第 5 章

多　　态

多态，按照字面意思去理解就是“多种状态”。在 C++ 中，它是指用同一个名字定义的接口，有多种不同的实现方式，从而实现“一个接口，多种实现方法”。多态是面向对象的第 3 大特征，分为静态多态和动态多态。

所谓静态多态（也称为编译时多态），是指在程序编译时就能够确定具体的使用形式。例如，函数重载就是静态多态的一种体现，编译时会根据参数个数和类型确定对同名函数的调用实际是哪个函数实现体。模板也是静态多态的一种，它采用数据类型作为参数，实现对通用程序设计的支持。

动态多态（也称为运行时多态）通过继承和虚函数实现，特点是编译时还不能确定实际调用的是哪个类的同名函数，要到程序实际运行时，才能根据当前对象的实际类型确定。

5.1　静态多态——模板

在某些情况下，可能需要对多种类型的数据进行类似的操作，以完成相同的功能。例如，编写函数 swapValue()实现对两个同类型的变量进行值交换。由于数据类型很多，此时如果使用函数重载，就需要编写多个重载函数，分别对每种数据类型进行实现。

```
void swapValue(int& x,int& y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
void swapValue(double& x,double& y)
{
    double temp;
    temp = x;
```

```
    x = y;
    y = temp;
}
void swapValue(QChar& x,QChar& y)
{
    QChar temp;
    temp = x;
    x = y;
    y = temp;
}
```

这些 swapValue()函数的结构都是类似的,不同的只是参数类型。多个重载函数不能保证功能实现上的一致性,效率也比较低下。C++中的模板技术可以使用参数化的类型创建一种通用功能的模板代码,以支持多种不同的数据类型,称为函数模板。

除函数之外,某些类之间也可能存在形式和功能都相似,只有类结构中使用的数据类型不同的情形。此时,也可以为这些类使用参数化的类型创建一个通用模板,称为类模板。模板的使用提高了代码的可重用性和开发效率。

5.1.1 函数模板


视频讲解

函数模板的定义形式如下。

```
template<类型参数列表>
函数类型 函数名(形参列表)
{
    //函数体
}
```

定义中的第 1 行称为模板前缀,template 是关键字,表示开始定义一个函数模板(或类模板)。一对尖括号括起的部分是类型参数列表,每个类型参数声明为

```
class 类型参数名
```

class 关键字也可以使用 typename 代替,指明它后面的标识符是一个类型参数名。凡是希望根据实际参数确定数据类型的变量,都可以使用类型参数进行声明。当有多个类型参数时,使用逗号分隔。

```
class 类型参数名 1,class 类型参数名 2,…
```

多数情况下的函数模板只需要一个类型参数就够了。例如,swapValue()函数模板的定义如下。

```
template<class T>
void swapValue(T& x,T& y)
```

```
{
    T temp;
    temp = x;
    x = y;
    y = temp;
}
```

类型参数 T 可以在使用时被替换为任何类型。

函数模板是一种抽象的形式，并不是一个可以直接执行的函数。编译时，编译器会使用实际的实参类型替换模板中的类型参数，以生成一个函数实例(称为模板函数)，然后再调用这个模板函数完成相应的功能。注意，模板中同样类型参数的地方，实参的类型也应当一致，否则会出现编译错误。例如：

```
int intA = 1, intB = 2;
swapValue(intA, intB);
qDebug()<< intA << intB;
```

传递给 swapValue()的是 int 类型的实参 intA 和 intB，编译器会用 int 替换掉类型参数 T，生成模板函数如下。

```
void swapValue(int& x, int& y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

然后再调用这个模板函数完成对 intA 和 intB 的值交换。上述代码输出的结果为

```
2    1
```

针对具体调用时的实参类型，编译器会为之生成一个独立的模板函数，但不会为没有真正用到的类型生成相应的模板函数。使用模板函数和普通函数在写法上并没有什么区别。

函数模板也可先声明，再定义和使用，声明时也需要加上模板前缀。函数模板还可以重载。例如，声明实现 3 个数据值交换(值依次向前轮换一个)的函数模板如下。

```
template < class T > void swapValue(T& x, T& y, T& z);
```

提示：因为许多编译器或者不支持函数模板声明，或者不支持函数模板的独立编译，或者即使支持，不同编译器的支持细节也可能呈现很大的不同，容易造成混乱，所以建议直接将函数模板定义在使用它的同一个文件中，并且确保定义在使用之前。

创建 Empty qmake Project 空项目 5_1，然后添加一个.cpp 文件，代码如下。

```
/************************************
 * 项目名：5_1
 * 说　明：函数模板及其使用
 ************************************/
```

```
#include <QDebug>

template <class T>
void swapValue(T& x,T& y)                          //函数模板
{    T temp;
     temp = x;
     x = y;
     y = temp;
}

template <class T> void swapValue(T& x,T& y,T& z)  //重载的函数模板
{    T temp;
     temp = x;
     x = y;
     y = z;
     z = temp;
}

int main(int argc, char *argv[])
{
     int intA = 1,intB = 2;
     swapValue(intA,intB);
     qDebug()<< intA << intB;

     double doubleA = 1.1,doubleB = 2.2,doubleC = 3.3;
     swapValue(doubleA,doubleB,doubleC);
     qDebug()<< doubleA << doubleB << doubleC;

     QChar qcharA('a'),qcharB('b');
     swapValue(qcharA,qcharB);
     qDebug()<< qcharA << qcharB;

     //swapValue(qcharA,intB);                      //与模板不一致,无法生成函数实例
}
```

使用时会根据实际的参数类型和个数,确定使用哪个函数模板以及生成什么样的模板函数。对于语句“swapValue(qcharA,intB);”,由于第 1 个实参为 QChar 类型,第 2 个实参为 int 类型,并不能与模板进行匹配,程序中也没有与之匹配的普通函数,因此会编译出错。

除了如代码中所示,使用调用过程推导出的模板类型参数类型(隐式实例化)外,还可以使用显式实例化的形式,如代码中的调用语句也可以分别写为

```
swapValue<int>(intA,intB);
swapValue<double>(doubleA,doubleB,doubleC);
swapValue<QChar>(qcharA,qcharB);
```

“应用程序输出”窗口的输出如图 5-1 所示。

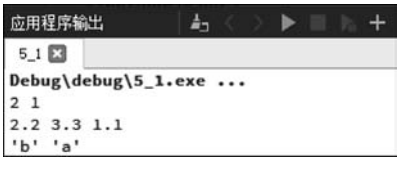

图 5-1 项目 5_1 的运行结果

函数模板还可以带普通参数,调用时需要显式实

例化，下面通过项目 5_2 展示带普通参数的函数模板的用法。

创建 Empty qmake Project 空项目 5_2，然后添加一个.cpp 文件，代码如下。

```
/************************************
 * 项目名：5_2
 * 说　明：带普通参数的函数模板及其使用
 ************************************/
#include <QDebug>
#include <QChar>

template <typename T,int size>
void ascendSort(T elements[])                    //冒泡排序函数模板
{
    T temp;
    for(int i = 0;i < size - 1;i++)
        for(int j = 0;j < size - 1 - i;j++)
            if(elements[j]> elements[j + 1])
            {
                temp = elements[j];
                elements[j] = elements[j + 1];
                elements[j + 1] = temp;
            }
}

int main(int , char *[])
{
    int intArr[5] = {3,4,1,2,5};
    ascendSort <int,5 >(intArr);
    qDebug()<< intArr[0]<< intArr[1]<< intArr[2]<< intArr[3]<< intArr[4];

    QChar qcharArr[5] = {'z','c','e','m','d'};
    ascendSort <QChar,5 >(qcharArr);
    qDebug()<< qcharArr[0]<< qcharArr[1]<< qcharArr[2]<< qcharArr[3]<< qcharArr[4];
}
```

编译器根据调用时的实参类型，分别实例化 int 和 QChar 类型的模板函数，完成本类型数据由小到大的排序，运行结果如图 5-2 所示。

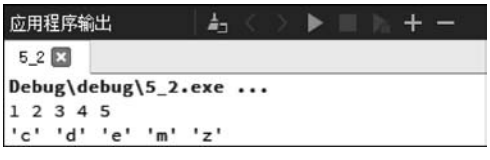

图 5-2　项目 5_2 的运行结果

5.1.2 类模板


视频讲解

类模板的写法如下。

