-by-value parameters are allocated here
  - Heap – dynamic memory is allocated here
- C
  - malloc() – memory allocation
  - free() – free memory
- C++
  - new – create space for a new object (allocate)
  - delete – delete this object (free)

动态对象

- 动态对象
  - 在*heap* 中创建
  - *new* / *delete*

为什么要引入new、delete操作符？
constructor/destructor

动态对象

```
class A
{ …
        public:
           A();
           A(int);
};
```

*A *p,*q;*

p = new A;

- *在程序的heap中申请一块大小为sizeof(A)的内存*
- *调用A的默认构造函数对该空间上的对象初始化*
- *返回创建的对象的地址并赋值给p*

q = new A(1);

- ……
- *调用A的另一个构造函数 A::A(int)*
- ……

delete p;

- *调用p所指向的对象的析构函数*
- *释放对象空间*

delete q;

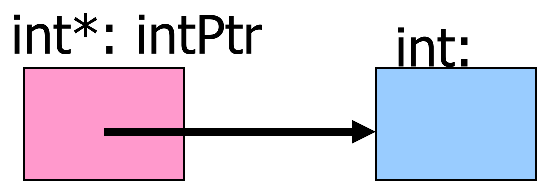
创建对象

- new
  - Works with primitives
  - Works with class-types
- Syntax:
  - `type* ptrName = new type;`
  - `type* ptrName = new type( params );`

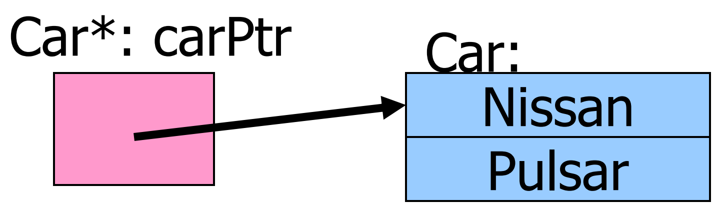
Constructor!

New Examples

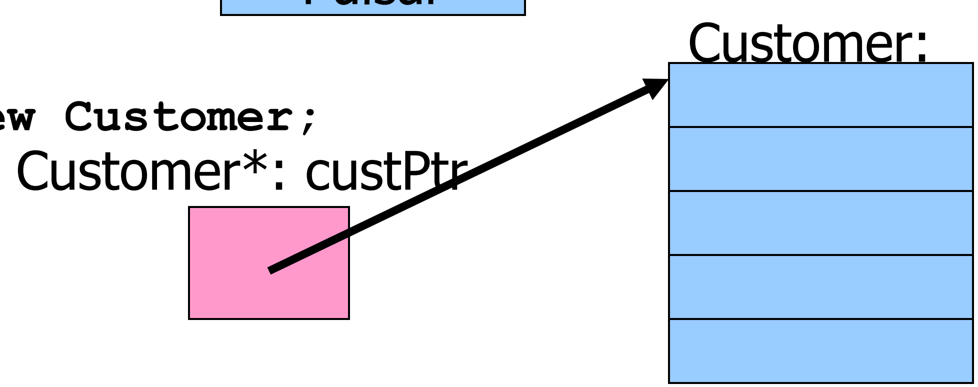
```
int* intPtr = new int;
```




```
Car* carPtr = new Car("Nissan", "Pulsar");
```




```
Customer* custPtr = new Customer;
```



Notice:

These are unnamed objects! The only way we can get to them is through the pointer!

Pointers are the same size no matter how big the data is!

动态对象

- *p = (A *) malloc (sizeof(A))*
  free(p)

  malloc 不调用构造函数
  free 不调用析构函数

- *new* 可重载

对象删除

- delete
  - Called on the pointer to an object
  - Works with primitives & class-types
- Syntax:
  - delete ptrName;
- Example:
  - `delete intPtr;`
  - `intPtr = NULL;`

  - `delete carPtr;`
  - `carPtr = NULL;`

  - `delete custPtr;`
  - `custPtr = NULL;`

Set to NULL so that you can use it later – protect yourself from accidentally using that object!

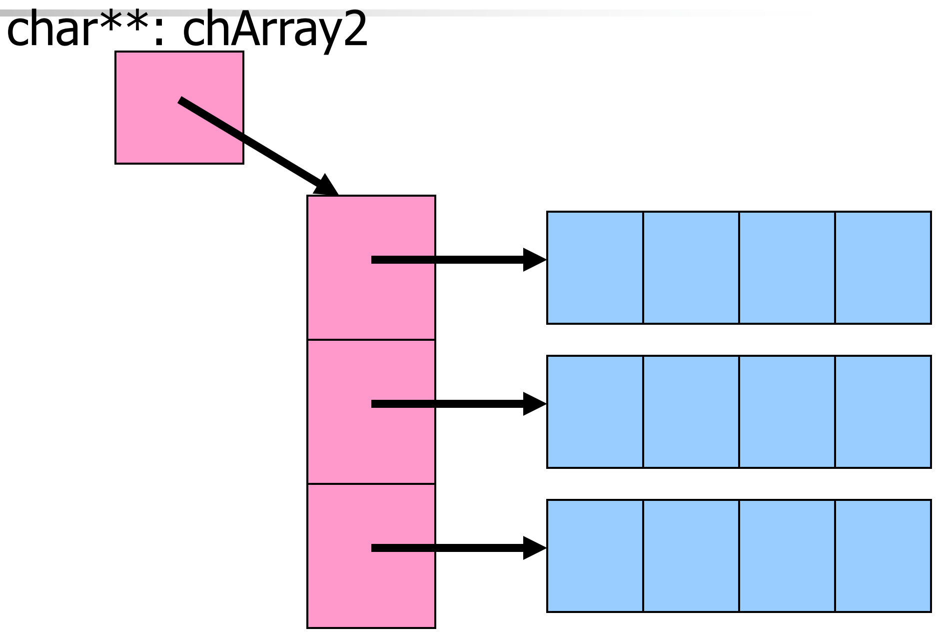
动态对象数组

- 动态对象数组的创建与撤消

```
A *p;
p = new A[100];
delete  []p;
```

- 注意
  - 不能显式初始化，相应的类必须有默认构造函数
  - delete中的[]不能省

动态2D数组



- Algorithm
  - Allocate the number of rows
  - For each row
    - Allocate the columns
- Example

```
const int ROWS = 3;
const int COLUMNS = 4;

char **chArray2;

// allocate the rows
chArray2 = new char* [ ROWS ];

// allocate the (pointer) elements for each row
for (int row = 0; row < ROWS; row++ )
   chArray2[ row ] = new char[ COLUMNS ];
```

