程序设计与算法语言 · 虚函数
C/C++ Programming and Algorithms
Virtual Function

Dongke Sun (孙东科)
dksun@seu.edu.cn

东南大学机械工程学院
School of Mechanical Engineering
Southeast University

April 12, 2018

提纲

提纲

静态联编和动态联编

联编Binding

是指一个计算机程序彼此关联的过程。在这里指函数间调用关系的确定。按照联编所确定的时刻不同，可分为两种：静态联编和动态联编。

静态联编Static Binding, Compile-Time Binding

是指联编出现在编译连接阶段，即函数调用关系的确定是在程序执行之前。这种联编又称早期联编，通过这种联编可实现静态多态。

动态联编Late Binding, Runtime Binding,

运行阶段才能确定函数的调用关系，这就是动态联编。动态联编又称滞后联编、晚期联编。 动态联编技术能实现动态多态。

必须将类的成员函数定义成虚函数（virtual function），才可以实现动态联编。

多态性Polymorphism

意味着具有多种形态。它起源于poly和morph，前者表示多，后者表示形态。多态性是将接口与实现分离，实现相同的方法但因数据差异而采用不同的策略。

普通函数的静态联编

静态联编

在编译连接阶段，就能根据参数的个数和类型确定调用的是哪一个函数。

```
1  #include <iostream>
2  using namespace std;
3  int add(int a, int b)              //重载函数1
4  {
5      return(a+b);
6  }
7  double add(double a, double b)     //重载函数2
8  {
9      return(a+b);
10 }
11 int main( )
12 {
13     cout<<add(1, 2)<<'\t';         //编译时确定调用重载函数1
14     cout<<add(1.3, 2.5)<<'\n';     //编译时确定调用重载函数2
15     return 0;
16 }
```

程序输出结果：

普通函数的静态联编

静态联编

在编译连接阶段，就能根据参数的个数和类型确定调用的是哪一个函数。

```
1  #include <iostream>
2  using namespace std;
3  int add(int a, int b)              //重载函数1
4  {
5      return(a+b);
6  }
7  double add(double a, double b)     //重载函数2
8  {
9      return(a+b);
10 }
11 int main( )
12 {
13     cout<<add(1, 2)<<'\t';         //编译时确定调用重载函数1
14     cout<<add(1.3, 2.5)<<'\n';     //编译时确定调用重载函数2
15     return 0;
16 }
```

程序输出结果：3　　3.8

成员函数的静态联编

```
1  class Point
2  {
3  protected:
4      double x, y;              // 点的坐标值
5  public:
6      Point(double a=0, double b=0)
7      {  x=a; y=b;   }
8      double Area( )            // 函数1
9      {  return 0.0; }
10 };
11 class Rectangle : public Point
12 {
13 protected:
14     double x1, y1;  // 长方形右下角点的坐标值，基类中的x，y为左上角坐标点
15 public:
16     Rectangle(double a=0, double b=0, double c=0, double d=0):Point(a, b)
17     {  x1=c; y1=d; }
18     double Area( )            // 函数2
19     {  return (x-x1)*(y-y1);  }
20 };
21 class Circle: public Point
22 {
```

成员函数的静态联编

```
23  protected:
24      double r; //半径，基类中x，y为圆心坐标点
25  public:
26      Circle(double a=0, double b=0, double c=0):Point(a, b)
27      {  r=c;  }
28      double Area( )              // 函数3
29      {  return 3.14*r*r;  }
30  };
31  double CalcArea(Point &p)     // 普通函数，编译连接时一定是调用函数1
32  {   return( p.Area( ) );  }
33  int main( )
34  {
35      Rectangle r(0, 0, 1, 1);  Circle c(0, 0, 1);
36      // 用派生类实参初始化基类形参，p只能引用基类的成员。
37      cout<<CalcArea(r)<<'\t'; // 参数传递：Point &p=r
38      cout<<CalcArea(c)<<'\n'; // 参数传递：Point &p=c
39      return 0;
40  }
```

成员函数的静态联编

```
23  protected:
24      double r; //半径，基类中x，y为圆心坐标点
25  public:
26      Circle(double a=0, double b=0, double c=0):Point(a, b)
27      {  r=c;  }
28      double Area( )              // 函数3
29      {  return 3.14*r*r;  }
30  };
31  double CalcArea(Point &p)     // 普通函数，编译连接时一定是调用函数1
32  {   return( p.Area( ) );  }
33  int main( )
34  {
35      Rectangle r(0, 0, 1, 1);  Circle c(0, 0, 1);
36      // 用派生类实参初始化基类形参，p只能引用基类的成员。
37      cout<<CalcArea(r)<<'\t'; // 参数传递：Point &p=r
38      cout<<CalcArea(c)<<'\n'; // 参数传递：Point &p=c
39      return 0;
40  }
```

