多　态　性

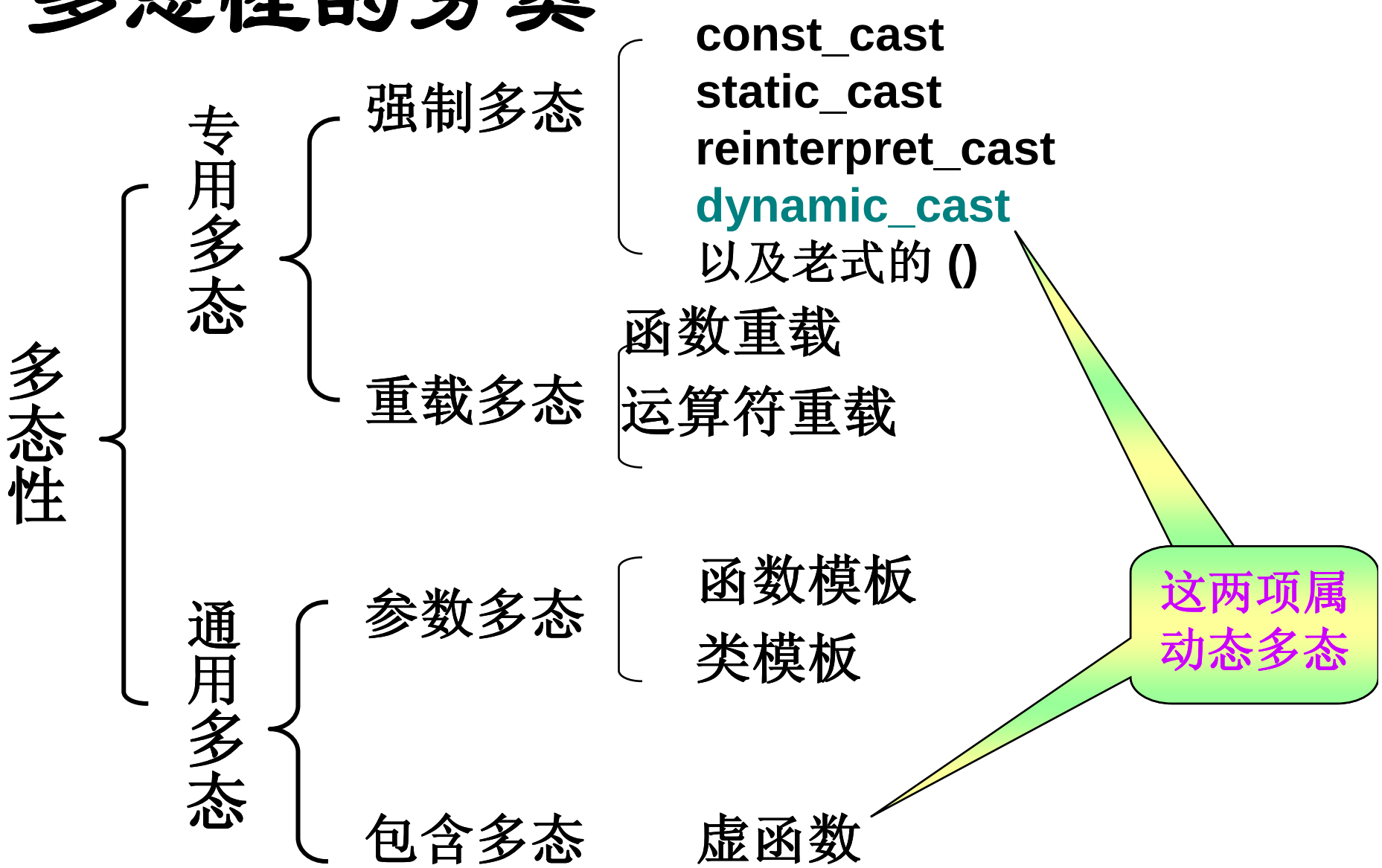
- 多态的概念
- 虚函数
- 纯虚函数
- 抽象类(虚拟基类)

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 。

多态性的分类


多态性
专用多态
通用多态
强制多态
重载多态
参数多态
包含多态
const_cast
static_cast
reinterpret_cast
dynamic_cast
以及老式的 ()
函数重载
运算符重载
函数模板
类模板
虚函数
这两项属
动态多态