动态2D数组

- Delete?
  - Reverse the creation algorithm
    - For each row
      - Delete the columns
    - Delete the rows
- Example

```
for (int row = 0; row < ROWS; row++)
{
   delete [ ] chArray2[ row ];
   chArray2[ row ] = NULL;
}

delete [ ] chArray2;
chArray2 = NULL;
```

Const 成员

- *const* 成员
  - *const* 成员变量

```
class A
{   const int x; }
```

- 初始化放在构造函数的成员初始化表中进行

```
class A
{         const  int x;
     public:
          A(int c): x(c) {  }
}
```

Const 成员

void f(A * const this);
void show(const A* const this);

- *const* 成员函数

```
class A
{    int x,y;
 public:
        A(int x1, int y1);
        void f();
        void show() const ;
};
```

const A a(0,0);

a.f(); X
a.show(); √

mutable

compiler

```
void A::f()
{  x = 1; y = 1; }

void A::show() const
{  cout <<x << y;}
```

```
class A
{
    int a;
    int & indirect_int;
public:
     A():indirect_int(*new int){ ... }
    ~A() { delete &indirect_int; }
    void f() const { indirect_int++; }
};
```

静态成员

- 静态成员
  - 类刻划了一组具有相同属性的对象
  - 对象是类的实例

  - 问题：同一个类的不同对象如何共享变量？
    - 如果把这些共享变量定义为全局变量，则缺乏数据保护
    - 名污染

静态成员

- 静态成员变量

```
class A
{   int  x,y;
    static int shared;
    .....
};
int A::shared=0;

A a, b;
```

  - 类对象所共享
  - 唯一拷贝
  - 遵循类访问控制

静态成员

- 静态成员函数

```
class A
{    static int shared;
     int x;
 public:
     static void f() { …shared…}
     void q() { …x…shared…}
};
```

  - 只能存取静态成员变量，调用静态成员函数
  - 遵循类访问控制

静态成员

- 静态成员的使用
  - 通过对象使用

    A a; a.f();
  - 通过类使用

    A::f();

- C++支持观点   “类也是对象”
  - Smalltalk

静态成员

```
class A
{      static int obj_count;
          …
  public:
         A()  {   obj_count++; }
         ~A()  {   obj_count--; }
         static int get_num_of_obj() ;
          …
};


int  A::obj_count=0;
int  A::get_num_of_obj() { return obj_count; }
```

示例

singleton

Resource Control
原则：谁创建，谁归还
解决方法：自动归还

```
class  singleton
{   protected:
            singleton(){}
            singleton(const singleton &);
    public:
            static singleton * instance()
            {   return  m_instance == NULL?
                                    m_instance = new singleton: m_instance;
            }
            static void destroy()  { delete m_instance; m_instance = NULL; }
    private:
            static singleton * m_instance;
};
singleton * singleton ::m_instance= NULL;
```

友元

- 友元
  - 类外部不能访问该类的*private*成员
    - 通过该类的*public*方法
    - 会降低对*private*成员的访问效率，缺乏灵活性

    - 例：矩阵类(Matrix)、向量类(Vector)和全局函数(multiply)，全局函数实现矩阵和向量相乘

友元

```
class Matrix
{     int  *p_data;
      int   lin,col;
  public:
     Matrix(int l, int c)
     {   lin = l;
         col = c;
         p_data = new int[lin*col];
     }
     ~Matrix()
     { delete []p_data; }

int &element(int i, int j)
{  return *(p_data+i*col+j); }

 void dimension(int &l, int &c)
 {   l = lin;
     c = col;
  }

  void display()
 {   int *p=p_data;
     for (int i=0; i<lin; i++)
     {   for (int j=0; j<col; j++)
         {  cout << *p << ' ';
            p++;
         }
         cout << endl;
     }
};
```

友元

```
class Vector
{   int  *p_data;
    int   num;
  public:
    Vector(int n)
    { num = n;
      p_data = new int[num];
    }
    ~Vector()
    {  delete []p_data;
    }
    int &element(int i)
    {  return p_data[i]; }

    void dimension(int &n)
    { n = num; }

    void display()
    {  int *p=p_data;
       for (int i=0; i<num; i++,p++)
           cout << *p << ' ';
       cout << endl;
    }
};
```

友元

```
void   multiply(Matrix &m, Vector &v, Vector &r)
{   int  lin,  col;
     m.dimension(lin,col);
    for (int i=0; i<lin; i++)
   {   r.element(i) = 0;
       for (int j=0; j<col; j++)
           r.element(i) += m.element(i,j)*v.element(j);
    }
}

void main()
{    Matrix m(10,5);
    Vector v(5);
    Vector r(10);
    ......
    multiply(m,v,r);
    m.display();
   v.display();
    r.display();
}
```

友元

- 分类
  - 友元函数
  - 友元类
  - 友元类成员函数
- 作用
  - 提高程序设计灵活性
  - 数据保护和对数据的存取效率之间的一个折中方案

友元

```
void func() ;
class B;
class C
{  .......
    void f();
};
class A
{       ...
        friend void func();          //友元函数
        friend class B;              //友元类
        friend void C::f();          //友元类成员函数
};
```