程序输出结果：0　　0

成员函数的静态联编

```
23  protected:
24      double r; //半径，基类中x，y为圆心坐标点
25  public:
26      Circle(double a=0, double b=0, double c=0):Point(a, b)
27      {  r=c;  }
28      double Area( )              // 函数3
29      {  return 3.14*r*r;  }
30  };
31  double CalcArea(Point &p)     // 普通函数，编译连接时一定是调用函数1
32  {   return( p.Area( ) );  }
33  int main( )
34  {
35      Rectangle r(0, 0, 1, 1);  Circle c(0, 0, 1);
36      // 用派生类实参初始化基类形参，p只能引用基类的成员。
37      cout<<CalcArea(r)<<'\t'; // 参数传递: Point &p=r
38      cout<<CalcArea(c)<<'\n'; // 参数传递: Point &p=c
39      return 0;
40  }
```

程序输出结果：0　　0

能否寻找或设计一种机制，让CalcArea( ) 函数变成一个通用的求面积的函数。

成员函数的静态联编

```
23  protected:
24      double r; //半径，基类中x，y为圆心坐标点
25  public:
26      Circle(double a=0, double b=0, double c=0):Point(a, b)
27      {  r=c;  }
28      double Area( )              // 函数3
29      {  return 3.14*r*r;  }
30  };
31  double CalcArea(Point &p)     // 普通函数，编译连接时一定是调用函数1
32  {   return( p.Area( ) );  }
33  int main( )
34  {
35      Rectangle r(0, 0, 1, 1);  Circle c(0, 0, 1);
36      // 用派生类实参初始化基类形参，p只能引用基类的成员。
37      cout<<CalcArea(r)<<'\t'; // 参数传递: Point &p=r
38      cout<<CalcArea(c)<<'\n'; // 参数传递: Point &p=c
39      return 0;
40  }
```

程序输出结果：0　　0

能否寻找或设计一种机制，让CalcArea( ) 函数变成一个通用的求面积的函数。

这就是C++提供的虚函数和动态联编所应该完成的工作。

提纲

使用虚函数实现多态

虚函数

在类中，那些被virtual关键字修饰的成员函数，就是虚函数。

```
virtual <type> <function_name> ( [<arguments>] )
{...}
```

虚函数的作用就是实现多态性，即根据不同的类对象调用其相应的函数。

动态联编示例

将成员函数定义成虚函数，以实现动态联编。

```
#include <iostream>
using namespace std;
class Point
{
protected:
    double x, y;                    // 点的坐标值
public:
    Point(double a=0, double b=0)
    {   x=a;  y=b;   }
    virtual double Area( )     // 虚函数1
    {   return 0.0;  }
```

使用虚函数实现多态

```
12  };
13  class Rectangle : public Point
14  {
15  protected:
16      double x1, y1; // 长方形右下角点的坐标值，基类中x，y为左上角坐标点
17  public:
18      Rectangle(double a=0, double b=0, double c=0, double d=0):Point(a,b)
19      {   x1=c;  y1=d; }
20      virtual double Area( )       // 虚函数2
21      {   return (x-x1)*(y-y1);  }
22  };
23  class Circle:public Point
24  {
25  protected:
26      double r;       // 半径，基类中x，y为圆心坐标点
27  public:
28      Circle(double a=0, double b=0, double c=0):Point(a, b)
29      {   r=c;  }
30      virtual double Area( )      // 虚函数3
31      {   return 3.14*r*r;   }
32  };
33  double CalcArea(Point &p)       // 基类对象引用派生类对象。
```

使用虚函数实现多态

```
34  {
35      return(p.Area( ));          // A
36  }
37  int main()
38  {
39      Point p(1, 2);
40      Rectangle r(0, 0, 1, 1);
41      Circle c(0, 0, 1);
42      // 用派生类的对象实例化基类对象的引用，用虚函数实现动态多态
43      cout<<CalcArea(p)<<'\t'     // B 参数传递: Point &p=p
44          <<CalcArea(r)<<'\t'     // C 参数传递: Point &p=r
45          <<CalcArea(c)<<'\n';    // D 参数传递: Point &p=c
46      return 0;
47  }
```

在A处，基类的对象调用派生类虚函数，Area( )是虚函数，可认为是三个虚函数（函数1、函数2、函数3）的入口。

在程序的运行阶段，根据实参的类型来确定调用哪一个虚函数。例如，程序运行到在B、C、D处可根据函数 CalcArea(Point &p) 中实参的不同而调用相应的函数。

使用虚函数实现多态

有关虚函数的说明

- 派生类的虚函数与基类的虚函数必须同名，且参数的类型、个数、顺序必须一致。
  如果仅仅名称相同，就会变成函数“覆盖（重写，override）”，触发隐藏机制。关于重载、覆盖、隐藏这三者之间的区别，我们暂不介绍、稍后深入讨论。
- 基类中虚函数前的关键字virtual不能缺省。
- 必须通过基类对象的指针或引用调用虚函数，才能实现动态多态。
- 内联函数不能定义为虚函数。
  这是因为：内联函数是个静态行为，而虚函数是个动态行为。这两者矛盾，不可同时出现。即使将虚函数申请为内联函数，编译器肯定不允许虚函数内联。
- 静态成员函数不能定义为虚函数。
  这是因为：静态成员函数属于类，与具体对象无关。
- 友元函数不能定义为虚函数。
  这是因为：友元函数不是类的成员函数。

