第22章 继承详解

本章将给出第6章中讲解的一些概念的形式化定义，并会详细分析它们。第6章中讲解了构造函数、析构函数和复制赋值运算符是如何产生并被派生类使用的，探讨了如何能将关键字public，private和protected用于基类和成员，还给出了多重继承的几个示例。

22.1 虚指针和虚表

每一个包含方法(虚函数)的类，都有一个虚跳表(virtual jump table)或者虚表(vtable)，它是作为“轻量级”C++执行环境的一部分而产生的。有多种方法可以实现虚表，其中最简单的实现(通常是最快和最轻量级的)包含一个指向该类全部方法的指针列表。根据优化策略的不同，虚表可以包含有助于调试的更多信息。编译器会将函数名称替换成方法调用的间接(相对于虚表列表)引用。

由此，可以显式地将多态类型定义成包含一个或者多个方法的类，从而要求使用虚表。多态类型的每一个实例都具有一个typeid属性，可以相当自然地将它实现成类的虚表地址。

使用虚表而不是switch语句

为了通过虚表实现间接的方法调用，编译器会为每一个多态类产生一个跳表，它与switch语句相似。程序员经常能够利用虚表而不必编写switch语句或者大型的复合条件语句。这样做隐含了大量的设计模式，比如命令模式、访问者模式、解释器模式以及策略模式。

只有当类中全部重写的方法被完全定义了并且能够被链接器找到时，才会对这个类建立虚表。

当执行完对象的构造函数后，就会设置它的typeid。如果存在基类，则在初始化每一个基类之后，会多次为对象设置typeid。

示例22.1中定义的类表明，从构造函数或者析构函数调用虚函数可能导致意外的后果。

示例22.1 src/derivation/typeid/vtable.h

```
[ . . . . ]
class Base {
 protected:
    int m_X, m_Y;
 public:
    Base();
    virtual ~Base();
    virtual void virtualFun() const;
};

class Derived : public Base {
```

```
    int m_Z;
 public:
    Derived();
    ~Derived();
    void virtualFun() const ;
};
[ . . . . ]
```

示例 22.2 表明了当从基类构造函数或析构函数调用虚函数时会发生什么。

示例 22.2 src/derivation/typeid/vtable.cpp

```
#include <QDebug>
#include <QString>
#include "vtable.h"

Base::Base() {
    m_X = 4;
    m_Y = 12;
    qDebug() << " Base::Base: " ;
    virtualFun();
}

Derived::Derived() {
    m_X = 5;
    m_Y = 13;
    m_Z = 22;
}

void Base::virtualFun() const {
    QString val=QString("[%1,%2]").arg(m_X).arg(m_Y);
    qDebug() << " VF: the opposite of Acid: " << val;
}

void Derived::virtualFun() const {
    QString val=QString("[%1,%2,%3]")
        .arg(m_X).arg(m_Y).arg(m_Z);
    qDebug() << " VF: add some treble: " ;
}

Base::~Base() {
    qDebug() << " ~Base() " ;
    virtualFun();
}

Derived::~Derived() {
    qDebug() << " ~Derived() " ;
}

int main() {
    Base *b = new Derived;                    1
    b->virtualFun();                          2
    delete b;                                 3
}
```

1 调用 Base::virtualFun( )。

2 使用虚表和运行时绑定调用 Derived::virtualFun( )。

3 调用 Base::virtualFun( )。

从随后的输出中可以看出，派生的 virtualFun()没有从 Base::Base()调用，因为基类初始化器位于还没有派生实例的一个对象中。

```
Base::Base:
VF: the opposite of Acid:  "[4,12]"
VF: add some treble:
~Derived()
~Base()
VF: the opposite of Acid:  "[5,13]"
```

不推荐从析构函数调用虚方法。从前一个输出中可以看出，基函数 virtualFun 总是从基类构造函数或者析构函数调用的。动态绑定不会在构造函数或者析构函数内发生。正如 Meyers 所说，“构造函数或者析构函数中不存在虚方法”。

22.2 多态与虚析构函数

当在继承层次中操作这些类时，经常需要维护包含有派生对象地址的基类指针的容器。示例 22.3 中定义的 Bank 类包含一个具有各种 Account 类型的容器。

示例 22.3 src/derivation/assigcopy/bank.h

```
[ . . . . ]
class Account;

class Bank {
 public:
    Bank& operator<< (Account* acct);                    1
    ~Bank();
    QString getAcctListing() const;
 private:
    QList<Account*> m_Accounts;
};
[ . . . . ]
```

