程序设计与算法语言 · 再谈重载
C/C++ Programming and Algorithms
Overload and Polymorphism

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

Spring semester, 2019

提纲

提纲

函数重载

函数重载（Function Overload）

函数重载是指：在同一作用域内，定义一组具有相同函数名、不同参数列表的函数。这组函数被称为重载函数。

重载函数通常用来命名一组功能相似的函数，这样做减少了函数名的数量，避免了名字空间的污染，对于程序的可读性有很大的好处。

```
1  #include<iostream>
2  using namespace std;
3
4  void print(int i)       {    cout<<"print␣a␣integer␣:␣"<<i<<endl;     }
5
6  void print(string str) {    cout<<"print␣a␣string␣:␣"<<str<<endl;    }
7
8  int main()
9  {
10     print(12);
11     print("hello␣world!");
12     return 0;
13 }
```

函数重载

- 为保持解析操作符或函数调用时独立于上下文，函数重载与返回类型无关。

```
1  float  sqr(float  f)
2  {
3      return f*f;
4  }
5  double sqr(double d)
6  {
7      return d*d;
8  }
9  void fun(double dbe, float flt)
10 {
11     float  fl = sqr(dbe);// 调用sqr(double)
12     double db = sqr(dbe);// 调用sqr(double)
13
14     fl = sqr(fla);       // 调用sqr(float)
15     db = sqr(fla);       // 调用sqr(float)
16 }
```

如果将返回类型考虑到函数重载中，在程序运行时就不能独立于上下文决定调用哪个函数。因此，就规定函数返回类型不能用于区分重载。

函数重载

- 除函数名、参数个数、参数类型外，作用域是判断函数重载的重要限定。

```
1  namespace mkt
2  {
3      void print(int i)  {    cout<<"print␣a␣integer␣:␣"<<i<<endl;     }
4  }
5  void print(string str) {    cout<<"print␣a␣string␣:␣"<<str<<endl;    }
6  int main()
7  {
8      mkt::print(12);
9      print("hello␣world!");
10     return 0;
11 }
```

程序运行结果：

函数重载

- 除函数名、参数个数、参数类型外，作用域是判断函数重载的重要限定。

```
1  namespace mkt
2  {
3      void print(int i)  {    cout<<"print a integer : "<<i<<endl;     }
4  }
5  void print(string str) {    cout<<"print a string : "<<str<<endl;    }
6  int main()
7  {
8      mkt::print(12);
9      print("hello world!");
10     return 0;
11 }
```

程序运行结果：print a integer : 12
print a string : hello world!

重载小结

函数重载

- 除函数名、参数个数、参数类型外，作用域是判断函数重载的重要限定。

```
1  namespace mkt
2  {
3      void print(int i)  {    cout<<"print a integer : "<<i<<endl;     }
4  }
5  void print(string str) {    cout<<"print a string : "<<str<<endl;    }
6  int main()
7  {
8      mkt::print(12);
9      print("hello world!");
10     return 0;
11 }
```

程序运行结果：print a integer : 12
print a string : hello world!

重载小结：函数的重载一定要在同一个作用域中，不能用函数的返回值来区分重载。重载的依据有：函数名（同名是前提）、参数个数（默认的参数不计入参数个数）、参数类型、有无关键词const。

函数在重载时，利用函数签名（即函数名、参数列表与关键词const）的不同来决定到底调用哪个方法。

提纲

同名隐藏现象

不同作用域声明的标识符的可见性原则

- 在派生层次结构中，基类的成员和派生类新增的成员都有各自的作用域：1) 它们的作用范围不同；2) 它们是相互包含的两个层；3) 基类在外层、派生类在内层。

作用域分辨符，就是"::"，它可以用来限定要访问的成员所在的类的名称。

同名隐藏现象

不同作用域声明的标识符的可见性原则

- 在派生层次结构中，基类的成员和派生类新增的成员都有各自的作用域：1) 它们的作用范围不同；2) 它们是相互包含的两个层；3) 基类在外层、派生类在内层。

```
1  class BaseClass
2  {
3      class DeriveClass
4      {
5      };
6  };
```

