第 3 章

继承与派生

继承与派生是面向对象程序设计的第二大特征，它允许在已有类（称为基类或父类）的基础上，根据自己的需要向类中添加更多的属性和方法，从而创建出一个新的类（称为派生类或子类）。继承与派生实际是从不同的角度来看的同一个过程：保持已有类的特性而构成新类的过程称为类的继承；在已有类的基础上新增自己的特性而产生新类的过程称为类的派生。派生类除了具有新增的属性和方法外，还自动具有基类的所有属性和方法。对于派生类，还可以继续派生出新类，因此基类和派生类是相对而言的。类层层派生，可形成复杂的继承结构。

继承机制的引入使程序设计时可以层层抽取出对象之间的共同点，从而减少了代码的冗余；避免了不必要的重复编程，增加了代码的可重用性。派生出的新类可以基于已有工作进一步扩展，以快速开发出高质量的程序。

3.1 类的继承与派生

3.1.1 派生类的定义

视频讲解

派生类只有一个直接基类的继承，称为单继承，否则称为多继承。在图 3-1 中，汽车和船都是单继承，它们都只有一个基类——交通工具；而房车有两个基类——房子和汽车；水陆两栖车也有两个基类——汽车和船，它们都是多继承。

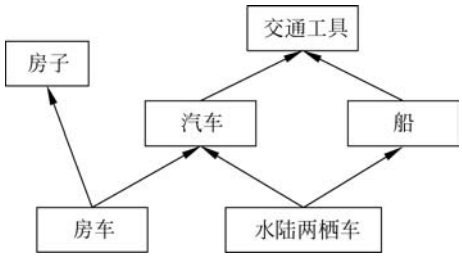


图 3-1　类的继承关系

派生类的定义格式如下。

```
class 派生类名:继承方式 基类 1,…,继承方式 基类 n
{       //基类可以有多个,之间用逗号隔开,每个基类都要写明继承方式
        派生类新增成员的声明
};
```

基类需要是已有的类,派生类是新定义的类,派生类的基类在派生类名后以“:”开头的基类列表中指明。当单继承时,此处只有一个基类;当多继承时,有多个基类,之间通过逗号隔开。每个基类都需要指明继承方式,且每个继承方式都只限制对紧随其后的基类的继承。继承有 public、private、protected 3 种方式。

(1) public:表示公有继承。

(2) private:表示私有继承。

(3) protected:表示保护继承。

下面举例说明。为了定义派生类,首先要有交通工具类 Conveyance 作为基类,其定义和实现如下。

```
class Conveyance                    //交通工具类
{
    double speed;                   //时速
public:
    double getSpeed()
    {
        return speed;
    }
};
```

然后在该类的基础上派生出汽车类 Car,其定义和实现如下。

```
class Car: public Conveyance        //汽车类
{
    int wheelsNum;                  //车轮数
public:
    int getWheelsNum()
    {
        return wheelsNum;
    }
};
```

在定义派生类时,只需要根据派生类所特有的特点添加新的属性和方法即可。从类的定义中可以看到,在派生类 Car 中并没有声明基类中的属性和方法,但此时派生类中已存在两个属性 wheelsNum 和 speed,以及两个方法 getSpeed()和 getWheelsNum()。原因是已指明 Conveyance 为基类,则派生类 Car 就自动拥有了基类的所有属性和方法。这种继承是整体的,不能只选择一部分成员而舍弃另一部分。

为了实现多继承,再定义和实现房子类 House 如下。

```
class House
```

```
{
    double area;                    //房屋的面积
public:
    double getArea()
    {
    return area;
    }
};
```

房车类 MotorHome 的定义和实现如下。

```
class MotorHome:public Car,public House
{
    int waterReserve;               //储水量,单位为升
public:
   int getWaterReserve()
     {
         return waterReserve;
     }
};
```

可见多继承只需要在基类列表中依次说明各个基类即可。在该例中,各个类的成员列表及继承关系如图 3-2 所示。

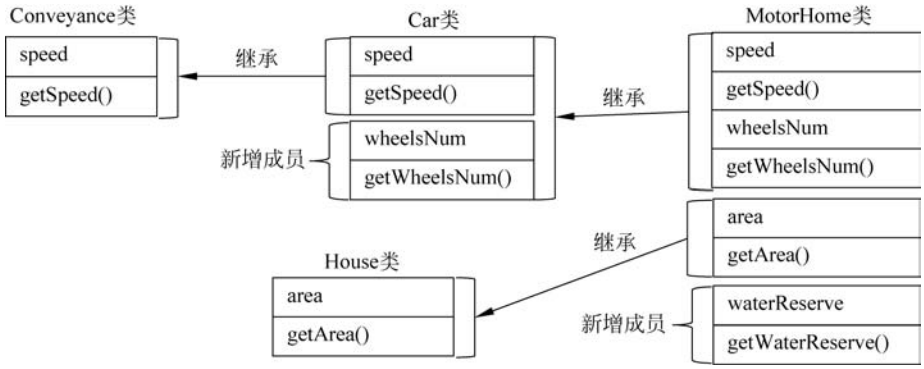


图 3-2　派生类的成员


视频讲解

3.1.2　继承方式

派生类虽然继承了基类中的全部成员,但是这些成员的访问权限可能会根据继承方式的不同而产生变化,其影响主要体现在派生类成员和派生类对象能否访问它们。

1. 公有继承

公有继承(public)具有以下特点。

(1) 基类中的 private 成员不可以访问(对派生类成员和派生类对象而言)。

(2) 基类中的 public 和 protected 成员的访问权限在派生类中保持不变。

(3) 派生类的成员函数可直接访问基类中的 public 和 protected 成员,但不能访问基类中的 private 成员。

(4) 派生类的对象只能访问基类中的 public 成员。

2. 保护继承

保护继承(protected)具有以下特点。

(1) 基类中的 private 成员不可以访问。

(2) 基类中的 public 和 protected 成员的访问权限在派生类中转为 protected。

(3) 派生类的成员函数可直接访问基类中的 public 和 protected 成员(现为派生类中的 protected 成员),但不能访问基类中的 private 成员。

(4) 派生类的对象不能访问基类中的任何成员。

3. 私有继承

私有继承(private)具有以下特点。

(1) 基类中的 private 成员不可以访问。

(2) 基类中的 public 和 protected 成员的访问权限在派生类中转为 private。

(3) 派生类的成员函数可直接访问基类中的 public 和 protected 成员(现为派生类中的 private 成员),但不能访问基类中的 private 成员。

(4) 派生类的对象不能访问基类中的任何成员。

表 3-1 对上述规则进行了总结。在实际应用中,公有继承是主要的继承方式,在设计类时多采用此方式。

表 3-1 继承方式和访问特性

<table>
<tr><th rowspan="2">基类成员</th><th rowspan="2">基类成员函数</th><th rowspan="2">基类对象</th><th colspan="2">private 继承方式</th><th colspan="2">protected 继承方式</th><th colspan="2">public 继承方式</th></tr>
<tr><th>派生类新增成员函数</th><th>派生类对象</th><th>派生类新增成员函数</th><th>派生类对象</th><th>派生类新增成员函数</th><th>派生类对象</th></tr>
<tr><td>基类 private 成员</td><td rowspan="3">可访问</td><td colspan="6">不可访问</td></tr>
<tr><td>基类 protected 成员</td><td>不可访问</td><td rowspan="2">可访问,访问权限转为 private</td><td rowspan="2">不可访问</td><td rowspan="2">可访问,访问权限仍为或转为 protected</td><td rowspan="2">不可访问</td><td>可访问,访问权限仍为 protected</td><td>不可访问</td></tr>
<tr><td>基类 public 成员</td><td>可访问</td><td>可访问,访问权限仍为 public</td><td>可访问</td></tr>
</table>

3 种继承方式的共同点如下。

(1) 不论以何种方式继承,派生类中新增的成员函数都可以访问基类中的 public 成员和 protected 成员。

(2) 不论以何种方式继承,基类的私有成员对派生类和派生对象都不可见,只能通过基类提供的公有接口间接操作。

3 种继承方式的不同点如下。

(1) 在不同继承方式下,基类中的 public 成员和 protected 成员在派生类中转为的访问权限不同。

(2) 派生类对象除了在 public 继承方式下可以访问基类中的 public 成员外,其他方式

均不可以访问基类中的任何成员。

从表 3-1 中可以看到，私有继承方式和保护继承方式的区别只在于：在私有继承下，原基类受保护成员和公有成员的访问权限均转为私有的；在保护继承下，原基类受保护成员和公有成员的访问权限均转为受保护的。从使用者的角度来说，派生类新增成员函数在两种继承方式下都可以访问原为基类的 protected 成员和 public 成员；派生类对象在两种继承方式下都不可以访问基类中的所有成员。似乎两种继承方式在用法上没有什么不同，那么区分这两种继承方式以及在类内部区分这两种访问权限的意义是什么呢？

在图 3-3 中，上、下两部分分别表示不同的继承关系，经观察可以发现，在父亲类对祖先类中成员的使用上确实没有什么区别。它们的区别主要体现在继续往下以公有继承方式派生出的新类上：在保护继承时，孩子类中的新增成员函数仍可访问到祖先类中的 protected 成员和 public 成员；而在私有继承时，祖先类中的所有成员均不可见。

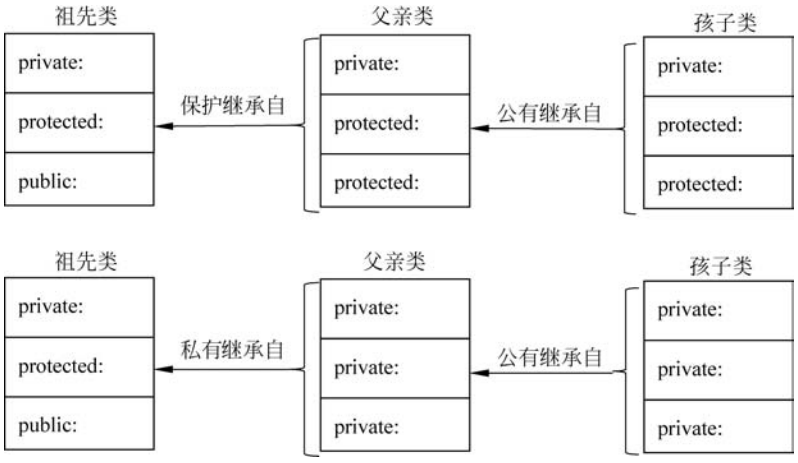


图 3-3　保护继承和私有继承的区别

提示：protected 成员的优点是既实现了数据隐藏又很好地实现了继承，而 private 成员只是很好地实现了数据隐藏。


视频讲解

3.1.3　重定义成员函数

派生类对象和基类对象可能会执行功能类似的同名动作，但对于不同的类，同名动作在具体实现细节上有所不同。例如，希望在交通工具基类 Conveyance 和汽车派生类 Car 中都设计一个输出内部所有属性的值的成员函数，这里有几个问题。

(1) 由于不同的类所包含的属性数量不同，所以若要输出不同类对象的全部属性，具体输出语句的代码一定不同，只设计一个成员函数不能解决问题。

(2) 因为上述原因，似乎应该给每个类都设置一个成员函数分别实现不同的输出。函数虽然可以重载，但不能同时声明多个具有同样函数原型(同函数名、同参数列表)的函数。因为派生类会继承基类中的所有成员，所以如果把基类和派生类中实现该功能的成员函数名修改成彼此不一样，记忆起来会非常麻烦。特别是当类层层继承时，对每个类都要记得它用于输出全部属性的成员函数名叫什么，因此这种处理方式是不可取的。

在派生类中，实际上是允许声明一个和基类中的成员函数原型完全相同的新成员函数的，新的同名成员函数有自己的新的函数实现，这种情况就是在派生类中对基类中成员函数的重新定义，称为重定义成员函数或重定义继承的函数。

例如，在3.1.1节例子的基础上为交通工具基类Conveyance和汽车派生类Car都添加一个成员函数showInfo()，在Conveyance类定义的public部分添加声明如下。

```
void showInfo();
```

在类外实现如下。

```
void Conveyance::showInfo()
{
    cout <<"Speed per hour:"<< speed << endl;
}
```

在Car类定义的public部分添加声明如下。

```
void showInfo();
```

在类外实现如下。

```
void Car::showInfo()
{
    cout <<"Speed per hour:"<< getSpeed()<< endl;
    cout <<"Number of wheels:"<< wheelsNum << endl;
}
```

由于Car新增的成员函数不能访问基类中的私有数据成员，所以在Car的showInfo()函数中调用了基类的getSpeed公有接口间接得到speed的值。

现在的问题是由于派生类对基类的成员是全部继承的，这就意味着在Car类中实际有两个一模一样的showInfo()成员函数，那么对于派生类的对象，在调用showInfo()函数时用的是哪个呢？

提示：

(1) 重载函数：指函数名相同，形参必须不同(不同类型或个数)的函数。

(2) 重定义函数：指派生类中新成员函数的原型和基类中成员函数的原型完全一致，只是实现体不同。

(3) 若派生类中新成员函数的形参类型或数量不同于基类中的同名函数，那么只是重载，而非重定义。

继承机制对此问题的处理原则是派生类重定义的成员函数屏蔽了基类的同原型成员函数。例如：

```
Car myCar;
myCar.showInfo();
```

调用的是Car类中重新定义的showInfo()函数。

那么是否还能通过派生类对象访问到基类中被屏蔽掉的同原型成员函数呢？答案是肯定的。但由于屏蔽，若想使用基类中的同原型成员函数，需要使用作用域运算符(::)指定基

类的名称。例如：

```
myCar.Conveyance::showInfo();
```

调用的是基类中的 showInfo()成员函数。

观察两个类中 showInfo()函数的实现，会发现在 Car 类中 showInfo()输出的前一部分实际就是 Conveyance 类中 showInfo()输出的内容，因此 Car 类中的 showInfo()函数可以修改为如下的形式。

```
void Car::showInfo()
{
    Conveyance::showInfo();
    cout <<"Number of wheels:"<< wheelsNum << endl;
}
```

如此不仅实现了代码的复用，而且在不同的派生类中也能保持对基类部分相关属性的输出格式的一致。

提示：要重定义一个成员函数，必须在派生类中给出它的声明，即使这个声明和基类中的声明完全相同。


视频讲解

3.1.4 赋值兼容规则

在公有继承时，派生类完整地继承了基类的所有属性和行为，基类成员相当于公有派生类成员的一个子集。基类对象具有的属性派生类对象都有，基类对象能做的事情，派生类对象都能做。因此，基类与派生类对象之间存在着赋值兼容关系，即派生类对象可以赋值给基类对象。此时派生类对象中原属于基类属性的那部分内存空间的值会被原样复制到基类对象的内存空间中，而派生类中新增的那部分属性的值在赋值过程中被舍弃，如图 3-4 所示；反之，由于基类对象中缺少派生类新增属性的值，所以基类对象不能赋值给派生类对象。也就是说，派生类对象对基类对象的赋值关系是单向的、不可逆的。

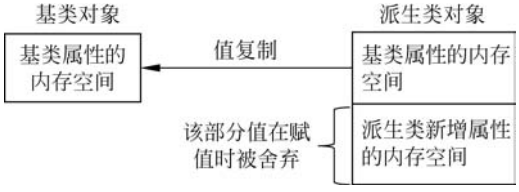


图 3-4　派生类对象赋值给基类对象时值的复制

在用到基类对象时，可以用其公有派生类对象代替，使用方式有以下几种。

(1) 可使用公有派生类对象赋值或初始化基类对象的引用，此时基类对象的引用名只是派生类对象中基类部分的别名。

(2) 若函数的参数是基类对象或基类对象的引用，对应的实参可以使用其派生类对象，但在函数中只能使用对象中的基类部分。

(3) 派生类对象的地址可以赋给指向基类对象的指针，但通过该指针只能访问到基类的部分。

例如定义了函数：

```
void func(Conveyance a)
{
    a.showInfo();
}
```

在已有 Car 类型对象 myCar 的情况下：

```
Conveyance x1,&x2 = myCar, * ptr;
ptr = &myCar;
x1 = myCar;
func(myCar);
```

上述语句都是合法的。引用 x2 只代表了 myCar 对象中属于 Conveyance 类的那部分属性的内存空间；指针 ptr 虽然实际指向派生类对象 myCar，但由于它自己是 Conveyance 类型的指针，所以只能调用 getSpeed()函数，而无法通过它去操作 getWheelsNum()函数；只从 myCar 中复制了 speed 属性的值给 x1 对象，wheelsNum 的值被舍弃；func()函数内部执行的是 Conveyance 类的 showInfo()函数。

在以后的编程中，读者经常会看到把派生类对象当作基类对象使用的情形。

3.2 派生类的构造与析构函数

之前说到派生类会继承基类中的全部成员，其实有一些例外，就是构造函数和析构函数。它们都不能被派生类所继承，在派生类中需要声明和实现自己的构造函数，以及在必要的时候自定义析构函数。

3.2.1 实现方式


视频讲解

派生类构造函数在实现时只需对本类中新增的成员进行初始化，基类部分的初始化会自动调用基类构造函数完成。这里有以下几点建议。

(1) 当基类中声明了带有形参的构造函数时，派生类也应声明带形参的构造函数，并传参数给基类构造函数。

(2) 当基类中声明了不带参的构造函数时，派生类构造函数可以不向基类构造函数传递参数，此时基类部分的初始化调用的是基类不带参构造函数。

(3) 若基类中未声明构造函数，在派生类中也可以不声明，此时都采用系统默认生成的构造函数。例如，3.1 节中各个派生类对象生成时均是调用了派生类和基类的默认构造函数。

构造函数的实现格式如下。

```
派生类类名::派生类类名(基类所需形参,本类数据成员所需形参):基类名(基类实参)
{
    新增数据成员赋初值;   //更建议写在初始化列表处
}
```

形参列表中形参的顺序无规定，前后可以调换。“基类实参”通常由派生类构造函数形参列表中的“基类所需形参”给出，但也可以是常量、全局变量等。对基类构造函数传递参数必须在派生类的初始化列表中进行。

例如，首先在 Conveyance 类定义的 public 部分添加构造函数如下。

```
Conveyance(int spd):speed(spd)
{
    cout <<"Constructor of Conveyance."<< endl;
}
```

然后为 Car 类在类内添加公有构造函数如下。

```
Car(int spd,int wN):Conveyance(spd),wheelsNum(wN)
{
    cout <<"Constructor of Car."<< endl;
}
```

在基类 Conveyance 中只有一个带参构造函数，派生类 Car 的对象初始化时，基类部分只能使用该带参构造函数进行初始化，因此派生类的构造函数也应有参数，并且部分参数是用于传递给基类初始化用的。在派生类 Car 的形参列表中包含了用于初始化基类部分的参数 spd 和用于初始化新增成员的参数 wN，并在初始化列表中完成对基类构造函数参数的传递和新增成员初始化值传递的工作。

对于有多个基类的情形，依次在初始化列表中向每个基类传递参数即可。例如，首先为 House 类在类内添加公有构造函数：

```
House(double a):area(a)
{
    cout <<"Constructor of House."<< endl;
}
```

对于 MotorHome 类，可在类内添加公有构造函数：

```
MotorHome(int spd,int wN,double a,int wR)
                :Car(spd,wN),House(a),waterReserve(wR)
{
}
```

则在派生类对象生成时，首先分别调用每个基类的构造函数，然后再调用派生类自己的构造函数。

派生类中新增的也可能是成员对象，此时也需要初始化成员对象，初始化时需要的值同样由派生类的构造函数传递（在基类中也可能有成员对象，但由基类构造函数初始化）。此时派生类构造函数的格式如下。

```
派生类类名::派生类类名(基类所需形参,新增成员对象所需形参,新增数据成员所需形参)
        :基类名(基类实参),新增对象成员名(新增成员对象的实参)
{
   新增普通数据成员赋初值;   //更建议写在初始化列表处
}
```

同样，形参的顺序并无规定。新增成员对象的实参一般由形参列表中的“新增成员对象所需形参”给出，但也可以是常量、全局变量等。成员对象初始化的值需要在初始化列表处传递，若列表中没有列出成员对象，系统会调用成员对象的默认构造函数。

例如，交通工具都有出厂日期，为了表示日期属性，定义 Date 类：

```
class Date
{
    int year,month,day;
public:
    Date(int yr,int mon = 1,int d = 1):year(yr),month(mon),day(d)
    {
    }
    void showDate()
    {
        cout <<"Date: "<< month <<","<< day <<" "<< year << endl;
    }
};
```

然后在基类 Conveyance 的定义中添加一个私有成员对象：

```
Date manuDate;
```

则类的构造函数需要修改为

```
Conveyance(int yr,int spd):manuDate(yr),speed(spd)
{
    cout <<"Constructor of Conveyance."<< endl;
}
```

这里只给成员对象 manuDate 的 year 属性传递了初始值，month 和 day 使用了默认值 1。在第 2 章已有的 Time 类的基础上，在派生类 Car 中添加一个私有成员对象表示车载系统每日自检的时间，代码如下。

```
Time testTime;
```