```
template <类型参数列表>
class 类名
{
    //类成员声明
};
```

类模板也可以是派生类模板，此时类名后要加上“：继承权限 父类名”。类型参数列表的规则与函数模板相同。

类模板的每个成员函数都是模板(成员函数模板)，它们在类模板内的声明与普通类的成员函数声明相同(除了使用模板参数)。但在类模板外进行定义时，语法有所不同，格式如下。

```
template <类型参数列表>
返回值类型 类模板名<类型参数名列表>::成员函数名(形参列表)
{
  //成员函数实现
}
```

即定义时需要指定模板前缀。且注意：使用的类名和一般的类名不同，需要在类模板名后添加一对尖括号，括号内指明类型参数名列表(不包括 class 关键字)。

类模板必须显式实例化(称为模板类)，且实际中使用的实参类型必须与显式指定的类型一致。

下面通过项目 5_3 熟悉类模板的定义和使用。考虑在项目 5_2 的基础上进一步改造，将数组和对数组的排序等操作都封装在一个类模板 sortedArr 中；然后通过图形界面分别输入不同类型的多个数据，再显示它们的排序结果。具体步骤如下。

(1) 创建带界面的基于 QWidget 的项目 5_3。

(2) 拖入 4 个标签、两个数字选择框和两个列表框。设置数字选择框的 maximum 属性为 5，界面设计参考图 5-3。

(3) 添加一个 sortedarr.h 头文件，用于声明和实现类模板 sortedArr。由于 minggw 编译器不支持模板的独立编译，因此将类模板的声明和成员函数模板的实现都放在该头文件中。sortedarr.h 头文件代码如下。

```
/************************************
 * 项目名: 5_3
 * 文件名: sortedarr.h
 * 说  明: 动态排序数组类模板 sortedArr 的定义
 ************************************/
```

```
#ifndef SORTEDARR_H
#define SORTEDARR_H

template <class T>
class sortedArr
{
public:
    sortedArr(int max = 1);
    ~sortedArr();
    T at(int pos);              //返回第 i 个数据
    void insert(T value);       //插入一个数据
private:
    int size,maxSize;           //size 为当前已存储数据个数,maxSize 为最大能存储数据个数
    T * data;                   //指向数据的指针
};

template <class T>
sortedArr<T>::sortedArr(int max):size(0),maxSize(max)
{
    data = new T[max];
}

template <class T>
sortedArr<T>::~sortedArr()
{
    delete[] data;
}

template <class T>
void sortedArr<T>::insert(T value)
{
    data[size++] = value;
    T temp;

    for(int i = size - 1;i > 0;i --)  //将新添加的数据插入排序好的队列
        if(data[i - 1]> data[i])
        {
            temp = data[i - 1];
            data[i - 1] = data[i];
            data[i] = temp;
        }
}

template <class T>
T sortedArr<T>::at(int pos)
{
```

```
    return data[pos];
}

#endif //SORTEDARR_H
```

该类模板表示的数组长度是可变的(私有数据成员 size 实时记录数组大小),可在定义对象时通过初始化参数指明数组的最大可能长度(用私有数据成员 maxSize 记录)。私有数据成员指针 data 用于指向数据数组。该数据数组在构造函数中动态申请,在析构函数中释放。

每通过 insert()函数插入一个数据时,首先会将数据放在队列的最后,并将 size 加 1;然后再通过交换,将新数据插入已排序好的队列,因此数组中的数据总是有序的。at()函数用于返回数组中指定位置的元素。

(4) 给两个数字选择框的 editingFinished 信号添加自关联槽,在 widget.cpp 文件中定义如下。

```
/*************************************
* 项目名: 5_3
* 文件名: widget.cpp
* 说  明: 类模板的使用
*************************************/
//其他包含的头文件参考默认生成的代码,这里不再列出
#include <QInputDialog>
#include <QMessageBox>

//其他函数等参考默认生成的代码,这里不再列出
void Widget::on_spinBox_editingFinished()
{
    int number = ui->spinBox->value();
    sortedArr<int> intArr(number);        //创建一个包含 number 个整型数据的排序数组

    ui->listWidget->clear();              //清空列表框

    for(int i = 0;i < number;i++)         //输入数据到排序数组
    {
        int inputValue = QInputDialog::getInt(nullptr,"输入",
                         "第" + QString::number(i + 1) + "个整型数: ",0);
        intArr.insert(inputValue);
    }

    for(int i = 0;i < number;i++)         //显示到 listWidget
        ui->listWidget->addItem(QString::number(intArr.at(i)));
}

void Widget::on_spinBox_2_editingFinished()
{
    int number = ui->spinBox_2->value();
```

```
    sortedArr < QString > strArr(number);   //创建一个包含 number 个字符串的排序数组

    ui -> listWidget_2 -> clear();          //清空列表框


    for(int i = 0;i < number;i++)           //输入数据到排序数组
    {
        QString inputStr = QInputDialog::getText(this,"输入",
                           "第" + QString::number(i + 1) + "个字符串: ");
        strArr.insert(inputStr);
    }