使用虚函数实现多态

- 不能将构造函数定义为虚函数，但可将析构函数定义为虚函数。
- 在构造函数中调用虚函数，不遵循动态多态规则，即：这种情况下调用的是类自身的虚函数。
- 虚函数与一般函数相比，程序调用时的执行速度要慢一些。
  这是因为：为了实现动态联编，编译器为每个含有虚函数的对象增加指向虚函数地址表的指针，通过该指针实现对虚函数的间接调用。

以上涉及虚函数的工作原理，我们对此暂不深究、稍后介绍。

在成员函数中调用虚函数

- 在成员函数中调用成员函数时，系统都是通过对象自身的指针this调用的。

```
1  class A
2  {
3  public:
4      virtual void fun1( ) { cout << "A::fun1" << '\t'; fun2( ); } //虚函数
5              void fun2( ) { cout << "A::fun2" << '\t'; fun3( ); } //E
6
7      virtual void fun3( ) { cout << "A::fun3" << '\t'; fun4( ); } //虚函数
8              void fun4( ) { cout << "A::fun4" << '\n'; }
9  };
10 class B: public A
11 {
12 public:
13             void fun3( ) { cout << "B::fun3" << '\t'; fun4( ); }
14             void fun4( ) { cout << "B::fun4" << '\n';          }
15 };
16 int main( )
17 {
18     A  a;  a.fun1( );
19     B  b;  b.fun1( );
20     return 0;
21 }
```

在成员函数中调用虚函数

程序（Polymorphism3.cpp）的运行结果是：
A::fun1 A::fun2 A::fun3 A::fun4
A::fun1 A::fun2 B::fun3 B::fun4

在成员函数中调用成员函数时，系统都是通过对象自身的指针this调用的。例如，类 A中的fun2( ) 的实际被处理成如下形式：

```
5       void fun2( )
6       {
7           cout << "A::fun2" << '\t';
8           this->fun3( );                          //E
9       }
```

在E行, this 是指向b的指针, 所以调用B的fun3( )函数。

在构造函数中调用虚函数

- 构造函数调用虚函数，调用的是类本身的虚函数；成员函数调用虚函数，遵循动态多态性原则。

```
1  class A
2  {
3  public:
4      A( )
5      { fun( ); }
6      virtual void fun( )
7      { cout<<"A::fun"<<'\t'; }
8  };
9  class B:  public A
10 {
11 public:
12     B( )
13     { fun( ); }
14     void fun( )
15     { cout<<"B::fun"<<'\t'; }
16     void g( )
17     { fun( );} // 调用虚函数
19 };
20 class C: public B
21 {
22 public:
23     C( )
24     { fun( ); }
25     void fun( )
26     { cout<<"C::fun"<<endl; }
27 };
28 int main( )
29 {
30     // 依次调用A、B、C三类的缺省
       构造函数
31     C c;
32     c.g( );
33     return 0;
34 }
```

运行结果：

在构造函数中调用虚函数

- 构造函数调用虚函数，调用的是类本身的虚函数；成员函数调用虚函数，遵循动态多态性原则。

```
1  class A
2  {
3  public:
4      A( )
5      { fun( ); }
6      virtual void fun( )
7      { cout<<"A::fun"<<'\t'; }
8  };
9  class B:  public A
10 {
11 public:
12     B( )
13     { fun( ); }
14     void fun( )
15     { cout<<"B::fun"<<'\t'; }
16     void g( )
17     { fun( );} // 调用虚函数
19 };
20 class C: public B
21 {
22 public:
23     C( )
24     { fun( ); }
25     void fun( )
26     { cout<<"C::fun"<<endl; }
27 };
28 int main( )
29 {
30     // 依次调用A、B、C三类的缺省
       构造函数
31     C c;
32     c.g( );
33     return 0;
34 }
```

运行结果：A::fun B::fun C::fun
C::fun

定义虚析构函数

虚析构函数

如果类的构造函数中有动态申请的存储空间，在析构函数中应释放该空间。此时，建议将析构函数定义为虚函数，以便实现通过基类的指针或引用撤消派生类对象时的多态性。

- 析构函数“不是虚函数”的情况

```
1  #include <iostream>
2  using namespace std;
3  class A
4  {
5      char *Aptr;
6  public:
7      A( )
8      { Aptr = new char[100]; }
9      ~A( )  // 析构函数不是虚函数
10     {
11         delete [ ]Aptr;
12         cout<<"Delete␣[␣]Aptr"<<
       endl;
13     }
14 };
15 class B : public A
16 {
17     char *Bptr;
18 public:
19     B( )
20     { Bptr=new char[100]; }
21     ~B( )
22     {
23         delete [ ]Bptr;
24         cout<<"Delete␣[␣]Bptr"<<
       endl;
25     }
26 };
```