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。

静态绑定与动态绑定

- 绑定
  - 程序自身彼此关联的过程，确定程序中的调用与执行代码间的关系。其实就是确定绝对地址的过程。
  - 然后就是个绑定的时机问题——何时绑定。
- 静态绑定(静态联编)
  - 由编译器在编译时发出调用，由链接器在链接时确定被调用函数的绝对地址。这都固定在代码中了。
- 动态绑定
  - 编译器在编译时仅对被调函数语法检查，直到程序运行时才通过附加(机制)代码确定要调用的函数。

静态绑定例

```
#include<iostream>

class Point
{ public:
      Point(double i, double j) : x(i), y(j) { }
      double Area()  const {  return 0.0; }
  private:
    double x, y;
};
class Rectangle : public Point
{ public:
    Rectangle(double i, double j, double k, double l);
    double Area() const  { return  w*h; }
  private:
    double w, h;
};
```

```
Rectangle::Rectangle(double i, double j, double k,
    double l)  : Point(i, j)
{   w=k;  h=l; }

void fun(const Point &s)
{   std::cout<<"Area="<<s.Area()<< std::endl;  }

int main()
{
    Rectangle rec(3.0, 5.2, 15.0, 25.0);
    fun(rec);
}
```

运行结果：

Area=0

动态绑定例

```
#include<iostream>

class Point
{ public:
     Point(double i, double j) {x=i; y=j;}
     virtual double Area()  const {  return 0.0; }
  private:
    double x, y;
};
class Rectangle:public Point
{ public:
     Rectangle(double i, double j, double k, double l);
     virtual double Area() const  { return  w*h; }
  private:
    double w,h;
};
```

```
Rectangle::Rectangle(double i, double j, double k,
    double l)  : Point(i, j)
{   w=k;  h=l; }

void fun(const Point &s)
{   std::cout<<"Area="<<s.Area()<< std::endl;  }

int main()
{
    Rectangle rec(3.0, 5.2, 15.0, 25.0);
    fun(rec);
}
```

运行结果：

Area=375

虚　函　数

- 虚函数是动态绑定的技术基础。
- 只能用于非静态非友元非内联的成员函数。
- 在类的声明中，在函数原型之前写**virtual**。如果一个函数被定义为虚函数，那么，使用基类类型的指针来调用该成员函数，**C++**能保证所调用的是特定于实际对象的成员函数。
- **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，同原型的函数都自动为虚函数。
- 本质：不是重载**(overload)**而是重写。

虚　函　数

- 调用方式：通过基类类型的指针或引用，执行时会根据指针指向的对象的类型，决定调用哪个类的成员函数。一个指向基类的指针可用来指向从基类公有派生的任何对象，这一事实非常重要，它是C++实现运行时多态的关键。
- **virtual** 的含义：迫使在继承于该类的派生类必须予以重新实现。
- **virtual** 的作用：使继承于该基类的各派生类，都肩负了重写虚函数的责任。
- 虚函数虽然不可以是友元函数，但可以外派成为另一个类的友元函数。

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虚指针vptr(virtual pointer)，并让其指向一个由类维护的虚函数表vtable(其实是个指针数组)，每个表项是一个虚函数名(地址)，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，都拥有各自虚指针和虚函数表(指针数组)。相同类型所生成的对象共享了同一个虚函数表。