1 这就是将对象指针添加到 m_Accounts 中的方法。

示例 22.4 中给出了这些 Account 类的定义。

示例 22.4 src/derivation/assigcopy/account.h

```
[ . . . . ]
class Account {
 public:
    Account(unsigned acctNum, double balance, QString owner);
    virtual ~Account(){
      qDebug() << "Closing Acct - sending e-mail "
               << "to primary acctholder:" << m_Owner; }
    virtual QString getName() const {return m_Owner;}
    // other virtual functions
```

```
 private:
    unsigned  m_AcctNum;
    double    m_Balance;
    QString    m_Owner;
};
class JointAccount : public Account {
 public:
  JointAccount (unsigned acctNum, double balance,
                QString owner, QString jowner);
  JointAccount(const Account & acct, QString jowner);
  ~JointAccount() {
     qDebug() << "Closing Joint Acct - sending e-mail "
              << "to joint acctholder:" << m_JointOwner; }
  QString getName() const {
    return QString("%1 and %2").arg(Account::getName())
                   .arg(m_JointOwner);
  }
  // other overrides
 private:
  QString m_JointOwner;
};
[ . . . . ]
```

通过对每一个所包含的 Account 调用 virtual 方法，Bank 类能够执行统一的操作。示例 22.5 中，delete acct 操作会导致对 Account 析构函数的间接调用，并随后会释放所分配的内存。

示例 22.5 src/derivation/assigcopy/bank.cpp

```
[ . . . . ]

#include <QDebug>
#include "bank.h"
#include "account.h"

Bank::~Bank() {
    qDeleteAll(m_Accounts);
    m_Accounts.clear();
}


Bank& Bank::operator<< (Account* acct) {
   m_Accounts << acct;
   return *this;
}

QString Bank::getAcctListing() const {
   QString listing("\n");
   foreach(Account* acct, m_Accounts)
      listing += QString("%1\n").arg(acct->getName());          1
   return listing;
}
```

1 getName()为虚函数。

尽管列表中的每一个地址都为 Account 对象，但其中的一些(甚至全部)有可能指向派生类对象，从而要求派生类析构函数调用。

如果析构函数为虚函数，则编译器会通过 Account 指针对任何析构函数启用运行时绑定，而不是简单地调用 Account::~Account()。如果没有在基类中将~Account()声明成虚函数，则运行示例 22.6 时将得到不正确的结果[①]。

示例 22.6 src/derivation/assigcopy/bank.cpp

```
[ . . . . ]

int main() {
  QString listing;
   {                                                          1
      Bank bnk;
      Account* a1 = new Account(1, 423, "Gene Kelly");
      JointAccount* a2 = new JointAccount(2, 1541, "Fred Astaire",
         "Ginger Rodgers");
      JointAccount* a3 = new JointAccount(*a1, "Leslie Caron");
      bnk << a1;
      bnk << a2;
      bnk << a3;
      JointAccount* a4 = new JointAccount(*a3);               2
      bnk << a4;
      listing = bnk.getAcctListing();
    }                                                         3
    qDebug() << listing;
    qDebug() << "Now exit program" ;
}
```

1 内部语句块的开始。

2 这条语句的作用是什么?

3 在这里，作为销毁 bank 类对象的一部分，全部四个 Account 对象都被销毁了。

以下输出是在~Account 中删除 virtual 后的结果：

```
Closing Acct - sending e-mail to primary acctholder:Gene Kelly
Closing Acct - sending e-mail to primary acctholder:Fred Astaire
Closing Acct - sending e-mail to primary acctholder:Gene Kelly
Closing Acct - sending e-mail to primary acctholder:Gene Kelly
[ ... ]
```

如果析构函数为虚函数，则这时示例中 Account 的两种类型都会被正确地销毁，且当销毁了 Bank 类对象时，联名账户中的两个账户持有人都能够得到合适的 E-mail 通知。

```
Closing Acct - sending e-mail to primary acctholder:Gene Kelly
Closing Joint Acct - sending e-mail to joint acctholder:Ginger Rodgers
```