虚析构函数

```
27  int main( )
28  {
29      B b;
30      return 0;
31  }
```

系统自动撤销派生类对象，不需要将析构函数定义为虚函数。

输出结果：

```
27  int main()
28  {
29      A *p1=new B;     // 基类的指针
30      delete p1;
31      A &p2=*(new B); // 基类的引用
32      delete &p2;
33      return 0;
34  }
```

输出结果：

虚析构函数

```
27  int main( )
28  {
29      B b;
30      return 0;
31  }
```

系统自动撤销派生类对象，不需要将析构函数定义为虚函数。

输出结果：

Delete [ ]Bptr

Delete [ ]Aptr

```
27  int main()
28  {
29      A *p1=new B;     // 基类的指针
30      delete p1;
31      A &p2=*(new B); // 基类的引用
32      delete &p2;
33      return 0;
34  }
```

输出结果：

虚析构函数

```
27  int main( )
28  {
29      B b;
30      return 0;
31  }
```

系统自动撤销派生类对象，不需要将析构函数定义为虚函数。

输出结果：
Delete [ ]Bptr
Delete [ ]Aptr

```
27  int main()
28  {
29      A *p1=new B;     // 基类的指针
30      delete p1;
31      A &p2=*(new B); // 基类的引用
32      delete &p2;
33      return 0;
34  }
```

输出结果：
Delete [ ]Aptr
Delete [ ]Aptr

如果将析构函数定义成虚析构函数，则运行结果为：

虚析构函数

```
27  int main( )
28  {
29      B b;
30      return 0;
31  }
```

系统自动撤销派生类对象，不需要将析构函数定义为虚函数。

输出结果：
Delete [ ]Bptr
Delete [ ]Aptr

```
27  int main()
28  {
29      A *p1=new B;     // 基类的指针
30      delete p1;
31      A &p2=*(new B); // 基类的引用
32      delete &p2;
33      return 0;
34  }
```

输出结果：
Delete [ ]Aptr
Delete [ ]Aptr

如果将析构函数定义成虚析构函数，则运行结果为：

```
9   virtual ~A( ) // 类B的析构函数自
        然也是虚析构函数
10  { ... }
```

虚析构函数

```
27  int main( )
28  {
29      B b;
30      return 0;
31  }
```

系统自动撤销派生类对象，不需要将析构函数定义为虚函数。

输出结果：
Delete [ ]Bptr
Delete [ ]Aptr

```
27  int main()
28  {
29      A *p1=new B;     // 基类的指针
30      delete p1;
31      A &p2=*(new B); // 基类的引用
32      delete &p2;
33      return 0;
34  }
```

输出结果：
Delete [ ]Aptr
Delete [ ]Aptr

如果将析构函数定义成虚析构函数，则运行结果为：

```
9   virtual ~A( ) // 类B的析构函数自
        然也是虚析构函数
10  { ... }
```

Delete [ ]Bptr //派生类析构
Delete [ ]Aptr //基类析构
Delete [ ]Bptr //派生类析构
Delete [ ]Aptr //基类析构

虚析构函数

```
27  int main( )
28  {
29      B b;
30      return 0;
31  }
```

系统自动撤销派生类对象，不需要将析构函数定义为虚函数。

输出结果：
Delete [ ]Bptr
Delete [ ]Aptr

```
27  int main()
28  {
29      A *p1=new B;     // 基类的指针
30      delete p1;
31      A &p2=*(new B); // 基类的引用
32      delete &p2;
33      return 0;
34  }
```

输出结果：
Delete [ ]Aptr
Delete [ ]Aptr

如果将析构函数定义成虚析构函数，则运行结果为：

```
9   virtual ~A( ) // 类B的析构函数自
        然也是虚析构函数
10  { ... }
```

Delete [ ]Bptr //派生类析构
Delete [ ]Aptr //基类析构
Delete [ ]Bptr //派生类析构
Delete [ ]Aptr //基类析构

结论：对于虚函数，若用派生类的指针或对象初始化基类的指针或引用，则通过该指针或引用调用虚函数，遵循派生类析构函数的“逐层”调用规则。

虚析构函数的作用

- 当一个类被用来作为基类的时候，把这个类的析构函数定义成虚函数。

```
1  class ClxBase
2  {
3  public:
4      ClxBase() {};
5      virtual ~ClxBase() {};
6
7      virtual void DoSomething() { cout << "Do something in class ClxBase!"
       << endl; };
8  };
9
10 class ClxDerived : public ClxBase
11 {
12 public:
13     ClxDerived() {};
14     ~ClxDerived() { cout << "Output from the destructor of class
       ClxDerived!" << endl; };
15
16     void DoSomething() { cout << "Do something in class ClxDerived!" <<
       endl; };
17 };
```

虚析构函数的作用

代码

```
18      ClxBase *pTest = new ClxDerived;
19      pTest->DoSomething();
20      delete pTest;
```

的输出结果是:

虚析构函数的作用

代码

```
18      ClxBase *pTest = new ClxDerived;
19      pTest->DoSomething();
20      delete pTest;
```

的输出结果是：

Do something in class ClxDerived!