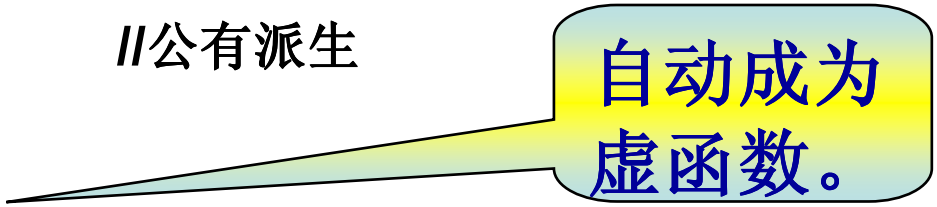
该项技术的实质是“将找谁变成到哪去找”——不用管找到的是哪一个。

- 派生类新增的虚函数依次排在表的后面。当然，派生类的**vtable**表项中放的是新的覆盖了父类同名函数的首址。
- 对象中加入的**vptr**的位置，会因类是**class**还是**struct**而不同：或在对象之前端，或在后端。于是造成了父子是**class**还是**struct**的不和(不兼容)。
- 对象中加入的**vptr**的位置，还会因编译器的不同而异：有的编译器将**vptr**加在对象之前端，有的加在后端。**VC++**是加在前端。
- 非虚函数不进入**vtable**表。

```
// 能昭示对象内部结构的例子，请调试该程序，作为习题。
#include<iostream>
#include<stdio.h>

class point3d
{
public:
    float x,y,z;
    point3d(float xx, float yy, float zz)
    {
        x=xx;
        y=yy;
        z=zz;
    }
    virtual ~point3d() {}
    void show() const       //普通成员函数
    {
        std::cout<<x<<"    "<<y<<"    "<<z<< std::endl;
    }
    static point3d origin;   //类定义中含有本类的静态对象
};
```

```
point3d point3d::origin( 10,18,20);//静态成员的类外初始化

int  main()
{
    using std::cout;
    using std::endl;
//  cout<<"&point3d::x = "<<&point3d::x<<endl;
    point3d dd( 10,18,20);
    cout<<"&point3d::dd = "<<&dd<<endl<<endl;
    cout<<"&point3d::x = "<<&dd.x<<endl;
    cout<<"&point3d::y = "<<&dd.y<<endl;
    cout<<"&point3d::z = "<<&dd.z<<endl<<endl;
    printf("&point3d::x = %p\n",&point3d::x);
    printf("&point3d::y = %p\n",&point3d::y);
    printf("&point3d::z = %p\n",&point3d::z);
}
```

```
#include <iostream>

class B0          //基类B0声明
{
public: //外部接口
  virtual void display() const   //虚成员函数
   { std::cout<<"B0::display()"<< std::endl; }
};

class B1: public B0              //公有派生
{
public:
   void display()  const
   {  std:: cout<<"B1::display()"<<endl;  }
};

class D1: public B1              //公有派生
{ public:
     void display() const
     {  std::cout<<"D1::display()"<< std::endl;  }
};
```



```
void fun(B0 *ptr)    //普通函数
{    ptr->display();    }

int 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()
```

动态多态的前提 (缺一不可)

- 必须有继承产生的类族；
- 必须是公有继承(类型兼容)；
- 基类的某成员函数使用了virtual；
- 派生类的成员函数要重写该虚函数；
- 派生类的对象要使用指针或引用来调用该虚函数；

“重写”的含义

在派生类重定义虚函数(重写)时，子类重写的函数必须与父类的函数具有相同的函数签名，包括返回类型，函数名、参数个数、参数类型以及声明次序，只是函数体实现不同。

这几条必须严格遵守，缺一不可。若仅函数名相同，那不是重写，是隐藏(覆盖)，或干脆错误，自然不享受动态联编的机制。

若在派生类中没有重新定义虚函数，则该类的对象将使用其基类的虚函数代码—只是继承，没有创新。

调用虚函数不一定动态联编

```
#include <iostream>
class A
{
public:
    A()   { fc(); }
    virtual void fc() const { std::cout << "in Class A“ << std::endl; }
    virtual ~A() { std::cout << "destructing A object……“ << std::endl; }
};

class B : public A
{
public:
    B() { fc(); }
    void f() const { fc(); }
    ~B() { fd(); }
    void fd() const   // 普通成员函数
    { std::cout << "destructing B object……“ << std::endl;   }
};
```

```
class C : public B
{
public:
  C() { }
  void fc() const { std::cout<<"in Class C"<< std::endl; }
  ~C() { fd(); }
  void fd() const  // 普通成员函数
   { std::cout<<"destructing C object……“ << std::endl; }
};

int main()
{   //以下是函数的调用顺序。思考：为什么没有动态联编？
    C c;                    // A()—>A::fc()—>B()—>A::fc()—>C()
    std::cout<<std::endl;
    c.fc();                 // C::fc()
    std::cout<<std::endl;
    A* p = new B;           // A()—>A::fc()—>B()—>A::fc()
    std::cout<<std::endl;
    delete p;               // ~B()—>B::fd()—>A::~A()
    std::cout<<std::endl;
}                   // ~C()—>C::fd()—>B::~B()—>B::fd()—>A::~A()
```