作用域分辨符，就是"::"，它可以用来限定要访问的成员所在的类的名称。

同名隐藏现象

不同作用域声明的标识符的可见性原则

- 在派生层次结构中，基类的成员和派生类新增的成员都有各自的作用域：1) 它们的作用范围不同；2) 它们是相互包含的两个层；3) 基类在外层、派生类在内层。

```
1  class BaseClass
2  {
3      class DeriveClass
4      {
5      };
6  };
```

- 如果存在两个或多个具有包含关系的作用域：外层声明了一个标识符，而内层没有再次声明同名标识符，那么外层标识符在内层依然可见，如果在内层声明了同名标识符，则外层标识符在内层不可见，这时称内层标识符隐藏了外层同名标识符，这种现象称为隐藏。

作用域分辨符，就是"::"，它可以用来限定要访问的成员所在的类的名称。

同名隐藏现象

不同作用域声明的标识符的可见性原则

- 在派生层次结构中，基类的成员和派生类新增的成员都有各自的作用域：1) 它们的作用范围不同；2) 它们是相互包含的两个层；3) 基类在外层、派生类在内层。

```
1  class BaseClass
2  {
3      class DeriveClass
4      {
5      };
6  };
```

- 如果存在两个或多个具有包含关系的作用域：外层声明了一个标识符，而内层没有再次声明同名标识符，那么外层标识符在内层依然可见，如果在内层声明了同名标识符，则外层标识符在内层不可见，这时称内层标识符隐藏了外层同名标识符，这种现象称为隐藏。

- 如果派生类声明了一个和基类某个成员同名的新成员，派生类的新成员就隐藏了外层同名成员，直接使用成员名只能访问到派生类的成员。
- 如果派生类中声明了与基类同名的新函数，即使函数的参数表不同，从基类继承的同名函数的所有重载形式也都被隐藏。
- 如果要访问被隐藏的成员，就需要使用作用域分辨符和基类名来限定。

作用域分辨符，就是"::"，它可以用来限定要访问的成员所在的类的名称。

同名隐藏现象

不同作用域声明的标识符的可见性原则

- 在派生层次结构中，基类的成员和派生类新增的成员都有各自的作用域：1) 它们的作用范围不同；2) 它们是相互包含的两个层；3) 基类在外层、派生类在内层。

```
1  class BaseClass
2  {
3      class DeriveClass
4      {
5      };
6  };
```

- 如果存在两个或多个具有包含关系的作用域：外层声明了一个标识符，而内层没有再次声明同名标识符，那么外层标识符在内层依然可见，如果在内层声明了同名标识符，则外层标识符在内层不可见，这时称内层标识符隐藏了外层同名标识符，这种现象称为隐藏。

- 如果派生类声明了一个和基类某个成员同名的新成员，派生类的新成员就隐藏了外层同名成员，直接使用成员名只能访问到派生类的成员。
- 如果派生类中声明了与基类同名的新函数，即使函数的参数表不同，从基类继承的同名函数的所有重载形式也都被隐藏。
- 如果要访问被隐藏的成员，就需要使用作用域分辨符和基类名来限定。

作用域分辨符，就是"::"，它可以用来限定要访问的成员所在的类的名称。

同名隐藏的验证

- 不能由派生类的对象直接调用被派生类隐藏的基类成员函数。

```
1  #include<iostream>
2  using namespace std;
3  class A
4  {
5  public:
6      void print()
7      {  cout<<"A print!"<<endl;    }
8  };
9  class B : public A
10 {
11 public:
12     void print(int x)
13     {  cout<<"B print!"<<x<<endl; }
14 };
15 int main()
16 {
17     B b;
18     b.print();          // 此处，编译器报错
19     return 0;
20 }
```

同名隐藏的验证

- 使用作用域分辨符和基类名访问被派生类隐藏的基类成员函数。

```
1  #include<iostream>
2  using namespace std;
3  class A
4  {
5  public:
6      void print()
7      {  cout<<"A print!"<<endl;    }
8  };
9  class B : public A
10 {
11 public:
12     void print(int x)
13     {  cout<<"B print!"<<x<<endl; }
14 };
15 int main()
16 {
17     B b;
18     b.A::print();      // 使用作用域分辨符和基类名
19     return 0;
20 }
```