注意

如果在一个类中声明了一个或者多个虚方法，则应该为这个类定义一个虚析构函数，即使它的函数体为空也应定义。

① 编译器会给出警告，指出析构函数中缺少 virtual，其行为是未定义的，所以在不同的系统上可能会得到不同的结果。

22.3 多重继承

多重继承是继承的一种形式，其中的类会继承多个基类的结构和行为。多重继承的常见用途如下。

- 用于组合存在一定重叠的不同类的功能，如图 22.1 所示。
- 用于以各种不同的方式实现共同的“纯接口”(只有纯虚函数的类)。

与单一继承一样，多重继承在类之间也定义了一种静态的关系。在运行时不能改变这种关系。

与单一继承层次相比，多重继承层次要复杂得多，更难以设计、实现和理解。可以用它来解决一些设计难题，但如果存在更简单的方法(如聚合)，则不应使用多重继承。

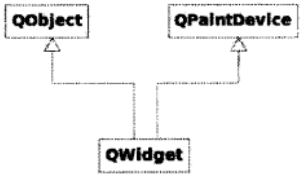

图 22.1 QWidget 的继承

22.3.1 多重继承语法

这一节中的例子给出了多重继承的语法及用法。

图 22.2 中给出了两个基类 Rectangle 和 ScreenRegion，每一个都在屏幕上扮演特定的角色。其中一个类关注的是形状和位置，而另一个关注的是颜色和可见性特征。Window 类必须同时为 Rectangle 类和 ScreenRegion 类，它们的定义参见示例 22.7。

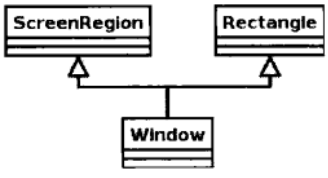

图 22.2 Window 类与 ScreenRegion 类和 Rectangle 类的关系

示例 22.7 src/multinheritance/window.h

```
[ . . . . ]

class Rectangle {
 public:
    Rectangle( Const Point& ul, int length, int width);
    Rectangle( const Rectangle& r) ;
    void move (const Point &newpoint);
 private:
    Point m_UpperLeft;
    int m_Length, m_Width;
};

class ScreenRegion {
 public:
    ScreenRegion( Color c=White);
    ScreenRegion (const ScreenRegion& sr);
    virtual color Fill( Color newColor) ;
    void show();
    void hide();
```

```
 private:
    Color m_Color;
    // other members...

};

class Window: public Rectangle, public ScreenRegion {
 public:
    Window( const Point& ul, int len, int wid, Color c)
        : Rectangle(ul, len, wid), ScreenRegion(c) {}              1

    Window( const Rectangle& rect, const ScreenRegion& sr)
        : Rectangle(rect), ScreenRegion(sr) {}                     2

    // Other useful member functions ...
};
```

1 使用基类 ctors。

2 使用基类复制 ctors。

Window 类的类首部中有一些语法项值得注意：

- 如果派生类不是 private 类型的，则访问指示符(public 或者 protected)必须出现在每一个基类名称的前面。
  - 默认的派生类是 private 类型的。
  - 派生类中可以混用 public，protected 和 private 类型。
- 多个基类用逗号分开。
- 初始化基类的顺序，应与类首部中基类出现的顺序相同。

示例 22.8 中提供了 Window 类的一些客户代码。

示例 22.8　src/multinheritance/window.cpp

```
#include "window.h"

int main() {
    Window w(Point(15,99), 50, 100, Color(22));
    w.show();                                                      1
    w.move (Point(4,6));                                           2
    return 0;
 }
```

1 调用 ScreenRegion::show()。

2 调用 Rectangle::move()。

成员初始化的顺序

对成员的默认初始化或者赋值的顺序，与类定义中声明数据成员的顺序相同：首先是基类，然后是派生类成员。

22.3.2　多重继承与 QObject

Qt 中的许多类都使用多重继承。图 22.3 中给出了 QGraphicsView 中使用的类之间的继承关系。QGraphicsItem 是一个轻量级对象，它不支持信号或者槽。QGraphicsObject

依次继承自 QObject 和 QGraphicsItem，因而具有这两种类的好处：QGraphicsObject 支持信号和槽，通常用在动画中。图 22.1 中所示的 QWidget 是使用了 QObject 多重继承的另一个类。

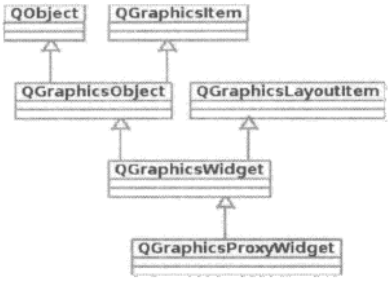

图 22.3 QGraphicsObject 与多重继承

图 22.3 中，QObject 是被多重继承了的基类之一。Qt 中的一个限制是：QObject 只能被任何类继承一次(不支持虚继承)。此外，在基类列表中必须首先列出派生自 QObject 的基类。如果破坏这个原则，则元对象编译器(moc)可能产生奇怪的代码错误信息。