此时，Car 类中的构造函数需要修改为

```
Car(int yr,int spd,int hr,int wN)
    :Conveyance(yr,spd),testTime(hr),wheelsNum(wN)
{
    cout <<"Constructor of Car."<< endl;
}
```

这里派生类构造函数需要给基类、派生类的新增成员对象、派生类的新增普通数据成员传递初始值，因此形参列表中有 4 个参数。从代码中可以看到，在对成员 testTime 初始化时调用的是它的带一个参数的构造函数，基类部分的成员对象 manuDate 是由基类构造函数统一初始化的。

因为析构函数不带参数，所以其声明与实现和普通类中的析构函数是一样的。在派生类对象的内存空间被回收前会依次调用成员对象及各基类的析构函数，分别完成各个部分的析构功能。


视频讲解

3.2.2 调用顺序

在派生类对象初始化时实际上有多个类(派生类、基类、成员对象的类)的构造函数体被执行,它们遵循规定好的调用顺序。整个过程如下。

(1) 将对象初始化时给出的实参值传递给派生类构造函数的形参。

(2) 按照派生类在定义时基类列表中的顺序依次调用各个基类的构造函数。基类没有在派生类初始化列表中列出时调用它的不带参构造函数,已列出时调用它的带参构造函数(构造函数使用的是传递来的参数)。

(3) 按照派生类中新增数据成员(包括成员对象)在类中声明的顺序依次初始化每个数据成员(为成员对象时调用其构造函数初始化)。

(4) 执行派生类构造函数的函数体。

提示:*构造函数的调用顺序只与基类列表中的基类顺序和新增成员对象在类中声明的顺序有关,与初始化区域中的顺序无关。*

例如,按照上述类的定义和继承关系生成 Car 类对象时,构造函数体执行的顺序依次为 Date 类的构造函数体、Conveyance 类的构造函数体、Time 类的构造函数体和 Car 类的构造函数体。其中,Date 类的构造函数体是由 Conveyance 类的构造函数初始化成员对象 manuDate 时被调用的。

析构函数执行的顺序和构造函数严格相反。例如,上述 Car 类对象在析构时依次执行 Car 类的析构函数体、Time 类的析构函数体、Conveyance 类的析构函数体和 Date 类的析构函数体。

项目 3_1 给出了上述类更为完整的定义,通过运行结果展示了派生类中构造函数和析构函数的执行顺序。时间类 Time 使用项目 2_3 中的 time.cpp 和 time.h,其他类的成员函数由于实现都较为简单,所以把它们都当作内联成员函数写在类定义中,于是每个类对应一个头文件。

Date 类定义在 date.h 头文件中,代码如下。

```
/*****************************************
 * 项目名: 3_1
 * 文件名: date.h
 * 说  明: 日期类
*****************************************/
#ifndef DATE_H
#define DATE_H

#include <iostream>
using namespace std;

class Date
{
    int year,month,day;
```

```
public:
   Date(int yr,int mon=1,int d=1):year(yr),month(mon),day(d)
   {
       cout<<"Constructor of Date."<<endl;
   }
   ~Date()
   {
       cout<<"Destructor of Date."<<endl;
   }
   void showDate()
   {
       cout<<"Date: "<<month<<","<<day<<" "<<year<<endl;
   }
};

#endif //DATE_H
```

交通工具类 Conveyance 定义在 conveyance.h 头文件中,代码如下。

```
/*****************************************
 * 项目名: 3_1
 * 文件名: conveyance.h
 * 说  明: 交通工具类
*****************************************/
#ifndef CONVEYANCE_H
#define CONVEYANCE_H

#include<iostream>
#include "date.h"
using namespace std;

class Conveyance
{
    Date manuDate;                  //生产日期
    double speed;                   //时速
public:
    Conveyance(int yr,int spd):manuDate(yr),speed(spd)
    {
        cout<<"Constructor of Conveyance."<<endl;
    }

    ~Conveyance()
    {
        cout<<"Destructor of Conveyance."<<endl;
    }

    double getSpeed()
```

```
    {
        return speed;
    }

    void showInfo()
    {
        cout <<"Speed per hour:"<< speed << endl;
    }
};

#endif //CONVEYANCE_H
```

汽车类 Car 定义在 car.h 头文件中，代码如下。

```
/*****************************************
 * 项目名: 3_1
 * 文件名: car.h
 * 说  明: 汽车类
 *****************************************/
#ifndef CAR_H
#define CAR_H

#include <iostream>
#include "time.h"
#include "conveyance.h"
using namespace std;

class Car:public Conveyance          //汽车类
{
    Time testTime;                   //系统每日自检的时间
    int wheelsNum;                   //车轮数
public:
    Car(int yr,int spd,int hr,int wN)
        :Conveyance(yr,spd),testTime(hr),wheelsNum(wN)
    {
        cout <<"Constructor of Car."<< endl;
    }

    ~Car()
    {
        cout <<"Destructor of Car."<< endl;
    }

    int getWheelsNum()
    {
        return wheelsNum;
    }

    void showInfo()
```

```
    {
        Conveyance::showInfo();
        cout <<"Number of wheels:"<< wheelsNum << endl;
    }
};

#endif //CAR_H
```

房子类 House 定义在 house.h 头文件中,代码如下。

```
/*****************************************
 * 项目名: 3_1
 * 文件名: house.h
 * 说  明: 房子类
*****************************************/
#ifndef HOUSE_H
#define HOUSE_H

#include<iostream>
using namespace std;

class House                          //房子
{
    double area;                     //面积
public:
    House(double a):area(a)
    {
        cout <<"constructor of House."<< endl;
    }

    ~House()
    {
        cout <<"Destructor of House."<< endl;
    }

    double getArea()
    {
        return area;
    }

    void showInfo()
    {
        cout <<"Area of house:"<< area << endl;
    }
};

#endif //HOUSE_H
```

房车类 MotorHome 定义在 motorhome.h 文件中，代码如下。

```
/*****************************************
 * 项目名：3_1
 * 文件名：motorhome.h
 * 说　明：房车类
 *****************************************/
#ifndef MOTORHOME_H
#define MOTORHOME_H

#include "car.h"
#include "House.h"
#include <iostream>
using namespace std;

class MotorHome:public Car,public House      //房车
{
    int waterReserve;                        //储水量
public:
    MotorHome(int yr,int spd,int hr,int wN,double a,int wR)
        :Car(yr,spd,hr,wN),House(a),waterReserve(wR)
    {
        cout<<"constructor of MotorHome."<<endl;
    }

    ~MotorHome()
    {
        cout<<"Destructor of MotorHome."<<endl;
    }

    int getWaterReserve()
    {
        return waterReserve;
    }

    void showInfo()
    {
        Car::showInfo();
        House::showInfo();
        cout<<"Water reserve:"<<waterReserve<<endl;
    }
};

#endif //MOTORHOME_H
```

主函数定义在 main.cpp 文件中，代码如下。

```
/*****************************************
 * 项目名：3_1
 * 文件名：main.cpp
```

```
 * 说　明：派生类中构造函数和析构函数的调用顺序
 ******************************************/
#include <iostream>
#include "MotorHome.h"
using namespace std;

int main()
{
    MotorHome mh(2019,100,23,6,8,20);
    return 0;
}
```

程序运行结果如图 3-5 所示。

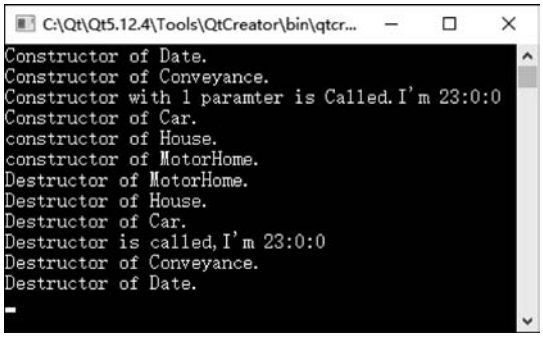


图 3-5　派生类中构造函数和析构函数的调用顺序

可以看到对象生成时按照基类构造函数体、新增成员对象构造函数体、派生类构造函数体的顺序进行。需要注意的是，基类可能还有它的基类，例如对于派生类 MotorHome 的基类 Car，它也有基类 Conveyance，因此 Car 类的构造函数体在执行前要先初始化它的基类 Conveyance 的部分，再初始化它的新增 Time 类的成员对象，然后再执行自己的构造函数体。

析构顺序和构造顺序是严格相反的，读者可参照图 3-5 进行查看。

3.3　二义性问题与虚基类

视频讲解

所谓二义性，是指在继承时基类之间或基类与派生类之间发生成员同名时出现的对成员访问的不确定性。

在 3.1.3 节中重定义成员函数时，派生类中存在两个同函数原型的成员函数，这就是一种二义性问题。C++对此的处理规则是：默认使用函数名调用的是派生类中新定义的成员函数，如须使用基类中同原型的成员函数，需要使用“基类名::”限定。这样问题就得到了解决。

但上述规则并不是在所有情形下都是一种最好的解决方式，在本节中将考虑一些特殊的情形，并引入虚基类的概念解决这些特殊情形带来的问题。

3.3.1 二义性问题

当一个派生类从多个基类派生，而这些基类又有一个共同的基类(祖先类)时，派生类中实际包含了多份祖先类的成员。例如在图 3-1 中，水陆两栖车同时继承自汽车和船，而这两者又都继承自同一个基类——交通工具。各个类中包含的成员如图 3-6 所示。

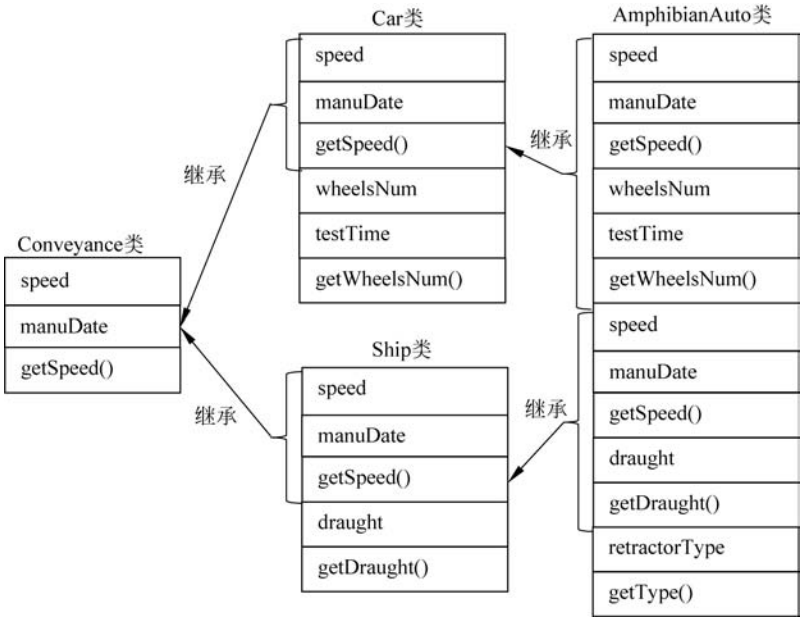


图 3-6 具有二义性成员的 AmphibianAuto 类

其中，船类 Ship 的定义如下。

```
class Ship:public Conveyance
{
    double draught;                    //船的吃水深度
public:
    double getDraught()
    {
        return draught;
    }
};
```

水陆两栖车类 AmphibianAuto 的定义如下。

```
class AmphibianAuto:public Car,public Ship
{
    int retractorType;                 //车轮收放装置类型编号
public:
    int getType()
    {
        return retractorType;
    }
};
```

可以看到，在 AmphibianAuto 类中有两个 speed、两个 manuDate 数据成员和两个 getSpeed()成员函数。它们实际上都来源于祖先类 Conveyance。应该怎样区分它们呢？

以公有接口 getSpeed 为例，对于 AmphibianAuto 类型的对象 ampAuto，语句：

```
ampAuto.getSpeed();
```

是不能通过编译的，因为存在两个 getSpeed()成员函数。语句：

```
ampAuto.Conveyance::getSpeed();
```

依然因无法区分而不能通过编译。在这种情形下，只能通过：

```
ampAuto.Car::getSpeed();
ampAuto.Ship::getSpeed();
```

进行区分。

虽然这里提供了一种区分的方式，但从类设计时的初衷来看，现在的 AmphibianAuto 类并不是设想的样子。因为对于水陆两栖车，它的出厂日期只有一个，额定速度也只有一个，而 ampAuto 对象中存储了这些属性的两份副本。数据冗余带来的问题不光是内存空间的浪费，更严重的是可能存在潜在的数据不一致问题。

3.3.2 虚基类

对于 3.3.1 节中介绍的情况，这里引入虚基类的概念。虚基类用于有共同基类的场合，主要用来解决多继承时可能发生的对同一基类继承多次而产生的二义性问题。它为最远的派生类提供唯一的一份虚基类成员，而不重复产生多份副本。

一个基类可以在生成一个派生类时作为虚基类，而在生成另一个派生类时不作为虚基类。因此虚基类是在定义派生类时声明的，形式如下。

```
class 派生类名: virtual 继承方式 基类名
{
        派生类新增成员的声明
};
```

即指明基类时在基类继承方式前加上 virtual 关键字。

需要注意，在设计类继承关系时，是在最开始处将共同基类设计为虚基类。对于 3.2 节中的例子，需要修改 Conveyance 类为 Car 类和 Ship 类的虚基类，即对 Car 类和 Ship 类的定义修改为

```
class Car: virtual public Conveyance
{
        //花括号中的代码和之前的代码相同,这里不再赘述
};
class Ship: virtual public Conveyance
{
        //花括号中的代码和之前的代码相同,这里不再赘述
};
```

而 AmphibianAuto 类无须改动。该类中只有一个 speed、manuDate 数据成员和一个 getSpeed()成员函数存在。此时,无论是以下哪条语句,调用的都是同一个唯一的 getSpeed()函数。

```
ampAuto.getSpeed();
ampAuto.Conveyance::getSpeed();
ampAuto.Car::getSpeed();
ampAuto.Ship::getSpeed();
```

由于派生类中虚基类部分的成员只存在一份,所以派生类构造函数的设计和之前会有所不同,需要注意以下 3 点。

(1) 由虚基类直接或间接派生出的所有派生类都应对虚基类初始化(通过派生类构造函数的初始化列表实现。若列表中没有列出,表示使用虚基类的默认构造函数)。

(2) 对象生成时,首先构建虚基类部分,然后是其他基类部分(这些基类在构建时会忽略掉对虚基类的构建,从而保证虚基类的部分只初始化一次),最后是自己新增的成员。

(3) 若包含多个虚基类,这些虚基类的部分构造顺序遵循(类层次上)从上到下、(同级时,按声明顺序)从左到右的规则。

项目 3_2 对此进行了说明。它在项目 3_1 的基础上对 Car 类的代码进行了修改。

```
/*****************************************
 * 项目名: 3_2
 * 文件名: car.h
 * 说  明: 汽车类,虚继承自 Conveyance
 *****************************************/
    //在代码中只修改了 Car 类定义的第 1 行,添加了 virtual 关键字如下
class Car: virtual public Conveyance //汽车类
    //其他的代码完全一致,这里不再赘述,请参考项目 3_1 中的 car.h
```

船类 Ship 定义在 ship.h 头文件中,代码如下。

```
/*****************************************
 * 项目名: 3_2
 * 文件名: ship.h
 * 说  明: 船类,虚继承自 Conveyance
 *****************************************/
#ifndef SHIP_H
#define SHIP_H

#include "conveyance.h"
#include<iostream>
using namespace std;

class Ship: virtual public Conveyance          //船类
{
```

```
    double draught;                                    //船的吃水深度
public:
    Ship(int yr,int spd,double drt):Conveyance(yr,spd),draught(drt)
    {
        cout <<"constructor of Ship."<< endl;
    }

    ~Ship()
    {
        cout <<"Destructor of Ship."<< endl;
    }

    double getDraught()
    {
        return draught;
    }

    void showInfo()
    {
        Conveyance::showInfo();
        cout <<"Ship draught:"<< draught << endl;
    }
};

#endif //SHIP_H
```

水陆两栖车类 AmphibianAuto 定义在 amphibianauto.h 头文件中，代码如下。

```
/*****************************************
 * 项目名：3_2
 * 文件名：amphibianauto.h
 * 说　明：水陆两栖车类
 *****************************************/
#ifndef AMPHIBIANAUTO_H
#define AMPHIBIANAUTO_H

#include<iostream>
#include "car.h"
#include "ship.h"
using namespace std;

class AmphibianAuto:public Car,public Ship        //水陆两栖车类
{
    int retractorType;                             //车轮收放装置类型编号
public:
    AmphibianAuto(int yr,int spd,int hr,int wN,double drt,int rT)
```

```
        :Conveyance(yr,spd),Car(yr,spd,hr,wN),Ship(yr,spd,drt),retractorType(rT)
    {
        cout <<"constructor of AmphibianAuto."<< endl;
    }

    ~AmphibianAuto()
    {
        cout <<"Destructor of AmphibianAuto."<< endl;
    }

    int getType()
    {
        return retractorType;
    }

    void showInfo()
    {
        Car::showInfo();
        Ship::showInfo();
        cout <<"Retractor type:"<< retractorType << endl;
    }
};

#endif //AMPHIBIANAUTO_H
```

主函数所在的文件为 main.cpp,代码如下。

```
/*****************************************
 * 项目名: 3_2
 * 文件名: main.cpp
 * 说  明: 派生类中构造函数和析构函数的调用顺序
 *****************************************/
#include<iostream>
#include "amphibianauto.h"
using namespace std;

int main()
{
    AmphibianAuto ampAuto(2020,100,23,8,1.5,2);
    return 0;
}
```

若想程序正常运行,还需要修改 motorhome.h 文件(或直接把该文件删除),因为虽然实际没有用到该类,但是由于它继承自 Car 类,而 Car 类虚继承了 Conveyance 类,所以在该类的构造函数中也需要对虚基类 Conveyance 初始化,然后整个项目才能编译通过。修改 motorhome.h 文件如下。

```
/*****************************************
 * 项目名：3_2
 * 文件名：motorhome.h
 * 说　明：房车类,基于虚基类的原因,修改了构造函数
 *****************************************/
    //在代码中只修改了构造函数的初始化列表,添加了对虚基类 Conveyance 的初始化
   MotorHome(int yr,int spd,int hr,int wN,double a,int wR)
        :Conveyance(yr,spd),Car(yr,spd,hr,wN),House(a),waterReserve(wR)
    //其他的代码完全一致,这里不再赘述,请参考项目 3_1 中的 motorhome.h
```

程序运行结果如图 3-7 所示。

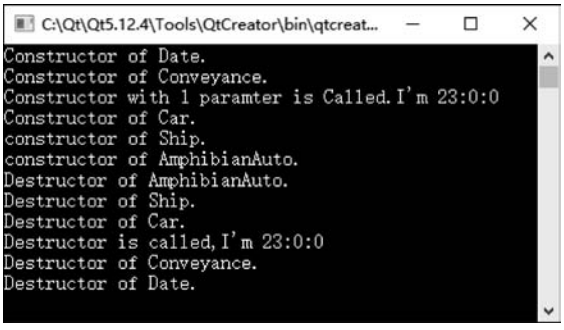


图 3-7　项目 3_2 的运行结果

AmphibianAuto 类的 ampAuto 对象在生成时首先调用虚基类 Conveyance 的构造函数(内部调用了 Date 类的构造函数对成员对象进行初始化),输出前两行；然后调用基类 Car 的构造函数(内部调用了 Time 类的构造函数对成员对象进行初始化),该函数在执行期间忽略了对基类 Conveyance 的初始化,输出第 3、4 行；再调用基类 Ship 的构造函数,该函数在执行期间也忽略掉了对基类 Conveyance 的初始化,结果输出第 5 行；最后执行自己的构造函数体,输出第 6 行。在对象的内存空间被回收时,析构顺序严格逆序于构造函数。

3.4　Qt 自定义派生类

本节介绍如何使用向导创建 Qt 项目,分析自动生成的自定义窗口派生类的代码,以及如何在此基础上进一步扩充自定义窗口派生类。

3.4.1　使用向导创建项目


视频讲解

选择菜单栏“文件”→“新建文件或项目”命令,在打开的窗口中选择 Application→Qt Widgets Application 选项,表示创建一个 Qt 窗口应用,如图 3-8 所示。