友元

```
class Matrix
{    ......
     friend void multiply(Matrix &m, Vector &v, Vector &r);
};
class Vector
{    ......
     friend void multiply(Matrix &m, Vector &v, Vector &r);
};
```

- 友元不具有传递性
- 能编译吗？

原则

Law of Demeter

- 避免将data member放在公开接口中

	Get	Set
R	√	
W		√
RW	√	√
NONE		

```
class AccessLevels {
public:
   int getReadOnly const { return readOnly; }
   void setReadWrite(int value) { readWrite = value; }
   int getReadWrite() { return readWrite; }
   void setWriteOnly(int value) { writeOnly = value; }
private:
   int noAccess;
   int readOnly;
   int readWrite;
   int writeOnly;
};
```

- 努力让接口完满 (complete) 且最小化

//错误声明
class Undergraduated_Student : public Student;
//正确声明
class Undergraduated_Student ;

单继承

protected

```
class Student
{       int id;
    public:
        char nickname[16];
        void set_ID(int x)  { id = x; }
        void SetNickName(char *s) { strcpy(nickname,s);}
virtual void showInfo()
        { cout << nickname << " : " << id <<endl; }
};
```

id
nickname
dept_no

```
class Undergraduated_Student : public Student
{       int dept_no;
    public:
        void setDeptNo(int x) { dept_np = x; }
        void set_ID(int x) {......}
        void showInfo()
         { cout << dept_no << ": "<< nickname <<endl; }
    private:
        Student::nickname;
        void SetNickName ();
};
```

继承方式

public

private、protected

友元和protected

```
class Base (
protected :
    int prot_mem;                          // protected 成员
} ;
class Sneaky : public Base {
    friend void clobber{Sneaky&) ;         //能访问Sneaky::prot_mem
    friend void clobber{Base&);            //不能访问Base::prot_mem
    int j;                                 // j 默认是private

void clobber(Sneaky &s) { s.j =            //正确：clobber 能访问Sneaky对象的
s.prot_mem = 0; )                          private和protected成员

void clobber(Base &b) {                    //错误： clobber 不能访问Base的
b.prot_mem = 0; }                          protected 成员
```

继承

- 派生类对象的初始化
  - 由基类和派生类共同完成
- 构造函数的执行次序
  - 基类的构造函数
  - 派生类对象成员类的构造函数
  - 派生类的构造函数
- 析构函数的执行次序
  - 与构造函数相反

B(const B& b){…} ??

class B: public A{
public:
using A::A; //继承A的构造函数

继承

- 基类构造函数的调用
  - 缺省执行基类默认构造函数
  - 如果要执行基类的**非默认构造函数**，则必须在派生类构造函数的**成员初始化表**中指出

```
class A
{    int x;
  public:
     A() { x = 0; }
     A(int i) { x = i; }
};
```

```
class B: public A
{   int y;
public:
    B() { y = 0; }
    B(int i) { y = i; }
    B(int i, int j):A(i)
    {   y = j;  }
};
```

```
B b1;         //执行A::A()和B::B()
B b2(1);      //执行A::A()和B::B(int)
B b3(0,1);    //执行A::A(int)和B::B(int,int)
```

虚函数

- 类型相容
  - 类、类型
  - 类型相容、赋值相容
  - 问题：a、b是什么类型时，a = b 合法？
    - A a;   B b;   class B: public A
      - 对象的身份发生变化
      - 属于派生类的属性已不存在
    - B* pb; A* pa = pb;   class B: public A
    - B b;   A &a=b;   class B: public A
      - 对象身份没有发生变化

虚函数

把派生类对象赋值
给基类对象

```
class A
{    int x,y;
 public:
      void f();
};
class B: public A
{    int  z;
public:
        void f();
        void g();
};
```

```
A  a;
B  b;

a = b;      //OK,
b = a;      //Error
a.f();      //A::f()
```

```
A &r_a=b;           //OK
A *p_a=&b;          //OK

B &r_b=a;           //Error
B *p_b=&a;          //Error
```

```
func1(A& a)
{ …  a.f();  … }

func2(A *pa)
{ …  pa->f(); …}

func1(b);
func2(&b);
```

A::f?
B::f?

基类的引用或指针可以引用
或指向派生类对象

虚函数

- 前期绑定（Early Binding）
  - 编译时刻
  - 依据对象的静态类型
  - 效率高、灵活性差
- 动态绑定（Late Binding）
  - 运行时刻
  - 依据对象的实际类型（动态）
  - 灵活性高、效率低
- 注重效率
  - 默认前期绑定
  - 后期绑定需显式指出 *virtual*

虚函数

- 定义
  - *virtual*

```
class A
{     …
   public:
      virtual void f();
};
```