同名隐藏的验证

- 利用using关键字将基类中的名字print引入到派生类作用域中。

```
1  #include<iostream>
2  using namespace std;
3  class A
4  {
5  public:
6      void print()
7      {  cout<<"A print !"<<endl;    }
8  };
9  class B : public A
10 {
11 public:
12     using A::print;    // 这里使用了关键词using
13     void print(int x)
14     {  cout<<"B print !"<<x<<endl; }
15 };
16 int main()
17 {
18     B b;
19     b.print();         // 不会报错
20     return 0;
21 }
```

同名隐藏的原因

隐藏发生的主要原因

如果派生类有与基类成员同名的成员（变量和函数），当派生类对象访问该成员时，为避免发生访问冲突，编译器优先考虑派生类域中的自身成员。

- 派生类对象访问某成员（包含成员变量和成员函数）时，编译器首先在派生类域中检索：
  - 如果在派生类域中找到该成员，则检索结束，返回该成员进行访问。
  - 如果在派生类域中找不到该成员，则去基类域中检索。
  - 如果基类域中存在，则返回该成员进行访问。如果基类域中也不存在，则编译错误，该成员无效。
- 当基类、派生类域都存在同名成员时，编译器优先在派生类域中检索，即使基类域中存在同名成员，基类中的该成员也不会被检索到。

同名隐藏的原因

隐藏发生的主要原因

如果派生类有与基类成员同名的成员（变量和函数），当派生类对象访问该成员时，为避免发生访问冲突，编译器优先考虑派生类域中的自身成员。

- 派生类对象访问某成员（包含成员变量和成员函数）时，编译器首先在派生类域中检索：
  - 如果在派生类域中找到该成员，则检索结束，返回该成员进行访问。
  - 如果在派生类域中找不到该成员，则去基类域中检索。
  - 如果基类域中存在，则返回该成员进行访问。如果基类域中也不存在，则编译错误，该成员无效。
- 当基类、派生类域都存在同名成员时，编译器优先在派生类域中检索，即使基类域中存在同名成员，基类中的该成员也不会被检索到。
- 当基类域中的成员被派生类域中的同名成员隐藏时：
  - 如果需要访问被派生类隐藏的基类成员，只能通过显式调用的方式访问。
  - 其他方式访问被隐藏的成员，编译器会认为该成员不存在从而导致失败。

同名隐藏的验证

- 同名隐藏：派生类对象优先考虑派生类域的自身成员（变量和函数）

```
1  class Father
2  {
3  public:
4      int f_a;
5      int f_b;
6      void ff1()
7      {cout<<"father␣ff1"<<endl;}
8  };
9  class Childer:public Father
10 {
11 public:
12     int c_a;
13     int f_b;
14     void cf1()
15     {cout<<"childer␣cf1"<<endl;}
16     void ff1()
15     {cout<<"childer␣ff1"<<endl;}
16 };
17 int main()
18 {
19     Childer ch;
20     cout<<ch.c_a<<endl; // 派生
       类域中的成员
21     cout<<ch.f_b<<endl; // 优先
       访问派生类域
22     cout<<ch.Father::f_b<<endl;
       // 显式访问被隐藏的基类成
       员
23     ch.cf1();
24     ch.ff1();
25     ch.Father::ff1();
26 }
```

同名成员ch.f_b和 ch.Father::f_b同时存在与派生类域和基类域。此时，派生类成员将基类成员隐藏。若要访问基类成员只能显式调用该成员，即：使用作用域分辨符和基类名进行访问。

提纲

函数覆盖(OVERRIDE)

函数覆盖(override)