单击 Choose 按钮后在弹出的对话框中输入项目名,并选择路径,然后单击“下一步”按

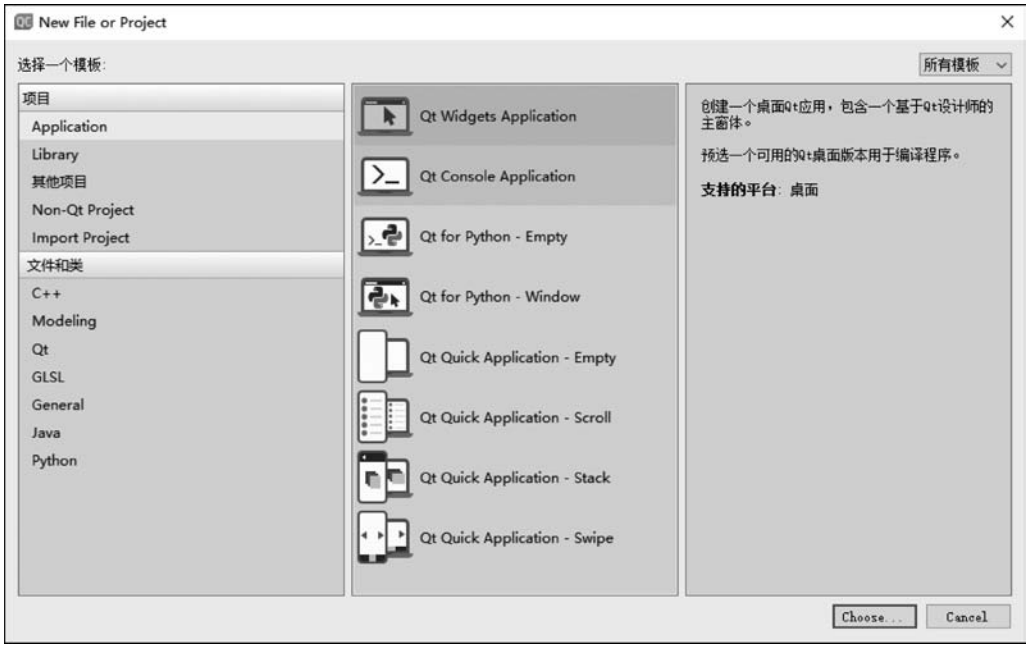


图 3-8　创建 Qt 窗口应用

钮选择工具包，再次单击“下一步”按钮将出现如图 3-9 所示的类信息界面。这里有 3 个基类可以选择，即 QDialog、QWidget 和 QMainWindow，有关它们的介绍请参考 2.3.1 节。本例选择 QWidget 作为基类，在类名处输入要创建的(基于 QWidget 的)派生类类名，向导会自动填充该类对应的头文件和源文件的文件名；并取消勾选“创建界面”复选框，如图 3-9 所示。

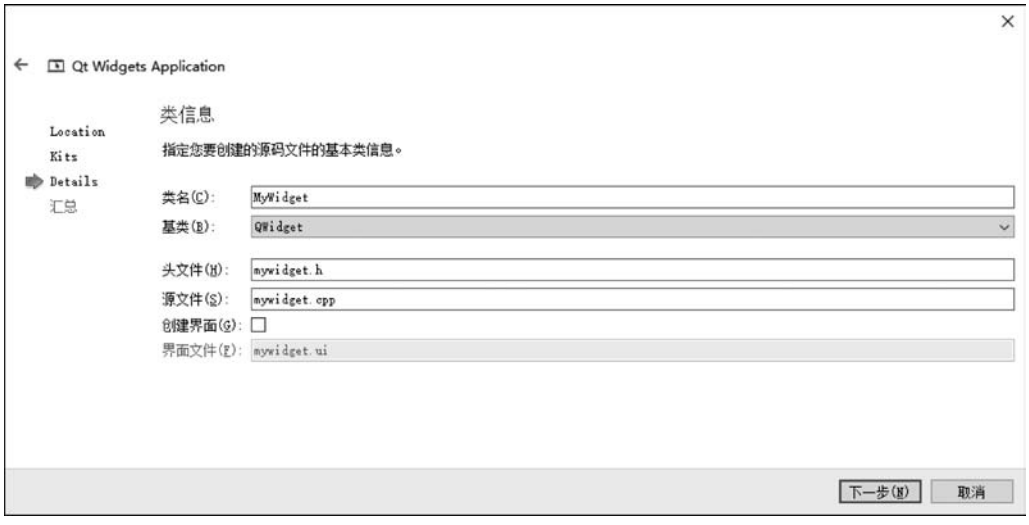


图 3-9　选择基于的基类

依次单击“下一步”按钮和“完成”按钮，即可生成一个完整的、可显示自定义窗口的项目，如图 3-10 所示。

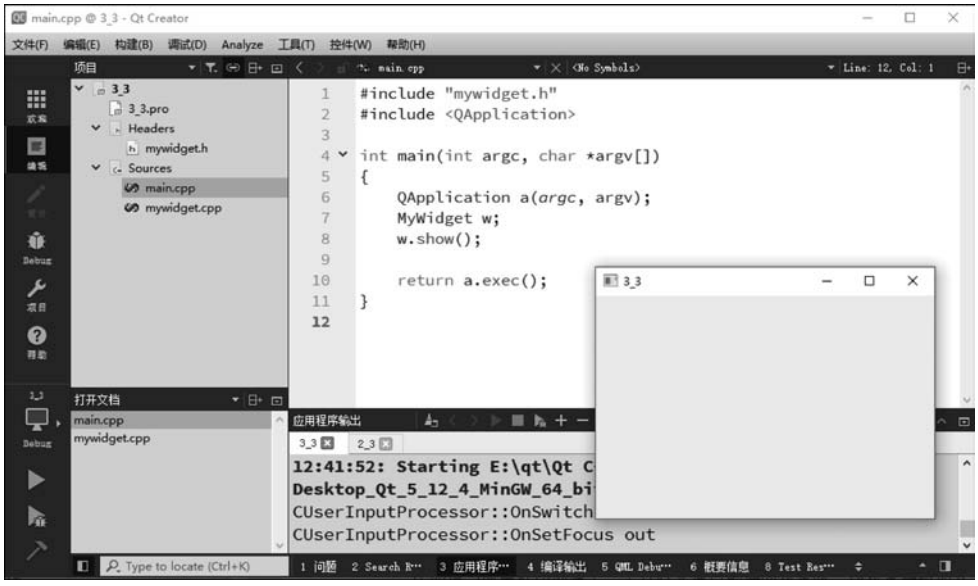


图 3-10　向导生成的项目及运行效果

向导的作用就是根据用户的设置自动地帮助用户生成各个文件(包括.pro 项目文件)中的代码。此项目的运行效果和项目 1_9 完全一样。

在项目 1_9 中创建的窗口只能是 QWidget 类规定好的外观和功能,要在窗口中添加部件,只能在 main()函数中定义部件并将窗口初始化为部件的父窗口,以便在显示窗口时一起显示出来。在这种方式下,窗口、部件、交互行为并没有封装在一起,本质上仍是一种面向过程的程序设计思路。

提示:父类和父窗口(父对象)不是一个概念。父类是指类继承与派生时被继承的那个基类;而父窗口是指被一个部件对象用作容纳、放置自身的一个窗口对象。

观察本项目可以发现,不同于项目 1_9 使用 QWidget 类生成窗口实例,这里使用了自定义的派生类 MyWidget 创建一个窗口。而从 MyWidget 类定义的代码中可以看到,MyWidget 类是一个继承自 QWidget 的派生类。

使用向导创建的项目采用了面向对象程序设计的思想,希望使用者在 QWidget 类的基础上进一步设计自己特定的窗口类,如给它添加一些部件、设计相应的动作等。这样就将窗口、窗口中的部件、各部件间的交互、部件和用户间的交互等都封装在自己定义的 MyWidget 类中,从而每个自定义窗口类 MyWidget 的实例都会拥有这些部件,以及可执行相应的动作。

下面在向导生成的代码的基础上进一步扩充和修改:对于派生类 MyWidget,在类中添加一个单行文本框成员、3 个按钮成员,以及一个为了实现信号映射而添加的 QSignalMapper * 指针。修改 mywidget.h 头文件如下。

```
/*****************************************
 * 项目名: 3_3
 * 文件名: mywidget.h
 * 说　明: 自定义窗口类
 *****************************************/
```

```
#ifndef MYWIDGET_H
#define MYWIDGET_H

#include <QWidget>
#include <QLineEdit>
#include <QPushButton>
#include <QSignalMapper>

class MyWidget: public QWidget
{
    Q_OBJECT

public:
    MyWidget(QWidget *parent = nullptr);
    ~MyWidget();

private:
    QLineEdit lineEdit;
    QPushButton btn1,btn2,btn3;
    QSignalMapper *mapper;
};

#endif //MYWIDGET_H
```

类开头处的 Q_OBJECT 宏是为了支持 Qt 的信号与槽机制，如果想在类中定义信号与槽，或者连接已有的信号与槽，都必须在类中添加该宏。Q_OBJECT 宏一般写在类定义最开头的私有区域中。

在构造函数中设置了一个 QWidget 类型的指针，用于指明自定义窗口类对象的父窗口，默认为无父窗口（指针初始化为空指针）。

为了实现和项目 2_14 同样的功能，需要修改 MyWidget 的构造函数，以实现对部件成员对象的初始化、属性设置，以及指针所指对象空间的动态申请和信号与槽的连接；需要修改析构函数，以释放申请的空间等。修改后的 mywidget.cpp 源文件如下。

```
/*****************************************
 * 项目名：3_3
 * 文件名：mywidget.cpp
 * 说　明：自定义窗口类的实现
 *****************************************/
#include "mywidget.h"

MyWidget::MyWidget(QWidget *parent)
    : QWidget(parent),lineEdit(this),btn1(this),btn2(this),btn3(this)
{
    //代码段 1：设置初始属性
```

```
    this->resize(400,300);                    //设置窗口和各部件的大小、位置,以及显示的文本等
    lineEdit.move(10,10);
    btn1.move(10,30);
    btn2.move(10,50);
    btn3.move(10,70);
    lineEdit.setText("我是单行文本框");
    btn1.setText("清除");
    btn2.setText("设置一段文字");
    btn3.setText("关闭窗口");

    //代码段 2: 信号与槽的连接
    QObject::connect(&btn1,SIGNAL(clicked()),&lineEdit,SLOT(clear()));
    QObject::connect(&btn3,SIGNAL(clicked()),this,SLOT(close()));

    //代码段 3: 通过 QSignalMapper 把无参信号 clicked 翻译成带 QString 参数的信号
    mapper = new QSignalMapper;
    QObject::connect(&btn2, SIGNAL(clicked()), mapper, SLOT(map()));
    mapper->setMapping(&btn2, "我是一行文字");
    QObject::connect(mapper, SIGNAL(mapped(const QString&)),
                     &lineEdit, SLOT(setText(const QString&)));

    //      //不起作用的 QSignalMapper
    //      QSignalMapper mapper;
    //      QObject::connect(&btn2, SIGNAL(clicked()), &mapper, SLOT(map()));
    //      mapper.setMapping(&btn2, "我是一行文字");
    //      QObject::connect(&mapper, SIGNAL(mapped(const QString&)),
    //                       &lineEdit, SLOT(setText(const QString&)));
}

MyWidget::~MyWidget()
{
    delete mapper;
}
```

当自定义窗口类对象有父窗口(由形参 parent 指针接收,默认值为空指针)时,将通过构造函数的初始化列表传递给基类(QWidget)的构造函数,以完成父窗口的设置工作。在初始化列表中也分别对成员对象 lineEdit、btn1、btn2 和 btn3 进行了初始化,即将它们的父窗口设置为 this 指针所指的对象(也就是当前自定义窗口类对象)。

代码段 1 设置本窗口及部件成员的大小、位置,以及显示的文字等。

代码段 2 实现信号与槽的连接。注意此时是在自定义窗口类的构造函数中实现连接的,因此关联 btn3 的 clicked 信号和本窗口的 close()槽时接收对象使用的是 this 指针(指向当前自定义窗口类对象)。

代码段 3 中对映射类的使用和项目 2_14 中的写法有所不同,主要区别在于:项目 2_14 中的 mapper 是一个定义的对象,而在本项目中是一个使用 new 运算符得到的对象(需要在析构函数中释放申请的空间)。这里 mapper 是一个指针,注意调用 setMapping()函数和

connect()函数时的代码与在项目 2_14 中的不同。

无须修改主函数所在的 main.cpp 文件,向导自动生成的代码如下。

```
/*****************************************
 * 项目名: 3_3
 * 文件名: main.cpp
 * 说  明: 自定义窗口类的使用
*****************************************/
#include "mywidget.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MyWidget w;
    w.show();
    return a.exec();
}
```

程序运行界面同图 2-18,运行效果与项目 2_14 完全相同。

提示:

(1) 实际上,程序员的大部分工作都是在创建自己的类,为其添加各个部件,实现对部件间、部件和用户间交互的响应等。而 main()函数通常都像项目 3_3 中一样,简单地完成定义 QApplication 类型的应用程序对象、生成窗口对象、调用应用程序对象的 exec()函数进行事件循环等工作即可。

(2) 因为可视化的窗口(或部件)类若完全由用户从头开始创建太过复杂(还涉及自我绘制等底层操作),所以绝大多数情况下自定义的可视化窗口(或部件)类都是从一个 Qt 框架中的基础可视化窗口(或部件)类派生得来的。

对于代码段 3,似乎比项目 2_14 中的更麻烦一点。但如果采用项目 2_14 的形式(即将代码段 3 替换为注释掉的部分,并注释掉析构函数中的 delete 语句),运行后会发现 btn2 按钮是不起作用的,这是为什么呢? 在 3.4.2 节讨论此问题。


视频讲解

3.4.2 静态创建类对象和动态创建类对象的区别

C++创建类对象的方式有两种。

一种是静态创建(注意,不是指定义为静态对象),即使用“类名 对象名;”的方式创建类对象,这种方式会在运行过程中,当进入对象作用域时给对象在栈中分配内存;在超出作用域范围时由系统自动析构对象并回收内存空间。

另一种是动态创建,即在程序运行过程中,使用 new 运算符给对象在堆空间中申请内存。动态创建的对象的内存空间不会被自动回收,需要编程者小心维护,并需要在不再使用该对象时显式地用 delete 运算符释放掉该内存空间(在释放前系统会自动调用对象的析构函数)。

项目 3_3 在构造函数中使用“QSignalMapper mapper;”方式(实例化)得到的对象 mapper 不起作用的原因在于：mapper 是构造函数中的局部对象，作用域只在该构造函数的内部。当 main()函数中生成了派生类 MyWidget 的对象 w 并调用构造函数初始化完毕后，该局部对象 mapper 就不存在了。当窗口显示后(语句“w.show();”执行完毕)，程序主要在“a.exec();”处不断地进行循环和事件处理，而此时已无 mapper 对象，因此当然不能完成信号映射的功能。实际上，如果把 lineEdit、btn1、btn2 和 btn3 对象也放在构造函数中定义而不是作为类的成员对象存在，在运行时这些部件也不会显示出来，原因是一样的(已不存在)。

对于需要在整个自定义窗口类对象生存期间都存在的部件及对象(这些部件和对象通常与这个窗口息息相关，如窗口中的子部件等)，这里有两种方法可以处理。

(1) 定义为窗口类的成员。如此，窗口类中的部件和窗口对象就具有同样的生存周期(是该窗口对象的一部分)。例如，项目 3_3 中 lineEdit、btn1、btn2 和 btn3 都采用了此种方式。

(2) 在类中定义指针成员，然后在构造函数中使用 new 运算符动态申请(需在析构函数中释放申请的空间)部件(或对象)空间，此空间只有在 delete 时才会被释放掉。例如，项目 3_3 中 mapper 采用了此种方式。

读者可试着在派生类中只声明部件指针，然后部件以 new 运算符动态申请；或者将 QSignalMapper 类对象以数据成员的形式添加到派生类中，在这两种方式的前提下修改代码，以熟悉类中部件的使用。

提示：

(1) 对于静态创建的对象，要注意它们的作用域。

(2) 对于使用 new 运算符动态申请的对象，注意当不再使用时要释放掉内存空间。

3.4.3　对象树机制

视频讲解

从 C++语言规范标准来说，使用 new 运算符动态申请的空间一定要在不再使用时释放掉，以免造成内存泄漏。在 Qt 框架中提供了对象树机制，极大地简化了程序员对动态申请对象空间的维护工作：对于以某窗口或部件为父对象(父窗口)的动态创建的对象，可以不用去编写 delete 语句，当父对象被销毁时，系统会自动调用相关的代码释放掉所有以此窗口或部件为父窗口的动态创建的对象。

提示：在项目 3_3 中，由于 mapper 并没有设置父窗口，所以依然需要使用 delete 语句显式地释放。

对象树机制实现的原理如下：如果一个子对象将另一个 QObject 类(它是 QWidget 的父类)对象或 QObject 派生类的对象作为自己的父对象，那么该子对象就会被添加到父对象的 children 列表中。当父对象被销毁时，对象树机制会从父对象的 children 列表中取出所有以它为父对象的对象，并依次销毁。

QObject 类(及其派生类)对象通常表示一个窗口(或部件)，一般只有显示在它们区域之中的子部件才会以它们为父窗口，当窗口(或部件)销毁时，也销毁它的子部件是一件理所应当的事情。在没有对象树机制之前，窗口销毁时需要程序员去关注并处理窗口内所有动

态申请的子部件内存空间回收的事情，而对象树机制大大减轻了程序员的负担，使其能将精力用在主要的业务逻辑上。

为了证实对象树机制，下面采用与 3.4.1 节同样的方式，使用向导创建一个不使用界面文件（取消“创建界面”复选框的勾选）的项目，自定义类名为 TestWidget。

在项目中定义一个自定义按钮类型 MyButton，操作步骤如下：给项目添加一个新的 C++类（添加方式参考 1.3.2 节，图 1-8 选择 C++class 选项），在定义类的界面（见图 3-11）中设置类的名字 MyButton。自定义按钮类基于 QPushButton 类创建，注意在该界面中填入基类 QPushButton。单击“下一步”按钮，按默认设置，单击“完成”按钮生成相关文件。

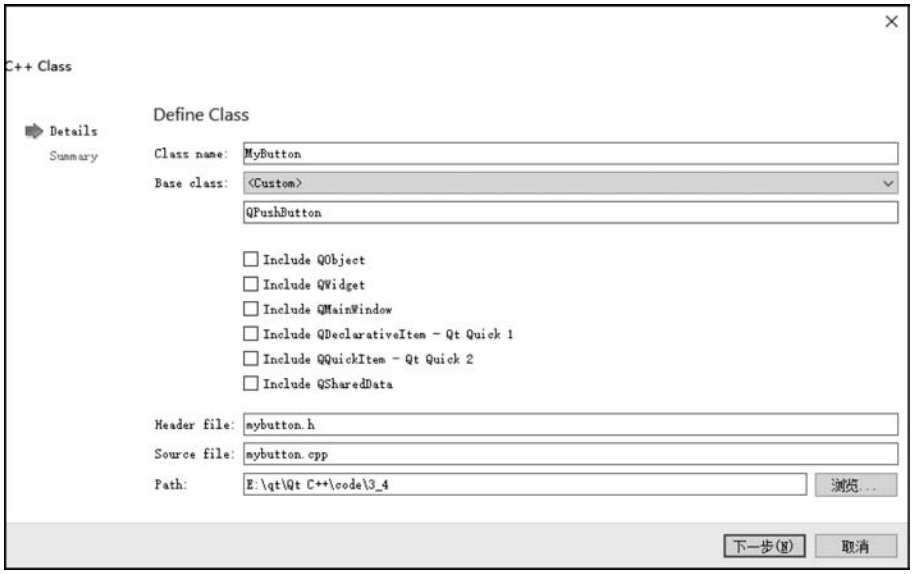


图 3-11　创建基于 QPushButton 的自定义按钮类 MyButton

提示：

（1）编写代码时注意添加需要的头文件。

（2）建议读者在上机时不要按照书上的代码直接敲进去，而是根据说明和给出的步骤自己试着添加代码，当编译不通过时再和书中的代码对比，查看漏掉的代码或写错的地方，以便及时注意到细节之处，从而快速提高自身的编程能力。

项目 3_4 中共有 5 个代码文件。MyButton 类是以 QPushButton 类为基类的，而该基类有带一个表示父对象的形参的构造函数（绝大多数可视化部件都有），因此派生类中应该也定义一个带有表示父对象的形参 parent 的构造函数，并将该形参传递给基类 QPushButton 的构造函数以设置父对象。在已生成代码的基础上修改 mybutton.h 如下。

```
/*****************************************
 * 项目名：3_4
 * 文件名：mybutton.h
 * 说　明：自定义按钮类
 *****************************************/
```

```
#ifndef MYBUTTON_H
#define MYBUTTON_H
#include <QPushButton>

class MyButton: public QPushButton
{
public:
    MyButton(QWidget *parent = nullptr);
    ~MyButton();
};

#endif //MYBUTTON_H
```

为了展示效果，这里声明了自定义按钮类 MyButton 的析构函数，并实现输出一句“自定义按钮的析构函数”。类实现文件 mybutton.cpp 的代码如下。

```
/*****************************************
 * 项目名：3_4
 * 文件名：mybutton.cpp
 * 说　明：自定义按钮类的实现
 *****************************************/
#include "mybutton.h"
#include <QDebug>

MyButton::MyButton(QWidget *parent):QPushButton(parent)
{
    this->setText("我是自定义按钮");
}

MyButton::~MyButton()
{
    qDebug()<<"自定义按钮的析构函数"<<endl;
}
```