  - 动态绑定
    - *根据*实际引用和指向的对象类型
  - 方法重定义

虚函数

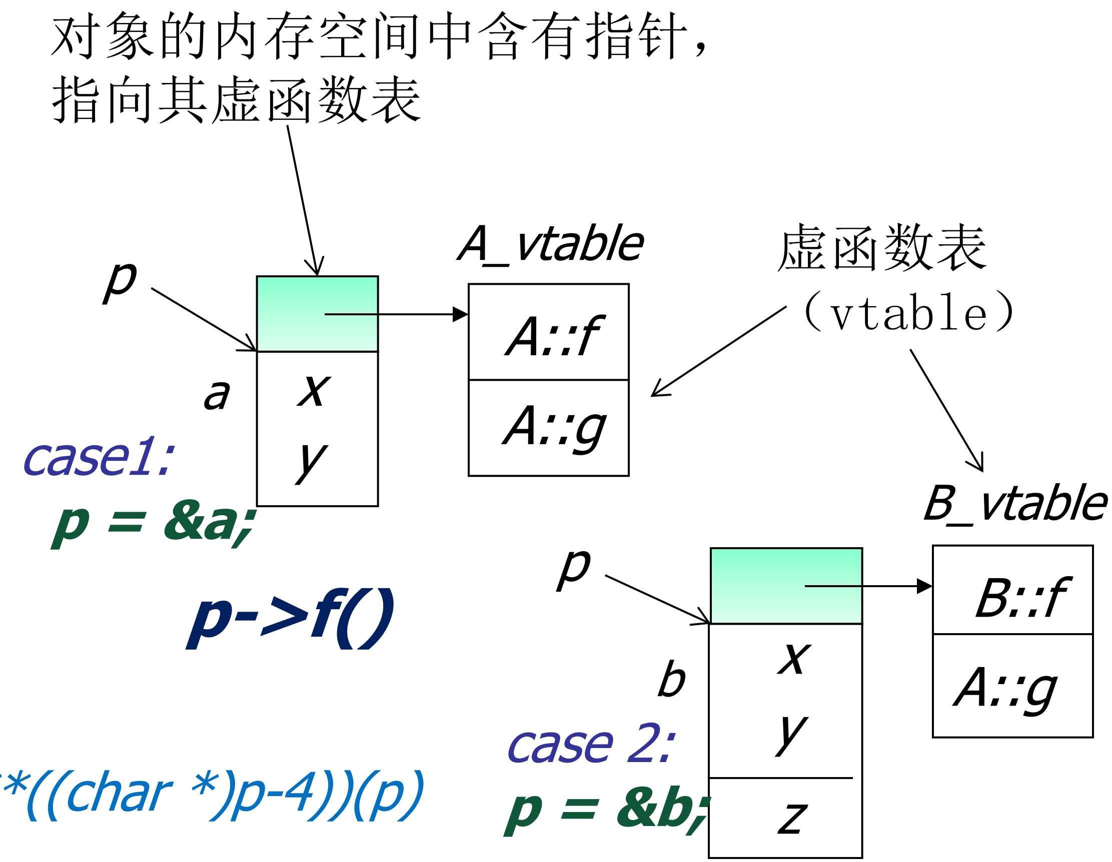
- 如基类中被定义为虚成员函数，则派生类中对其重定义的成员函数均为虚函数

- 限制
  - 类的成员函数才可以是虚函数
  - 静态成员函数不能是虚函数
  - 内联成员函数不能是虚函数
  - 构造函数不能是虚函数
  - 析构函数可以（往往）是虚函数

虚函数

- 后期绑定的实现

```
class A
{     int x,y;
public:
      virtual f();
      virtual g();
      h();
};
class B: public A
{     int z;
public:
      f();
      h();
};
A a; B b;
A *p;
```



虚函数

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

直到构造函数返回之后，
对象方可正常使用

```
...
B b;          // A::A()，A::f, B::B(),
A *p=&b;
p->f();       //B::f
p->g();       //A::g
p->h();       //A::h, B::f, A::g
```

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

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

B b;
A* p = &b;
p->f();   //b.B::g
```

B* const this

this->g();

final, override

```
struct B {
    virtual void f1(int) const ;
    virtual void f2 ();
    void f3 () ;
    virtual void f5 (int) final;

};
struct D: B {
    void f1(int) const override ;  //正确： f1与基类中的f1 匹配
    void f2(int) override ;        //错误： B没有形如f2(int) 的函数。int f2()?
    void f3 () override ;          //错误： f3不是虚函数
    void f4 () override ;          //错误： B没有名为f4的函数
    void f5 (int) ;                //错误： B已经将f5声明成final
}
```

虚函数

- 纯虚函数和抽象类
  - 纯虚函数
    - 声明时在函数原型后面加上 = 0
    - 往往只给出函数声明，不给出实现

Means “ not there”

```
virtual int f()=0;

class AbstractClass
{     …
  public:
     virtual int f()=0;
};
```

  - 抽象类
    - 至少包含一个纯虚函数
    - 不能用于创建对象
    - 为派生类提供框架，派生类提供抽象基类的所有成员函数的实现

_pure_virtual_called

虚函数

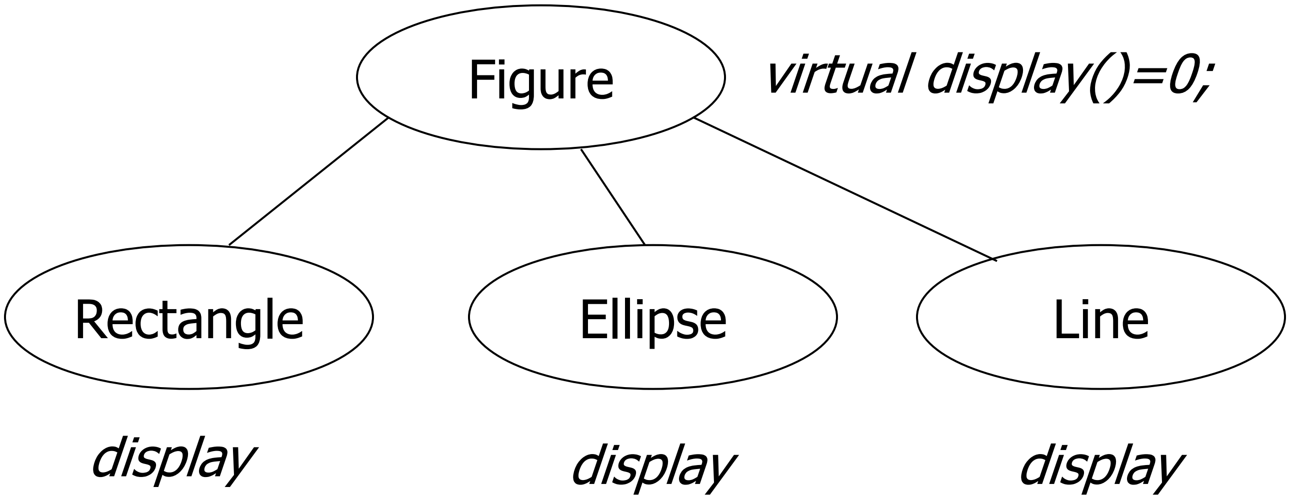


*Figure *a[100];*
a[0] = new Rectangle();
a[1] = new Ellipse();
a[2] = new Line();
...
for (int i=0; i<num_of_figures; i++)   a[i]->display();