如果派生类创建一个返回类型与函数签名（函数名称与参数列表）都与基类虚函数相同的函数，并为该函数提供新的函数体，这就是函数覆盖（重写）。

```
1  class Mammal{
2  public:
3      virtual void move() const {cout << "Mammal moves one step\n";}
4         void move(int d) const {cout << "Mammal moves " <<d<<" steps\n";}
5  protected:
6      int age;     int weight;
7  };
8  class Dog : public Mammal{
9  public:
10     void move() const { cout << "Dog moves 5 steps\n"; } // hiding!
11 };
12 int main(){
13     Mammal bigAnimal;     Dog tinyPuppy;
14     bigAnimal.move();     bigAnimal.move(2);     tinyPuppy.move();
15     tinyPuppy.move(10); // You'll receive an error: no matching function!
16     return 0;
17 }
```

函数覆盖(OVERRIDE)

- 请判断如下派生类中的成员函数是否为覆盖（重写，override）。

```
1  struct B
2  {
3      virtual void f1(int) const;
4      virtual void f2();
5               void f3();
6  }
7  struct D : B
8  {
9               void f1(int) const; //
10              void f2(int);       //
11              void f3();          //
12              void f4();          //
13 }
```

函数覆盖(OVERRIDE)

- 请判断如下派生类中的成员函数是否为覆盖（重写，override）。

```
1  struct B
2  {
3      virtual void f1(int) const;
4      virtual void f2();
5               void f3();
6  }
7  struct D : B
8  {
9               void f1(int) const; //
10              void f2(int);       //
11              void f3();          //
12              void f4();          //
13 }
```

- 覆盖是指派生类中存在重新定义的函数，其函数名、参数列、返回值类型必须同基类中的相对应被覆盖的函数严格一致。
- 覆盖函数和被覆盖函数，分别位于基类和派生类中，并且只有函数体不同。
- 当派生类对象调用满足覆盖特征的函数时会自动调用派生类中的函数版本，而不是基类中的被覆盖函数版本。

函数覆盖(OVERRIDE)

- 请判断如下派生类中的成员函数是否为覆盖（重写，override）。

```
1  struct B
2  {
3      virtual void f1(int) const;
4      virtual void f2();
5               void f3();
6  }
7  struct D : B
8  {
9               void f1(int) const; //override
10              void f2(int);       //
11              void f3();          //
12              void f4();          //
13 }
```

- 覆盖是指派生类中存在重新定义的函数，其函数名、参数列、返回值类型必须同基类中的相对应被覆盖的函数严格一致。
- 覆盖函数和被覆盖函数，分别位于基类和派生类中，并且只有函数体不同。
- 当派生类对象调用满足覆盖特征的函数时会自动调用派生类中的函数版本，而不是基类中的被覆盖函数版本。

函数覆盖(OVERRIDE)

- 请判断如下派生类中的成员函数是否为覆盖（重写，override）。

```
1  struct B
2  {
3      virtual void f1(int) const;
4      virtual void f2();
5               void f3();
6  }
7  struct D : B
8  {
9               void f1(int) const; //override
10              void f2(int);       //not override, but hiding B::f2()
11              void f3();          //
12              void f4();          //
13 }
```

- 覆盖是指派生类中存在重新定义的函数，其函数名、参数列、返回值类型必须同基类中的相对应被覆盖的函数严格一致。
- 覆盖函数和被覆盖函数，分别位于基类和派生类中，并且只有函数体不同。
- 当派生类对象调用满足覆盖特征的函数时会自动调用派生类中的函数版本，而不是基类中的被覆盖函数版本。

函数覆盖(OVERRIDE)

- 请判断如下派生类中的成员函数是否为覆盖（重写，override）。

```
1  struct B
2  {
3      virtual void f1(int) const;
4      virtual void f2();
5               void f3();
6  }
7  struct D : B
8  {
9               void f1(int) const; //override
10              void f2(int);       //not override, but hiding B::f2()
11              void f3();          //not override, but hiding B::f3()
12              void f4();          //
13 }
```

- 覆盖是指派生类中存在重新定义的函数，其函数名、参数列、返回值类型必须同基类中的相对应被覆盖的函数严格一致。
- 覆盖函数和被覆盖函数，分别位于基类和派生类中，并且只有函数体不同。
- 当派生类对象调用满足覆盖特征的函数时会自动调用派生类中的函数版本，而不是基类中的被覆盖函数版本。