Output from the destructor of class ClxDerived!

如果把类ClxBase析构函数前的virtual去掉，那输出结果是：

虚析构函数的作用

代码

```
18      ClxBase *pTest = new ClxDerived;
19      pTest->DoSomething();
20      delete pTest;
```

的输出结果是：
Do something in class ClxDerived!
Output from the destructor of class ClxDerived!

如果把类ClxBase析构函数前的virtual去掉，那输出结果是：
Do something in class ClxDerived!

把基类的析构函数定义成虚函数的原因在于：

虚析构函数的作用

代码

```
18      ClxBase *pTest = new ClxDerived;
19      pTest->DoSomething();
20      delete pTest;
```

的输出结果是：
Do something in class ClxDerived!
Output from the destructor of class ClxDerived!

如果把类ClxBase析构函数前的virtual去掉，那输出结果是：
Do something in class ClxDerived!

把基类的析构函数定义成虚函数的原因在于：
一般情况下类的析构函数应完成内存资源释放，而析构函数不被调用的话就会造成内存泄漏。当用一个基类的指针删除一个派生类的对象时，要确保派生类的析构函数会被调用，即确保不会内存泄漏。

当然，**并不是要把所有类的析构函数都写成虚函数**。因为：

虚析构函数的作用

代码

```
18      ClxBase *pTest = new ClxDerived;
19      pTest->DoSomething();
20      delete pTest;
```

的输出结果是：
Do something in class ClxDerived!
Output from the destructor of class ClxDerived!

如果把类ClxBase析构函数前的virtual去掉，那输出结果是：
Do something in class ClxDerived!

把基类的析构函数定义成虚函数的原因在于：
一般情况下类的析构函数应完成内存资源释放，而析构函数不被调用的话就会造成内存泄漏。当用一个基类的指针删除一个派生类的对象时，要确保派生类的析构函数会被调用，即确保不会内存泄漏。

当然，**并不是要把所有类的析构函数都写成虚函数**。因为：当类里面有虚函数的时候，编译器会给类添加一个虚函数表用来存放虚函数指针，这样就会增加类的存储空间。所以，只有当一个类被用来作为基类的时候，才把析构函数写成虚函数。

提纲

虚函数的工作原理

再谈虚函数

虚函数是C++多态性的主要体现，指向基类的指针在操作它的多态类对象时，会根据不同的类对象，调用其相应的函数，这个函数就是虚函数。

虚函数的工作原理

再谈虚函数

虚函数是C++多态性的主要体现，指向基类的指针在操作它的多态类对象时，会根据不同的类对象，调用其相应的函数，这个函数就是虚函数。

虚函数工作原理

在C++中，虚函数是通过虚函数表（Virtual Table，VTABLE）来实现的。在这个表中，存储的是虚函数的地址表。这张表解决了继承、覆盖的问题，保证其容真实反应实际的函数。

虚函数的工作原理

再谈虚函数

虚函数是C++多态性的主要体现，指向基类的指针在操作它的多态类对象时，会根据不同的类对象，调用其相应的函数，这个函数就是虚函数。

虚函数工作原理

在C++中，虚函数是通过虚函数表（Virtual Table，VTABLE）来实现的。在这个表中，存储的是虚函数的地址表。这张表解决了继承、覆盖的问题，保证其容真实反应实际的函数。

每当创建一个包含有虚函数的类（基类）或从包含有虚函数的类派生一个类派生类时，编译器就会为这个类创建一个VTABLE保存该类所有虚函数的地址。这个VTABLE的作用就是保存自己类中所有虚函数的地址。可以把VTABLE形象地看成一个函数指针数组，其每个元素存放的就是虚函数的地址。

虚函数的工作原理

再谈虚函数

虚函数是C++多态性的主要体现，指向基类的指针在操作它的多态类对象时，会根据不同的类对象，调用其相应的函数，这个函数就是虚函数。

虚函数工作原理

在C++中，虚函数是通过虚函数表（Virtual Table，VTABLE）来实现的。在这个表中，存储的是虚函数的地址表。这张表解决了继承、覆盖的问题，保证其容真实反应实际的函数。

每当创建一个包含有虚函数的类（基类）或从包含有虚函数的类派生一个类派生类时，编译器就会为这个类创建一个VTABLE保存该类所有虚函数的地址。这个VTABLE的作用就是保存自己类中所有虚函数的地址。可以把VTABLE形象地看成一个函数指针数组，其每个元素存放的就是虚函数的地址。

在每个带有虚函数的类（基类和派生类）中，编译器秘密地置入虚函数指针（V-Pointer，VPTR ）指向类的对象的VTABLE。 当构造该派生类对象时，其成员VPTR被初始化并指向该派生类的VTABLE。因此，可认为VTABLE 是该类的所有对象共有，在定义该类时被初始化；而VPTR则是每个类对象都独有一份，且在该类对象被构造时被初始化。