```
AbstractFactory* fac;

case MAC:
    fac = new MacFactory;
case WIN:
    fac = new WinFactory;
```

WinButton
WinLabel
……

Step1:
提供Windows GUI类库

Step2:
增加对Mac的支持

```
    Button *pb= fac->CreateButton();
……
pb->SetStyle( … );

    Label *pl= fac->CreateLabel();
……
pl->SetText( … );
```

MacButton
MacLabel
……

Step3:
增加对用户跨平台设计的支持

Button SetStyle()=0;

WinButton   MacButton

AbstractFactory CreateButton() =0;
CreateLabel()=0;

WinFactory   MacFactory

```
WinButton* CreateButton()
{  return new WinButton; }
WinLabel*   CreateLabel()
{ return new WinLabel; }
```

```
MacButton* CreateButton()
{  return new MacButton; }
MacLabel* CreateLabel()
{  return new MacLabel; }
```

```
class Button; // Abstract Class
class MacButton: public Button {};
class WinButton: public Button {};
class Label; // Abstract Class
class MacLabel: public Label {};
class WinLabel: public Label {};
```

```
class AbstractFactory {
public:
    virtual Button* CreateButton() =0;
    virtual Label* CreateLabel() =0;
};
class MacFactory: public AbstractFactory {
public:
    MacButton* CreateButton() { return new MacButton; }
    MacLabel* CreateLabel() {  return new MacLabel; }
};
class WinFactory: public AbstractFactory {
public:
    WinButton* CreateButton() { return new WinButton; }
    WinLabel*   CreateLabel() { return new WinLabel; }
};
```

```
AbstractFactory* fac;
switch (style) {
case MAC:
    fac = new MacFactory;
    break;
case WIN:
    fac = new WinFactory;
    break;
}
Button* button = fac->CreateButton();
Label* Label = fac->CreateLabel();
```

抽象工厂模式
Abstact Factory

虚函数

- 虚析构函数

class B {…};
class D: public B{…};
B p = new D;*　　　　?:　**delete p;**

class mystring {…}
class B {…}
class D: public B{
mystring name; …}
B p = new D;*　　　　?:　**delete p;**

虚函数

```
class FlyingBird
class NonFlyingBird
```

Penguin

```
virtual void fly() { error("Penguins can't fly!"); }
```

- 确定public inheritance, 是真正意义的 “is_a”关系
- 不要定义与继承而来的非虚成员函数同名的成员函数

```
class Rectangle
{
    public:
    virtual void setHeight(int);
    virtual void setWidth(int);
            int height() const;
            int width() const;};
```

```
void Widen(Rectangle& r, int w)
{
    int oldHeight = r.height();
    r.setWidth(r.width() + w);
    assert(r.height() == oldHeight);
}
```

```
assert(s.width() == s.height());
```

```
class Square: public Rectangle {
    public:
        void setLength (int);
    private:
        void setHeight(int );
        void setWidth(int );
… };
```

```
Square s(1,1);
Rectangle *p = &s;
p->setHeight(10);
```

```
class B
{   public:
            void mf();
    …   };
class D: public B {
    public:
            void mf();
… };

D x;
B* pB = &x;
pB->mf();    //B:mf
D* pD = &x;
pD->mf();   //D:mf
```

虚函数

- 明智地运用private Inheritance
  - Implemented-in-term-of
    - 需要使用Base Class中的protected成员，或重载virtual function
    - 不希望一个Base Class被client使用
  - 在设计层面无意义，只用于实现层面

```
class CHumanBeing { … };

class CStudent: private CHumanBeing { … };

void eat(const CHumanBeing& h)
{ … }

CHumanBeing a;  CStudent b;
eat(a);
eat(b);  //Error
```

虚函数

```
class Shape {
public:
    virtual void draw() const = 0;

    virtual void error(const string& msg);

    int objectID() const;
};
```

- 纯虚函数

  只有函数接口会被继承

  - 子类必须继承函数接口
  - （必须）提供实现代码

- 一般虚函数

  函数的接口及缺省实现代码都会被继承

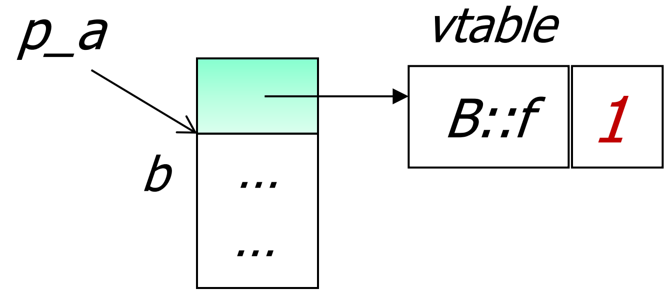
  - 子类必须继承函数接口
  - 可以继承缺省实现代码

- 非虚函数

  函数的接口和其实现代码都会被继承

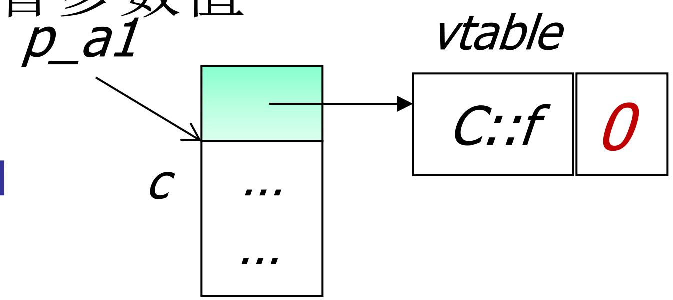
  - 必须同时继承接口和实现代码