函数覆盖(OVERRIDE)

- 请判断如下派生类中的成员函数是否为覆盖（重写，override）。

```
1  struct B
2  {
3      virtual void f1(int) const;
4      virtual void f2();
5               void f3();
6  }
7  struct D : B
8  {
9               void f1(int) const; //override
10              void f2(int);       //not override, but hiding B::f2()
11              void f3();          //not override, but hiding B::f3()
12              void f4();          //not override, new-defined
13 }
```

- 覆盖是指派生类中存在重新定义的函数，其函数名、参数列、返回值类型必须同基类中的相对应被覆盖的函数严格一致。
- 覆盖函数和被覆盖函数，分别位于基类和派生类中，并且只有函数体不同。
- 当派生类对象调用满足覆盖特征的函数时会自动调用派生类中的函数版本，而不是基类中的被覆盖函数版本。

重载、隐藏与覆盖

函数重载（overload）是指在同一作用域中定义同名函数，特征是：

- 相同的范围，在同一个类中；
- 函数名字相同；
- 参数（类型、个数、顺序）不同；
- virtual关键字可有可无。

函数隐藏（hide）是指派生类的函数屏蔽了与其同名的基类函数，规则如下：

- 如果派生类的函数与基类的函数同名，但是具有不同参数。此时，不论有无virtual关键词，基类的函数将被隐藏（注意别与重载混淆）；
- 如果派生类的函数与基类的函数同名，并且具有相同参数，但是基类函数没有virtual关键词。此时，基类的函数被隐藏（注意别与覆盖混淆）。

函数覆盖（override）是指派生类的函数覆盖基类的函数，特征是：

- 不同的范围，分别位于派生类与基类；
- 函数名字相同；
- 参数列表（类型、个数、顺序）相同；
- 基类函数必须有关键词virtual。

重载、隐藏与覆盖

函数重载（overload）是指在同一作用域中定义同名函数，特征是：

- 相同的范围，在同一个类中；
- 函数名字相同；
- 参数（类型、个数、顺序）不同；
- virtual关键字可有可无。

函数隐藏（hide）是指派生类的函数屏蔽了与其同名的基类函数，规则如下：

- 如果派生类的函数与基类的函数同名，但是具有不同参数。此时，不论有无virtual关键词，基类的函数将被隐藏（注意别与重载混淆）；
- 如果派生类的函数与基类的函数同名，并且具有相同参数，但是基类函数没有virtual关键词。此时，基类的函数被隐藏（注意别与覆盖混淆）。

函数覆盖（override）是指派生类的函数覆盖基类的函数，特征是：

- 不同的范围，分别位于派生类与基类；
- 函数名字相同；
- 参数列表（类型、个数、顺序）相同；
- 基类函数必须有关键词virtual。

重载、隐藏与覆盖

函数重载（overload）是指在同一作用域中定义同名函数，特征是：

- 相同的范围，在同一个类中；
- 函数名字相同；
- 参数（类型、个数、顺序）不同；
- virtual关键字可有可无。

函数隐藏（hide）是指派生类的函数屏蔽了与其同名的基类函数，规则如下：

- 如果派生类的函数与基类的函数同名，但是具有不同参数。此时，不论有无virtual关键词，基类的函数将被隐藏（注意别与重载混淆）；
- 如果派生类的函数与基类的函数同名，并且具有相同参数，但是基类函数没有virtual关键词。此时，基类的函数被隐藏（注意别与覆盖混淆）。

函数覆盖（override）是指派生类的函数覆盖基类的函数，特征是：

- 不同的范围，分别位于派生类与基类；
- 函数名字相同；
- 参数列表（类型、个数、顺序）相同；
- 基类函数必须有关键词virtual。

重载、隐藏与覆盖

- 请判断基类成员函数与派生类成员函数的关系：重载、隐藏与覆盖？

```
1  class Base
2  {
3  public:
4      void f(int   x)       { cout << "Base::f(int) " << x << endl;   }
5      void f(float x)       { cout << "Base::f(float) " << x << endl; }
6      virtual void g(void) { cout << "Base::g(void)" << endl;         }
7  };
8  class Derived : public Base
9  {
10 public:
11     virtual void g(void) { cout << "Derived::g(void)" << endl;      }
12 };
13 void main(void)
14 {
15     Derived d;    Base *pb = &d;
16     pb->f(42);            // 调用:
17     pb->f(3.14f);         // 调用:
18     pb->g();              // 调用:
19 }
```