    for(int i = 0;i < number;i++)           //显示到 listWidget_2
        ui -> listWidget_2 -> addItem(strArr.at(i));
}
```

在槽函数 on_spinBox_editingFinished()中,首先根据类模板创建了一个整型的模板类 sortedArr < int >; 接着清空列表框后,通过 for 循环依次弹出对话框请用户输入各整型数据,并插入数组和完成排序; 最后通过 for 循环依次将排序数组中的元素显示到列表框(listwidget)。

槽函数 on_spinBox_2-editingFinished()中完成类似的操作,只是生成的是 QString 类型的模板类 sortedArr < QString >,并将结果显示在列表框 2(listwidget_2)中。

main.cpp 文件没有改动,widget.h 头文件中相比默认生成的代码,只添加了自关联槽的声明,这里不再列出。

程序运行时,首先修改左边数字选择框中的值为 4,然后在该部件之外的任意地方单击完成编辑,在弹出的对话框中依次输入 9,8,7,6; 类似地,修改右边数字选择框中的值,并在弹出的对话框中输入字符串,最终运行效果如图 5-3 所示。

图 5-3 项目 5_3 的运行结果

提示:

(1) 一个类如果要使用信号与槽,必须要加入 Q_OBJECT 宏进行预处理。但 Q_OBJECT 宏不支持 C++类模板,所以通常类模板中不使用信号与槽。

(2) 如果确实希望定义一个可使用信号与槽机制的类模板,可以首先定义一个普通的中间类,在中间类中定义信号与槽,然后再使用中间类派生出类模板即可。

5.2 Qt 中的容器


视频讲解

容器是用于包含和管理其他对象(数据)的一个对象。例如,可形象地把数组类比为一个容器,它可以容纳多个数组元素,并能通过下标对元素进行操作。

Qt 提供了多种容器类型,分别用于表示动态数组、链表、从一个类型到另一个类型的映射等结构。它们通常以类模板的形式存在,都在 QTL(Qt Template Library)模板库中。常见的容器如表 5-1 所示。

表 5-1 Qt 常见容器

容器类型	功能
列表 QList<T>	存储指定 T 类型数据的列表。可以通过整数索引快速地访问数据,与依赖于迭代器进行查找的容器相比更快捷
链表 QLinkedList<T>	存储指定 T 类型数据的链表。只能通过迭代器访问数据。在数据量大且经常进行中间插入等操作时性能更好
向量 QVector<T>	存储指定 T 类型数据的向量(动态数组),数据存储于连续的内存空间中
栈 QStack<T>	是 QVector<T>的子类,提供后进先出(Last In First Out,LIFO)的数据结构和相关操作,如弹出、压入、查找当前栈顶等
队列 QQueue<T>	是 QList<T>的子类,提供了先进先出(First In First Out,FIFO)的数据结构和相关操作,如入队、出队、查找当前队头等
集合 QSet<T>	是一个能够快速查询指定 T 类型数据值的集合
映射表 QMap<key,T>	提供了一个关联数组。将 key 类型的键值映射到 T 类型的数据值上。内部按照键值的顺序存储数据
多值映射 QMultiMap<key,T>	是 QMap 的子类,提供多值映射(一个键值可关联多个 T 类型值)
散列表 QHash<key,T>	类似于 QMap,但以任意顺序存储,因此查找速度更快
多值散列表 QMultiHash<key,T>	是 QHash 的子类,提供多值散列的存储

标准 C++中也提供了若干容器,在 STL(Standard Template Library)标准模板库中。Qt 应用程序中既可以使用标准 C++的容器,也可以使用 Qt 提供的容器。鉴于 Qt 容器“平台无关”和“隐式数据共享”等优势,以及考虑到在一些嵌入式平台中 STL 或许不可用,建议读者使用 Qt 容器。

Qt 容器中存储的项(数据)必须是可以赋值的数据类型(即具有默认构造函数、复制构造函数,可进行赋值操作的数据类型)。例如,基础数据类型(int、double、指针等)和部分 Qt 数据类型(如 QChar、QString、QDateTime 等)都可以存储到容器中,但 QObject 类及其子类们(如 QWidget)则不能(但可以存储指向它们的指针)。

部分容器可以使用索引(类似于数组中的下标)操作包含的项,所有容器都可以使用迭代器(也是类模板,作用类似于索引)操作包含的项。每个容器类型都提供了 Java 风格的迭代器和 STL 风格的迭代器。每种风格的迭代器又可以分为只读迭代器和读写迭代器,前者只能读取数据,后者还能修改存储的数据。

Java 风格的迭代器如表 5-2 所示，表中迭代器的名字非常规则：只读迭代器均为在容器(或其父容器类)名的类型参数列表前加上 Iterator；读写迭代器在只读迭代器的基础上，再在 Q 后面加上 Mutable。

表 5-2　Java 风格的迭代器

容　　器	只读迭代器	读写迭代器
QList＜T＞ QQueue＜T＞	QListIterator＜T＞	QMutableListIterator＜T＞
QVector＜T＞ QStack＜T＞	QVectorIterator＜T＞	QMutableVectorIterator＜T＞
QLinkedList＜T＞	QLinkedListIterator＜T＞	QMutableLinkedListIterator＜T＞
QSet＜T＞	QSetIterator＜T＞	QMutableSetIterator＜T＞
QMap＜key,T＞ QMultiMap＜key,T＞	QMapIterator＜key,T＞	QMutableMapIterator＜key,T＞
QHash＜key,T＞ QMultiHash＜key,T＞	QHashIterator＜key,T＞	QMutableHashIterator＜key,T＞

STL 风格的迭代器如表 5-3 所示。

表 5-3　STL 风格的迭代器

容　　器	只读迭代器	读写迭代器
QList＜T＞ QQueue＜T＞	QList＜T＞::const_iterator	QList＜T＞:: iterator
QVector＜T＞ QStack＜T＞	QVector＜T＞::const_iterator	QVector＜T＞:: iterator
QLinkedList＜T＞	QLinkedList＜T＞::const_iterator	QLinkedList＜T＞::iterator
QSet＜T＞	QSet＜T＞::const_iterator	QSet＜T＞:: iterator
QMap＜key,T＞ QMultiMap＜key,T＞	QMap＜key,T＞::const_iterator	QMap＜key,T＞:: iterator
QHash＜key,T＞ QMultiHash＜key,T＞	QHash＜key,T＞::const_iterator	QHash＜key,T＞:: iterator

各类容器的区别在于内部保存和操作数据的方式不一样。后续将介绍列表、向量、链表 3 种类型，并通过它们了解 QTL 中的容器、迭代器及相关算法。


视频讲解

5.2.1　列表

QList＜T＞是提供列表的模板，内部存储为一组指向被存储元素的指针数组(当 T 本身就是指针类型或大小不超过指针类型的基本类型时，QList＜T＞会直接在数组中存储这些元素)。创建列表对象时，默认初始化为空列表。

列表的优势在于能对存储的数据进行快速索引，还提供了快速插入列表项、删除列表项等操作。由于 QList 在列表两端都预先分配缓存，因此在列表两端插入或删除元素的操作也很快。

列表可以使用插入运算符进行连续输入。例如，下面的语句会在 intList1 中依次插入 3 个整型元素。

```
QList < int > intList1, intList2;
intList1 << 1 << 2 << 3;
```

列表可进行整体赋值操作。执行下面的语句后，intList2 中也包含了 3 个同样的整型元素。

```
intList2 = intList1;
```

列表可进行比较运算(==和!=)，当两个列表内部的项和顺序都完全一样时，==运算的结果为 true；还可以使用+、+=等运算实现列表连接操作。

QList < T >的索引与数组的下标类似，都是从 0 开始的，也可以使用[]运算符访问位于索引处的值，代码如下。

```
intList1[0] = 0;
qDebug()<< intList1[1];
```

上述代码会将列表中的第 0 个元素更新为 0(原值为 1)，然后输出第 1 个元素的值 2。如果只是读取索引处元素的值，建议调用 at()成员函数，其操作会比下标运算[]更快(因为前者不需要进行深拷贝)。

列表提供了对应的迭代器。但由于也能使用索引值访问元素，因此很少使用它的迭代器，更多的是使用索引值进行访问。QList < T >列表常用的成员函数如表 5-4 所示。

表 5-4 QList < T >列表常用的成员函数

成员函数	功能
bool isEmpty() const;	列表中没有项(数据元素)时返回 true，否则返回 false
int size() const;	返回列表中的项数
const T at(int i) const;	返回位于索引位置 i 处的项。使用时应保证 0 <= i < size()
T& first();	返回列表中第 1 项的引用
T& last();	返回列表中最后 1 项的引用
const T& constFirst() const;	返回列表中第 1 项的常引用(不能通过引用修改此项)
const T& constLast() const;	返回列表中最后 1 项的常引用(不能通过引用修改此项)
int indexOf(const T& value, int from=0) const;	从 from 位置开始，搜索 value 第 1 次出现的位置并返回；未找到时返回-1
int lastIndexOf(const T &value, int from = -1) const;	从 from 位置开始，反向搜索 value 第 1 次出现的位置并返回，未找到时返回-1。from 为-1 时从最后 1 项开始搜索
bool contains(const T &value) const;	列表中包含有值为 value 的项时返回 true，否则返回 false
int count(const T &value) const;	返回列表中 value 出现的次数
void insert(int i, const T &value);	在列表的索引位置 i 处插入 value 项
void append(const T &value);	在列表末端插入 value 项
void prepend(const T &value);	在列表头前插入 value 项
void replace(int i, const T &value);	将索引位置 i 处的项替换成 value

续表

成员函数	功　能
void removeAt(int i);	删除索引位置 i 处的项
bool removeOne(const T &value);	删除第 1 次出现的 value 项，若成功删除返回 true；若未找到，返回 false
int removeAll(const T &value);	删除所有出现的 value 项，返回删除的项数
void removeFirst();	删除列表中的第 1 项
void removeLast();	删除列表中的最后 1 项
T takeFirst();	删除列表的第 1 项，并将该项返回
T takeLast();	删除列表的最后 1 项，并将该项返回
void clear();	删除列表中的所有项
void swap(int i, int j);	交换位于索引位置 i 和 j 的项
void move(int from, int to);	将位置 from 处的项移动到位置 to
QList<T> mid(int pos, int length = -1) const;	返回从位置 pos 开始，长度为 length 的子列表。如果 length=-1，从 pos 开始到列表尾
QVector<T> toVector() const;	返回转换为 QVector<T>类型的对象
static QList<T> fromVector(const QVector<T> &vector);	静态成员，返回根据 QVector<T>类型的参数 vector 转换的 QList<T>类型对象

需要注意，为了提高效率，除了 isEmpty()成员函数之外，其他的成员函数都默认列表是非空的，在使用前并不会验证其参数是否有效；并且使用索引值进行操作的成员函数均假设其索引值都在有效范围内。因此，在调用成员函数前，通常先使用 isEmpty()函数判断列表是否为空，以避免对空列表的错误操作；对于以索引值为参数的成员函数，还要注意保证索引的有效性(位于有效范围内)。

队列 QQueue<T>是 QList<T>的派生类，提供了先进先出的数据结构。新增的成员函数“void enqueue(const T &t);”将项目添加到队列尾(入队操作)；成员函数“T dequeue();”将队列第 1 项删除(出队操作)；成员函数“T& head();”用于获取队列第 1 项(不删除)的引用。


视频讲解

5.2.2 向量

QVector<T>是一个向量(或称为动态数组)模板，内部数据的存储位置彼此相邻，并可以基于索引快速访问。它提供了和 QList<T>类似的功能，包含的成员函数及其原型和 QList<T>中的也基本一致。

QVector<T>类型和 QList<T>类型通常可以互换，既可以通过 QList<T>中的成员函数 toVector()和静态成员函数 fromList()实现，也可以通过 QVector<T>中的成员函数 toList()和静态成员函数 fromVector()实现。两个类之间的区别如下。

(1) QVector<T>始终将项顺序存储在栈内存(栈区，由系统自动分配的存储空间)中；而 QList<T>会根据情况，将其项分配在堆内存(堆区，通过 new 运算符申请的存储空间)或栈区中(当 sizeof(T)小于 sizeof(void *)时)。在不要求存放数据的内存空间必须连续时，建议使用 QList<T>。

(2) 二者的查询复杂度差不多。但对于像 prepend()、insert()这样的操作,通常 QList<T>会比 QVector<T>快得多。

(3) 如果需要开辟连续的内存空间存储数据,或者单个元素的尺寸比较大(此时需要避免个别插入操作,以免出现堆栈溢出)时,建议使用 QVector<T>。

如果只是需要一个可变大小的数组,还可以使用 QVarLengthArray<T,length>类模板,它可以预先在栈区中分配 length 长度大小的数组空间,如果超过这个长度,会在堆区中每次增量申请固定大小的空间用于存储。

栈 QStack<T>是 QVector<T>的派生类,提供了后进先出的数据结构。新增的成员函数原型如下。