```
(**((char *)p_a1 - 4))(p_a1)
char *q = *((char *)p_a1 - 4);
(*q)(p_a1, *q+4);
```



虚函数

- 绝对不要重新定义继承而来的缺省参数值
  - 静态绑定
  - 效率




```
class A
{
public:
    virtual void f(int x=0) =0;
};

class B: public A
{
public:
    virtual void f(int x=1)
    { cout << x;}
};
```

```
class C: public A
{
public:
    virtual void f(int x) { cout<< x;}
};
```

```
A *p_a;
B b;
p_a = &b;
p_a->f();
```

```
A *p_a1;
C c;
p_a1 = &c;
p_a1->f();
```

对象中只记录虚函数的入口地址

多继承

- 多继承
  - 定义

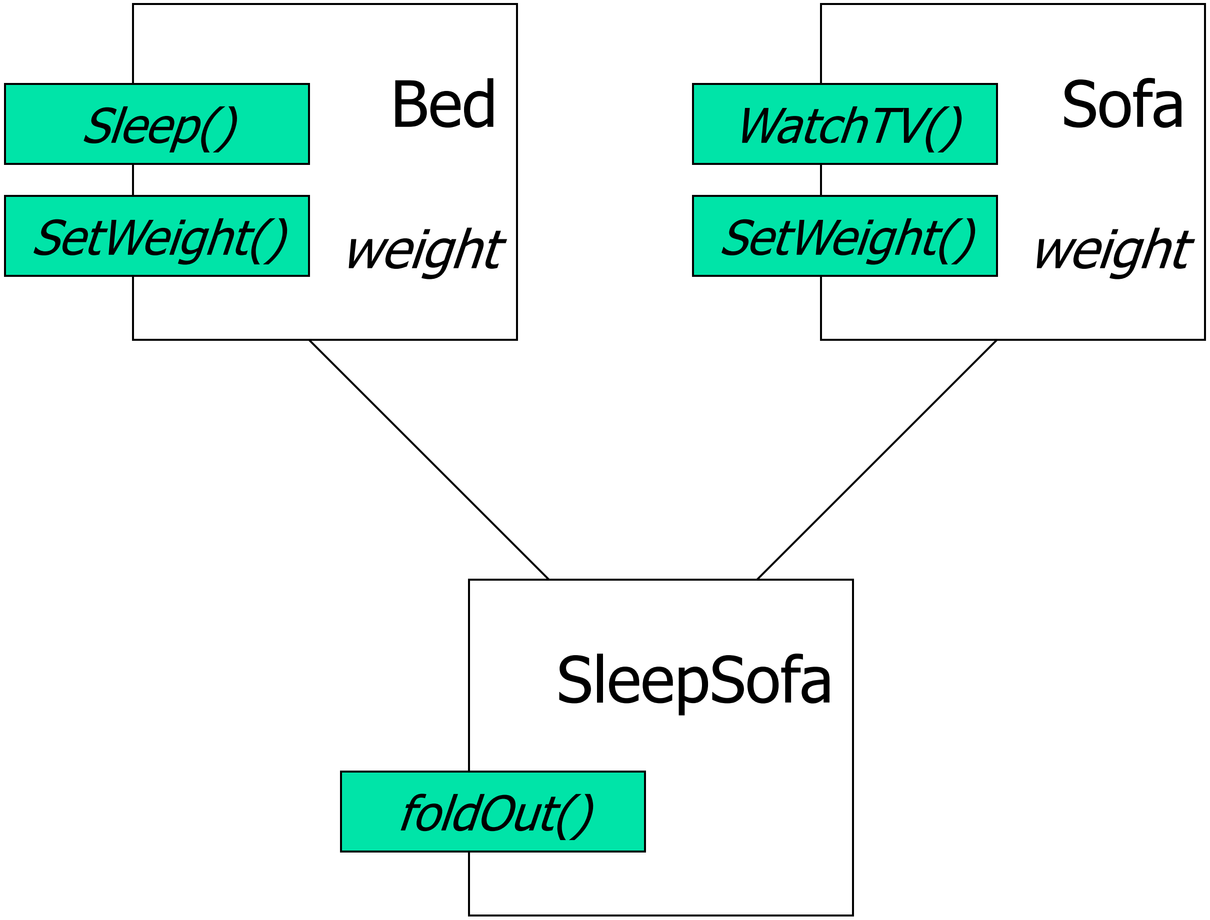