在自定义窗口类 TestWidget 中添加一个私有的 MyButton 类型的指针，用于指向窗口内的自定义按钮子部件，将代码修改如下。

```
/*****************************************
 * 项目名：3_4
 * 文件名：testwidget.h
 * 说　明：自定义窗口类
 *****************************************/
#ifndef TESTWIDGET_H
```

```
#define TESTWIDGET_H

#include <QWidget>
#include "mybutton.h"

class TestWidget: public QWidget
{
    Q_OBJECT
public:
    TestWidget(QWidget *parent = nullptr);
    ~TestWidget();
private:
    MyButton *myBtn;
};

#endif //TESTWIDGET_H
```

在构造函数中给自定义按钮动态申请空间(由 myBtn 指向)。另外,为了展示效果,这里也声明了 TestWidget 类的析构函数,并实现输出一句“自定义窗口类的析构函数被调用”。类实现文件 testwidget.cpp 的代码如下。

```
/*****************************************
 * 项目名: 3_4
 * 文件名: testwidget.cpp
 * 说  明: 自定义窗口类的实现
*****************************************/
#include "testwidget.h"
#include "mybutton.h"
#include <QDebug>

TestWidget::TestWidget(QWidget *parent): QWidget(parent)
{
    this->resize(100,100);
    myBtn = new MyButton(this);
}

TestWidget::~TestWidget()
{
    qDebug()<<"自定义窗口类的析构函数被调用"<< endl;
}
```

主函数所在的 main.cpp 文件由项目向导生成后未做任何修改,代码在这里不再列出。

在程序运行时,首先会显示如图 3-12(a)所示的窗口。此时在应用程序输出窗口中还看不到输出,因为窗口和按钮对象都还存在。在关闭窗口后,应用程序输出窗口中显示的内容如图 3-12(b)所示。

(a) 运行界面　　(b) 输出窗口

图 3-12　项目 3_4 的运行界面及应用程序输出窗口中的内容

观察 TestWidget 类可以发现，在构造函数中使用了 new 运算符动态创建自定义按钮对象，但该对象没有在析构函数或其他任何地方显式地使用 delete 命令释放空间。通过应用程序的输出可以证实：当窗口关闭时，程序运行结束，窗口对象析构，由于对象树机制的存在，窗口对象的子部件——按钮对象也跟着析构了。

读者还可以试着将自定义窗口类构造函数中的语句：

```
myBtn = new MyButton(this);
```

替换为

```
myBtn = new MyButton;
myBtn -> show();
```

此时，由于 myBtn 所指的对象并未指定当前窗口是自己的父对象，所以会单独显示（调用 show()函数后才会显示）为一个窗口。当自定义按钮和自定义窗口都关闭后，在应用程序输出窗口中也只能看到自定义窗口的析构函数被调用了，而 myBtn 所指向的动态创建的按钮对象并未被安全释放。

提示：

（1）在 Qt 中，所有具有父对象的动态创建的部件（或窗口）都可以不用程序员显式地编写 delete 代码，对象树机制会在其父对象销毁前完成这些动态创建的对象的空间的回收。

（2）Qt 对象树机制在用户图形界面编程上是非常有用的，它能够帮助程序员将主要精力放在系统的业务上，提高编程效率；同时也减小了内存泄漏的压力，保证了系统的稳健性。

（3）在编程时请务必记得给各个子部件都设置合适的父窗口，以充分利用对象树机制。

3.4.4　自定义信号和槽

Qt 中信号与槽机制的实现需要元系统的支持。从编程的角度，需要满足以下条件：发送者类和接收者类必须直接或间接地继承自 QObject 类（可视的部件类都是它的子类）；类定义要放在头文件中；在类定义的私有访问权限处要加上宏 Q_OBJECT。

自定义的信号和槽也都在类中进行声明。信号在声明前用“signals:”限定，槽在声明前用“[访问权限] slots:”限定，即带有信号和槽的类定义形式为

```
class 类名: 继承权限 父类名
{
    Q_OBJECT
    //类其他成员的声明,包括指定访问权限
signals:
    信号的声明语句
public slots:
    公有槽的声明语句
protected slots:
    保护槽的声明语句
private slots:
    私有槽的声明语句
};
```

信号没有返回值,返回类型一般都是 void。信号函数只需要声明,可根据情况设置形参列表。信号可由用户的动作触发(实际的过程是:用户事件导致系统底层产生了消息,消息处理时发出信号。通常用户不关心这些底层的细节),也可在某一时刻由代码主动触发,使用 emit 命令触发信号,格式如下。

```
emit 信号名(实参列表);
```

emit 说明此时发送了一个信号,信号可以带参数(可以带零个或多个)。当与之关联的槽执行时,信号里带的实参依次传递给槽的形参。槽本质上是一个普通的成员函数,被声明为槽后可以作为接收者接收信号的处理函数被调用。槽函数需要实现,实现方式与普通成员函数相同。


视频讲解

1. 自定义槽

下面的项目 3_5 实现如下功能:窗口中包含两个单行文本框,当在第 1 个文本框中输入内容后按 Enter 键时,会在第 2 个文本框中显示出在第 1 个文本框中输入的内容。在该项目中添加了一个自定义的槽函数实现对文本框的“按 Enter 键”信号的响应。

采用与 3.4.1 节同样的方式,使用向导创建一个不使用界面文件(取消“创建界面”复选框的勾选)的项目,自定义类名为 MyWidget。

在自定义类 MyWidget 中添加两个 QLineEdit 类型的指针 edit1 和 edit2(注意添加相关的头文件);添加一个自定义槽函数 EnterPressedSlot(),用于实现在单行文本框 edit1 内按下 Enter 键时要执行的动作。mywidget.h 头文件的代码如下。

```
/*****************************************
 * 项目名: 3_5
 * 文件名: mywidget.h
 * 说  明: 自定义窗口类
*****************************************/
#ifndef MYWIDGET_H
```

```
#define MYWIDGET_H

#include <QWidget>
#include <QLineEdit>

class MyWidget: public QWidget
{
    Q_OBJECT

public:
    MyWidget(QWidget *parent = nullptr);
    ~MyWidget();
private:
    QLineEdit *edit1, *edit2;
public slots:
    void EnterPressedSlot();
};

#endif //MYWIDGET_H
```

信号和槽要先连接上才能使用，在多数情况下都是在窗口的构造函数中进行连接的。单行文本框在 Enter 键被按下时会发出 returnPressed 信号(可选中类名，然后按 F1 键查看类中的信号列表，再通过帮助文档获得有关该信号的说明)，需要将此信号和自定义的 EnterPressedSlot()槽连接起来。槽也需要实现，MyWidget 类的实现文件 mywidget.cpp 的代码如下。

```
/*****************************************
 * 项目名: 3_5
 * 文件名: mywidget.cpp
 * 说  明: 自定义窗口类的实现
*****************************************/
#include "mywidget.h"

MyWidget::MyWidget(QWidget *parent)
    : QWidget(parent)
{
    edit1 = new QLineEdit(this);
    edit2 = new QLineEdit(this);
    this->resize(300,100);
    edit1->move(10,10);
    edit2->move(10,50);

    connect(edit1,SIGNAL(returnPressed()),this,SLOT(EnterPressedSlot()));
}

void MyWidget::EnterPressedSlot()
```

```
{
    QString str = edit1 -> text();
    edit2 -> setText("输入为:" + str);}

MyWidget::~MyWidget()
{
}
```

在构造函数中首先动态申请了两个单行文本框对象,并分别由类指针成员 edit1、edit2 指向,然后对当前窗口及其子部件的属性进行了设置。

因为自定义类 MyWidget 继承自 QWidget,QWidget 又继承自 QObject,所以 connect()函数也是 MyWidget 类中的成员函数,无须加上"QObject::"限定就可以直接使用。注意,因为槽是在自定义窗口类 MyWidget 中声明的,所以信号的接收者是当前窗口,它由 this 指针指向。

槽的实现比较简单。首先定义一个 QString 类型的字符串 str 并初始化为单行文本框 edit1 中的文字(通过 text()成员函数获取),然后将单行文本框 edit2 中的文字设置为字符串常量"输入为:"连接上字符串 str。

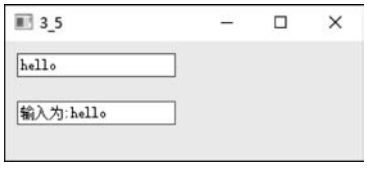


图 3-13　项目 3_5 的运行效果

主函数所在的 main.cpp 文件由项目向导生成后没有修改,这里不再列出。

程序运行效果如图 3-13 所示,在上面的文本框中输入 hello 并按 Enter 键,下面的文本框中会按照指定的格式显示上方文本框中的内容。

2. 取消关联

当一个对象被销毁后,Qt 会自动取消所有连接到这个对象的信号上的槽。对象也可以主动取消所有关联到自己信号上的槽。例如,在 EnterPressedSlot()函数体的最后添加语句:

```
edit1 -> disconnect();
```

则程序运行时,只能在上面的文本框中第一次按 Enter 键时,下面的文本框会按照指定的格式显示内容。然后,由于槽函数 EnterPressedSlot()执行了单行文本框 edit1 的解绑所有连接操作,所以再次输入内容并按 Enter 键时,发出的信号已无槽与之关联,下面的文本框也就不会再发生变化。

部件的 disconnect()成员函数有多种重载的形式,如参数可以是自己的信号:

```
edit1 -> disconnect(SIGNAL(returnPressed()));
```

该语句表示只解绑与单行文本框 edit1 的 returnPressed()信号相关的连接。参数还可以是接收者,例如:

```
edit1 -> disconnect(this);
```

该语句表示解绑单行文本框 edit1 与接收者 this(当前窗口)之间的所有信号槽的连接。

解绑信号槽的连接实际上还有很多方式，感兴趣的读者可以在 Qt Creator 环境中通过帮助系统进一步了解。

3. 自定义信号

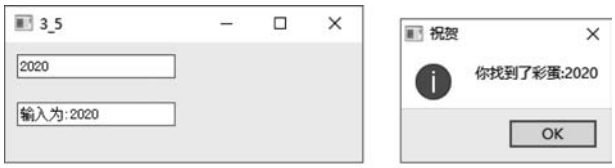

视频讲解

项目 3_5 只给出了槽的声明和定义，接下来学习如何自定义一个信号，并在合适的时候发送它。在项目 3_5 的基础上扩充如下功能：如果用户输入 2020 并按 Enter 键，将弹出一个“祝贺”消息框，告知用户找到了程序中的彩蛋。

考虑上述功能的实现：用户在单行文本框 edit1 中输入内容并按 Enter 键时，首先需要完成槽中的工作，然后对输入的内容进行检查，如果是“2020”（这就是信号发送的条件），则弹出消息框。该功能可以通过在 EnterPressedSlot()函数体的最后添加以下语句实现。

```
if(str == "2020")
    QMessageBox::information(this, "祝贺", "你找到了彩蛋:" + str);
```

information()函数是 QMessageBox 类中的静态成员函数，可以通过“类名::”的形式调用，用于显示一个信息框。其 3 个参数分别表示父窗口、消息框标题、消息框内容。运行时，在文本框 edit1 中输入 2020 并按 Enter 键后，运行效果如图 3-14 所示，不仅在图 3-14(a)所示窗口的下方的文本框中按指定的格式显示上方文本框的内容，还会弹出如图 3-14(b)所示的消息框。



(a) 显示输入的内容　　(b) 弹出消息框

图 3-14　项目 3_5 添加代码之后的运行效果

为了展示信号的使用和发送过程，在项目 3_6 中考虑把弹出消息框的功能独立出来作为一个槽函数。当槽函数 EnterPressedSlot()判断出输入的内容是特殊的“2020”字符串时，发送一个表示发现了“特殊字符串”的信号，该信号与实现弹出消息框的槽函数连接，从而实现与上述同样的功能。

在自定义窗口类 MyWidget 中添加一个自定义信号 specialStrSig 的声明，表示发现了特殊字符串；再声明一个槽 specialStrSlot()，以便在前述信号发生时进行处理（弹出消息框）。项目 3_6 实现时分别给此信号和槽设计了一个 QString 类型的参数，以传递该特殊的字符串。项目 3_6 在项目 3_5 的基础上进行修改，mywidget.h 头文件如下。

```
/*****************************************
 * 项目名: 3_6
 * 文件名: mywidget.h
 * 说　明: 自定义窗口类,添加了自定义信号和槽
*****************************************/
#ifndef MYWIDGET_H
```

```
#define MYWIDGET_H

#include <QWidget>
#include <QLineEdit>

class MyWidget: public QWidget
{
    Q_OBJECT

public:
    MyWidget(QWidget *parent = nullptr);
    ~MyWidget();
private:
    QLineEdit *edit1, *edit2;
public slots:
    void EnterPressedSlot();
    void specialStrSlot(QString);
signals:
    void specialStrSig(QString);
};

#endif //MYWIDGET_H
```

在类构造函数实现时,须完成对这对信号和槽的连接操作;在槽函数 EnterPressedSlot()中也要进行修改,以做到在条件满足时发送信号。修改后的 mywidget.cpp 的代码如下。

```
/*****************************************
 * 项目名: 3_6
 * 文件名: mywidget.cpp
 * 说  明: 自定义窗口类的实现,添加了自定义信号和槽
*****************************************/
#include "mywidget.h"
#include <QMessageBox>

MyWidget::MyWidget(QWidget *parent)
    : QWidget(parent)
{
    edit1 = new QLineEdit(this);
    edit2 = new QLineEdit(this);
    this->resize(300,100);
    edit1->move(10,10);
    edit2->move(10,50);

    connect(edit1,SIGNAL(returnPressed()),this,SLOT(EnterPressedSlot()));
    connect(this,SIGNAL(specialStrSig(QString)),
                        this,SLOT(specialStrSlot(QString)));
```

```
}

void MyWidget::EnterPressedSlot()
{
    QString str = edit1 -> text();
    edit2 -> setText("输入为:" + str);
    if(str == "2020")
        emit specialStrSig(str);
}

void MyWidget::specialStrSlot(QString str)
{
    QMessageBox::information(this, "祝贺", "你找到了彩蛋:" + str);
}

MyWidget::~MyWidget()
{
}
```

在发送信号 specialStrSig(str)时,将字符串 str(值为 QString("2020"))作为参数进行了传递,它被传递给关联的 this 所指向对象(当前窗口)槽函数 specialStrSlot()的形参 str,并在消息框显示时作为内容的一部分进行显示。

程序运行效果同图 3-14。

4. 普通成员函数作为槽

在 Qt5 中还提供了 connect()函数的另外一种重载形式,格式如下。

```
QObject::connect(& 发送对象, & 类名::信号名,
                 & 接收对象, & 类型::槽名或普通成员函数名);
```

与 2.6.2 节介绍的 connect()函数方式相比,这种形式关联的槽不必用"slots:"声明(当然也可以如此声明),即普通成员函数也可以作为槽,并且这种形式在编译时会对信号和槽进行检查(要求槽和信号的参数至少能进行隐式转换),因此更建议使用这种形式的 connect()函数。

例如,在项目 3_6 中 MyWidget 类定义(mywidget.h)的公有部分加入以下普通成员函数声明。

```
void function(QString);
```

在类实现文件(mywidget.cpp)中添加其实现,代码如下。

```
void MyWidget::function(QString)
{
    QMessageBox::information(this, "^_^", "我是普通成员函数被调用");
}
```

然后在构造函数的函数体中添加 connect()函数调用。

```
connect(this,&MyWidget::specialStrSig,this,&MyWidget::function);
```

运行程序,并在上方文本框中输入 2020 后按 Enter 键,function()函数也会被执行,但由于消息框是模态对话框,在弹出一个消息框后单击 OK 按钮才会继续弹出另外一个消息框,如图 3-15 所示。

图 3-15　项目 3_6 添加代码之后的运行效果

3.5　Qt 中的界面

在使用向导新建项目的过程中勾选“创建界面”复选框后,生成的项目可以通过 Qt Designer 以一种鼠标拖曳、选择的方式组织自定义窗口中的子部件,设置它们的属性,以及进行一些信号与槽连接的操作,从而使快速开发成为可能。

所有通过 Qt Designer 工具进行的操作,最终都会形成一个界面文件(.ui 文件)。编译器就是根据这个文件生成相应的头文件,并将其包含到项目中,以实现自动化的界面设计。为了更好地理解界面文件在编译后生成的头文件,本节首先将介绍命名空间的概念,然后介绍如何通过 Qt Designer 快速实现界面设计。


视频讲解

3.5.1　命名空间

以中国的人名为例,虽然理论上可以由汉字任意排列组合而成,但人们总喜欢使用有意义的名字,因此即使在一个小范围内,也总会大概率地出现重名的情况。在程序中也存在同样的情形,不同项目协同开发人员编写的代码之间、项目代码与库文件(如 C++标准库、Qt 库)之间、库文件和库文件之间都可能出现重名的情况。对单独一位程序员而言,还有可能小心地起名以避免重名,但当多名程序员共同开发时,若要求使用的变量(或函数、类等)彼此之间不重名,难度很大。但如果不加以处理,名称之间就会出现被屏蔽或彼此冲突等情况。

对于派生类中成员函数的重定义,在调用基类的同原型成员函数时可以用“基类名::成员函数名(参数列表)”的形式来指明。这里“基类名::”就限定了成员函数名是这个基类范围内的成员函数,如此解决了重名(实际上这还是重定义成员函数所希望的)问题。类的成员天然地具有类这样一个范围域,而程序中定义的全局变量、函数、类等却不具有这样一个范围域,名字冲突又该怎么办呢?

提示:*名字冲突指在同一作用域中有两个或多个同名的实体。*

以现实生活中重名的问题举例，为了解决名字冲突，会给名字限定一个范围。例如，班级中有两位名叫“王伟”的同学，若上课时教师只点名“王伟”就会指代不清，引起名字冲突，此时如果能将两位王伟同学分属于不同的小组，如“一组”“二组”，那么在点名时就可以使用“一组的王伟”“二组的王伟”指明唯一的那位王伟同学，从而合理地解决了名字冲突问题。

在 C++中引入了类似的概念，称为命名空间，以解决名字冲突问题。命名空间是一个可以由用户自己定义（命名）的空间域。类似于上述的“小组”的概念，各个实体（函数、类等）分别被放在不同的命名空间中，以便和其他命名空间中的同名实体区分开。

在实际编程中，每位项目开发人员、每个库的开发人员都可以定义自己的命名空间域，把自己开发出的实体放在自己的命名空间域内（同一命名空间中要避免重名），这样即使有重名的情况，也因分属于不同的命名空间得以区分。

命名空间使用关键字 namespace 定义，格式如下。

```
namespace 命名空间名
{
    实体的声明和定义;
}
```

说明如下。

（1）花括号中的实体可以是变量、常量、函数、结构体、类、模板等。

（2）可以在项目中的多个地方定义命名空间，如果命名空间的名称相同，则会自动合并为一个命名空间，可以理解为追加。

（3）命名空间可以嵌套定义，花括号中还可以有命名空间的定义。

（4）命名空间定义在全局范围内，不可以定义在函数中。

为了使用带有命名空间的实体，需要在实体名前面加上命名空间的名称，并使用作用域解析符::限定，格式如下。

```
命名空间名::实体名
```

当命名空间嵌套时，依次使用命名空间名限制，即使用“外层命名空间名::内层命名空间名::实体名”的形式。如果在某一代码段范围内需要多次使用某命名空间中的实体，也可以使用 using 声明语句，格式如下。

```
using namespace 命名空间名;
```

using 和 namespace 都是关键字，using 声明语句的作用范围从声明位置开始，到 using 语句所在的作用域结束。在此范围内的代码，当使用该命名空间中的实体时，都无须添加“命名空间名::”限定。需要注意的是，在同一作用域内可能有多个 using 语句起作用，使用的（前面不加命名空间限制的）实体名不能在这些起作用的命名空间中重名。

项目 3_7 是一个纯 C++项目，代码如下。

```
/*****************************************
 * 项目名: 3_7
 * 说  明: 命名空间的使用
*****************************************/
#include <iostream>
using namespace std;

namespace MySpace1
{
void func()
{
    cout <<"function in namespace MySpace1."<< endl;
}
int i = 1;
namespace MyInterSpace
{
int i = 2;
}
}

namespace MySpace2
{
void func()
{
    cout <<"function in namespace MySpace2."<< endl;
}
}

namespace MySpace2
{
int i = 3;
int j = 4;
}


int main()
{
    cout << MySpace1::i <<"\t"<< MySpace1::MyInterSpace::i <<"\t"<< MySpace2::i << endl;
    MySpace1::func();
    MySpace2::func();
            //cout << i << j << endl;          //出错,未声明的标识符 i 和 j
            //func();                          //出错,未声明的函数
    using namespace MySpace1;
    cout << i << endl;
    func();
    using namespace MySpace2;
    cout << j << endl;
            //cout << i << endl;               //出错,i 是二义的
            //func();                          //出错,对 func()的调用是二义的
    return 0;
}
```

在该程序中有两个自定义命名空间 MySpace1 和 MySpace2，命名空间 MySpace1 中包括变量 i、函数 func 和一个命名空间 MyInterSpace；命名空间 MySpace2 进行了两次定义，第 1 次包括了一个函数 func()，第 2 次追加了变量 i 和变量 j；嵌套在命名空间 MySpace1 中的命名空间 MyInterSpace 中包括一个变量 i。