```
T pop();                    //出栈操作: 从栈顶弹出一个元素(删除列表的最后 1 项)并返回
void push(const T &t);      //入栈操作: 将 t 压入栈顶(添加为列表的最后 1 项)
T& top();                   //返回栈顶元素(列表的最后 1 项)的引用
```

top()函数还有一个重载的常成员函数形式。

项目 5_4 展示了栈的用处,具体步骤如下。

(1) 创建带界面的、基于 QWidget 的应用程序。

(2) 在界面中拖入两个按钮、3 个标签、一个列表框、一个数字选择框和一个单行文本框,界面设计如图 5-4 所示。

图 5-4 项目 5_4 的运行结果

(3) 在 Widget.h 中包含头文件:

```
#include <QStack>
```

并在自定义类 Widget 中添加一个私有数据成员:

```
private: QStack<int> stack;
```

(4) 为两个按钮的 clicked 信号添加自关联槽。widget.cpp 文件中的代码如下。

```
/************************************
 * 项目名: 5_4
 * 文件名: widget.cpp
 * 说  明: 栈的使用
 ************************************/
```

```
//其他包含的头文件参考默认生成的代码,这里不再列出
#include <QMessageBox>

//其他函数等参考默认生成的代码,这里不再列出
void Widget::on_pushButton_clicked()
{
    stack.push(ui->spinBox->value());
    ui->listWidget->clear();
    for(int i=0;i<stack.size();i++)     //在 listWidget 中显示栈中的元素
        ui->listWidget->addItem(QString::number(stack.at(i)));
}

void Widget::on_pushButton_2_clicked()
{
    if(!stack.isEmpty())
    {
        int val=stack.pop();
        ui->lineEdit->setText(QString::number(val));
        ui->listWidget->clear();
        for(int i=0;i<stack.size();i++)
            ui->listWidget->addItem(QString::number(stack.at(i)));
    }
    else
        QMessageBox::information(this,"提示","栈已空");
}
```

程序运行后,依次入栈 6、5、4、8、7;然后出栈 7;再入栈 3,结果如图 5-4 所示。


视频讲解

5.2.3 链表

QLinkedList<T>是链式列表(链表),使用非连续的内存块保存数据,所以只能基于迭代器访问。链表提供了和列表类似的功能,如 append()、isEmpty()、size()、first()等,插入和删除操作也比较快。链表中存储的数据项称为节点。

各成员函数中的参数使用的是迭代器而非索引值。例如,在链表中,insert()成员函数的原型为

```
QLinkedList::iterator insert(QLinkedList::iterator before, const T &value);
```

表示在迭代器 before 后插入 value 节点。

STL 和 Java 两种风格的迭代器的实现不太一样:STL 风格的迭代器是建立在指针操作基础上的,指向当前节点;而 Java 风格的迭代器位于节点之间。

下面通过项目 5_5 熟悉链表和迭代器的使用。创建 Empty qmake Project 空项目 5_5,然后添加一个.cpp 文件,代码如下。

```
/**********************************
 * 项目名: 5_5
 * 文件名: main.cpp
 * 说  明: 链表及迭代器的使用
 **********************************/
#include <QDebug>
#include <QLinkedList>
int main(int argc, char *argv[])
{
    QLinkedList<QString> strLList;
    strLList <<"string1"<<"string2"<<"string3";

    //Java 风格的只读迭代器
    QLinkedListIterator<QString> rIterJ(strLList);  //使用容器对象初始化迭代器
    rIterJ.toBack();                                //修改迭代位置为位于最后 1 个节点后
    while(rIterJ.hasPrevious())                     //迭代位置前面有节点
        qDebug()<< rIterJ.previous();               //返回位置前的节点,再前跳一个节点
    qDebug()<< endl;

    //Java 风格的读写迭代器
    QMutableLinkedListIterator<QString> rwIterJ(strLList);
    while(rwIterJ.hasNext())                        //迭代位置后面有节点
    {
        rwIterJ.next();                             //迭代位置后跳一个节点
        rwIterJ.setValue("aaa:" + rwIterJ.value());  //修改最近跳过的节点
        if(rwIterJ.value() == "aaa:string1")
            rwIterJ.remove();                       //删除最近跳过的节点
        else if(rwIterJ.value() == "aaa:string2")
            rwIterJ.insert("new string");           //在迭代位置处插入节点
    }
    rwIterJ.toFront();                              //修改迭代位置位于第 1 个节点前
    while(rwIterJ.hasNext())                        //迭代位置后面有节点
        qDebug() << rwIterJ.next();                 //返回位置后的节点,再后跳一个节点
    qDebug()<< endl;

    //STL 风格的读写迭代器
    QLinkedList<QString>::iterator rwIterS;
    for(rwIterS = strLList.begin();rwIterS!= strLList.end();rwIterS++)
    {
        if((*rwIterS).endsWith("2"))
            *rwIterS = "bbb" + (*rwIterS).mid(3);   //修改当前节点
        else if(*rwIterS == "new string")
            strLList.insert(rwIterS,"another string");  //插入新节点
        else
            strLList.erase(rwIterS);                //删除当前节点
    }

    //STL 风格的只读迭代器
```

```
    QLinkedList<QString>::const_iterator rIterS;
    for(rIterS = strLList.constBegin();rIterS!= strLList.constEnd();rIterS++)
        qDebug()<< *rIterS;                                    //显示当前节点
    qDebug()<< endl;

    return 0;
}
```

Java风格的迭代器默认位于整个链表第1个节点之前。只读迭代器提供判断(如hasNext()函数判断是否有后继节点)、迭代位置修改(如toBack()函数将迭代置于最后1个节点之后)、读取指定节点(如peekNext()函数返回后继节点,但保持迭代位置不动;next()函数返回后继节点,且迭代位置后跳一个节点)等操作。读写迭代器还提供了setValue()、remove()、insert()等函数用于实现对节点的修改、删除、插入等操作。

图5-5 项目5_5的运行结果

STL风格的迭代器指向当前节点,可以通过*运算返回当前节点。迭代器每加1时向后跳一个节点,减1时向前跳一个节点。可直接通过读写迭代器修改当前节点。

容器提供了一些成员函数,用于获取指定节点的迭代,如begin()函数返回第1个节点的迭代、constBegin()函数返回第1个节点的只读迭代等;还提供了一些成员函数,用于根据迭代器操作节点,如insert()函数用于在迭代指向的节点前插入节点、erase()成员函数删除迭代指向的节点等。

程序运行结果如图5-5所示。


视频讲解

5.3 动态多态

把不同的子类对象都当作父类对象来看,可以屏蔽不同子类之间的差异,写出通用的代码。此时,子类对象只能使用原属于父类的那些属性和行为。而动态多态使以父类形式出现的子类对象仍能够以子类的方式进行工作。

例如,有动物类Animal,以及它派生出的狮子类Lion、牛类Cattle、人类Person等。假设各类中均实现了准备食物(prepairing()成员函数)、进食(eating()成员函数)、休息(resting()成员函数)等功能,每个功能在每个类中具体的实现方式都有所不同。例如,对牛来说,准备食物就是要发现一片草地;对狮子来说,准备食物就是要捕猎到猎物;而对人来说,准备食物就是要烧好饭等。

现在想实现函数showLifeProcess(),功能是按照寻找食物、进食、休息的顺序依次调用对象的成员函数,以模拟对象维持生命的过程。由于具体的动物类型的功能实现方式都不同,需要针对狮子类、牛类、人类等各种类型分别写一个这样的同名重载函数完成模拟。但

当具体的动物类型不断增加时，要写的就太多了。

自然就会想到，它们都属于动物，能否只针对 Animal 类写一个这样的函数就够了？这就是“屏蔽子类间的差异，写出通用的代码”的思路，代码如下。