    class 〈派生类名〉：[〈继承方式〉] 〈基类名1〉，

    [〈继承方式〉] 〈基类名2〉，…

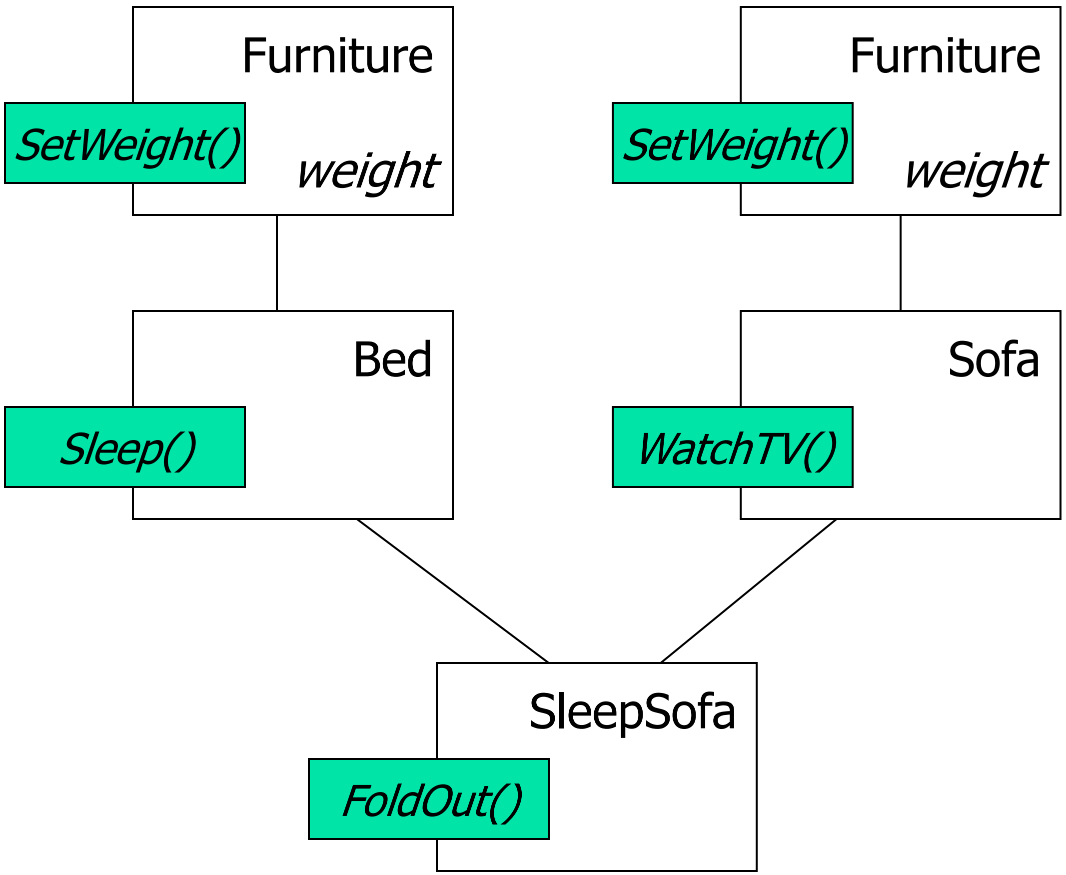
    { 〈成员表〉 }

    - 继承方式
      - *public*、*private* 、*protected*
    - 继承方式及访问控制的规定同单继承
    - 派生类拥有所有基类的所有成员

多继承

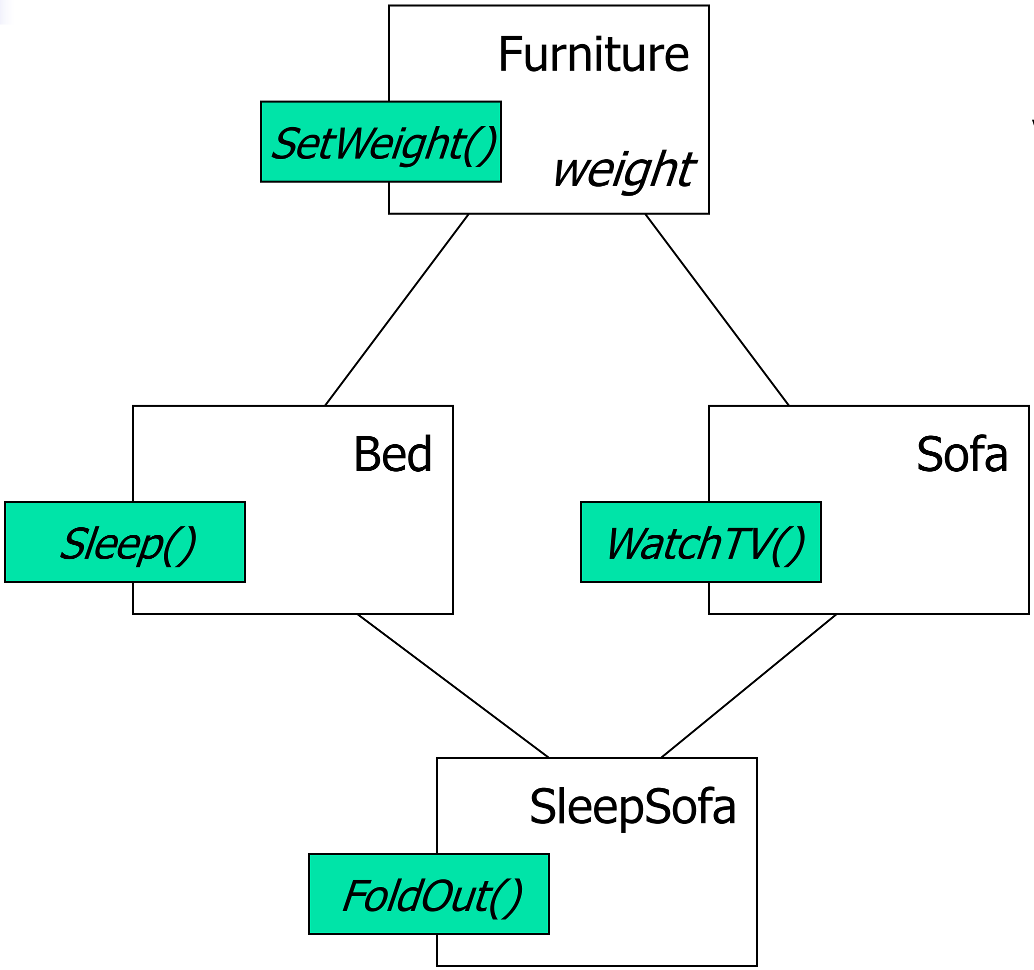

Bed
Sleep()
SetWeight()
weight
Sofa
WatchTV()
SetWeight()
weight
SleepSofa
foldOut()

多继承



Base-Class
Decomposition

多继承



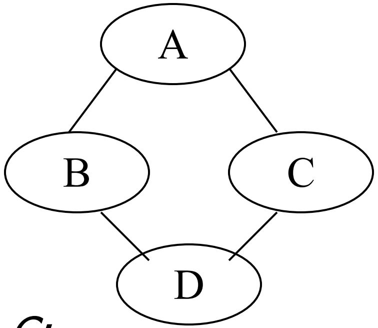
Virtual Inheritance

多继承

- 基类的声明次序决定：
  - 对基类构造函数/析构函数的调用次序
  - 对基类数据成员的存储安排

- 名冲突
  - 〈基类名〉::〈基类成员名〉