仔细观察会发现，其实程序一开始还声明了如下语句。

```
using namespace std;
```

即使用了系统标准命名空间 std，它是 cout 和 cin 等流对象所在的命名空间。该语句起作用的范围到本文件结束。如果在编程时不加此语句，则在使用 cin 或 cout 时需要写成 std::cin 和 std::cout 的形式。

在使用时，可以通过“命名空间名::”说明是哪个命名空间中的 i 变量或 func()函数，如主函数中的前 3 行所示。这时若直接使用变量 i 或 func()函数，系统会报错“未声明的变量 i 和函数 func”，见主函数第 4、5 行注释掉的代码。

语句“using namespace MySpace1;”的作用范围到主函数结束，因此接下来主函数第 7、8 行中用到的变量 i 和函数 func()均是 MySpace1 中的名字。然后第 9 行通过 using 语句使用了命名空间 MySpace2，之后的代码再使用变量 i 和函数 func()就无法区分是哪个命名空间中的名字了(见第 11、12 行，此区域中命名空间 std、MySpace1、MySpace2 均起作用)，系统会报错“变量 i 和函数 func 是二义的”。第 10 行使用的变量 j 由于只在 MySpace1 命名空间中有定义，并不冲突，因此可以正常输出。

图 3-16　项目 3_7 的运行结果

程序运行结果如图 3-16 所示。

3.5.2　快速实现界面设计

视频讲解

按照 3.4.1 节使用向导创建项目的步骤创建项目 3_8，注意勾选图 3-9 中的“创建界面”复选框，创建一个 Qt 窗口应用。在创建好后程序可以直接运行，实现显示一个窗口的效果。

观察项目构成可以发现，自定义窗口类的实现代码和之前有所不同，在项目中还多了一个以.ui 为扩展名的文件，它称为“界面文件”。该文件实际是一个 XML 格式的文件，用于记录用户使用 Qt Designer 工具设计了界面之后的结果。双击界面文件，可以打开 Qt Designer 工具，如图 3-17 所示。

左侧的部件面板以抽屉盒的形式列出了常见的可视化部件，拖动此处的部件到中间的窗口，即可在窗口中添加一个部件。右上方的对象浏览器窗口用树状视图显示了窗口中各部件之间的布局包含关系(因为还未添加子部件，这里只有一个 MyWidget 对象)。属性窗口在右下方，用于设置当前选中对象的属性值，如显示的文字以及部件的宽、高等，根据选中对象的不同，此处会显示不同的属性。在中间下方的信号与槽编辑器处可以对已有信号和槽进行连接。它的旁边还有一个动作编辑器(Action Editor)，用于设计动作(将在以后用到时再做介绍)。

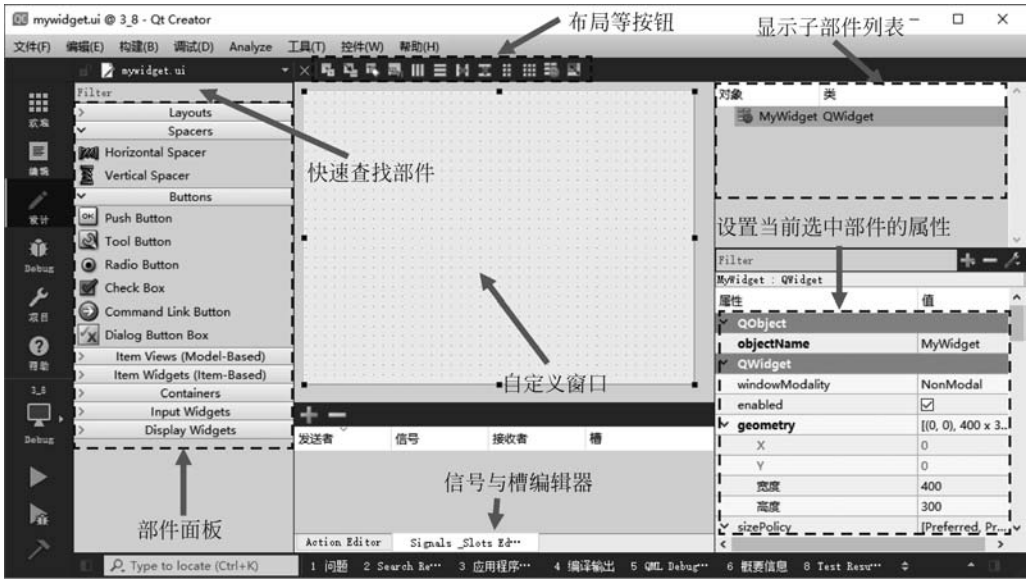


图 3-17　设计界面工具 Qt Designer 的主界面

读者可按以下步骤给项目 3_8 添加新的内容，初步熟悉 Qt Designer 的使用。

(1) 拖动左侧 Buttons 列表下的 Push Button 到窗口中，并通过拖动的方式设置它的大小和位置等，此时可见右上方的对象浏览器窗口的列表中多了一个 QPushButton 对象。

(2) 在窗口中选中刚才拖入的按钮，然后在右下方的属性窗口中找到 text 属性，在后面的输入框中输入 close window，可以看到按钮上的文字改变了。

(3) 单击信号与槽编辑器上方的绿色加号，添加一个信号与槽连接，发送者选择 pushButton，信号选择 clicked()，接收者选择 MyWidget，槽选择 close()。该步操作还可以按如下方式完成(作用是一样的)：单击图 3-18 中由虚线框起来的“编辑信号与槽”按钮，切换到“编辑信号/槽”模式，将按钮部件一直拖动到窗体边界，在弹出的“配置连接”窗口中首先勾选左下方的“显示从 QWidget 继承的信号和槽”复选框，然后分别为 pushButton 选中 clicked 信号，为 MyWidget 选中 close()槽，单击“确定”按钮。如果要切换回原窗口编辑界面，单击图 3-18 中虚线框左侧的“编辑窗体”按钮即可。

操作完成后的主界面如图 3-18 所示。此时运行即可显示一个窗口，在窗口中有一个显示文字为 close window 的按钮，单击它会关闭窗口。

上述操作没有手写任何一句代码，但已实现了简单的界面设计、属性设置和与用户的交互等功能。打开程序代码，读者会发现，除了界面文件，其他的程序代码和之前项目刚创建完时的代码没有任何变化，但运行效果确实不一样了，这是为什么呢？

在项目构建后，在构建目录下面会生成一个名为 ui_派生类类名.h 的头文件，它是系统根据界面文件自动生成的。在项目编译时，实际是这个头文件参与了项目的生成。回到自定义窗口类实现的源文件中可以看到，里面包含了此头文件。

提示：

(1) Qt Designer 主界面中添加子部件、设置属性的操作对应着界面文件内容的修改(也有操作对应着直接在源文件中添加代码)。

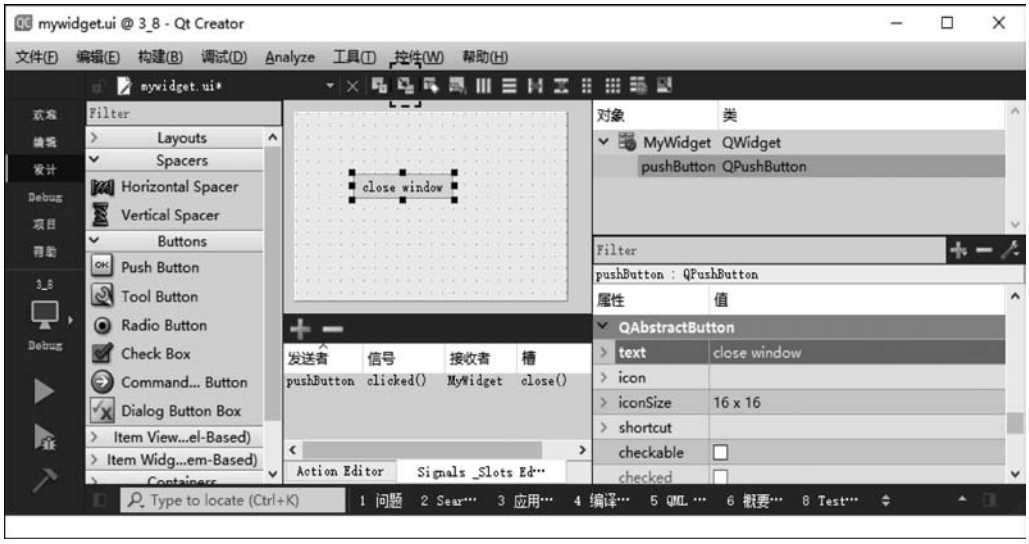


图 3-18　Qt Designer 中部件、属性及信号与槽的设置

(2) 界面文件是 XML 格式的文本文件，并不是源代码。它用于生成构建目录下名为 ui_派生类类名.h 的头文件。

头文件中的内容是有关如何生成界面的程序代码，用户无须也不要去修改此头文件(如需修改界面，在 Qt Designer 中直接修改即可)。当界面设计不同时，头文件中的内容也不同，但总体结构是一致的。

下面是按照前述步骤得到的界面文件生成的 ui_mywidget.h 头文件的代码。接下来对自动生成的该文件及项目中的程序代码进行分析，观察如何实现设置的功能。

```
/********************************************
 * 项目名: 3_8
 * 文件名: ui_mywidget.h
 * 说　明: 由系统根据界面文件自动生成的头文件
 ********************************************/
/*****************************************************************
** Form generated from reading UI file 'mywidget.ui'
**
** Created by: Qt User Interface Compiler version 5.12.4
**
** WARNING! All changes made in this file will be lost when recompiling UI file!
*****************************************************************/
#ifndef UI_MYWIDGET_H
#define UI_MYWIDGET_H

#include <QtCore/QVariant>
#include <QtWidgets/QApplication>
#include <QtWidgets/QPushButton>
```

```
#include <QtWidgets/QWidget>

QT_BEGIN_NAMESPACE

class Ui_MyWidget
{
public:
    QPushButton * pushButton;

    void setupUi(QWidget * MyWidget)
    {
        if (MyWidget->objectName().isEmpty())
            MyWidget->setObjectName(QString::fromUtf8("MyWidget"));
        MyWidget->resize(241, 157);
        pushButton = new QPushButton(MyWidget);
        pushButton->setObjectName(QString::fromUtf8("pushButton"));
        pushButton->setGeometry(QRect(50, 50, 91, 23));

        retranslateUi(MyWidget);
        QObject::connect(pushButton, SIGNAL(clicked()),MyWidget, SLOT(close()));
        QMetaObject::connectSlotsByName(MyWidget);
    } //setupUi

    void retranslateUi(QWidget * MyWidget)
    {
        MyWidget->setWindowTitle(QApplication::translate("MyWidget", "MyWidget", nullptr));
        pushButton->setText(QApplication::translate("MyWidget", "close window",nullptr));
    } //retranslateUi

};

namespace Ui {
    class MyWidget: public Ui_MyWidget{};
} //namespace Ui

QT_END_NAMESPACE

#endif //UI_MYWIDGET_H
```

可以看到,在该头文件中实现了一个 Ui_MyWidget 界面类,里面包含了一个 QpushButton 类型的指针、一个 setupUi()成员函数和一个 retranslateUi()成员函数。setupUi()成员函数在实现中依次设置窗口的对象名和大小、动态生成按钮对象、设置按钮对象名、设置按钮的大小、调用本类的成员函数 retranslateUi()(该函数实现动态翻译的功能)、连接按钮的 clicked 信号和窗口的 close()槽、设置通过槽名自动连接信号(将在 3.5.3 节介绍)。可见用户之前在 Qt Designer 中的操作最终都转换成这里的代码。

在头文件中还定义了一个命名空间 Ui,该命名空间中定义了一个 MyWidget 类(注意,

类名虽然和 setupUi()成员函数中的 MyWidget 形参同名，但它们一个是类名，一个是函数中的形参指针，它们是不同的实体，作用在不同的区域），MyWidget 类只是简单地继承了本头文件中的 Ui_MyWidget 界面类，并未添加新的成员。

QT_BEGIN_NAMESPACE 和 QT_END_NAMESPACE 是 Qt 中的宏，它们成对使用，作用是把上面的定义放到 Qt 的命名空间中。

自定义窗口类定义在 mywidget.h 头文件中，内容如下。

```
/*****************************************
 * 项目名：3_8
 * 文件名：mywidget.h
 * 说　明：由项目向导自动生成的自定义窗口类
*****************************************/
#ifndef MYWIDGET_H
#define MYWIDGET_H

#include <QWidget>

//前置声明，说明里面的标识符 MyWidget 是 Ui 命名空间中的类名
namespace Ui {
class MyWidget;
}

class MyWidget: public QWidget
{
    Q_OBJECT

public:
    explicit MyWidget(QWidget *parent = nullptr);
    ~MyWidget();

private:
    Ui::MyWidget *ui;
};

#endif //MYWIDGET_H
```

在自定义窗口类中因在声明私有指针成员 ui 时用到 Ui 命名空间中的 MyWidget 类，而编译器并不知道 Ui::MyWidget 里的 MyWidget 是什么，所以要在最开头做一个前置声明，告诉编译器它是 Ui 命名空间中的一个类。

接下来是自定义窗口类 MyWidget 的定义，这才是实际生成自定义窗口实例对象时使用的类。注意，它和 Ui::MyWidget 是两个不同的类（名字一样，但分属于不同的命名空间）。可以看到，该类中并没有与子部件有关的数据成员（或指针）的定义，所有与界面设置相关的工作都是通过 ui 指针指向的界面类（Ui::MyWidget）对象来完成的。

自定义窗口类 MyWidget 的实现文件 mywidget.cpp 的代码如下。

```
/*****************************************
 * 项目名: 3_8
 * 文件名: mywidget.cpp
 * 说  明: 由项目向导自动生成的自定义窗口类实现
 *****************************************/
#include "mywidget.h"
#include "ui_mywidget.h"

MyWidget::MyWidget(QWidget *parent):
    QWidget(parent),
    ui(new Ui::MyWidget)
{
    ui->setupUi(this);
}

MyWidget::~MyWidget()
{
    delete ui;
}
```

在构造函数中申请了动态的 Ui::MyWidget 类型的对象，并调用它的 setupUi()函数对界面进行了设置。主函数文件 main.cpp 由向导自动生成，其代码和项目 3_3 中的 main.cpp 完全相同，这里不再列出。

本例中对自定义窗口类 MyWidget 的设计实际采用了一种界面设计和其他业务逻辑实现相分离的设计方式，如图 3-19 所示。

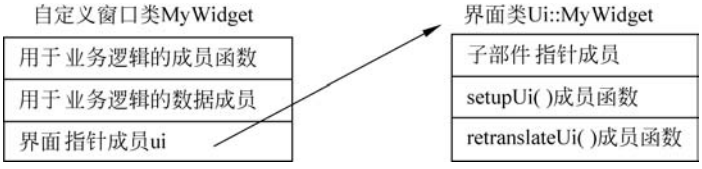


图 3-19　MyWidget 类在设计时业务实现和界面设计相分离

在 MyWidget 中，所有和界面相关的工作(添加子部件、设置部件属性等)都被分离出来，用一个 Ui::MyWidget 界面类型的对象实现，MyWidget 类中设置了一个指针成员 ui，指向在自定义窗口生成时动态生成的界面对象，并在构造函数中调用界面对象的 setupUi()成员函数完成和界面相关的工作。

这样做的好处如下。

(1) 避免了在头文件中暴露私有细节。在采用此种方式书写的类定义中看不到私有指针指向的界面类对象中的成员(如 pushButton 等子部件)。

(2) 分离了业务和界面。在项目 3_3 中，窗口中每添加一个子部件，就要在自定义窗口类中添加一个指针，那么当界面有改动时就要不停地修改自定义窗口类。而在此种设计方式下，和界面相关的操作都在界面类中。在项目由多人协同开发时，界面设计者可只关注于界面设计，程序员可(在界面的基础上)只关注于业务的逻辑实现。

提示：虽然项目 3_8 的代码都是自动生成的，但仍强烈建议读者仔细分析各个文件中代码间的逻辑关系，以便在后续项目较为复杂时仍能把握整个程序的结构，这对初学者来说是非常必要的。

3.5.3　信号与槽的自动关联


视频讲解

在 3.5.2 节的讲解中提到了通过槽名自动连接信号的连接设置操作，代码如下。

```
QMetaObject::connectSlotsByName(MyWidget);
```

此语句的功能是分别将所有 MyWidget 中以“on_发射者对象名_信号名”命名的槽与(名字中出现的)发射者的(名字中出现的)信号相连。这样对于多个如此命名的槽和信号，就不需要一次次地调用 connect()函数分别连接了。

下面通过一个例子说明信号与槽自动关联操作的实现。考虑实现以下功能：在界面上放置两个按钮，分别显示文字“确定”和“取消”，实现单击时分别弹出消息框，以提示用户单击了“确定”或“取消”按钮。

操作步骤如下。

(1) 使用项目向导创建一个基于 QWidget 的 Qt 项目 3_9(勾选“创建界面”复选框)。

(2) 在 Qt Designer 界面拖动两个 Push Button 按钮到窗口中，并在按钮上双击，分别修改显示的文字为“确定”“取消”(也可选中按钮后在属性窗口中设置)。

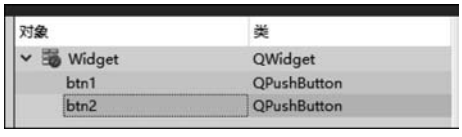


图 3-20　对象列表

(3) 此时右上角的对象列表中新增了两个 QPushButton 子部件，可双击子部件的对象名进行修改。如图 3-20 所示，将两个按钮的名字分别修改为 btn1 和 btn2(此步可忽略，使用系统默认的名字也可以)。

提示：

最好不要在对象的自关联槽函数创建之后再在对象浏览器窗口中修改对象名，否则会导致自动生成的代码之间不衔接。

作为实验，可以修改图 3-20 中的 Widget 窗口名，然后进行编译，查看各个源文件，观察什么地方改动了，什么地方的代码没有修改导致不能编译通过，需要修改哪些地方的代码才可以正常运行。

(4) 在自定义窗口的“确定”按钮上右击，在弹出的快捷菜单中选择“转到槽”命令，此时会弹出如图 3-21 所示的对话框，选择 clicked 信号。

(5) 单击 OK 按钮会自动转到代码文件。仔细观察，在类的实现文件中多了一个名为 on_btn1_clicked()的槽(如果在步骤(3)中未修改按钮对象名，这里的 btn1 会是系统默认的对象名)，只是实现体为空；切换到类的头文件，可以看到该私有槽的声明已经被添加到类的定义中。

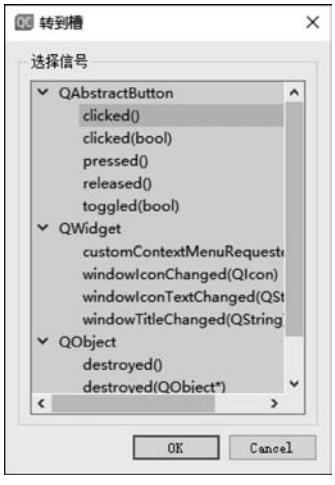


图 3-21　选择与槽相关联的信号

(6) 切换回类的实现文件,在开头处添加以下语句。

```
#include <QMessageBox>
```

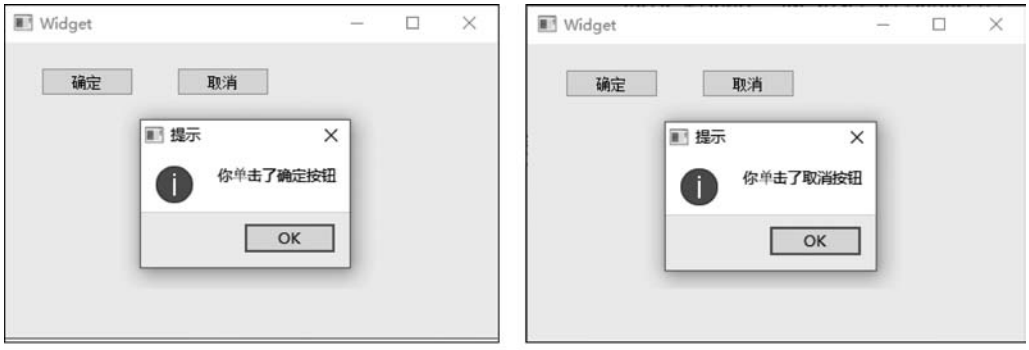
(7) 修改槽函数 on_btn1_clicked 的实现代码如下。

```
void Widget::on_btn1_clicked()
{
    QMessageBox::information(this, "提示", "你单击了确定按钮");
}
```

(8) 按照步骤(4)～步骤(7),为“取消”按钮的 clicked 信号也添加一个自关联槽,实现代码如下。

```
void Widget::on_btn2_clicked()
{
    QMessageBox::information(this, "提示", "你单击了取消按钮");
}
```

编译运行程序,运行效果如图 3-22 所示。



(a) 单击“确定”按钮时　　(b) 单击“取消”按钮时

图 3-22　项目 3_9 的运行效果

观察项目构建文件夹中生成的界面类头文件,其中并没有任何 connect 语句,项目中的两对信号与槽都是通过 connectSlotsByName()函数实现连接的。