虚函数的工作原理

假设我们有这样的一个类Base及其对象b：

```
1  class Base
2  {
3  public:
4      virtual void f() { cout << "Base::f" << endl; }
5      virtual void g() { cout << "Base::g" << endl; }
6      virtual void h() { cout << "Base::h" << endl; }
7  }b;
```

总结一下VTABLE、VPTR和类对象的关系

- 每一个具有虚函数的类都有一个虚函数表VTABLE，按虚函数在类中声明的顺序存放着虚函数的地址，这个虚函数表VTABLE是这个类的所有对象所共有。
- 在每个具有虚函数的类的对象里面都有一个VPTR虚函数指针，这个指针指向VTABLE的首地址，每个类的对象都有这么一种指针。

虚函数的工作原理

- 当派生类中的函数与其基类虚函数原型一致时，该函数才是“虚函数”。

```
1  class A
2  {
3  public:
4      virtual void fun()
5      {  cout <<"A::fun"<<endl;  }
6  };
7  class B: public A
8  {
9  public:
10     void fun(int x=0)//非虚函数
11     {  cout <<"A::fun"<<endl; }
12 };
13 int main
14 {
15     B b;          A *pa = &b;
16     pa->fun();    b.fun();
17     return 0;
18 }
```

```
14 class A
15 {
16 public:
17     virtual void fun()
18     {  cout <<"A::fun"<<endl;  }
19 };
20 class B: public A
21 {
22 public:
23     void fun()  // 虚函数
24     {  cout <<"B::fun"<<endl;  }
25 };
26 int main
27 {
28     B b;          A *pa = &b;
29     pa->fun();    b.fun();
30     return 0;
31 }
```

虚函数的工作原理

- 当派生类中的函数与其基类虚函数原型一致时，该函数才是“虚函数”。

```
1  class A
2  {
3  public:
4      virtual void fun()
5      {  cout <<"A::fun"<<endl;  }
6  };
7  class B: public A
8  {
9  public:
10     void fun(int x=0)//非虚函数
11     {  cout <<"A::fun"<<endl; }
12 };
13 int main
14 {
15     B b;          A *pa = &b;
16     pa->fun();    b.fun();
17     return 0;
18 }
```

```
14 class A
15 {
16 public:
17     virtual void fun()
18     {  cout <<"A::fun"<<endl;  }
19 };
20 class B: public A
21 {
22 public:
23     void fun()  // 虚函数
24     {  cout <<"B::fun"<<endl;  }
25 };
26 int main
27 {
28     B b;          A *pa = &b;
29     pa->fun();    b.fun();
30     return 0;
31 }
```

运行结果：A::fun
B::fun

虚函数的工作原理

- 当派生类中的函数与其基类虚函数原型一致时，该函数才是“虚函数”。

```
1  class A
2  {
3  public:
4      virtual void fun()
5      {  cout <<"A::fun"<<endl;  }
6  };
7  class B: public A
8  {
9  public:
10     void fun(int x=0)//非虚函数
11     {  cout <<"A::fun"<<endl; }
12 };
13 int main
14 {
15     B b;          A *pa = &b;
16     pa->fun();    b.fun();
17     return 0;
18 }
```

运行结果：A::fun
B::fun

```
14 class A
15 {
16 public:
17     virtual void fun()
18     {  cout <<"A::fun"<<endl;  }
19 };
20 class B: public A
21 {
22 public:
23     void fun()  // 虚函数
24     {  cout <<"B::fun"<<endl;  }
25 };
26 int main
27 {
28     B b;          A *pa = &b;
29     pa->fun();    b.fun();
30     return 0;
31 }
```

运行结果：B::fun
B::fun

提纲

纯虚函数与抽象类

在定义基类时，会遇到这样的情况：基类中虚函数的具体实现依赖于派生类，这就导致无法定义基类中虚函数的具体实现。此时，就需要引入纯虚函数。

纯虚函数与抽象类

在定义基类时，会遇到这样的情况：基类中虚函数的具体实现依赖于派生类，这就导致无法定义基类中虚函数的具体实现。此时，就需要引入纯虚函数。

例如，定义基类Shape实现对形状的通用操作，包括求这个形状的面积、绘制这个形状的图形等。对形状的通用操作会随派生类（类Point、 类Rectangle、类Circle等）的不同而发生变化，所以需要在派生类中实现相关具体操作。

纯虚函数与抽象类

在定义基类时，会遇到这样的情况：基类中虚函数的具体实现依赖于派生类，这就导致无法定义基类中虚函数的具体实现。此时，就需要引入纯虚函数。

例如，定义基类Shape实现对形状的通用操作，包括求这个形状的面积、绘制这个形状的图形等。对形状的通用操作会随派生类（类Point、 类Rectangle、类Circle等）的不同而发生变化，所以需要在派生类中实现相关具体操作。