```
class A
{  int x;
    ...
};
class B: A;
class C: A;
class D: B, C;
```



- 虚基类
  - 如果直接基类有公共的基类，则该公共基类中的成员变量在多继承的派生类中有多个副本

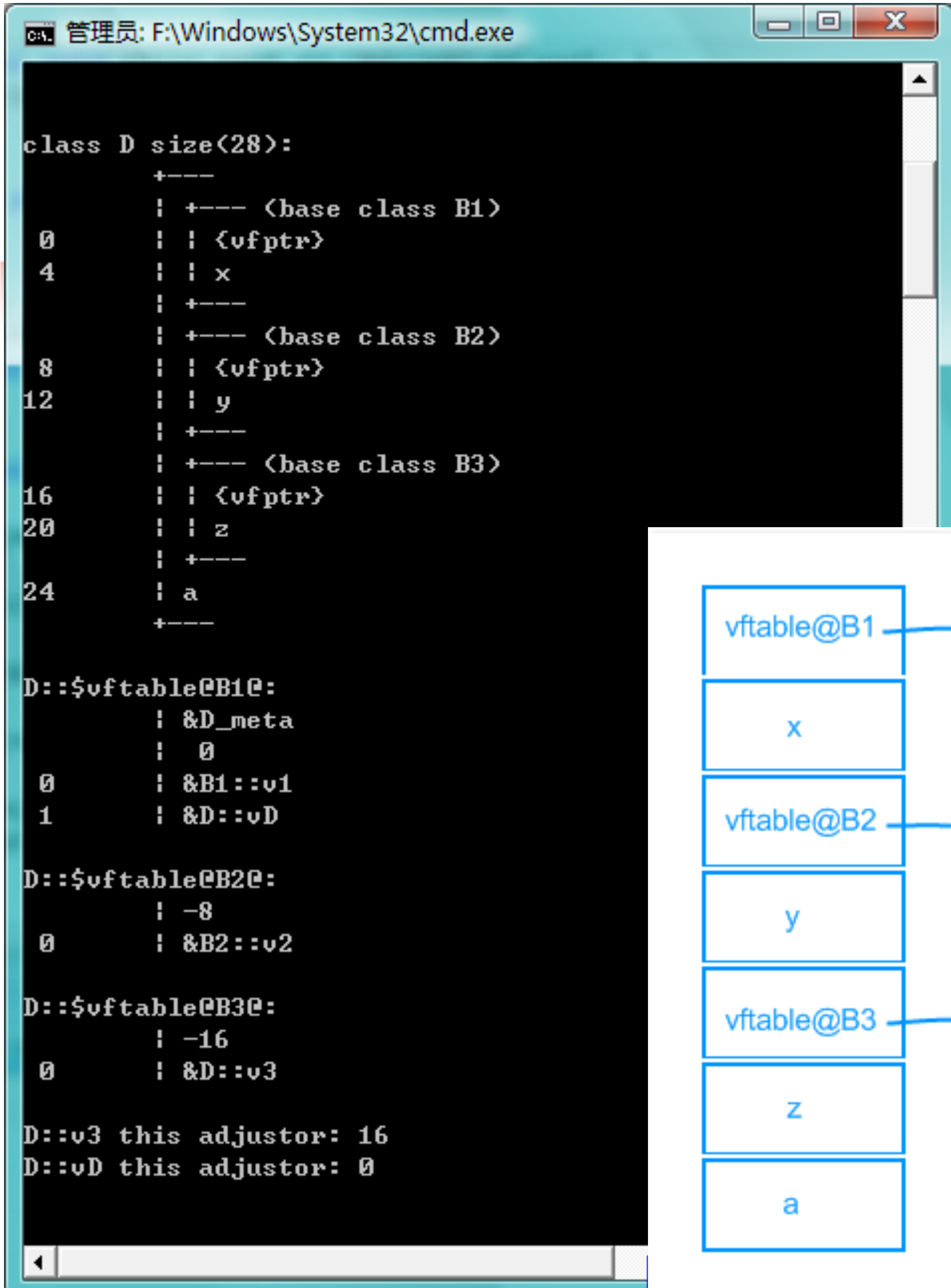
多继承

- 类D拥有两个x成员：B::x和C::x

- 虚基类
  - 合并

```
class A;
class B: virtual public A;
class C: public virtual A;
class D: B, C;
```

- 注意
  - 虚基类的构造函数由最新派生出的类的构造函数调用
  - 虚基类的构造函数优先非虚基类的构造函数执行



管理员: F:\Windows\System32\cmd.exe

```
class D size(28):
        +---
        | +--- (base class B1)
 0      | | {vfptr}
 4      | | x
        | +---
        | +--- (base class B2)
 8      | | {vfptr}
12      | | y
        | +---
        | +--- (base class B3)
16      | | {vfptr}
20      | | z
        | +---
24      | a
        +---

D::$vftable@B1@:
        | &D_meta
        |  0
 0      | &B1::v1
 1      | &D::vD

D::$vftable@B2@:
        | -8
 0      | &B2::v2

D::$vftable@B3@:
        | -16
 0      | &D::v3

D::v3 this adjustor: 16
D::vD this adjustor: 0
```