视频讲解

3.6　Qt 常用部件

本节介绍 Qt Designer 中的部分常用部件,并给出部件的类名,读者可以直接使用这些部件,也可以在这些部件类的基础上派生出自己特有的部件类。

3.6.1　按钮部件

按钮部件为用户下达执行命令、选择某个功能等提供了交互操作。Qt 中的按钮部件如表 3-2 所示。

表 3-2 按钮部件

部件	名 称	作 用	类 名
	Push Button	命令按钮，通过按下按钮回答问题或命令计算机执行某些操作	QPushButton
	Tool Button	工具按钮，通常添加在工具栏内，为命令或选项提供快速访问	QToolButton
	Radio Button	单选按钮，可以选中或取消选中，通常是一组单选按钮互斥使用	QRadioButton
	Check Box	复选框，除选中或取消选中外，还有第 3 种可选的“无更改”状态，通常成组不互斥使用	QCheckButton
	Command Link Button	命令链接按钮，带有一个箭头图标，类似于平面按钮的外观，作用类似于单选按钮	QCommandLinkButton
	Dialog Button Box	按钮盒子，用于统一管理一组 QButton 按钮	QDialogButtonBox

命令按钮为矩形，通常在按钮上显示一个描述其功能的文本标签。典型的命令按钮常用于完成诸如“确定”“取消”“是”“否”“应用”“关闭”之类的操作。用户可以通过在文本标签中某字符的前面加上符号 & 指定快捷键，例如：

```
QPushButton * button = new QPushButton("O&K", this);
```

运行时，在按钮上并不显示符号 &，文本标签只显示为 OK。在按下 Alt 键时，可看到字符 K 下有一条横线；在按下 Alt+K 组合键时，其作用等同于单击此按钮。

如果要在文本标签中显示出字符 &，需要使用 && 表示一个可显示的 &。

提示：除了命令按钮以外，其他带有标签的单选按钮、复选框、命令链接按钮等都可以通过在文本标签上添加 & 以指定快捷键。

QToolButton 是与工具操作相关的按钮，通常和 QToolBar 工具栏搭配使用，对应着一个动作(QAction)，一般在创建 QAction 动作的同时创建。不同于命令按钮显示文本标签，工具按钮通常显示为图标(也可以有多种显示样式)。

单选按钮用于从多个选择中选取一个。属于同一个父窗口的单选按钮默认(自动独占 autoExclusive 属性为 True)是互斥的。在这种情况下，如果需要将它们分成多个单选按钮组，则每组按钮都要放入一个 GroupBox 容器(或 ButtonGroup)中。

复选框通常用于从多个选择中选取多个。当一组复选框的 autoExclusive 属性都设置为 True 时，也可以实现互斥的效果。

命令链接按钮一般不单独使用，通常用于替代向导和对话框中的单选按钮，完成选择某个功能并直接(代替了单击“下一步”按钮)进入下一步的操作。

按钮盒子中有一组按钮，可使用系统默认标准按钮，也可以自定义按钮。

下面通过一个例子熟悉这些按钮的使用。

(1) 使用项目向导创建一个基于 QMainWindow 带界面的 Qt 项目 3_10。

(2) 按照图 3-23 所示拖入各种部件，并通过在标签文本中添加字符 & 的形式为各个部件设置快捷键；然后运行程序，测试一下快捷键的使用和单选按钮及复选框的效果。读者还可以在属性窗口中修改每组中单选按钮、复选框的 autoExclusive 属性，以观察运行时的不同效果。

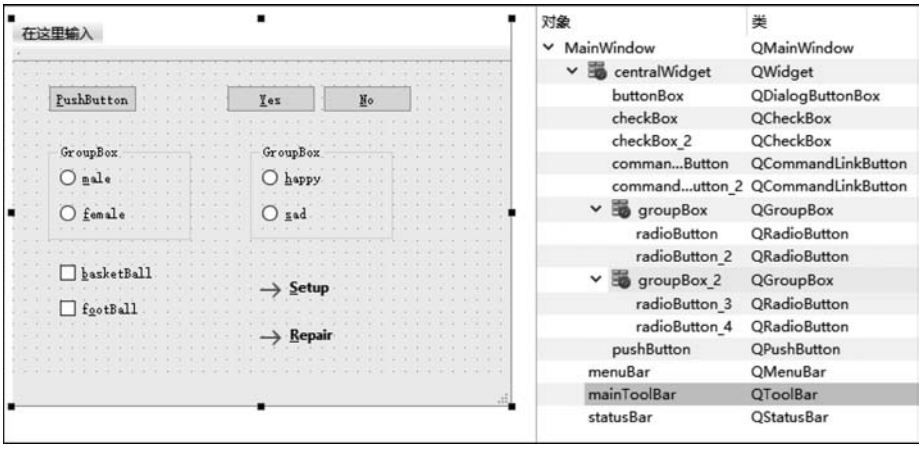


图 3-23　项目 3_10 的界面

(3) 选中按钮盒子,在属性窗口中找到 StandardButtons,单击以打开列表,为按钮盒子选中 Yes 和 No 两个按钮,可以看到图中原本的按钮换成了这两个系统标准按钮。

(4) 在按钮盒子上右击,在弹出的快捷菜单中选择"转到槽"命令,然后在弹出的对话框中选择 clicked(QAbstractButton *)信号,在转到的自关联槽 on_buttonBox_clicked(QAbstractButton * button)的函数体中添加以下实现代码。

```
if(button == ui -> buttonBox -> button(QDialogButtonBox::Yes))
{
    if(ui -> radioButton -> isChecked())
        QMessageBox::information(this,"提示","你选择了性别: 男");
    else if(ui -> radioButton_2 -> isChecked())
        QMessageBox::information(this,"提示","你选择了性别: 女");
    else
        QMessageBox::information(this,"提示","你未选择性别.");
}
```

在 mainwindow.h 中添加头文件:

```
#include <QAbstractButton>
```

在 mainwindow.cpp 中添加头文件:

```
#include <QMessageBox>
```

运行程序,效果为单击 Yes 按钮时会弹出消息框,给出用户选择的性别信息。

提示:在 MainWindow 类的成员函数内访问界面中部件对象的写法如下。

```
ui->部件名
```

ui 是 MainWindow 类中指向界面类对象的指针,部件名即为图 3-23 右边的对象浏览器窗口中列出的部件名,它们在界面类内以成员对象的形式存在。

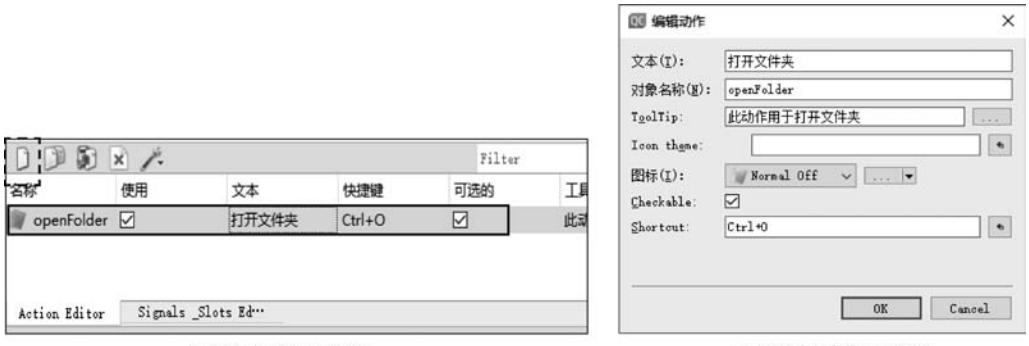
(5) 给项目添加资源文件和图标资源,以备后面使用。操作如下:在项目名处右击,在弹出的快捷菜单中选择 Add New 命令,然后在弹出的窗口中选择 Qt 中的 Qt Resource File

选项，如图 3-24 所示。接着按向导步骤为资源文件命名(本例中名为 res)，完成后可以看到项目中多了一个名为 res.qrc 的资源文件。

图 3-24　添加资源文件

在资源文件名上右击，在弹出的快捷菜单中选择“添加已有文件”命令，添加两个图标文件。为了方便管理，建议将图标文件放在项目文件夹下。资源数据通常被编译成一个应用程序库，可被压缩。

(6) 在动作编辑框(位于 Qt Designer 界面的中下方，见图 3-25(a))中单击“新建”按钮(虚线框处)，在打开的“编辑动作”对话框中按图 3-25(b)所示填入信息。在图标处为 Normal Off 和 Normal On 分别选择刚添加的图标资源；勾选 Checkable 复选框；在 Shortcut 后面的文本框中单击，然后直接按 Ctrl+O 组合键以填入快捷键。



(a) 单击“新建”按钮　　(b) “编辑动作”对话框

图 3-25　添加 QAction 动作

接着将图 3-25(a)中实线框所示的区域向上拖动到工具栏区域，直到显示出红色的短竖线时松开鼠标，此时工具栏上就添加了一个工具按钮。运行程序，观察选中和未选中该工具

按钮时图标的变化情况。

(7) 在图 3-25(a)中实线框所示的区域右击,在弹出的快捷菜单中选择“转到槽”命令,然后信号选择 triggered(当工具按钮被按下时会发出该信号),在转到的槽函数 on_openFolder_triggered()的函数体中添加语句:

```
QMessageBox::information(this,"提示","你单击了工具按钮");
```

运行程序,效果为单击工具按钮时会弹出消息框。

(8) 单击“编辑 Tab 顺序”按钮(图 3-26 中的虚线框所示)时会显示如图 3-26 所示的效果,数字大小代表 Tab 顺序的前后。依次单击每个部件,可以修改其 Tab 顺序。

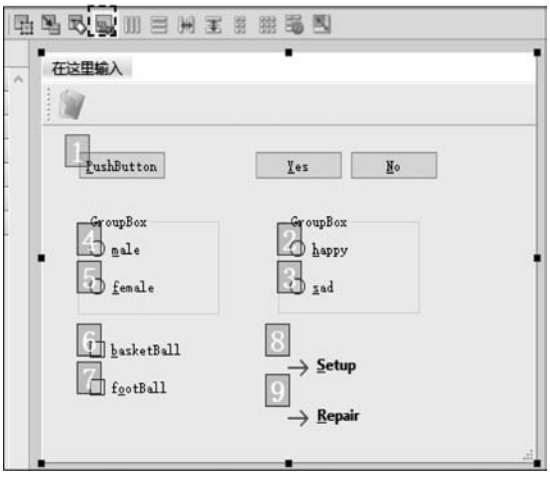


图 3-26 Tab 顺序

提示:Tab 顺序指在运行时按下键盘上的 Tab 键,各部件依次获得焦点(指当前处于被选中状态)的顺序。

3.6.2 输入部件

输入部件为用户输入信息、设置数据提供支持。Qt 提供的输入部件如表 3-3 所示。

表 3-3 输入部件

部件	名 称	作 用	类 名
	Combo Box	组合框,提供一组下拉选项	QComboBox
	Font Combo Box	字体组合框,下拉选项为系统已有的字体列表	QFontComboBox
	Line Edit	单行文本框,用于输入和编辑一行纯文本信息	QLineEdit
	Text Edit	多行文本框,可以输入多行内容,支持使用 HTML 样式的富文本信息	QTextEdit
	Plain Text Edit	纯文本编辑框,可以输入多行信息,只支持纯文本信息	QPlainTextEdit

续表

部件	名　　称	作　　用	类　　名
	Spin Box	数字选择框，用于设置一个整数值	QSpinBox
	Double Spin Box	浮点数旋转框，用于设置一个浮点数值	QDoubleSpinBox
	Time Edit	时间编辑框，用于编辑时间	QTimeEdit
	Date Edit	日期编辑框，用于编辑日期	QDateEdit
	Date/Time Edit	日期/时间编辑框，是 QTimeEdit 和 QDateEdit 的父类，用于编辑日期和时间	QTimeDateEdit
	Dial	转盘，用来设置一定范围内的整数值	QDial
	Horizontal Scroll Bar	水平滚动条，当工作界面不能完全显示时，可使用滚动条调节显示的内容	QScrollBar
	Vertical Scroll Bar	垂直滚动条	
	Horizontal Slider	水平滑动部件，用于设定一个整数值	QSlider
	vertical Slider	垂直滑动部件，用于设定一个整数值	
	Key Sequence Edit	按键序列编辑框，用于记录用户的输入按键序列	QKeySequenceEdit

组合框是按钮和弹出列表的组合，提供了一种以占用最少屏幕空间的方式向用户显示选项列表的方法。设计时，用户可在组合框部件上双击，以添加、删除选项及修改选项的顺序；在运行时，如果当前选项发生更改（通过编程或用户交互实现），会发出 currentIndexChanged 信号；如果是由用户交互动作引起的选项更改，还会发出 activated 信号；组合框可设置为可编辑的（editable 属性为 True），这样运行时允许用户修改列表中的每个项目。

字体组合框用在工具栏中，通常和用于控制字体大小的组合框以及用于粗体、斜体的工具按钮结合使用。在运行时，若选择新字体会发出 currentIndexChanged 和 currentFontChanged 信号。

单行文本框通过设置可以为“只读”或“只写”（用于输入密码）模式；可以限制其最大输入长度、控制输入文本的格式；它支持剪切、粘贴、撤销、重做、拖放等编辑功能。

多行文本框经过了优化，可处理大型文档并快速响应用户的输入。它支持纯文本和富文本，适用于段落，还可以显示图像、列表和表格等。如果文本内容太多，当超出可显示的行数时会出现滚动条。若只需要显示一小段富文本，也可使用 QLabel。

纯文本编辑框和多行文本框的功能类似，但只支持纯文本（不支持富文本显示），并针对纯文本处理进行了优化。

对于数字选择框，用户通过单击上/下按钮或按键盘上的上/下方向键增加/减少当前设定的整数值，也可以由用户手动输入。用户可以给它们设定步长（指每次单击上/下按钮时数值增减的大小）、最大值、最小值等属性。在每次值更改时，都会发出两个 valueChanged 信号，一个信号提供数值，另一个信号提供 QString 类型的字符串。

浮点数旋转框支持浮点数，其他与数字选择框相同。

日期/时间编辑框是用于编辑日期和时间的部件。用户可通过键盘或箭头按钮以增减日期和时间值的形式编辑日期和时间；可以设置最大/最小日期、最大/最小时间等；可以设置显示格式；可以通过 calendarPopup 属性设置是否启用日历弹出窗口。

日期/时间编辑框派生出日期编辑框和时间编辑框，分别用于日期和时间的编辑。

转盘部件通常用于一个可以环绕的范围内值的设置，如角度范围(0°～360°)等，可设置最大/最小值、步长等属性。在转动转盘时，会连续发出 valueChanged 信号。

当在一个有限的窗口大小中显示很多数据时通常用到滚动条，它包括滑块、滚动箭头和页面部件等组成部分。滑块可提供快速定位；滚动箭头是按钮，每单击一次增/减步长个数值；页面部件是滚动条中拖动滑块的区域(滚动条的背景)，单击此处将滚动条增/减一个“页面”的数值。其方向可以通过 orientation 属性设置。

滑块实现和转盘部件类似的功能，只是以长条的形式显示。通过 orientation 属性可设置其为水平或垂直摆放。

按键序列编辑框通常用于接收用户设置的快捷键。在运行中选中该部件时开始记录用户的按键行为，并显示在编辑框中，直到用户释放最后一个键 1 秒后结束。例如，在图 3-25(b)中 Shortcut 后面的编辑框就是一个按键序列编辑框。

提示：各个部件类提供了更多属性和行为，请参考 Qt Creator 中的帮助。

下面通过一个例子熟悉这些输入部件的使用。

(1) 使用项目向导创建一个基于 QWidget 带界面的 Qt 项目 3_11。

(2) 按照图 3-27 所示拖动各个部件到窗口中。双击组合框部件，在打开的窗口中添加多条项目；选中字体组合框，在对象属性窗口中设置默认字体为隶书(currentFont 属性)；双击文本框部件，在打开的窗口中添加内容；分别选中数字选择框、浮点数旋转框、转盘、滚动条、滑动部件，在对象属性窗口中设置 value 属性为 40；设置浮点数旋转框保留一位小数(decimals 属性为 1)；对于日期/时间编辑框，在对象属性窗口中设置默认显示的日期和时间(date、time 属性)、格式(displayFormat 属性)等。

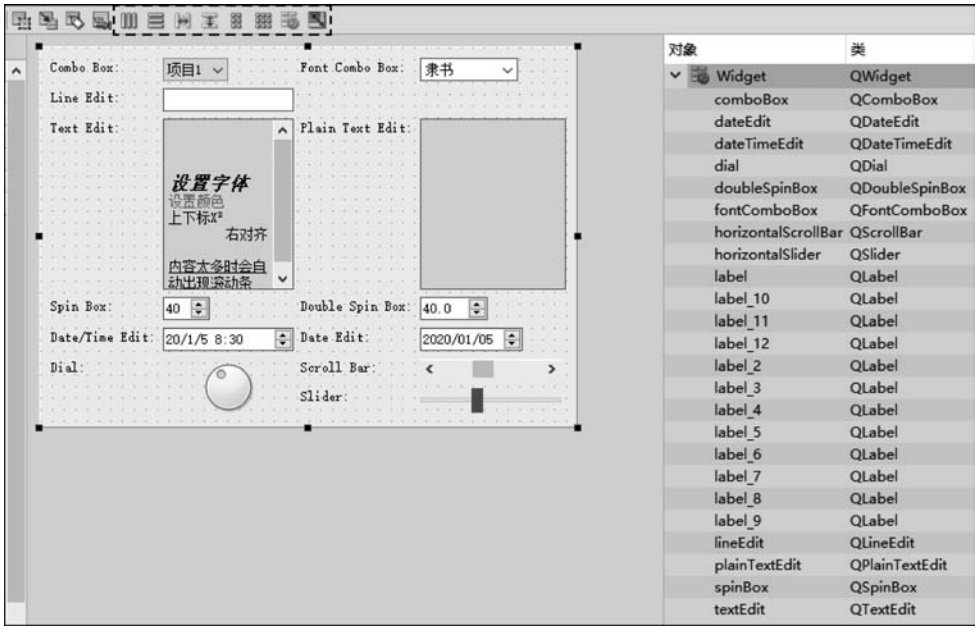


图 3-27　项目 3_11 的界面

(3) 窗体上的所有部件，在尺寸合适、排列整齐时界面看起来才美观。Qt 提供了一些简单且有效的方式用于自动排列窗口子部件的布局。在图 3-27 中，虚线框中的按钮从左到

右分别为水平布局、垂直布局、使用分裂器水平布局、使用分裂器垂直布局、在窗体布局中布局、栅格布局、打破布局、调整大小。读者可以在窗体中选中整个窗体(或若干部件),然后应用以上布局,以观察不同的布局效果。

提示:

① Qt 中的布局是通过布局类完成的。当应用布局到整个窗口时,窗口设置为具有布局的窗口;当选中若干个部件并应用布局时,会生成一个布局对象,读者可以在对象浏览器窗口中观察布局对象的生成情况。

② 在 Qt Designer 的部件面板中还有 Layout 和 Spacers 两组面板,其中的部件所起的功能与这些布局按钮类似,但可以进行更加详细的布局设计。

(4) 在自定义窗口类 Widget 构造函数(在 widget.cpp 文件中)的函数体结尾添加语句:

```
ui->plainTextEdit->setPlainText(ui->textEdit->toPlainText());
```

运行程序,可见纯文本编辑框中显示了和多行文本框中相同的文字,但没有格式,如图 3-28 所示。

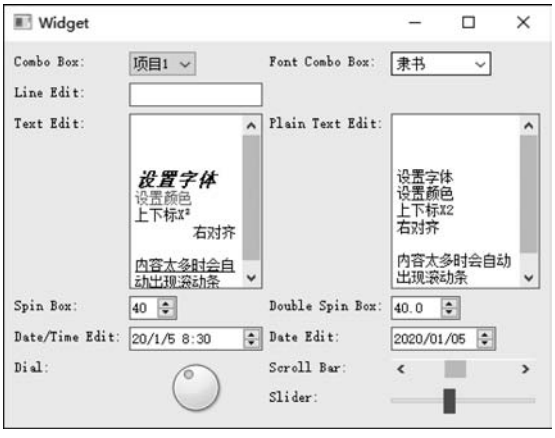


图 3-28　项目 3_11 中纯文本编辑框的内容显示

(5) 在转盘部件上右击,添加一个槽,信号选择 valueChanged(int),在打开的槽函数 on_dial_valueChanged(int value)的函数体中添加语句:

```
ui->horizontalSlider->setValue(value);
ui->horizontalScrollBar->setValue(value);
ui->spinBox->setValue(value);
ui->doubleSpinBox->setValue(value);
ui->lineEdit->setText(QString::number(value));
```

运行程序,可以看到在转动转盘时滚动条和滑动部件也随之变化,数字选择框中的数字也会随之改变。形参 value 的值由信号发射时传递而来,在代码中"QString::number(value)"是调用了 QString 类中的静态成员函数将整型的 value 值转换为对应的 QString 类型的字符串(有关静态成员的概念请参考第 4 章)。