同名成员函数的表现

同名的成员函数可分为两种：普通和虚函数。

- 普通的同名函数们在同一个类中只能是重载关系，

  例如：**void Show(int , char) ;**

  void Show(char *, float) ;

- 若在继承类族中，子类又新增了与父类同名的普通函数，此时是隐藏，调用时可以用“ :: ”区分，

  例如：**A :: Show( );**

  B :: Show( );

- 若在继承类族中，子类又新增了与父类同函数签名的虚函数，此时是重写， 但能表现出动态多态。

  例如： **Aobj.Show( );**是调用 **A :: Show( )** 。

  Bobj.Show( );是调用 **B :: Show( )** 。

重载、覆盖和重写

- 重载：在同一作用域内，函数名相同却有不同的参数列表和代码实现。
- 覆盖：在继承中，子类中再度出现了父类的同名函数，无论形参是否相同，都是覆盖，覆盖也常被称为隐藏。
- 重写：在继承中，子类中再现了父类的用虚函数或纯虚函数修饰的同函数签名（此时的同名最严格：函数名、返回类型、形参表都必须相同），这种现象叫重写。注意，若返回类型符合“类型兼容”亦可。即子类中的同名函数返回了父类类型或子类类型。这称为“协变类型”。

```
#include<iostream>
using namespace std;
class Base
{  public:
    void f(int x) { cout<<“Base::f(int)”<<x<<endl;}
    void f(float x) { cout<<“Base::f(float)”<<x<<endl;}
    void g(float x) { cout<<“Base::g(float)”<<x<<endl;}
    void h(float x) { cout<<“Base::h(float)”<<x<<endl;}
    virtual  void m(float x){cout<<“Base::m(float)”<<x<<endl;}
};
class derived : public Base
{  public:
    void f(int x) { cout<<“derived ::f(int)”<<x<<endl;}
    void g(float x) { cout<<“derived ::g(float)”<<x<<endl;}
    void h(int x) { cout<<“derived ::h(int )”<<x<<endl;}
    void m(float x){cout<<“derived ::m(float)”<<x<<endl;}
    void m(int x){cout<<“derived ::m(int)”<<x<<endl;}
};
```

重载、覆盖还是重写？

```
class Base
{
public:
    virtual void foo()  const;
};
class Derived : public Base
{
public:
    void foo() const;  // 是重写吗？
};
```

虚函数的设计原则

若在类族中，某对象的行为与其类型相关，且表现为动态特性，则应在基类中将该行为的函数设计为虚函数；

若在类族中，某对象的行为与其类型相关，且表现为静态特性，则应在子类中将该行为的函数覆盖；

若在类族中，某对象的行为与其类型无关，则不必设计为虚函数，且别将其覆盖。

为何使用指针时会动态联编

当用指针指向一个对象，用指针调用对象的虚函数时，
比如有：class B { … }; 这个类含有虚函数fun(),
当 B obj;

B * p = &obj;

系统会将调用语句：p->fun(arg_list);

某编译器将之改写为：

(* (p ->_vptr[slotNum])) (p , arg_list);

将对象首址传给**this**。

通过间址找所指向的函数体

是虚函数在**vtable**表中的下标位置。

虚函数所带的实参列表

无虚函数的弊端

多层继承后，若无虚函数，尽管用指针指向各类对象，但仍不能调用子类的同名函数。

```
#include <iostream>

class k                                  //k基类声明
{
public:
  void at(int a) const
  { std::cout << “K ” << a << std::endl; }
};
class kk: public k                       //kk基类声明
{
public:
  void at(char a) const {    std::cout <<"KK "<< std::endl; }
};
```

```
class xq : public kk
{
public:
  void at(float f)  const     //(局部)对象成员
   { std::cout <<“XQ “ << f << std::endl; }
};

int main()
{
  k* p;
  p = new k;       //局部父对象
  p->at(10);
  p = new kk;     //局部子对象
  p->at('i');
  p = new xq;     //局部孙对象
  p->at(10.88);
}
```