```
void showLifeProcess(Animal * ptr)
{
    ptr->prepairing();
    ptr->eating();
    ptr->resting();
}
```

然而，根据 3.1.4 节，即使传递给函数形参 ptr 的是一个牛类对象的地址，在函数内部使用 ptr 指针指向时，也只能把它当作基类 Animal 的对象使用，调用的成员函数是 Animal 类中的成员函数，无法调用实际牛类中的同名成员函数。

这时，如果将 prepairing()、eating()、resting()等成员函数声明为虚函数，则传递牛类对象给函数时，内部实际调用的就都是牛类的成员函数了，即实现了“以父类形式出现的子类对象仍能够以子类的方式进行工作”的效果。

5.3.1 虚函数


视频讲解

和成员函数重定义类似，基类和派生类中的同原型虚函数也通常用来执行一些功能类似(但又不完全相同)的操作。但重定义时，对成员函数的调用是确定的；而虚函数则是在运行时根据实际情况动态地决定使用哪个类中的同名虚函数。

通过在基类定义语句中，在成员函数声明的返回类型前加上 virtual 关键字指明虚函数，格式如下。

```
virtual 返回类型 成员函数名(形参列表);
```

基类中声明了虚函数，则派生类中所有的同原型函数均自动成为虚函数。

为了演示虚函数的声明和使用，新建纯 C++项目 5_6，并分别实现动物类 Animal 和狮子类 Lion。Animal 类的定义如下。

```
/*****************************************
 * 项目名: 5_6
 * 文件名: animal.h
 * 说  明: 动物类定义
 *****************************************/
#ifndef ANIMAL_H
#define ANIMAL_H

class Animal
{
public:
```

```
    virtual void prepairing();
    virtual void eating();
    virtual void resting();
    virtual ~Animal();
};

#endif //ANIMAL_H
```

虚函数只在类定义中声明,类外实现虚函数时不能再加关键字 virtual。Animal 类的实现代码如下。

```
/*****************************************
 * 项目名: 5_6
 * 文件名: animal.cpp
 * 说  明: 动物类实现
 *****************************************/
#include "animal.h"
#include <iostream>
using namespace std;

void Animal::prepairing()
{
    cout <<"prepairing food."<< endl;
}

void Animal::eating()
{
    cout <<"eating food."<< endl;
}

void Animal::resting()
{
    cout <<"resting."<< endl;
}

Animal::~Animal()
{
}
```

派生类 Lion 的定义如下。

```
/*****************************************
 * 项目名: 5_6
 * 文件名: lion.h
```

```
 * 说  明: 狮子类定义
 ********************************************/
#ifndef LION_H
#define LION_H
#include "animal.h"

class Lion:public Animal
{
public:
    void prepairing();              //自动为虚函数
    void eating();                  //自动为虚函数
    void resting();                 //自动为虚函数
};

#endif //LION_H
```

由于基类中已经声明过虚函数,派生类中的同原型成员函数会自动成为虚函数。因此,派生类中这些函数前面不需要再加 virtual 关键字(也可以加上)。Lion 类的实现代码如下。

```
/*******************************************
 * 项目名: 5_6
 * 文件名: lion.cpp
 * 说  明: 狮子类实现
 ********************************************/
#include "lion.h"
#include <iostream>
using namespace std;

void Lion::prepairing()
{
    cout <<"Hunt an animal."<< endl;
}

void Lion::eating()
{
    cout <<"Shred animals and eat."<< endl;
}

void Lion::resting()
{
    cout <<"Find a cool place and rest."<< endl;
}
```

在基类 Animal 中声明了 3 个虚成员函数 prepairing()、eating()、resting()和一个虚析构函数(原因见提示),Lion 类以 Animal 为基类,那么它的 3 个同原型的成员函数也自动为虚函数。

提示：定义了虚函数的基类中最好要定义虚析构函数。因为使用语句“delete 基类指针;”销毁基类指针所指的动态派生类对象时，若基类析构函数不是虚函数，将只调用基类的析构函数，而不调用派生类的析构函数，这是不合理的。


视频讲解

5.3.2 调用方式

必须使用基类的指针或引用调用虚函数，执行时才会根据所指向(引用)对象的实际类型调用实际对象类型中的同原型虚函数。

```
/*****************************************
 * 项目名：5_6
 * 文件名：main.cpp
 * 说　明：虚函数的调用
 *****************************************/
#include <iostream>
#include "animal.h"
#include "lion.h"
using namespace std;

void showLifeProcess1(Animal *ptr)
{
    ptr->prepairing();
    ptr->eating();
    ptr->resting();
}

void showLifeProcess2(Animal& obj)
{
    obj.prepairing();
    obj.eating();
    obj.resting();
}

void showLifeProcess3(Animal obj)
{
    obj.prepairing();
    obj.eating();
    obj.resting();
}

int main()
{
    Lion lion;
    showLifeProcess1(&lion);          //传指针方式
    cout<<endl;
    showLifeProcess2(lion);           //传引用方式
```

```
    cout << endl;
    showLifeProcess3(lion);          //传值方式
    return 0;
}
```

程序运行结果如图 5-6 所示。

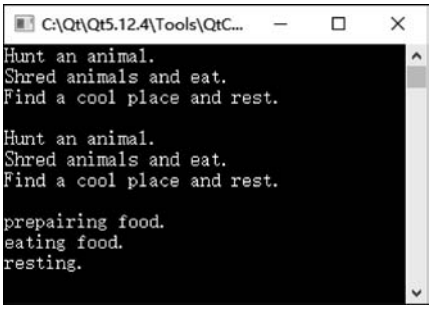

图 5-6 项目 5_6 的运行结果

showLifeProcess1()函数的参数为传指针方式,可以实现动态多态。指针 ptr 所指向的对象为 lion,根据实际对象为 Lion 类型而调用 Lion 类中的同原型虚函数。

showLifeProcess2()函数的参数为传引用方式,可以实现动态多态。引用 obj 实际代表的是派生类对象 lion 的内存空间,根据实际对象为 Lion 类型调用了 Lion 类中的同原型虚函数。

showLifeProcess3()函数的参数为传值方式,不能实现动态多态。在函数调用时,系统会给形参 obj 分配内存空间(一个 Animal 对象所需要的内存空间),然后将实参对象 lion 中属于基类部分的数据成员值复制给形参 obj。在函数内部实际使用的是一个 Animal 类型对象 obj,因此调用的是 Animal 类的成员函数。

5.3.3 实现原理

视频讲解

首先分析下面的程序,你认为输出应该是多少呢?