我们把基类中的通用操作定义为纯虚函数，并把至少包含一个纯虚函数的类称为抽象类（抽象数据类型）。定义纯虚函数的一般格式为：

```
1   virtual <type> <function_name> ( [<arguments>] ) = 0
```

函数参数列表圆括号后面的“= 0”，表示将函数名的值赋为0。

纯虚函数与抽象类

在定义基类时，会遇到这样的情况：基类中虚函数的具体实现依赖于派生类，这就导致无法定义基类中虚函数的具体实现。此时，就需要引入纯虚函数。

例如，定义基类Shape实现对形状的通用操作，包括求这个形状的面积、绘制这个形状的图形等。对形状的通用操作会随派生类（类Point、 类Rectangle、类Circle等）的不同而发生变化，所以需要在派生类中实现相关具体操作。

我们把基类中的通用操作定义为纯虚函数，并把至少包含一个纯虚函数的类称为抽象类（抽象数据类型）。定义纯虚函数的一般格式为：

```
1  virtual <type> <function_name> ( [<arguments>] ) = 0
```

函数参数列表圆括号后面的“= 0”，表示将函数名的值赋为0。

关于纯虚函数和抽象类的使用

- 抽象类只能做派生类的基类，不能定义抽象类的对象。
- 一般而言，纯虚函数没有函数体，但可以给出纯虚函数的函数体。
- 若派生类实现了基类所有的纯虚函数，则派生类就不再是抽象类。
- 若派生类没有实现基类所有的纯虚函数，则派生类依然是抽象类。

纯虚函数与抽象类

- 定义一个抽象类，给出纯虚函数的函数体。

```
1  #include <iostream>
2  using namespace std;
3
4  class A
5  {
6  public:
7      virtual void fun( ) = 0;
8  };
9  void A::fun( )
10 {
11     cout<<"virtual␣fun=0"<<endl;
12 }
13
14 class B: public A
15 {
16 public:
17     virtual void fun( )
18     { cout<<"virtual␣fun"<<endl; }
19 };
```

纯虚函数与抽象类

```
20  int main( )
21  {
22      B b;
23      b.fun( );
24      b.A::fun( );  // 调用纯虚函数的函数体
25      return 0;
26  }
```

程序的运行结果是:
virtual fun
virtual fun=0

注意
即使给出了纯虚函数的函数体，该函数依然是纯虚函数，含有该函数的类依然是抽象类。
在本例中，不可以定义类A的对象。

PureVirtualFunction2.cpp

纯虚函数与抽象类

- 定义一个抽象类并派生出若干类，在派生类中实现纯虚函数。

```
#include <iostream>
using namespace std;

class Shape
{
public:
    virtual double Area( )=0; //抽象的求面积的纯虚函数
    virtual void   Draw( )=0; //抽象的绘制图形的纯虚函数
};
class Point:public Shape
{
protected:
    double x, y;    //点的坐标值
public:
    Point(double a=0, double b=0)
    {
        x=a;
        y=b;
    }
```

纯虚函数与抽象类

```
20      double Area( )  //基类纯虚函数Area( )的具体实现
21      {
22          return 0.0;
23      }
24      void Draw( )    //基类纯虚函数Draw( )的具体实现
25      {
26          cout <<"Draw Point!\n";
27      }
28  };
29  class Rectangle:public Point
30  {
31  protected:
32      double x1, y1;  //长方形右下角点的坐标值，基类中x，y为左上角坐标点
33  public:
34      Rectangle(double a=0, double b=0, double c=0, double d=0):Point(a, b)
35      {
36          x1=c;
37          y1=d;
38      }
39      double Area( )  //基类纯虚函数Area( )的具体实现
40      {
41          return (x-x1)*(y-y1);
```

纯虚函数与抽象类

```
42      }
43      void Draw( )     //基类纯虚函数Draw( )的具体实现
44      {
45          cout <<"Draw␣Rectangle!\n";
46      }
47  };
48  class Circle:public Point
49  {
50  protected:
51      double r;       //半径，基类中x，y为圆心坐标点
52  public:
53      Circle(double a=0, double b=0, double c=0):Point(a, b)
54      {
55          r=c;
56      }
57      double Area( ) //基类纯虚函数Area( )的具体实现
58      {
59          return 3.14*r*r;
60      }
61      void Draw( )    //基类纯虚函数Draw( )的具体实现
62      {
63          cout <<"Draw␣Circle!\n";
```

纯虚函数与抽象类

```
64      }
65  };
66
67  double CalcArea(Shape &s)
68  {
69      return(s.Area( )); //A通过基类对象的引用实现动态多态
70  }
71
72  void DrawShape(Shape *sp)
73  {
74      sp->Draw( );        //B通过基类对象的指针实现动态多态
75  }
76
77  int main( )
78  {
79      Point p(1, 2);
80      Rectangle r(0, 0, 1, 1);
81      Circle c(0, 0, 1);
82      cout<<CalcArea(p)<<'\t'<<CalcArea(r)<<'\t'<<CalcArea(c)<<'\n';
83      DrawShape(&p);
84      DrawShape(&r);
85      DrawShape(&c);
```

纯虚函数与抽象类

```
86      return 0;
87  }
```

程序（PureVirtualFunction.cpp）的运行结果是：

0 1 3.14
Draw Point!
Draw Rectangle!
Draw Circle!