(6) 为日期编辑框添加一个自关联槽,信号选择 userDateChanged(QDate),在槽函数 on_dateEdit_userDateChanged(const QDate &date)的函数体中添加语句:

```
ui->dateTimeEdit->setDate(date);
```

运行程序，并修改日期编辑框中的日期，可以看到日期/时间编辑框中的日期也随之改变。

(7) 给字体组合框添加一个自关联槽，信号选择 currentFontChanged(QFont)，在槽函数 on_fontComboBox_currentFontChanged(const QFont &f)的函数体中添加语句：

```
ui->lineEdit->setFont(f);
```

运行程序，并更改字体组合框中选择的字体，可见单行文本框中文本的字体改变了。

(8) 将显示为“Line Edit”的标签文字改为“&Line Edit”，然后和后面的单行文本框设定伙伴关系，操作如下：单击图 3-29 虚线框中的“编辑伙伴”按钮以切换到“编辑伙伴”模式，将标签拖动到右边的单行文本框处，当显示出如图 3-29 所示的箭头时松开。如果要切换到窗口编辑模式，单击位于图 3-29 左上角的“编辑窗体”按钮即可。

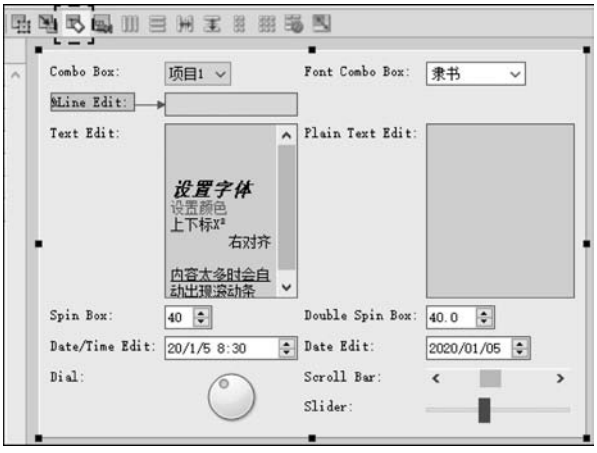


图 3-29　设定伙伴关系

运行程序，按 Alt+L 组合键可以发现单行文本框获得了焦点，这就是伙伴关系的作用。

提示：“伙伴”是标签中的一个概念。因为标签经常被作为一个交互式部件的说明，所以设置了伙伴关系，使得能通过在标签上设置的快捷键将焦点快速定位到对应部件(它的伙伴)上，从而达到方便索引的目的。

3.6.3　显示部件

显示部件用于向用户展示各种信息。常用的显示部件如表 3-4 所示。

表 3-4　显示部件

部件	名　称	作　用	类　名
	Label	标签，用于显示文字或图片信息	QLabel
	Text Browser	文本浏览器，类似于 Text Edit，但无法编辑，只用于显示	QTextBrowser
	Graphics View	视图窗口部件，用于可视化 QGraphicsScene (二维图形项目构成的场景)的内容	QGraphicsView
	Calendar Widget	日历部件，以月的形式显示日历，提供用户选择日期的功能	QCalendarWidget

续表

部件	名　　称	作　　用	类　　名
	LCD Number	液晶数字部件，显示带有类似 LCD 效果的数字，可以是十进制、十六进制、八进制或二进制等	QLCDNumber
	Progress Bar	进度条，用于向用户指示操作的进度	QProgressBar
	Horizontal Line	水平线，提供一条水平线段	QLine
	Vertical Line	垂直线	
	OpenGL Widget	OpenGL 窗口，提供了用于显示集成到 Qt 应用程序中的 OpenGL 图形的功能	QOpenGLWidget
	Quick Widget	Quick 窗口，用于显示 Qt Quick 用户界面。当提供主源文件的 URL 时，它将自动加载并显示 QML 场景	QQuickWidget

标签支持富文本，还可以显示图像，但没有提供用户交互的功能。

文本浏览器支持富文本，但只提供了浏览功能，不能修改。它实际上是 QTextEdit 类的派生类，相当于只读版本的 QTextEdit，并添加了一些导航功能，以便用户可以跟踪超文本文档中的链接。

日历部件默认使用当前的日期进行了初始化，也提供了一些公有槽函数用于更改显示的日期。用户还可以使用鼠标和键盘选择日期，将 selectionMode 属性设置为 NoSelection 可以禁止用户的选择功能。

液晶数字部件可以显示很大范围内的数字，当显示的内容超出范围时会发出 overflow 信号。

进度条通常用于安装程序的等待过程、复制文件的等待过程等，通过进度指示告知用户操作的进展程度，可以给进度条设置最大和最小步数以及步长等。

线部件可以设置线宽、起始点等属性。

下面通过一个例子熟悉部分显示部件的使用。

(1) 使用项目向导创建一个基于 QWidget 带界面的 Qt 项目 3_12。

(2) 按照项目 3_10 中步骤(5)的方式给该项目添加一个资源文件，并分别添加一个 GIF 格式的动画图片资源和一个 JPG 格式的静态图片资源。

(3) 拖动两个标签到窗口中，并将它们都拉大一些以便能容纳图片，勾选它们的 scaledContents 属性以便图片能自动缩放。对其中的一个标签设置 pixmap 属性为步骤(2)中刚添加的静态图片资源，运行程序可见图片显示在标签中。

(4) GIF 图片的显示需要编写代码实现，在自定义窗口类 Widget 构造函数的函数体末尾(在 widget.cpp 文件中)添加语句：

```
QMovie * mv = new QMovie(":/res/a.gif");
ui -> label -> setMovie(mv);
mv -> start();
```

代码中第 1 句为使用资源创建一个 QMovie 对象，注意资源路径的写法，":/"代表使用本项目中的资源(也可以使用绝对路径资源，如"C:/a.gif"，但不建议使用绝对路径)，res 是

资源的前缀(它类似于一个虚拟的,从功能逻辑上区分的子文件夹的作用)。本例实现时,资源是放在项目目录下的 res 子文件夹中的,因此添加了资源后自动为资源添加了一个 res 前缀,如图 3-30 所示。在书写资源路径时,如果有前缀需要添加上完整的前缀。类似于子文件夹,前缀也可以有多级,在书写时用"/"隔开。

图 3-30　资源前缀

代码中第 2 句设置名为 label 的标签用于显示动态图,读者可以根据情况将其替换为自己操作时实际的标签对象名。

(5) 添加一个文本浏览器,并双击该对象打开内容编辑窗口,给内容添加一个超链接,标题为"上海电力大学",URL 为 http://www.shiep.edu.cn,将文本浏览器的 openExternalLinks 属性设置为勾选。运行程序,在文本浏览器的超链接上单击,会自动调用系统浏览器打开上海电力大学网站。

(6) 拖动一个日历部件到窗体,并添加一个自关联槽,信号选 clicked(QDate),在槽函数 on_calendarWidget_clicked(const QDate &date)的函数体中添加语句:

```
ui->textBrowser->append(date.toString());
```

运行程序,则每单击一次日历部件上的某个日期,就会在文本浏览器中添加一行该日期的文本。

(7) 添加液晶数字部件,设置 value 属性值为 40.5,并观察勾选和不勾选 smallDecimalPoint 属性(小数点)时效果的不同;添加一个水平线条,并设置线宽 lineWidth 属性为 10,观察显示效果;添加一个进度条部件和一个滑动部件;给滑动部件设置 maximum 属性为 100;给滑动部件添加一个自关联槽,信号选 valueChanged(int),在槽函数 on_horizontalSlider_valueChanged(int value)的函数体中添加语句:

```
ui->progressBar->setValue(value);
```

运行程序,观察拖动滑动部件的滑块时进度条的显示效果,如图 3-31 所示。

图 3-31　项目 3_12 的运行效果

3.7 编程实例——计算器

本节将实现一个能进行实数间加、减、乘、除运算的简易计算器。首先创建一个基于QWidget的带界面的Qt项目3_13，然后按照以下步骤进行操作。

1. 计算器界面设计

在界面中拖入两个单行文本框和17个按钮，按钮上显示的文字、按钮对象和单行文本框对象名如图3-32所示。为了美观，设置窗口为“栅格布局”以对齐部件(操作参考3.6.2节)。

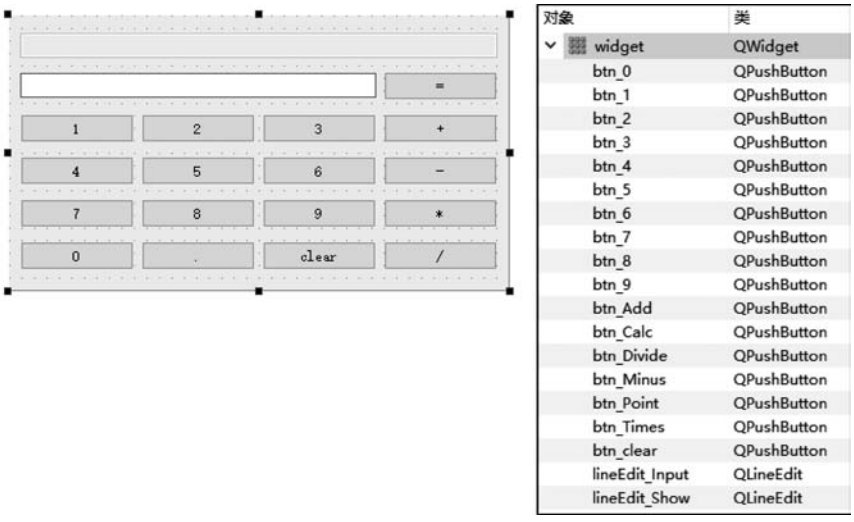


图3-32 计算器界面设计

将窗口对象的windowTitle属性设置为“计算器”；取消勾选最上方第1个单行文本框(lineEdit_Show对象，仅用于显示结果)的enable属性，使该单行文本框变为灰色；勾选第2个单行文本框(lineEdit_Input对象)的readOnly属性(限制用户不能直接在文本框中通过键盘输入内容)，将其alignment属性设置为AlignRight。

2. 计算器功能的实现

在进行算术运算时，须存储之前输入的左操作数和操作符(右操作数可直接从文本框lineEdit_Input中读取)，因此在自定义窗口类Widget中添加私有数据成员如下。

```
QString operandStr1;                    //用于存储字符串形式的左操作数
QString operatorStr;                    //用于存储操作符
```

并在Widget类的构造函数体中添加以下语句，将它们初始化为空串。

```
operandStr1 = "";
operatorStr = "";
```

接下来实现单击各个按钮时触发的功能：给每个按钮的clicked信号都添加自关联槽。以按钮1为例，自关联槽的实现代码如下。

```
void Widget::on_btn_1_clicked()
{
    ui->lineEdit_Input->setText(ui->lineEdit_Input->text() + "1");
}
```

按钮 2～按钮 9 的功能实现和按钮 1 是类似的，只需把上述代码中的字符"1"改成对应的数字字符即可。对于按钮 0，由于一般不会出现诸如 00 形式的数字 0，所以代码中对这种情况进行了处理，自关联槽定义如下。

```
void Widget::on_btn_0_clicked()
{
    if(ui->lineEdit_Input->text()!= "0")
        ui->lineEdit_Input->setText(ui->lineEdit_Input->text() + "0");
}
```

小数点按钮需要考虑按下时前面没有数字的情形(此时默认为整数部分为 0)、按下时数字串中已有了小数点的情形，最终自关联槽定义如下。

```
void Widget::on_btn_Point_clicked()
{
    if(ui->lineEdit_Input->text() == "")
        ui->lineEdit_Input->setText("0.");
    else if(ui->lineEdit_Input->text().contains(".") == true)
        ;                                    //数字串中已有小数点,不能再输入
    else
        ui->lineEdit_Input->setText(ui->lineEdit_Input->text() + ".");
}
```

单击 clear 按钮时，只需将文本框 lineEdit_Input 中的内容清空即可，代码如下。

```
void Widget::on_btn_clear_clicked()
{
    ui->lineEdit_Input->clear();
}
```

加、减、乘、除按钮的实现是类似的。以加法按钮为例，分情况进行处理。如果按下时文本框 lineEdit_Input 中是空串，则不进行任何处理直接结束；否则说明用户提供了一个操作数，接下来判断它是左操作数还是右操作数。若 operandStr1 为空，说明文本框中是左操作数，则将其存储到 operandStr1，将＋运算符存储到 operatorStr，将文本框清空以待用户再次输入右操作数，将已输入的内容显示于文本框 lineEdit_Show 中；若 operandStr1 不为空，说明文本框中已是右操作数，此时按下加法按钮和按下等号按钮的作用是相同的，直接调用单击等号按钮关联的槽函数 on_btn_Calc_clicked()进行处理即可。其实现代码如下。

```
void Widget::on_btn_Add_clicked()
{
    if(ui->lineEdit_Input->text() == "")  //没有输入数据
        return;
    else if(operandStr1 == "")            //没有左操作数
    {
        operandStr1 = ui->lineEdit_Input->text();
```

```
        operatorStr = " + ";
        ui -> lineEdit_Input -> clear();      //输入文本框清空,以待输入右操作数
        ui -> lineEdit_Show -> setText(operandStr1 + operatorStr);
    }
    else
        on_btn_Calc_clicked();
}
```

其他 3 个运算符的实现是一样的,只需将上述代码中赋值给 operatorStr 的字符串改成相应的运算符即可。

最后是等号按钮的实现,代码如下。

```
void Widget::on_btn_Calc_clicked()
{
   if(operandStr1!= ""&&ui -> lineEdit_Input -> text()!= ""&&operatorStr!= "")
   {
        double result;
        double operand1 = operandStr1.toDouble();
        double operand2 = ui -> lineEdit_Input -> text().toDouble();
        if(operatorStr == " + ")
            result = operand1 + operand2;
        else if(operatorStr == " - ")
            result = operand1 - operand2;
        else if(operatorStr == " * ")
            result = operand1 * operand2;
        else if(operatorStr == "/")
            if(operand2!= 0.0)
                result = operand1/operand2;
            else
            {
                QMessageBox::warning(this,"提示","除数不能为零");
                result = 0;
            }
        ui -> lineEdit_Show -> setText(QString::number(result));
        operandStr1 = "";                     //计算完毕,操作数清空
        operatorStr = "";                     //计算完毕,操作符清空
        ui -> lineEdit_Input -> clear();      //计算完毕,数据输入文本框清空
    }
}
```

首先判断左、右操作数和运算符是否都存在,然后将 QString 字符串形式的操作数转换为 double 类型的操作数(由 QString 类的成员函数 toDouble()实现);再根据运算符的不同分别进行不同的计算,结果放在 result 中;对于除法,代码中添加了对除数为 0 时的出错处理;最后将算出的结果显示在文本框 lineEdit_Show 中(QString 的静态成员函数 number 用于将给定的参数转换为字符串形式),并清空相关数据以待下一次计算。

上述代码中使用到 QMessageBox 类,因此还需在 widget.cpp 文件中添加头文件:

```
#include <QMessageBox>
```

到此功能已全部实现,运行程序可查看效果。图 3-33 所示为按下按钮 2、3、+之后再按下按钮 5、.、1、=的效果。

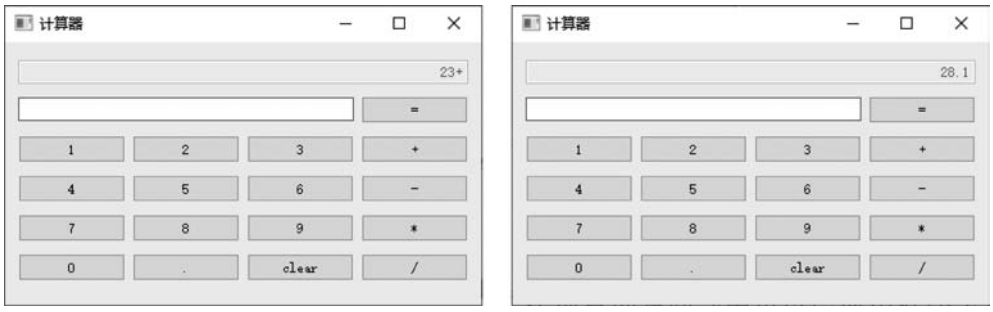


图 3-33　计算器运行效果示例(23+5.1)

3. 登录界面设计

接下来考虑在上述已实现计算器的基础上添加登录的功能：程序运行时首先显示一个登录界面，只有当输入了正确的用户名和密码后才能打开计算器。

实际上，除了可以在项目创建向导中给应用程序主窗口选择使用界面外，在项目中新增自定义 C++窗口(或部件)类时也可以使用界面。接下来在项目中添加一个带界面的登录对话框类，操作步骤如下。

在项目名处右击，在弹出的快捷菜单中选择 Add New 命令，打开如图 3-34 所示的界面。选择 Qt 下的“Qt 设计师界面类”选项，以创建一个 Qt 设计师界面类。

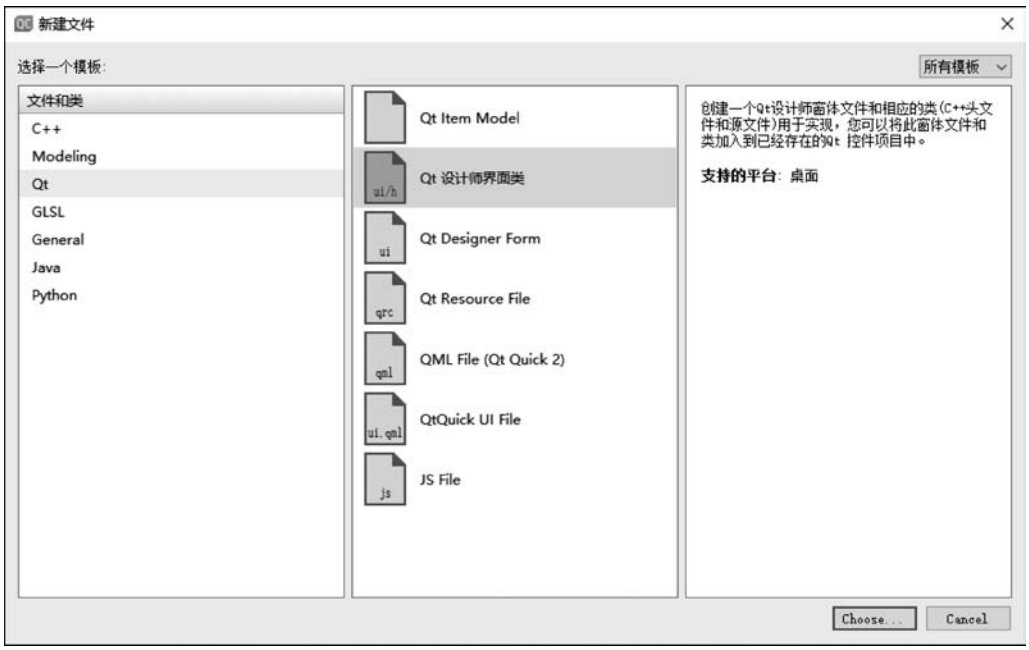


图 3-34　选择“Qt 设计师界面类”选项

单击图 3-34 中的 Choose 按钮后进入选择界面模板的步骤，如图 3-35 所示。所谓选择界面模板，是指准备在哪种窗口(或部件)的基础上进行更多的设计(即选择窗口或部件类作为自定义窗口类的基类)，本例中选择 Dialog without Buttons 选项，即使用 QDialog 类作为基类。

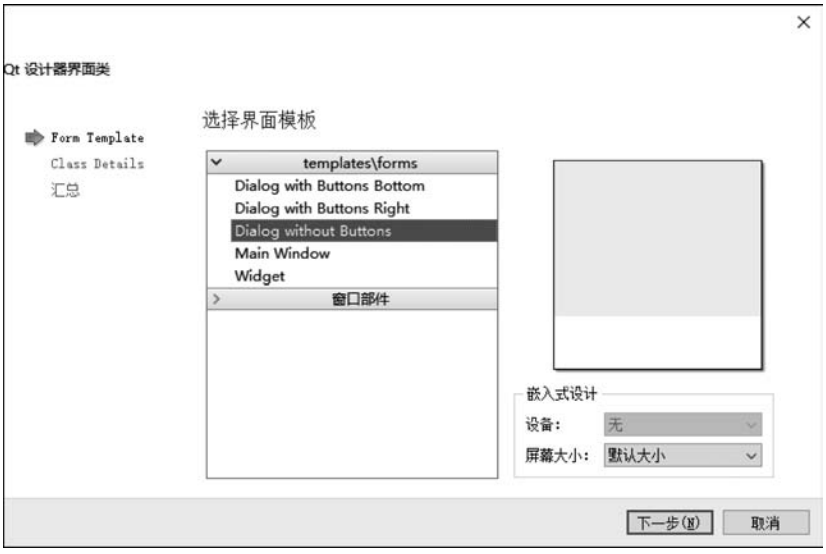


图 3-35　选择界面模板

提示：此处界面模板的含义与 C++模板的概念不同（有关 C++模板，请参考 5.1 节）。

单击“下一步”按钮，进入图 3-36 所示的界面。在“类名”文本框中输入自定义登录对话框类的名字(本例使用 LoginDialog)，下方会自动生成相关的头文件、源文件和界面文件的名字，读者也可以修改这些文件的文件名(但不建议修改)，默认以当前工程目录为存放路径。