```
/*****************************************
 * 项目名: 5_7
 * 文件名: main.cpp
 * 说  明: 类对象的内存空间分配
 *****************************************/
#include <iostream>
using namespace std;

class A
{
    int a,b;
    double c;
public:
    void fun(){}
};

class B
{
    int a;
```

```
    double c;
    int b;
public:
    void fun(){}
};

class C
{
    int a,b;
    double c;
public:
    virtual void fun(){}
};

int main()
{
    cout << sizeof(A)<<"\t"<< sizeof(B)<<"\t"<< sizeof(C)<< endl;
    return 0;
}
```

程序运行结果如图 5-7 所示。

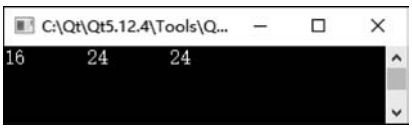

图 5-7　项目 5_7 的运行结果

由于成员函数存储在代码区，由类的所有对象所共享。因此，给对象分配内存空间时，只需要给各数据成员分配内存空间即可，类数据类型的内存长度通常为类内各数据成员的长度之和。例如，类 A 中包含了两个整型和一个 double 类型的数据成员，以及一个成员函数 fun()，其占据的空间为 sizeof(int)＋sizeof(int)＋sizeof(double)，即 4＋4＋8＝16 字节，这与输出结果是一致的。

类 B 和类 A 包含了同样的数据成员和成员函数，本质上是完全一样的。运行结果却显示两种数据类型对象占据内存空间大小不同。再仔细观察，会发现类 B 中调整了数据成员声明的顺序。为什么只是声明顺序不同，就会导致所占空间的不同呢？这是因为进行了“数据对齐”的操作。

为了避免读一个数据成员时要多次访问存储器，提高存储器的访问效率，操作系统对基本数据类型的合法地址做了限制，Windows 操作系统的对齐要求是：任何 K（K＝2，4 或 8）字节的变量，地址都必须是 K 的整数倍。数据成员 a 和 b 为 int 类型，占 4 字节；数据成员 c 为 double 类型，占 8 字节。类 A 和类 B 对象的数据成员内存情况如图 5-8 所示。

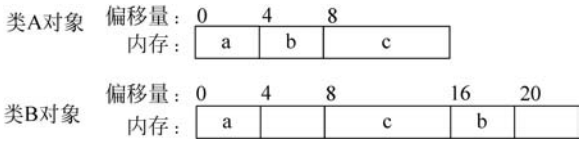

图 5-8　内存空间中的数据对齐

数据成员的位置是按照声明的前后顺序依次排列的。对于类 A，成员 b 占 4 字节，排在 4 号位置；成员 c 占 8 字节，排在 8 号位置，都满足 Windows 系统的对齐要求。而对于类 B，安排完 a 后，c 不能紧跟在这块空间的后面，因为 c 占 8 字节，而起始位置是 4，不满足整数倍的要求。因此，中间 4 字节为对齐填充部分，不代表有效数据。

似乎类 B 的对象长度为 20 字节就可以了。但之所以在数据成员 b 后面还要填充 4 字节，是考虑到对象连续存储的情况，如“B arr[2];”，如果末尾不填充，arr[1]对象中的数据成员 c 就不能满足对齐要求了。由于不同操作系统数据对齐要求不同，该程序在其他操作系统环境下的运行结果可能不同。

类 C 和类 A 的数据成员声明顺序相同，可见它们的大小与数据对齐无关。仔细观察可知，两者唯一的不同之处在于类 C 中成员函数 fun()前面加了一个 virtual 关键字，因此多出的 8 字节应当与虚函数有关。

类中有虚函数时，编译器会自动给每个由该基类及其派生类所定义的对象加上一个叫作 v-pointer 的指针，简称 VPTR(虚指针，占 8 字节)，通常位于对象内起始的位置。事实上，编译器还为每个含有虚函数的类加上了一个叫作 v-table 的表，简称 VTABLE(虚表)，同一个类的不同对象拥有相同虚函数表，在对象生成时，就将 VPTR 初始化为指向类的 VTABLE。VPTR 和 VTABLE 用于实现动态多态，如图 5-9 所示。

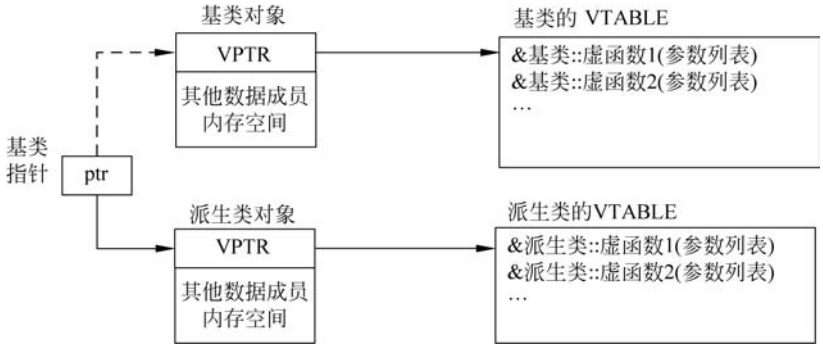

图 5-9 虚指针与虚表

通过基类指针(或引用)调用成员函数，在基类中没有虚函数时，是根据基类到代码区的固定位置去找函数段。而有虚函数的基类，使用基类指针(或引用)调用对象的虚函数时，是根据实际所指对象的虚指针找到类的虚表，然后再根据表中列出的虚函数的地址，到代码区找相应的函数段。

由此可见，图 5-9 中，若基类指针 ptr 指向派生类对象，根据它的 VPTR 找到的是派生类的 VTABLE，若改为如虚线所示的指向基类对象，则沿 VPTR 找到的是基类的 VTABLE，从而实现了动态多态。

如果在派生类中没有重定义基类的虚成员函数，则派生类虚表中记录的是基类虚成员函数的地址；若重定义过，则记录的是派生类虚成员函数的地址。

5.4 抽象类与纯虚函数

5.4.1 抽象类


视频讲解

有时候可能会希望避免使用基类定义新的对象。例如，一个实际的动物属于一个具体的派生类，如狮子、牛、狗、羊等。Animal 类只是对这些类进行了一个抽象，代表了很多具有

共同特点的类。它是一个抽象的概念，不存在只是 Animal 类型但不属于任何具体派生类的动物。因此，定义一个 Animal 类型的对象没有太大意义，更常见的情形是用具体的派生类，如狮子、牛等，去定义对象。

在这种情况下，基类也没有必要（可能也不能确定如何）去实现一个虚函数的具体功能，只由各个派生类分别提供虚函数具体的实现版本即可。例如，对于 Animal 类的 preparing() 等虚函数，可以不用实现它。此时这些虚函数由于没有函数定义体，需要一种特殊的写法来说明，即声明为纯虚函数。

纯虚函数声明的语法格式如下。

```
virtual 返回类型 成员函数名(形参列表) = 0;
```

只要在虚函数声明语句的分号前加上“=0”就可成为纯虚函数。纯虚函数在类中可以只声明，不实现。含有纯虚函数的类称为抽象类。如果类内的所有虚函数都是纯虚函数，则此基类称为纯抽象类。不论是抽象类还是纯抽象类，都不能被用来实例化对象，它们是不完整的，只能作为基类派生其他类。

提示：

① 如果使用抽象类定义对象，会出现编译错误。

② 可以定义抽象类的指针或引用。

对项目 5_6 的 Animal 类进行修改，使之成为抽象类。头文件修改如下。

```
/*****************************************
 * 项目名: 5_8
 * 文件名: animal.h
 * 说　明: 抽象类: 动物类
*****************************************/
#ifndef ANIMAL_H
#define ANIMAL_H

class Animal
{
public:
    virtual void prepairing() = 0;
    virtual void eating() = 0;
    virtual void resting() = 0;
    virtual ~Animal();
};

#endif //ANIMAL_H
```

类实现文件修改如下。

```
/*****************************************
 * 项目名: 5_8
 * 文件名: animal.cpp
 * 说  明: 抽象类: 动物类的实现
*****************************************/
#include "animal.h"