22.3.3 解决多重继承的冲突

图 22.4 给出的 UML 框图中，接口和实现中都错误地使用了多重继承。为了使事情更复杂一些，其中的一个类从同一个基类继承了两次。

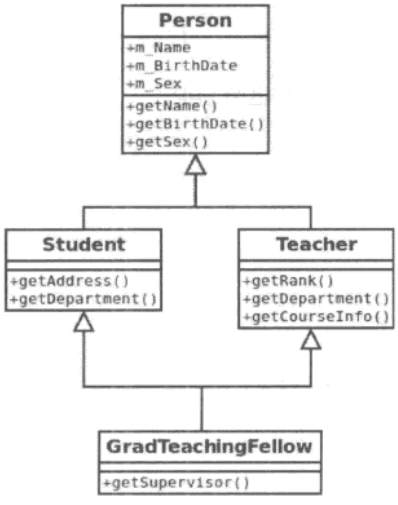

图 22.4 Person-Student-Teacher 继承层次

这里的 GradTeachingFellow 类派生自两个类：Student 和 Teacher。

```
class GradTeachingFellow : public Student,
                           public Teacher {
    // class member functions and data members
```

如果错误地使用了多重继承，就会出现命名冲突和设计问题。这个例子中，Student 类和 Teacher 类都具有 getDepartment() 函数。这样，学生既可以在某个系学习，又可以在另一个系教书。

问题

当对 GraduateTeachingFellow 调用 getDepartment() 函数时，会出现什么问题？

```
GraduateTeachingFellow gtf;
Person* pptr = &gtf;
Student * sptr = &gtf;;
Teacher* tptr = &gtf;
gtf.Teacher::getDepartment();
gtf.Student::getDepartment();
sptr->getDepartment()
tptr->getDepartment()
pptr->getDepartment();   // Ambiguous: runtime error if virtual
gtf.getDepartment();  // Compiler error: ambiguous function call
```

当然，问题在于：没有在 GradTeachingFellow 类中提供 getDepartment() 函数。当编译器查找 getDepartment() 函数时，Student 类和 Teacher 类具有同样的优先级。

应该避免类似这样的继承冲突，因为它会在以后招来许多设计上的麻烦。不过，如果能够通过作用域解析得到解决的话，也可以这样做。

22.3.3.1　虚继承

图 22.4 中，GraduateTeachingFellow 从同一个基类中继承了多次。这种模型会存在另一个问题：冗余。这种多次继承类的实例会如图 22.5 那样。

Person 类的属性应当只被继承一次。对 GradTeachingFellow 而言，具有两个生日和两个名称没有什么意义。虚继承(virtual inheritance)能够消除这种冗余。

当将多重继承用在有歧义的环境下时，会出现奇怪的问题，尤其是将虚继承/函数与非虚继承/函数混用，增加了复杂性时，这样就似乎提示了 Java 的设计者不要在 Java 中使用多重继承。实际上，Java 允许程序员定义只由抽象函数(纯虚函数)组成的接口。这样，Java 类就能够利用 implements 子句根据需要实现任意多的接口。

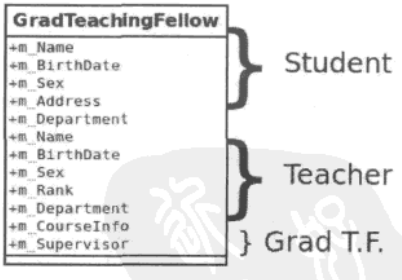

图 22.5　GradTeachingFellow，没有虚继承

22.3.3.2　虚基类

可以将基类声明成虚基类。虚基类将它的表示与具有同一个虚基类的其他类共享。

在 Student 类和 Teacher 类的类首部中增加关键字 virtual，而让类定义的其他细节保持不变，就得到示例 22.9。

示例 22.9 src/multinheritance/people.h

```
#include "qdatetime.h"

class Person {
public:
    Person(QString name, QDate birthdate)
    QObject(name.ascii()),
    m_Birthdate(birthdate) {}

    Person(const Person& p) : QObject(p),
    m_Birthdate(p.m_Birthdate) {}

private:
    QDate m_Birthdate;
};

class Student : virtual public Person {                    1
    // other class members
};

class Teacher : virtual public Person {                    2
    // other class members
}
class GraduateTeachingFellow :
    public Student, public Teacher {                       3
public:
    GraduateTeachingFellow(const Person& p,
                           const Student& s, const Teacher& t):
    Person(p), Students(s), Teacher(t) {}                  4
}
```