虚函数以及静态转换对继承的影响

```
#include <iostream>

struct B   // 这是一个虚拟基类，不能用来创建对象，只能继承
{
  B () {  std::cout << "created B\n"; }
  virtual ~B () { std::cout << "destroyed B\n"; }
  virtual void op1() = 0;   // 纯虚函数
};
struct D1 : public  B
{
  D1 () {  std::cout<< "created D\n"; }
  void op1() { std::cout << "D1::op1() called\n"; }
  void op2() { std::cout << "D1::op2() called\n"; }
  virtual int thisop() const
  {
    std::cout << "D1::thisop () called\n";
    return 10;
  }  //  新增的虚函数
};
```

```
struct D1 : public  B
{
  D1 () {  std::cout<< "created D\n"; }
  void op1() { std::cout << "D1::op1() called\n"; }
  void op2() { std::cout << "D1::op2() called\n"; }
  virtual int thisop() const
  {
    std::cout << "D1::thisop () called\n";
    return 10;
  }  // 新增的虚函数
};

struct D2 : public  B
{
  D2 () {  std::cout << "created D2\n"; }
  void op1() { std::cout << "D2::op1() called\n"; }
  void op2() { std::cout << "D2::op2 () called\n"; }
  virtual char thatop() const    // 新增的虚函数
  {
    std::cout << "D2::thatop() called\n";
    return 'A';
  }
};
```

```
int main()
{
  B* bp = new D2;                          //对象交抽象类的指针把持
  D1* dp = static_cast<D1*>(bp);   // 静态的转换成子类型
  dp->op1();                               // 基类中op1是纯虚函数
  dp->op2();                               // op2是普通成员函数
  int  a = dp->thisop();                   // thisop是虚函数
  cout<<"a = "<< a <<endl;
}
// 一种可能运行结果如下：
created B
created D2
D2 ::op1() called                  // 为什么？
D1::op2() called                   // 为什么？
D2::thatop() called                // 为什么？
a = 1875677761                     // 为什么？
```

多继承时使用虚函数

```
#include<iostream>
class A
{
public:
  virtual void get() {   std::cout << "A::i " << i << std::endl;  }
  int i;
};

class B : virtual public A
{
public:
  void get()
  {
    i = 10;
    std::cout << "visit A::i in B " << i << std::endl;
  }
};
```

```
class C: virtual protected A
{
  int i;
public :
  C()  { A::i = 4; }
  void get()
  {
    std::cout<< "visit A::i in C " << A::i << std::endl;
  }
};

class D : public B, public C
{
public:
  void get()  {   std::cout << "visit A::i in D "<< A::i << std::endl;  }
};
```

```
int main()
{
    A a,*p;
    B b;
    C c;
    D d;
    a.i = 10;
    p = &a;
    p->get();
    p = &b;
    p->get();
    p = &c;          // error, A is an inaccessible base of C
    p->get();        // error
    p = &d;
    p->get();
}
```

虚函数的意义

- 虚函数的出现，使相关的语法元素具有了新意。
- 对象出现了静态类型和动态类型两重性。

  静态类型是指其语法表面所表现的类型。

  动态类型是指运行时实际起作用的类型。
- 这两个方面其实表现了对象同时具有的“类型性”和“实例性”。在没有虚函数时这二者是统一的，虚函数使得二者分离。
- 指向对象的指针和对象的引用则仅具有“类型性”。

空的虚函数在派生链中的纽带作用

```
#include <iostream>

class A
{
public:
  virtual  void p() const
  {   std::cout << "in Class A" << std::endl; }
};

class B : public A
{
public:
    void p() const { }  //去掉此句试一试!
};

class C : public B
{
public:
    void p()  const  { std::cout << "in Class C" << std::endl; }
};
```

```
void show(A *a)
{
    a->p();
}
// 此例子作为课后习题
int main()
{
    A* a = new A;
    A* b = new B;
    A* c = new C;
    show(a);              // 调用A类的p()函数
    show(b);              // 什么也不做
    show(c);              // 调用B类的p()函数
   delete a;
   delete b;
   delete  c;
}
```

析构函数为虚函数

为何需要虚析构函数？

避免析构对象不彻底造成内存泄漏。

当基类的析构函数为virtual时,子类的析构函数自然为virtual。反之，当基类的析构函数为非virtual,而子类的析构函数为virtual时，子类对象不具有多态性。即，释放对象时会发生内存泄漏。