Base::f(int)与Base::f(float)是　　，而Base::g(void)与Derived::g(void) 是　　。

重载、隐藏与覆盖

- 请判断基类成员函数与派生类成员函数的关系：重载、隐藏与覆盖？

```
1  class Base
2  {
3  public:
4      void f(int   x)       { cout << "Base::f(int) " << x << endl;   }
5      void f(float x)       { cout << "Base::f(float) " << x << endl; }
6      virtual void g(void) { cout << "Base::g(void)" << endl;         }
7  };
8  class Derived : public Base
9  {
10 public:
11     virtual void g(void) { cout << "Derived::g(void)" << endl;      }
12 };
13 void main(void)
14 {
15     Derived d;    Base *pb = &d;
16     pb->f(42);           // 调用: Base::f(int) 42
17     pb->f(3.14f);        // 调用: Base::f(float) 3.14
18     pb->g();             // 调用: Derived::g(void)
19 }
```

Base::f(int)与Base::f(float)是　　，而Base::g(void)与Derived::g(void) 是　　。

重载、隐藏与覆盖

- 请判断基类成员函数与派生类成员函数的关系：重载、隐藏与覆盖？

```
1  class Base
2  {
3  public:
4      void f(int   x)       { cout << "Base::f(int) " << x << endl;   }
5      void f(float x)       { cout << "Base::f(float) " << x << endl; }
6      virtual void g(void) { cout << "Base::g(void)" << endl;         }
7  };
8  class Derived : public Base
9  {
10 public:
11     virtual void g(void) { cout << "Derived::g(void)" << endl;      }
12 };
13 void main(void)
14 {
15     Derived d;    Base *pb = &d;
16     pb->f(42);           // 调用: Base::f(int) 42
17     pb->f(3.14f);        // 调用: Base::f(float) 3.14
18     pb->g();             // 调用: Derived::g(void)
19 }
```

Base::f(int)与Base::f(float)是重载，而Base::g(void)与Derived::g(void) 是　　。

重载、隐藏与覆盖

- 请判断基类成员函数与派生类成员函数的关系：重载、隐藏与覆盖？

```
1  class Base
2  {
3  public:
4      void f(int   x)       { cout << "Base::f(int) " << x << endl;   }
5      void f(float x)       { cout << "Base::f(float) " << x << endl; }
6      virtual void g(void) { cout << "Base::g(void)" << endl;         }
7  };
8  class Derived : public Base
9  {
10 public:
11     virtual void g(void) { cout << "Derived::g(void)" << endl;      }
12 };
13 void main(void)
14 {
15     Derived d;    Base *pb = &d;
16     pb->f(42);           // 调用: Base::f(int) 42
17     pb->f(3.14f);        // 调用: Base::f(float) 3.14
18     pb->g();             // 调用: Derived::g(void)
19 }
```

Base::f(int)与Base::f(float)是重载，而Base::g(void)与Derived::g(void) 是覆盖。

重载、隐藏与覆盖

- 请判断基类成员函数与派生类成员函数的关系：重载、隐藏与覆盖？

```
1  #include <iostream>
2  class Base
3  {
4  public:
5      virtual void f(float x)
6      {   cout << "Base::f(float)␣" << x << endl;   }
7              void g(float x)
8      {   cout << "Base::g(float)␣" << x << endl;   }
9              void h(float x)
10     {   cout << "Base::h(float)␣" << x << endl;   }
11 };
12 class Derived : public Base
13 {
14 public:
15     virtual void f(float x)           //
16     {   cout << "Derived::f(float)␣" << x << endl; }
17             void g(int x)             //
18     {   cout << "Derived::g(int)␣"  << x << endl;  }
19             void h(float x)           //
20     {   cout << "Derived::h(float)␣" << x << endl; }
21 };
```