定义抽象类

可以使编译器自动查找“创建抽象类的对象”这种错误、避免运行阶段的出错，还可以让开发者与其他应用程序或程序员共享抽象数据类型。

纯虚函数和抽象类小结

- 纯虚函数让C++能够更健壮地支持面向对象编程。
- 包含一个纯虚函数的类就是抽象类（抽象数据类型）。
- 抽象类定义了派生类都需要的成员变量和成员函数。
- 抽象类是不能实例化的类，不能定义抽象类的对象。

提纲

总结与思考

思考

1. 为什么不将类的所有成员函数都声明为虚函数？

总结与思考

思考

1. 为什么不将类的所有成员函数都声明为虚函数？
   创建VTABLE的开销伴随第一个虚函数的创建而发生。在此之后，创建其他虚函数带来的开销将很小。有C++程序员认为，如果一个函数为虚函数，那么其他所有函数也应为虚函数。但也有C++程序员认为，无论做什么都应有充分的理由。
2. 为何要创建抽象类？为何不避免创建抽象数据类型？

总结与思考

思考

1. 为什么不将类的所有成员函数都声明为虚函数？
   创建VTABLE的开销伴随第一个虚函数的创建而发生。在此之后，创建其他虚函数带来的开销将很小。有C++程序员认为，如果一个函数为虚函数，那么其他所有函数也应为虚函数。但也有C++程序员认为，无论做什么都应有充分的理由。
2. 为何要创建抽象类？为何不避免创建抽象数据类型？
   C++的很多规则都旨在让编译器查找bug，从而避免出现运行阶段的bug。通过在类中包含纯虚函数使类成为抽象类，编译器就可以查找“创建抽象类的对象”这种错误，从而避免运行阶段的错误。此外，通过定义抽象数据类型，可以让编程者与其他应用程序或程序员共享抽象数据类型。

练习

1. 哪些函数不能是虚函数？（　　）
   A. 构造函数　B. 析构函数　C. 克隆函数　D. 拷贝构造函数

2. 对象存储在基类变量中时，C++如何知道该调用哪个虚函数？（　　）
   A. 关键词virtual　B. 根据VTABLE　C. 这不可能　D. 无法判断

总结与思考

思考

1. 为什么不将类的所有成员函数都声明为虚函数？
   创建VTABLE的开销伴随第一个虚函数的创建而发生。在此之后，创建其他虚函数带来的开销将很小。有C++程序员认为，如果一个函数为虚函数，那么其他所有函数也应为虚函数。但也有C++程序员认为，无论做什么都应有充分的理由。
2. 为何要创建抽象类？为何不避免创建抽象数据类型？
   C++的很多规则都旨在让编译器查找bug，从而避免出现运行阶段的bug。通过在类中包含纯虚函数使类成为抽象类，编译器就可以查找“创建抽象类的对象”这种错误，从而避免运行阶段的错误。此外，通过定义抽象数据类型，可以让编程者与其他应用程序或程序员共享抽象数据类型。

练习

1. 哪些函数不能是虚函数？ （A D）
   A. 构造函数　B. 析构函数　C. 克隆函数　D. 拷贝构造函数

2. 对象存储在基类变量中时，C++如何知道该调用哪个虚函数？（　）
   A. 关键词virtual　B. 根据VTABLE　C. 这不可能　D. 无法判断

总结与思考

思考

1. 为什么不将类的所有成员函数都声明为虚函数？
   创建VTABLE的开销伴随第一个虚函数的创建而发生。在此之后，创建其他虚函数带来的开销将很小。有C++程序员认为，如果一个函数为虚函数，那么其他所有函数也应为虚函数。但也有C++程序员认为，无论做什么都应有充分的理由。
2. 为何要创建抽象类？为何不避免创建抽象数据类型？
   C++的很多规则都旨在让编译器查找bug，从而避免出现运行阶段的bug。通过在类中包含纯虚函数使类成为抽象类，编译器就可以查找“创建抽象类的对象”这种错误，从而避免运行阶段的错误。此外，通过定义抽象数据类型，可以让编程者与其他应用程序或程序员共享抽象数据类型。

练习

1. 哪些函数不能是虚函数？ （A D）
   A. 构造函数　B. 析构函数　C. 克隆函数　D. 拷贝构造函数

2. 对象存储在基类变量中时，C++如何知道该调用哪个虚函数？（ B ）
   A. 关键词virtual　B. 根据VTABLE　C. 这不可能　D. 无法判断

总结与思考 · 作业

阅读题：请写出如下多重继承中构造函数的调用顺序