如果你不能确定你编写的类是否将来会被继承，请把析构函数设置为虚函数。

```
#include <iostream>
#include <cstring>

class Base
{
public:
  Base() {}
  ~Base() { std::cout << "~Base called\n"; } // non-virtual!!
};

class Derived : public Base
{
public:
  Derived(const char* str = "") : mstr(0)
  {
    int len = std::strlen(str);
    mstr = new char[len+1];
    std::strcpy(mstr, str);
    mstr[len] = 0;
}
```

```
  ~Derived()
  {
    delete[] mstr;
    std::cout << "~Derived called.\n";
  }
protected:
  char* mstr;
};

int main(int argc, char** argv)
{
  Base* pstr = new Derived("1234567890");
  delete pstr;
}

// 以上程序运行界为：
~Base called
// 可见派生类对象在堆上分配的内存没有被释放。
```

抽象类(虚拟基类)

先来看个例子：

假设，汽车最大速度的函数为**Speed**，潜艇最大速度的函数为**Speed**。有个两栖交通工具，它可以奔跑在马路上，也可以航行在大海中，那么它就同时拥有两种交通工具的最大速度特性**Speed** 。于是可以定义一个交通工具类，它的函数为**Speed** ，但并不实现(也无法实现)，而是由各派生类去实现，这个函数也被称为接口。甚至这个基类也被称为接口。

抽象类(虚拟基类)

带有纯虚函数的类称为抽象类。如:

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

其中：“ =0 ”的含义：是告诉编译器，该函数的地址为0，即不用为该函数编址。

于是这个类是“残缺不全”的类，当然不能实例化出对象来。只有virtual函数才可以=0。

抽象类(虚拟基类)

- 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。
  - 对于暂时无法实现或不想给出有意义的定义的函数，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。
  - 抽象类不能作函数的参数，也不能作返回值。
  - 构造函数不能是虚函数，析构函数可以是虚函数。
  - 构造函数和析构函数不允许调用纯虚函数，否则会导致程序运行错误。但其他成员函数可以调用。

```
#include <iostream>

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() const { std::cout<<"B1::display()"<< std::endl; }
};
class D1: public B1           //公有派生
{
public:
    void display() const { std::cout<<"D1::display()"<< std::endl; }
};
```

```
void fun(B0 *ptr)    //普通函数
{   ptr->display();   }

int 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
}
```