重载、隐藏与覆盖

- 请判断基类成员函数与派生类成员函数的关系：重载、隐藏与覆盖？

```
1  #include <iostream>
2  class Base
3  {
4  public:
5      virtual void f(float x)
6      {   cout << "Base::f(float)␣" << x << endl;   }
7              void g(float x)
8      {   cout << "Base::g(float)␣" << x << endl;   }
9              void h(float x)
10     {   cout << "Base::h(float)␣" << x << endl;   }
11 };
12 class Derived : public Base
13 {
14 public:
15     virtual void f(float x)           // 覆盖
16     {   cout << "Derived::f(float)␣" << x << endl; }
17             void g(int x)             //
18     {   cout << "Derived::g(int)␣"  << x << endl;  }
19             void h(float x)           //
20     {   cout << "Derived::h(float)␣" << x << endl; }
21 };
```

重载、隐藏与覆盖

- 请判断基类成员函数与派生类成员函数的关系：重载、隐藏与覆盖？

```
1  #include <iostream>
2  class Base
3  {
4  public:
5      virtual void f(float x)
6      {   cout << "Base::f(float)␣" << x << endl;   }
7              void g(float x)
8      {   cout << "Base::g(float)␣" << x << endl;   }
9              void h(float x)
10     {   cout << "Base::h(float)␣" << x << endl;   }
11 };
12 class Derived : public Base
13 {
14 public:
15     virtual void f(float x)            // 覆盖
16     {   cout << "Derived::f(float)␣" << x << endl; }
17             void g(int x)              // 隐藏，非重载
18     {   cout << "Derived::g(int)␣"  << x << endl;  }
19             void h(float x)            //
20     {   cout << "Derived::h(float)␣" << x << endl; }
21 };
```

重载、隐藏与覆盖

- 请判断基类成员函数与派生类成员函数的关系：重载、隐藏与覆盖？

```
1  #include <iostream>
2  class Base
3  {
4  public:
5      virtual void f(float x)
6      {   cout << "Base::f(float)␣" << x << endl;   }
7              void g(float x)
8      {   cout << "Base::g(float)␣" << x << endl;   }
9              void h(float x)
10     {   cout << "Base::h(float)␣" << x << endl;   }
11 };
12 class Derived : public Base
13 {
14 public:
15     virtual void f(float x)           // 覆盖
16     {   cout << "Derived::f(float)␣" << x << endl; }
17             void g(int x)             // 隐藏，非重载
18     {   cout << "Derived::g(int)␣"  << x << endl;  }
19             void h(float x)           // 隐藏，非覆盖
20     {   cout << "Derived::h(float)␣" << x << endl; }
21 };
```

重载、隐藏与覆盖

- 基于上述代码中的函数及分析，请判断程序输出结果。

```
22  void main(void)
23  {
24      Derived d;    Base *pb = &d;   Derived *pd = &d;
25
26  //
27      pb->f(3.14f); //
28      pd->f(3.14f); //
29
30  //
31      pb->g(3.14f); //
32      pd->g(3.14f); //
33
34  //
35      pb->h(3.14f); //
36      pd->h(3.14f); //
37  }
```

重载、隐藏与覆盖

- 基于上述代码中的函数及分析，请判断程序输出结果。

```
22  void main(void)
23  {
24      Derived d;    Base *pb = &d;   Derived *pd = &d;
25
26  //
27      pb->f(3.14f); // Derived::f(float) 3.14
28      pd->f(3.14f); // Derived::f(float) 3.14
29
30  //
31      pb->g(3.14f); // Base::g(float) 3.14
32      pd->g(3.14f); // Derived::g(int) 3 (surprise!)
33
34  //
35      pb->h(3.14f); // Base::h(float) 3.14 (surprise!)
36      pd->h(3.14f); // Derived::h(float) 3.14
37  }
```