Animal::~Animal()
{
}
```

代码中没有实现纯虚函数,它不需要实现。

此时,main.cpp 文件(见项目 5_6)中的 showLifeProcess3()函数会报错,因为该函数的形参是 Animal 类型的对象,而此时 Animal 是一个抽象类,不能定义对象。

showLifeProcess1()和 showLifeProcess2()函数的形参分别是 Animal 类型的指针和 Animal 类型的引用,并不是 Animal 类型的对象,是可以定义的。它们虽然没有自己类型的对象可以指向(引用),但是可以用于指向(引用)Animal 类的派生类的对象。Lion 类中重新定义并实现了与纯虚函数同原型的成员函数,所以 Lion 类已不再是抽象类,而是可以实例化对象的普通类。

代码中去掉 showLifeProcess3()函数及对该函数的调用后,可以正常执行,运行结果如图 5-10 所示。

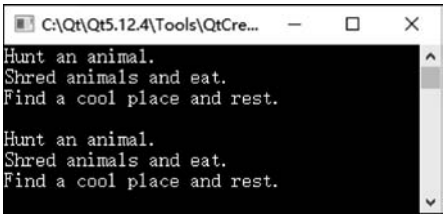

图 5-10 项目 5_8 的运行结果

提示:派生类中要重新声明与纯虚函数同原型的成员函数并实现它,才能用于定义对象,否则该派生类仍然是一个抽象类。

5.4.2 纯虚函数的定义

视频讲解

一般情况下,纯虚函数不用实现。纯虚函数的存在,一是为了给它的所有派生类声明一个统一的接口(各派生类中各自实现纯虚函数);二是使类成为抽象类,以阻止使用它定义对象。

但也可以实现纯虚函数。在某些情况下,可能既想要阻止使用抽象类定义对象,同时又想有实质的内容可以使用,例如,可能会想利用这个纯虚函数提供一段程序代码,让所有派生类或是一部分的派生类共享,以避免这段程序代码在派生类中重复出现。这时候可以在基类中实现纯虚函数的定义。

例如,复制项目 5_8 为项目 5_9,并实现 Animal 类的 prepairing()纯虚成员函数,Animal 类实现文件修改如下。

```
/*****************************************
 * 项目名: 5_9
 * 文件名: animal.cpp
```

```
 * 说　明：抽象类：动物类的实现，实现了纯虚函数
 ********************************************/
#include "animal.h"
#include <iostream>
using namespace std;

Animal::~Animal()
{
}

void Animal::prepairing()
{
    cout <<"Now is prepairing period:"<< endl;
}
```

定义过的纯虚函数可以作为所有派生类完成该函数任务时都要执行的基本动作，被所有(或部分)派生类调用。例如，可修改 Lion 类的 prepairing()函数实现如下，从而既实现了代码复用，又阻止了抽象类定义对象。

```
/*****************************************
 * 项目名：5_9
 * 文件名：lion.cpp
 * 说　明：狮子类实现，调用了基类的纯虚函数
 ********************************************/
//除了修改了以下 prepairing()函数，其他代码同项目 5_6 中的 lion.cpp 文件
void Lion::prepairing()
{
    Animal::prepairing();
    cout <<"Hunt an animal."<< endl;
}
//除了修改了以上 prepairing()函数，其他代码同项目 5_6 中的 lion.cpp 文件
```

运行结果如图 5-11 所示。

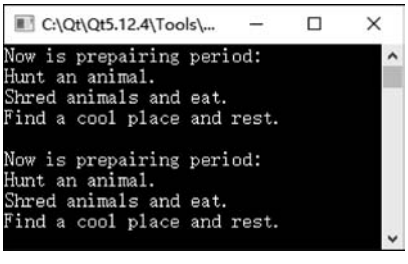

图 5-11　项目 5_9 的运行结果

可见，纯虚函数仍然可以作为抽象类的普通函数进行实现，完成一些在所有派生类实现该纯虚函数时都要完成的共同动作。

5.5 编程实例——猴子选大王

一群猴子(共 number 只)要选新猴王了。选择方法是：先按照 1～number 的顺序给每个猴子一个编号，再将猴子按照编号顺序围坐成一个圆圈(number 号猴子之后是 1 号猴子)，接着从 1 号猴子开始往下数，每数到指定数字(用 m 表示)的猴子，该猴子就要离开圆圈，接着从紧邻的下一只猴子重新从 1 开始往下数，如此直到圈中最后只剩下一只猴子为止，最后的这只猴子就是选出的新猴王。本节将编写程序模拟并展示选新猴王的过程。

创建一个基于 QWidget、带界面的应用程序，设计如图 5-12 所示的界面。

图 5-12 项目 5_10 的界面

由于初始时需要用户设置猴子的总数以及指定数字 m，考虑在主界面显示之前先请用户输入这两个数据，并使用这两个数据对主界面进行一些初始化的工作。因此，主函数代码实现如下。

```
/*****************************************
 * 项目名: 5_10
 * 文件名: main.cpp
 * 说  明: 主函数实现
*****************************************/
#include "widget.h"
#include <QApplication>
#include <QInputDialog>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    int number;
    number = QInputDialog::getInt(nullptr,"初始设置","有多少只猴子?",1,1,100);
    int m;
    m = QInputDialog::getInt(nullptr,"初始设置","数到几的猴子退出?",1,1,100);

    Widget w;
```

```
    w.show();
    w.initial(number,m);

    return a.exec();
}
```

代码中定义的 number 用于接收输入对话框中用户输入的猴子数量，m 用于接收输入对话框中用户输入的指定数，接着再显示主要的界面 w。可以看到，主函数中还调用了自定义窗口类 Widget 中的 initial()成员函数(这是自定义的成员函数，用来完成对窗口显示内容的初始设置，将在随后介绍)。

为了处理的方便，在 Widget 类中声明以下数据成员。

```
private:
    int m;
    QLinkedList<QString> monkeyList;
    QLinkedList<QString>::iterator currentMonkey;
```

其中，m 为指定数(数到 m 时，猴子退出)；链表 monkeyList 表示猴子围成的圆圈，每只未退出的猴子都是链表中的一个节点；链表迭代器 currentMonkey 表示当前正数到的猴子。

为了处理方便，还需在类中添加私有的成员函数 showMonkeyList()，用于在界面的猴子序列区域(多行文本框)中显示猴子序列，代码如下。

```
void Widget::showMonkeyList()          //遍历显示猴子链表
{
    QString str;
    QLinkedList<QString>::iterator iter;
    for(iter = monkeyList.begin();iter!= monkeyList.end();iter++)
    {
        if(iter == currentMonkey)
            str += '*' + *iter + ' ';
        else
            str += *iter + ' ';
    }
    ui->textEdit->setText(str);
    if(monkeyList.size() == 1)
    {
        QString bigBoss = *monkeyList.begin();
        QMessageBox::information(this,"结果","新大王为猴子" + bigBoss);
        this->close();
    }
}
```

依次遍历链表，并将各节点中的内容连接成字符串，然后显示在多行文本框中。如果链表中只有一个节点，则说明此时已找到新大王，弹出显示结果的消息框，之后关闭整个窗口结束程序。

添加公有 initial()成员函数，用于初始化窗口中显示的内容，代码如下。

```
void Widget::initial(int number, int m)
{
```

```
    this->m=m;
    ui->labelInfo->setText("共"+QString::number(number)+"只猴子,数到"
                           +QString::number(m)+"退出");
    for(int i=1;i<=number;i++)    //初始化猴子链表
        monkeyList.append(QString::number(i));
    currentMonkey=monkeyList.begin();
    showMonkeyList();
}
```

该函数的功能为将传入的固定数 m 的值存储于对象内部的数据成员 m 中,设置界面上方标签显示的文字,使用循环初始化猴子链表(将 number 只猴子依次链接到链表),设置当前开始数的位置为第 1 只猴子,然后再调用 showMonkeyList()函数显示初始状态的猴子序列。此处使用链表表示猴子围成的圆圈(当访问到链表结尾时,继续接着从链表头开始访问)。

单击图 5-12 中的“下一只出列的猴子”按钮时,执行的自关联槽定义如下。