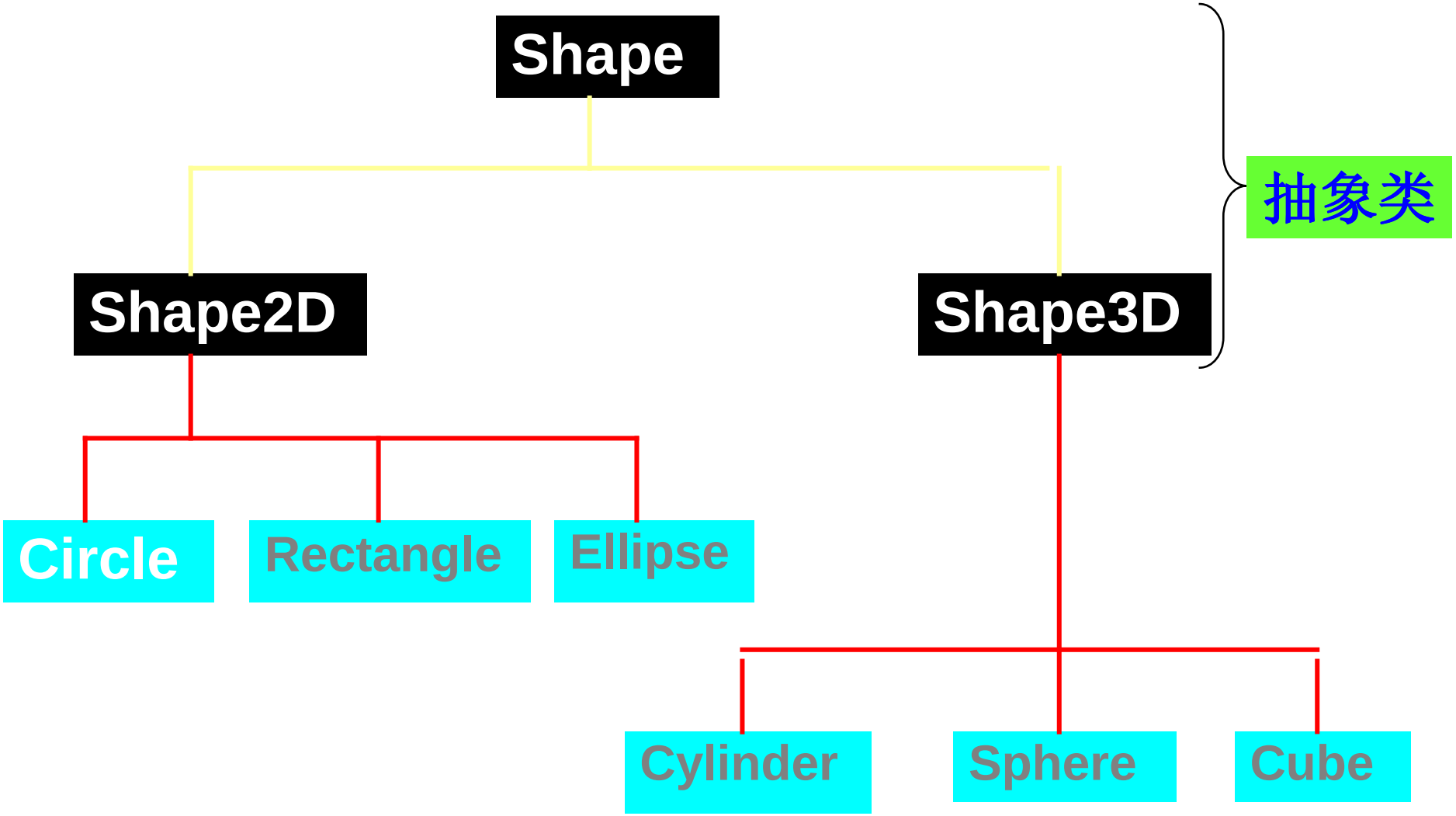
运行结果：

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

多态类和抽象类

- 多态类
  - 拥有虚函数的类，若公有派生出了子类，且在子类中重写了父类的虚函数，就称为多态类。
- 抽象类
  - 如果一个类中存在至少一个纯虚函数，这个类就是抽象类。
  - 抽象类不能创建该类的对象；
  - 抽象类的最重要的用途就是提供一个界面，而不暴露任何实现的细节，迫使子类去实现它。

抽象类可以有多层


Shape
Shape2D
Shape3D
抽象类
Circle
Rectangle
Ellipse
Cylinder
Sphere
Cube

继承的再认识

某函数若可实施于一个类，则定可实施于它的子类——这个类为其子类提供了公共的操作接口。

（公有）继承有两层含义：函数接口的继承，实现的继承；

- 设计纯虚函数的目的是，父类没有或无法提供一个先验的行为，这使子类们只得到无实现的接口，迫使子类们各行其是地予以实现；

  当然，也可以有实现，子类们只能以

  父类指针-〉父类名::纯虚函数名()

  的方式调用。这实际为子类们提供了一块公共功能。

当一个类的全部函数成员都是纯虚函数时，这个类就是个“接口”，——要求子类定给出自己的函数实现。

- 设计普通虚函数的目的是，使子类既得到接口，也得到一个默认的实现，子类可以用重写予以超越，也可以留用；这就是说，子类们既得到了接口也得到了行为(有选择余地)。

- 设计非虚函数的目的是，使子类既得到接口，也得到一个不可改变的行为——强制实现。子类不打算超越父类的功能，则采用此法，（若打算超越，则用虚函数）。这又引出了一个法则：绝不要重新定义继承来的非虚函数。因为那将引起隐藏，会造成本应是同一个功能，而子类和父类的同名函数却干了不同的事。这是应该由虚函数来完成的，非虚函数不要插手。那样做会导致子类的对象精神分裂。(Scott条款37)