重载、隐藏与覆盖

- 基于上述代码中的函数及分析，请判断程序输出结果。

```
22  void main(void)
23  {
24      Derived d;    Base *pb = &d;   Derived *pd = &d;
25
26  // Good :  behavior depends solely on type of the object!
27      pb->f(3.14f); // Derived::f(float) 3.14
28      pd->f(3.14f); // Derived::f(float) 3.14
29
30  // Bad :  behavior depends on type of the pointer!
31      pb->g(3.14f); // Base::g(float) 3.14
32      pd->g(3.14f); // Derived::g(int) 3 (surprise!)
33
34  // Bad :  behavior depends on type of the pointer!
35      pb->h(3.14f); // Base::h(float) 3.14 (surprise!)
36      pd->h(3.14f); // Derived::h(float) 3.14
37  }
```

当发生函数隐藏时，指针类型（而非所指对象）决定了所访问的成员函数。

重载、隐藏与覆盖

- 基于上述代码中的函数及分析，请判断程序输出结果。

```
22  void main(void)
23  {
24      Derived d;    Base *pb = &d;   Derived *pd = &d;
25
26  // Good :  behavior depends solely on type of the object!
27      pb->f(3.14f); // Derived::f(float) 3.14
28      pd->f(3.14f); // Derived::f(float) 3.14
29
30  // Bad :  behavior depends on type of the pointer!
31      pb->g(3.14f); // Base::g(float) 3.14
32      pd->g(3.14f); // Derived::g(int) 3 (surprise!)
33
34  // Bad :  behavior depends on type of the pointer!
35      pb->h(3.14f); // Base::h(float) 3.14 (surprise!)
36      pd->h(3.14f); // Derived::h(float) 3.14
37  }
```

当发生函数隐藏时，指针类型（而非所指对象）决定了所访问的成员函数。
函数隐藏“往往”神出鬼没、令人迷惑，在一些情况下要避免函数隐藏。

重载、隐藏与覆盖

- 基于上述代码中的函数及分析，请判断程序输出结果。

```
22  void main(void)
23  {
24      Derived d;    Base *pb = &d;    Derived *pd = &d;
25
26  // Good :  behavior depends solely on type of the object!
27      pb->f(3.14f); // Derived::f(float) 3.14
28      pd->f(3.14f); // Derived::f(float) 3.14
29
30  // Bad :  behavior depends on type of the pointer!
31      pb->g(3.14f); // Base::g(float) 3.14
32      pd->g(3.14f); // Derived::g(int) 3 (surprise!)
33
34  // Bad :  behavior depends on type of the pointer!
35      pb->h(3.14f); // Base::h(float) 3.14 (surprise!)
36      pd->h(3.14f); // Derived::h(float) 3.14
37  }
```

当发生函数隐藏时，指针类型（而非所指对象）决定了所访问的成员函数。函数隐藏“往往”神出鬼没、令人迷惑，在一些情况下要避免函数隐藏。

然而，隐藏规则可在程序编译阶段由编译器明确指出一些错误、消灭若干意外，这是同名隐藏存在的意义所在。

Acknowledgement and References

致谢

Thanks to Dr. Tao Hu 胡涛 and Prof. Dr. Xiaojun Liu 刘晓军 for their insightful discussions and helpful .ppt slides.

书目

《Visual C++程序设计》张岳新 编著

《C++程序设计教程 第3版》王珊珊 臧冽 张志航 编著

《高质量C/C++编程指南》林锐 韩永泉 编著

《C++ Primer 中文版 第5版》Stanley B. Lippman, Josee Lajoie, Barbara E. Moo著, 王刚 杨巨峰 译

《C++ Primer Plus 中文版 第6版》Stephen Prata著, 张海龙 袁国忠 译

《Professional C++ 3rd Edition》Marc Gregoire著

《Effective C++ 中文版》Scott Meyers著，侯捷 译

《More Effective C++ 中文版》Scott Meyers著，侯捷 译

网络

CSDN社区 http://www.csdn.net

联系我

孙东科

QQ: 12897898
WeChat: DongkeSun
E-Mail: dksun@seu.edu.cn

办公室：机械楼230室（机械楼北侧）

东南大学机械工程学院
江苏省微纳生物医疗器械设计与制造重点实验室