图 3-36　设置自定义登录对话框类的类名

再次单击“下一步”按钮，完成带界面自定义登录对话框类的初始创建。可以看到，工程中已新增了 3 个与该类有关的文件。

双击 logindialog.ui 文件打开界面设计师。将对话框窗口标题(windowTitle 属性)设置为“登录”；然后向窗口中拖入两个标签、一个按钮和两个单行文本框；标签和按钮显示的文字如图 3-37 所示，各部件的名字见图中的对象浏览器窗口。

图 3-37　登录对话框界面设计

设置标签为右对齐(alignment 属性值为 AlignRight)；设置密码文本框的 echoMode 为 Password,使输入数据时显示为表示密码的黑色圆点；将窗口设置为“栅格布局”,以方便地对齐各个部件；为了方便用户使用,还可指定部件的 Tab 键顺序(参考 3.6.1 节)、设置标签的快捷键以及和单行文本框的伙伴关系(参考 3.6.2 节)。

4. 登录功能的实现

首先给登录对话框类添加一个信号,在类定义中(logindialog.h 文件)添加代码：

```
signals:
    void LoggedIn();
```

然后给“登录”按钮的 clicked 信号添加自关联槽,代码实现如下。

```
void LoginDialog::on_loginBtn_clicked()
{
    if(ui->userEdit->text() == "admin"&&ui->pwdEdit->text() == "123456")
    {
        emit(LoggedIn());
        hide();                                    //隐藏登录窗口
    }
    else
        QMessageBox::information(this, "提示", "用户名和密码错误");
}
```

当输入了正确的用户名和密码时,将发射 LoggedIn 信号(目的是通知计算器窗口把自己显示出来,信号槽关联在随后的主函数中),然后将登录对话框隐藏,否则会提示用户名和密码出错。

在 logindialog.cpp 文件中还需添加头文件：

```
#include <QMessageBox>
```

为了实现先显示登录界面,成功后再显示计算器界面的操作,以及将登录对话框发射的 LoggedIn 信号和计算器窗口的显示槽函数 show()进行关联,主函数也需要进行修改,代码如下。

```
/*****************************************
 * 项目名: 3_13
 * 文件名: main.cpp
 * 说　明: 主函数的实现
 *****************************************/
```

```
#include "widget.h"
#include <QApplication>
#include "logindialog.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    Widget w;
    LoginDialog login(&w);
    login.show();
    QObject::connect(&login,SIGNAL(LoggedIn()),&w,SLOT(show()));

    return a.exec();
}
```

运行时,首先会显示出图 3-38 所示的登录界面。在输入正确的用户名和密码并单击“登录”按钮后才打开计算器界面。

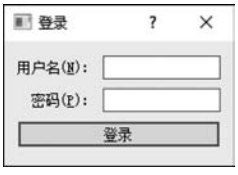


图 3-38 登录对话框的运行效果

课后习题

一、选择题

1. 关于基类和派生类,下列说法中错误的是(　　)。
   A. 派生类是在基类的基础上定义的新类
   B. 派生类至少有一个基类
   C. 派生类成员可以直接访问继承自基类的所有成员
   D. 一个派生类可以做另一个派生类的基类
2. 在派生类中重定义成员函数后,下列说法中正确的是(　　)。
   A. 原继承自基类的成员函数不再存在
   B. 原继承自基类的成员函数无法再被调用
   C. 原继承自基类的成员函数仍然可以按照以往常规的方式被调用
   D. 重定义成员函数的原型必须和原继承自基类的成员函数的原型相同
3. 下列关于继承和派生中的赋值规则错误的是(　　)。
   A. 派生类对象可以初始化基类的引用

B. 基类的对象可以赋值给派生类的对象

C. 派生类的对象的地址可以赋值给指向基类的指针变量

D. 在需要基类对象的任何地方都可以使用公有派生类的对象代替

4. 设 ClassA 类是 ClassB 类的派生类，则定义 ClassA 类的对象时和该对象的空间被回收时调用构造函数和析构函数的次序为(　　)。

A. ClassA 的构造函数、ClassB 的构造函数；ClassB 的析构函数、ClassA 的析构函数

B. ClassA 的构造函数、ClassB 的构造函数；ClassA 的析构函数、ClassB 的析构函数

C. ClassB 的构造函数、ClassA 的构造函数；ClassA 的析构函数、ClassB 的析构函数

D. ClassB 的构造函数、ClassA 的构造函数；ClassB 的析构函数、ClassA 的析构函数

5. 在派生类构造函数的成员初始化列表中不能包含(　　)。

A. 基类的构造函数

B. 派生类中成员对象的初始化

C. 基类中成员对象的初始化

D. 派生类中一般数据成员的初始化

6. 关于派生类的构造函数和析构函数，下列描述错误的是(　　)。

A. 当基类只定义带参数的构造函数时，派生类不需要定义构造函数

B. 如果基类分别定义不带参数和带参数的构造函数，则派生类可以定义不带参数的构造函数

C. 在派生类中，各个析构函数的执行顺序总是和构造函数的执行顺序严格相反

D. 当有多个非虚基类时，按照派生类定义时的继承顺序分别依次调用各个基类的构造函数

7. 下列关于多继承二义性的描述中错误的是(　　)。

A. 若派生类的两个基类中都有某同名成员，存在二义性问题

B. 二义性问题都是无法解决的

C. 对成员名进行类作用域限定可以解决部分二义性问题

D. 若一个派生类有两个基类，而这两个基类又有一个共同的基类，则对该共同基类的成员进行访问时可能出现二义性

8. 对于有虚祖先基类的多层派生类，在构造函数初始化列表中给出了虚基类的构造函数，这样会导致虚基类部分的初始化次数为(　　)。

A. 与虚基类下面的派生类个数有关　　B. 多次

C. 两次　　D. 一次

9. 多继承的构造顺序可分为以下 4 步：

(1) 所有非虚基类的构造函数按照它们被继承的顺序构造；

(2) 所有虚基类的构造函数按照它们被继承的顺序构造；

(3) 所有派生类新增子对象(成员对象)的构造函数按照它们被声明的顺序构造；

(4) 派生类自己的构造函数体。

这 4 个步骤的正确顺序是(　　)。

A. (2)(1)(3)(4)　　B. (4)(3)(2)(1)

C. (3)(4)(1)(2)　　D. (2)(4)(3)(1)

10. 有关父窗口和父类(基类),下列说法中正确的是(　　)。

A. 在 Qt 中,当父窗口被销毁时,所有以它为父窗口的部件都会被自动销毁,这是由对象树机制实现的

B. 父类是类,是相对于派生类而言的,是派生类的基类;而父窗口是一个基类对象,以它为父窗口的对象一定是父窗口类的派生类的对象

C. 它们是同一个概念

D. 每个部件或窗口都必须要设置它的父窗口

11. 关于静态创建的对象和动态创建的对象,下列说法中错误的是(　　)。

A. 静态创建的对象有名字,动态创建的对象通过指针指向

B. 静态创建对象的内存空间在超出作用域范围后由系统自动回收

C. 动态创建对象的内存空间由编程者维护和释放

D. 静态创建的对象可以通过 delete 语句释放

12. 下列关于自定义信号和槽的描述正确的是(　　)。

A. 不能定义具有私有(private)访问权限的槽

B. 信号具有私有的访问权限

C. 在类中要定义信号或槽,必须直接或间接地继承自 QObject 类,且在类的 private 区域中添加宏 Q_OBJECT

D. 信号通过 emit 语句发送,只能发送给信号所在类的对象

13. 下列关于命名空间的说法不正确的是(　　)。

A. 在函数、类中可以定义命名空间

B. 在不同命名空间中可以定义同名的类

C. 在一个命名空间中可以包含另一个命名空间,即命名空间可以嵌套

D. 在命名空间中可以定义函数、类等

14. 关于使用 Qt Designer 设计界面,下列说法中不正确的是(　　)。

A. 界面文件经编译后会生成对应的.h 头文件

B. 在界面中进行的设置是无法通过编写 C++代码实现的

C. 界面文件是一个文件名以.ui 结尾的 XML 文件

D. 即使不使用 Qt Designer,程序员还是可以通过编写代码创建出可视化界面

15. 下列关于自动关联槽的说法不正确的是(　　)。

A. 需要通过 QMetaObject::connectSlotsByName 进行自动关联设置

B. 槽函数的名字格式为“on_信号的发射对象_发射的信号名”

C. 自动关联的槽函数需要程序员编写槽函数体实现

D. 槽函数的名字格式为“on_信号的接收对象_发射的信号名”

16. 设 ClassB 类是公有派生自 ClassA 类的派生类，并有语句“ClassA objA，* ptrA＝&objA；Class B objB，* ptrB＝&ojbB；”，则下列赋值语句正确的是(　　)。

A. ptrA＝ptrB；　　B. ptrB＝ptrA；

C. objB＝objA；　　D. ptrB＝&objA；

17. 阅读下列程序，出现编译错误的语句是(　　)。

```
class A{public: int a;};
class B:public A{public: int b;};
class C:public A{public: int c;};
class D:public B, public C{public:int d;};
int main()
{
    D d;
    d.a = 1; //①
    d.b = 3; //②
    d.c = 4; //③
    d.d = 5; //④
    return 0;
}
```

A. ①　　B. ①③　　C. ②③　　D. ③④

18. 关于虚基类，下列描述错误的是(　　)。

A. 当既有虚基类又有非虚基类时，先调用虚基类的构造函数，再调用非虚基类的构造函数

B. 用于解决继承最远共同基类时的二义性问题

C. 需要在第一层基类继承时引入关键字 virtual

D. 在派生类(包括直接或间接派生的类)中不需要通过构造函数的初始化表对虚基类进行初始化

二、程序分析题

1. 程序的运行结果为

```
1
2
3
4
```

请根据运行结果和代码中的注释填空。

```
#include <iostream>
using namespace std;
class ClassA
{
private:
    int a;
protected:
    int b;
public:
    int c;
```

```
    ClassA(int _a, int _b, int _c):a(_a),b(_b),c(_c)
    {
    }
    void show()
    {
        cout << a << endl;
        cout << b << endl;
        cout << c << endl;
    }
};
class ClassB ____①____                          //公有继承
{
private:
    int d;
public:
    ____________②____________                   //构造函数头
    {
        d = _d;
    }
    void show()
    {
        ____③____
        cout << d << endl;
    }
};
int main()
{
    ClassB obj(1,2,3,4);
    obj.show();
    return 0;
}
```

2. 请阅读程序，给出运行结果。

```
#include <iostream>
using namespace std;
class Date
{
protected:
    int year,month,day;
public:
    Date(int y = 2000,int m = 1,int d = 1):year(y),month(m),day(d)
    {
        cout <<"Date constructor.";
        show();
    }
    ~Date()
    {
        cout <<"Date destructor."<< endl;
    }
    void show()
```

```
    {
        cout << year <<"/"<< month <<"/"<< day << endl;;
    }
};
class Person
{
protected:
    string name;
public:
    Person()
    {
        name = "John";
        cout <<"Person constructor."<< endl;
    }
    ~Person()
    {
        cout <<"Person destructor."<< endl;
    }
};
class Student:virtual public Person
{
protected:
    Date admissionDate;
public:
    Student():admissionDate(2019,9,1)
    {
        cout <<"Student constructor."<< endl;
    }
    ~Student()
    {
        cout <<"Student destructor."<< endl;
    }
};

class Worker:virtual public Person
{
public:
    Worker()
    {
        cout <<"Worker constructor."<< endl;
    }
    ~Worker()
    {
        cout <<"Worker destructor."<< endl;
    }
};
class Intern: public Student, public Worker
{
protected:
    Date internshipStartDate;
public:
```

```
    Intern():internshipStartDate(2020,7,1)
    {
        cout <<"Intern constructor."<< endl;
    }
    ~Intern()
    {
        cout <<"Intern destructor."<< endl;
    }
    void show()
    {
        cout <<"Name:"<< name << endl;
        cout <<"Date of enrollment:";
        admissionDate.show();
        cout <<"Internship start date:";
        internshipStartDate.show();
    }
};
int main()
{
    Intern intern;
    intern.show();
    return 0;
}
```

3. 请阅读程序,给出运行结果。

项目文件 ch3exam2_3.pro 的代码如下。

```
QT += widgets
SOURCES += main.cpp
HEADERS += myclass.h
```

头文件 myclass.h 的代码如下。

```
#include <QObject>
#include <iostream>
class MyClass:public QObject
{
    Q_OBJECT
    int value;
public:
    void setValue(int v)
    {
        value = v;
        emit valueChanged();
    }
public slots:
    void showValue()
    {
        std::cout << value << std::endl;
    }
signals:
```

```
    void valueChanged();
};
```

源文件 main.cpp 的代码如下。

```
#include "myclass.h"
int main()
{
    MyClass obj;
    QObject::connect(&obj, &MyClass::valueChanged,
                     &obj,&MyClass::showValue);
    obj.setValue(3);
    obj.setValue(5);
    return 0;
}
```

4. 请阅读程序，给出运行结果。

```
#include<iostream>
using namespace std;
namespace ns1 {
int i=1;
}
int main()
{
    using namespace ns1;
    cout<<i<<" ";
    int i=2;
    {
        int i=3;
        cout<<i<<" ";
    }
    cout<<i;
    return 0;
}
```

三、编程题

1. 定义一个哺乳动物类 Mammal，它含有 weight（出生重量，单位为克）数据成员和 outInfo()成员函数（输出出生重量信息），要求定义带参构造函数、不带参构造函数、析构函数；由哺乳动物类派生出狗类 Dog，新增 color（颜色）数据成员，并重定义 outInfo()成员函数（在该函数中调用基类的 outInfo()成员函数，以完成对基类派生得到的属性的输出，然后输出新增的数据成员信息），要求定义带参构造函数、不带参构造函数、析构函数；在主函数中分别定义使用了默认构造函数的 Dog 类对象和带参构造函数的 Dog 类对象，以测试类的使用与观察构造函数与析构函数的调用顺序。

2. 按如下要求编写程序：(1)定义一个 Shape 基类，它包含 area 数据成员（表示面积），有 getArea()成员函数返回面积值，定义构造函数对面积进行初始化；(2)Shape 类作为虚基类派生出 Square 类，增加边长属性，构造函数需有参数初始化边长，并根据边长传递参数（面积＝边长×边长）给基类，以初始化面积；(3)Shape 类作为虚基类派生出 Circle 类，增加

半径属性，构造函数需有参数初始化半径，并根据半径传递参数给基类，以初始化面积；(4)Square 类和 Circle 类共同派生出铜钱类 CopperCash(外圆内方的)，定义构造函数正确设置其外圆半径、内孔边长和面积(为圆面积减去中间镂空的方块面积)；(5)定义主函数测试铜钱类的使用。

3. 创建一个窗口，将大小设置为 300×100 像素(宽×高)，标题为你的姓名；在窗口中添加一个标签，标签的初始内容为“我用于显示”；然后在窗口中添加一个按钮，文字为“退出”，功能为单击时关闭当前窗口；继续在窗口中添加一个按钮，文字为“显示信息”，功能为单击时将标签显示的内容修改为“你单击了按钮”。

4. 创建如图 3-39 所示的界面(初始时姓名输入框、右侧多行文本框中均没有文字)，要求年龄可选范围为 0～120，性别默认为男，窗口标题名为“信息录入与显示”；使用布局排版部件；按下组合键 Alt＋M 时可以定位到数字选择框，按下组合键 Alt＋N 时可以定位到姓名输入框；单击“添加”按钮时可将信息添加到右边的列表框中。多次添加的效果如图 3-39 所示。

图 3-39　多次添加的效果

四、思考题

1. 不论是程序设计语言还是自然语言，都可能有二义性，也称为歧义。请针对 C++语言思考在哪些情况下会产生二义性，以及如何去处理(解决)这些二义性。

2. Qt 框架中的部件类是如何将界面设计和业务逻辑分开的？UI 界面文件又起到什么作用？它是如何跟部件类产生联系的？

实验 3　派生类、信号与槽和界面设计

一、实验目的

1. 掌握派生类的定义，能在 Qt 已有窗口类的基础上编写自己的类。
2. 了解虚基类的概念。
3. 掌握界面设计师工具的使用。
4. 掌握自定义信号、槽的编写。

二、实验内容

1. 编写一个输入和显示学生和教师信息的程序

学生数据有学号、姓名、性别、身份证号、联系电话、专业；教师数据有工号、姓名、性别、身份证号、联系电话、职称、部门。要求编写3个类实现：Person类包含最基本的共同信息；Student类继承Person类，并包含自己的成员(数据成员和成员函数)；Teacher类继承Person类，并包含自己的成员(数据成员和成员函数)。

2. 在实验内容1的基础上按要求扩充和修改程序

定义一个派生自教师类和学生类的助教类，它除了具有教师类和学生类的属性之外，还具有course(辅助教授的课程名)数据成员以及相关的成员函数(如输入、输出信息的成员函数，构造函数等)。请根据虚基类的知识合理地设计各个类的继承方式，并编写主函数测试助教类的使用。

提示：为实验简便起见，可把部分数据成员定义成受保护的。

3. 创建基于QWidget基类的Qt应用程序

实现以下功能。

(1) 在界面上添加一个标签，标签的初始内容为“我是第1个标签”。

(2) 通过编写代码在创建的自定义窗口类中添加一个标签(QLabel)成员，标签的初始内容为“我是第2个标签”。

(3) 在界面上添加两个按钮，一个按钮初始显示的内容为“显示坐标信息”，单击时在第1个标签中显示窗口(带框架)的左上角的坐标和右下角的坐标；在第2个标签中显示本窗口用户区的左上角和右下角的坐标。另一个按钮显示的内容为“退出”，当单击时关闭窗口(要求在界面设计师的信号与槽编辑器中实现关联)。

提示：QString::number(int)将整型转换成QString类型；通过自定义类对象中指向界面对象的数据成员ui指针访问界面对象里的成员。

4. 创建基于QWidget基类的Qt项目

实现以下功能。

(1) 添加一个单行文本框(QLineEdit)，可以输入姓名。

(2) 添加一个下拉列表(QComboBox)，可以选择科目(语文、数学、英语)。

(3) 创建一个组合框(QGroupBox)并在其中放置单选按钮，以选择成绩级别(A、B、C、D，默认选中A)。

(4) 创建一个列表框(QListWidget)，在其中添加两条记录“张三英语成绩：A”和“李四语文成绩：B”。

(5) 创建一个“添加成绩”按钮(QPushButton)，可以在列表框中的最后添加一行成绩记录。

(6) 创建一个“插入成绩”按钮，可以在列表框选定项的上面插入一行成绩记录。

(7) 在添加或插入成绩时首先判断QListWidget中的项目数，当少于10个时可以添加；当大于10个时发射自定义信号，关联的槽函数弹出警告对话框，内容为“条目数已达10个最大值”。

(8) 创建一个“关于”按钮，单击时弹出一个消息对话框，对话框中的内容为你的姓名。

(9) 在列表框中双击某项，可以删除这行成绩记录。

(10) 在适当位置添加标签用于说明，并要求使用布局管理窗口中的部件，界面参考如图3-40所示。

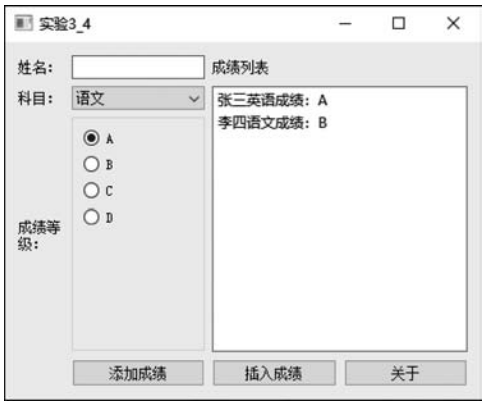


图 3-40　实验内容 4 的界面

提示：可能用到的成员函数如下。

```
QRadioButton 类：isChecked()          //是否被选中
QComboBox 类：  currentText()         //返回当前选中项字符串
QListWidget 类：currentRow()          //获取当前项的编号
                addItem(item)         //在末尾插入 item 字符串项
                insertItem(int,item)  //在指定位置插入 item 字符串项
                takeItem(int)         //删除该编号行
                count()               //返回总行数
```

5. 完成一个进制转换器。

要求实现以下功能，界面如图 3-41 所示。

图 3-41　实验内容 5 的界面

(1) 创建一个文本框，初始时值为 0，要求显示时右对齐；创建一个不可编辑的文本框，用于显示转换结果。

(2) 创建 0～9 数字按钮，使用户能够通过单击按钮的方式输入数据。

(3) 创建单选按钮组，能够选择将十进制转换成二进制、八进制还是十六进制。

(4) 创建“计算”按钮，单击时根据单选按钮组中的选择进行转换；创建“清空”按钮，单击时将文本框中的数字设置为 0；创建“退出”按钮，单击时退出程序。

提示：QString 里的 number 成员函数可实现十进制到各种进制的转换。