  结论：若一个类不含有virtual，则意味着它不打算被用作基类。实在要继承也只是扩充新的。

不要重新定义继承来的有默认形参值的虚函数

决不要重新定义继承来的有默认形参值的虚函数，那也会导致子类的对象精神分裂。要么基类的虚函数不要带默认形参值，那只会对子类添乱。

理由很简单：虚函数属于动态联编，而带默认形参值的函数属静态联编，它们水火不相容。

父类的有默认形参值的虚函数，在继承时被子类重新定义，则有可能没再给出默认值，这样一来，动态调用时如何去调？静态联编是在编译时完成的，可让它去对付动态运行时才能确定的函数，它能应付得了吗？看下面的例子：

```
class A
{
public:
    virtual void fa(int x = 10)     //父类的成员带了默认值
    { std::cout<<"A::fa() called! x = "<< x <<std::endl; }
};
class B : public A
{
public:
    void fa(int x = 100)  //一个子类的成员也带了默认值
    { std::cout<<"B::fa() called! x = "<< x <<std::endl; }
};
class C : public A
{
public:
    void fa(int x)            //另一个子类的虚成员函数不带默认值
    { std::cout<<"C::fa() called! x = "<< x <<std::endl; }
};
```

```
int main()
{
    A*pab= new B;
    B* pb= new B;
    A* pac= new C;
    C* pc= new C;
    pab->fa();
    pb->fa();
    pac->fa();
    pc->fa(20);
    delete pab;
    delete pb;
   delete pac;
    delete pc;
}
```

```
// 运行结果
B::fa() called! x = 10
B::fa() called! x = 100
C::fa() called! x = 10
C::fa() called! x = 20
```

为何是这样的结果？

因为动态条件完全满足，必然是调用子类的虚函数。但由于默认形参数值属静态联编，受制于指针类型，结果调的是子类的“重写”函数，但使用的却是指针类型指示的值。

不要对虚函数使用指向函数的指针

```
#include <iostream>
class B   // 抽象类
{
public:
    B ( ) {  std::cout<<“ a B object called\n"; }
    virtual ~B ( ) {  std::cout<<“a B object distroyed\n"; }
    virtual void op1() const = 0;      //纯虚函数
    virtual void op2() const = 0;
};
class D1 : public  B   //公有继承
{
public:
    D1 ( ) {  cout<<“a D1 object called\n";  }
    void op1() const { std::cout << "D1::op1() called\n"; }
    void op2() const { std::cout << "D1::op2() called"; }
};
```

```
int main()
{
   D1 d;                    //派生类对象
   B *bp = &d;              //基类的指针指向子类的对象
   typedef void (D1::* MemFp)();
   MemFp fp2 = D1::op2;     //指向了子类的虚函数

   bp->op1();
   (d.*fp2)();
   //(bp->*fp2)();若用指向函数的指针指向虚函数则失去了多态性
}
```

课后练习

1. 本次课中前面指出的习题。
2. 编译运行本次课中的示例代码，如有**bug**请自行改正。
3. 定义一个基类**Animal**，它包含两个私有数据成员，一个是**string**成员，存储动作的名称（**“Fido”**），一个是整数成员**weight**，存储了动 物的重量（单位是磅）。该基类还包含一个公共的虚拟成员函数**who()**和一个纯虚函数**sound()**，公共的虚拟成员函数**who()**返回一个**string** 对象，该对象包含了**Animal**对象的名称和重量，纯虚函数**sound()**在派生类中应返回一个**string**对象，表示该动物发出的声音。把**Animal** 类作为一个公共基类，派生至少三个类**Sheep**、**Dog**和**Cow**，在每个类中实现**sound()**函数。
   定义一个类**Zoo**，它至多可以在一个数组中存储**50**种不同类型的动作（使用指针数组）。编写一个**main()**函数函数，创建给定数量的派生类对象的随机序 列，在**Zoo**对象中存储这些对象（使用指针数组）。编写一个**main()**函数，创建给定数量的派生类对象的随机序列，在**Zoo**对象中存储这些对象的指针。 使用**Zoo**对象的一个成员函数，输出**Zoo**中每个动物的信息，以及每个动物发出的声音。

4. 任务模拟。(节选自The C++ Programming Language)

Design and implement a library for writing event-driven simulations. Hint: <task .h> However, that is an old program, and you can do better. There should be a class task. An object of class task should be able to save its state and to have that state restored (you might define task::save()and task::restore ()) so that it can operate as a coroutine. Specific tasks can be defined as objects of classes derived from class task. The program to be executed by a task might be specified as a virtual function. It should be possible to pass arguments to a new task as arguments to its constructor(s). There should be a scheduler implementing a concept of virtual time. Provide a function task ::delay (long)that ‘‘consumes’’ virtual time. Whether the scheduler is part of class task or separate will be one of the major design decisions. The tasks will need to communicate. Design a class queue for that. Devise a way for a task to wait for input from several queues. Handle run-time errors in a uniform way. How would you debug programs written using such a library?