1 注意这里的关键字 virtual。

2 虚继承。

3 这里并不需要关键字 virtual。

4 在多重派生的类中，有必要显式地初始化全部的虚基类，以消除应如何将它们初始化的歧义。

使用了虚继承之后，GradTeachingFellow 的实例如图 22.6 所示。

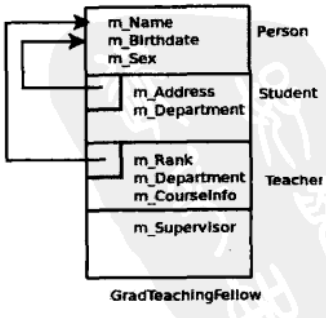

图 22.6 GradTeachingFellow，有虚继承

从另一个类虚继承的每一个实例，都具有一个指向它的虚基类子对象的指针(或者有可变的偏移量)。对程序员而言，虚基类指针是不可见的，而且通常情况下不需要改变它。

利用多重继承，每一个虚基类指针都指向同一个对象，从而有效地使基类对象能被所有的派生类部分共享。

对于具有虚基类的任何类，对于这个虚基类的成员初始化项必须出现在这个类的成员初始化表中。否则，虚基类将会被默认初始化。

22.4 public, protected 和 private 派生

多数情况下，我们看到的子句都采用 public 派生，例如：

```
class Square : public Shape {
// ... };
```

这种派生关系描述了两个类之间的接口关系。这意味着基类的接口(public 部分)会与派生类的接口合并。如果能够将派生类对象称为“是”基类对象时(即二者存在“是”关系)，就适合使用 public 派生。例如，正方形“是”一种(具有更多属性的)形状。

较少看到的是 protected 派生或者 private 派生，它们被认为是“实现”关系而不是“是”关系。基类接口(public 部分)会与派生类的实现(根据派生类型的不同，可以是 private 类型或者 protected 类型)合并。

在效果上，private 派生就如同向派生类添加了一个额外的对象作为其 private 数据成员。

类似地，protected 派生就如同向派生类添加了一个额外的对象作为其 protected 数据成员，但这个对象会共享 this 指针。

示例 22.10 就是这种情形的一个具体例子，其中的 private 派生就是合适的。模板类 Stack 是从 QList 类 private 派生的。这样做的理由是：根据定义，栈是一种被限制成只能访问栈顶项的数据结构。从 QVector 类 public 派生的 QStack 类，为栈提供了一个所期望的 public 接口，而且还允许客户代码无限制地访问栈中的项，因为 QStack 类包含 QVector 类的全部 public 接口。Stack 类是从 QList 类 private 派生的，所以它的 public 接口限制客户代码对栈的访问，以与这种数据结构的定义相一致。

示例 22.10 src/privatederiv/stack.h

```
#ifndef _STACK_H_
#define _STACK_H_

#include <QList>

template<class T>
class Stack : private QList<T> {

public:
    bool isEmpty() const {
        return QList<T>::isEmpty();
    }
    T pop() {
        return takeFirst();
    }
    void push(const T& value) {
        prepend(value);
    }
    const T& top() const {
        return first();
    }
    int size() const {
```

```
        return QList<T>::size();
    }
    void clear() {
        QList<T>::clear();
    }
};
#endif
```

示例 22.11 表明，客户代码企图使用 Stack 的基类(QList)接口是不允许的。

示例 22.11 src/privatederiv/stack-test.cpp

```
#include "stack.h"
#include <QString>
#include <qstd.h>
using namespace qstd;

int main() {
    Stack<QString> strs;
    strs.push("hic");
    strs.push("haec");
    strs.push("hoc");
//  strs.removeAt(2);                                    1
    int n = strs.size();
    cout << n << " items in stack" << endl;
    for (int i = 0; i < n; ++i)
        cout << strs.pop() << endl;
}
```

1 错误：继承的 QList 方法是 private 类型的。

因此，当基类只用于实现目的时，private 派生提供了隐藏基类 public 接口的一种途径。对于 protected 派生又如何呢?

假设希望派生 XStack 类，它是来自于这个 Stack 类的一种特殊的栈。对于从 QList 类 private 派生的 Stack 类，不能在 XStack 的实现中使用任何 QList 的成员函数。如果在实现 XStack 时需要使用某些 QList 函数，则从 QList 派生 Stack 时，必须使用 protected 派生。Protected 派生使 QList 的 public 接口在 Stack 中成为 protected 接口。在内部，这使得派生自 Stack 的类能够使用继承的 QList protected 接口。

22.5 复习题

1. 什么是虚表?
2. 什么是多态类型?
3. 哪些类型的成员函数不会被继承? 为什么?
4. 在什么情况下应使用虚析构函数?
5. 当从基类构造函数调用虚函数时会发生什么?
6. 什么是虚继承? 它能用来解决什么问题?
7. 为什么要使用非 public 派生?