```
void Widget::on_pushButton_clicked()
{
    for(int i=1;i<m;i++)
    {
        currentMonkey++;
        if(currentMonkey==monkeyList.end())
            currentMonkey=monkeyList.begin();
    }
    ui->lineEdit->setText(*currentMonkey);     //显示将被删除的猴子
    currentMonkey=monkeyList.erase(currentMonkey);
    if(currentMonkey==monkeyList.end())
        currentMonkey=monkeyList.begin();
    showMonkeyList();
}
```

该槽函数的功能:从当前猴子处往下数,到第 m 只猴子停止(如果中间有已到链表尾的情形,则切换到链表头继续)。此时 currentMonkey 指向的即为当前要出列的猴子节点,将其显示在界面单行文本框中,从链表中删除此猴子节点,然后显示更新后的猴子序列。

程序运行结果如图 5-13 所示。图 5-13(a)为指定共 10 只猴子,数到 3 出列的初始界面;图 5-13(b)为最终得到的结果。

(a) 初始界面　　(b) 最终结果

图 5-13　项目 5_11 的运行效果

本节的例子实际是计算机和数学领域中经典的约瑟夫问题，程序只是模拟了原始的求解过程。还有许多高效率的解法，感兴趣的读者可以参考更多的资料进行了解。

课后习题

一、选择题

1. 声明一个用于求解并返回两个同类型数据中较小值的函数模板，下列写法正确的是(　　)。

A. template < class T > T min(T x,T y);

B. template < class T > min(T x,int y);

C. template < class T > T min(x,y);

D. template < class T > T min(T x,y);

2. 类模板的模板类型参数(　　)。

A. 可以作为成员函数的返回类型　　B. 可以作为数据成员的类型

C. 可以作为成员函数的形参类型　　D. 以上 3 个选项均可

3. 已知函数模板声明为

```
template < class T >
void show(T a)
{
    cout << a << endl;
}
```

下列能正确调用实例化模板函数的语句有(　　)。

① show(5);

② show < int >(5);

③ show(int 5);

A. ①　　B. ①②　　C. ①②③　　D. ②

4. 下列关于模板的叙述中正确的是(　　)。

A. 类模板的主要作用是生成抽象类

B. 函数模板不是函数，在调用时会根据给出的实参类型实例化一个模板函数

C. 类模板不能有数据成员

D. 类模板实例化时，编译器将根据类模板生成一个对象

5. 下列说法中错误的是(　　)。

A. 列表、向量、链表的区别在于内部存储数据项的格式不同

B. 队列是一种先进先出的数据结构

C. 列表、向量、链表中的项都可基于迭代器进行访问，也可基于索引值进行访问

D. 栈是一种先进后出的数据结构

6. 下列说法中错误的是(　　)。

A. Qt 中的列表、向量、链表都是以类模板的形式提供的

B. Qt 中,向量中的数据项是顺序存储的,而列表中的不一定

C. Qt 中的栈派生自向量,队列派生自列表

D. 迭代器用于访问容器中的数据项,只能读取,不能修改

7. 下列关于虚析构函数的描述错误的是(　　)。

A. 没有定义虚析构函数时,系统会自动生成默认虚析构函数

B. 如果基类没有将析构函数声明为 virtual,则在通过基类指针销毁派生类对象时,只会调用基类析构函数,而派生类对象比基类对象多出来的部分则不会被销毁

C. 在类中声明了虚函数后,也应将类的析构函数声明成虚函数

D. 基类的析构函数是虚函数,则派生类的析构函数自动成为虚析构函数

8. 下列关于静态多态和动态多态的描述,错误的是(　　)。

A. 通过将派生类的对象赋值给基类对象,可以实现动态多态

B. 静态多态在程序编译期间就能确定具体的使用形式

C. 动态多态在程序运行期间才能确定要调用的函数

D. 虚函数是用于实现动态多态的一种机制

9. 下列纯虚函数的描述中错误的是(　　)。

A. 纯虚函数是类的成员函数

B. 纯虚函数在类内声明时需要加 virtual 关键字

C. 纯虚函数是返回值等于 0 的函数

D. 声明有纯虚函数的类是抽象类,不能用来定义对象

10. 下列函数原型中,为纯虚函数声明的是(　　)。

A. void func()=0;　　B. virtual void func();

C. virtual void func()=0;　　D. virtual void func(){ }

11. 下列描述中正确的是(　　)。

A. 在虚函数中不能使用 this 指针　　B. 纯虚函数只能声明,不能定义

C. 抽象类是只有纯虚函数的类　　D. 抽象类指针可指向它的派生类对象

12. 下列说法中正确的是(　　)。

A. 虚函数是没有实现的函数

B. 基类中定义了虚函数,派生类的同原型函数自动成为虚函数

C. 纯虚函数不是虚函数

D. 构造函数和析构函数都不能是虚函数

二、程序分析题

1. 程序填空。

```
#include<iostream>
using namespace std;
template<class T>
class A{
private:
    T x;
public:
    A(T _x);
```

```
};

                ①                :x(_x)
{
    cout <<"构造函数被调用"<< endl;
}

int main()
{
    A < int > obj(0);
}
```

2. 下列程序的输出结果为

```
2 0
```

请填空。

```
#include < iostream >
#include < QStack >
using namespace std;
int main()
{
    int a(0),b(1),c(2);
                ①
    vec.push(a);
    vec.push(b);
        ②
    vec.push(c);
    cout << vec.pop()<<' '<< vec.pop()<< endl;
}
```

3. 下列程序的输出结果为

```
ClassA::fun1
ClassB::fun2
```

请填空。

```
#include < iostream >
using namespace std;
class ClassA
{
    public:
        void func1();
                ①
        virtual ~ClassA(){}
};
class ClassB:public ClassA
{
    public:
        void func2();
```

```
        virtual ~ClassB(){}
};
void ClassA::func1()
{
    cout << "ClassA::func1" << endl;
}
void ClassA::func2()
{
    cout << "ClassA::func2" << endl;
}
void ClassB::func2()
{
    cout << "ClassB::func2" << endl;
}
void call(ClassA    ②    )
{
    p.func1();
    p.func2();
}
int main()
{
    ClassB obj;
    call(obj);
    return 0;
}
```

三、编程题

1. 编写一个函数模板，对两个同类型数据（如两个整数、两个浮点数或两个字符串等）比较大小，返回较大的那个，并编写主函数进行测试。

2. 设计一个 DynamicArray 类模板（通用动态数组），包含一个表示数组元素个数的数据成员、一个用于指向动态申请数据空间的指针成员；分别定义成员函数，用于设置、获取指定的数组元素；定义构造函数，用于初始化数据成员以及申请用于存储元素的空间、定义析构函数释放申请的空间。编写并测试该类模板，使之可以用于 int、char、double 等数据类型。

3. 创建如图 5-14 所示的界面。在程序内维护一个字符串队列（QQueue 容器）对象，初始为空，队列中的所有项实时地显示于中间的多行文本框中；单击“进队列”按钮时，将上面文本框中的文字放入队列；单击“出队列”按钮时，从队列中取出最前面的一项，显示在下面的文本框中。

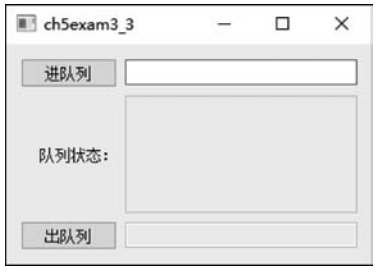

图 5-14 队列操作界面

4. 定义一个抽象基类 BaseShapes，其中包含公有访问权限的纯虚成员函数 area()和虚析构函数；定义两个类 Square、Circle 为抽象基类 BaseShapes 的派生类，其中 Square 类新引入数据成员长 length 和宽 width，Circle 类中新引入数据成员半径 radius，并分别实现成员函数 area()；在 main()函数中定义基类指针，并实现通过它调用各个类对

象的 area()函数。

四、思考题

1. 现实世界中也经常会遇到栈、队列等组织形式，你能根据自己的生活所见或专业所学举出一些它们的例子么？

2. 类模板和模板类是否是同一个概念，如果不是，它们之间有何联系和区别呢？

实验 5　多态的实现与容器的使用

一、实验目的

1. 掌握模板的定义与使用。

2. 熟悉 Qt 中常见的容器。

3. 掌握虚函数、抽象类等的概念与使用。

二、实验内容

1. 编写一个函数模板，求数组中最小的那个元素并返回，编写主函数并使用整型、浮点型和字符串 QString 类型进行测试。

2. 定义栈类模板(要求自己编写，非 Qt 中提供的已有类)，要求有存储数据项的数据成员、用于指示当前栈顶位置的数据成员、用于弹出栈顶元素的成员函数、用于压入栈的成员函数、用于输出栈中各个项内容的成员函数及必要的构造函数等。编写主函数，使用整型和字符串类型进行测试。

3. 设计如图 5-15 所示的界面，并完成以下要求的功能。

图 5-15　实验 3 内容界面

(1) 在自定义窗口类中添加一个 QList < QString >类型的数据成员 list。

(2) 每单击一次“在列表中添加上述文字”按钮，就在 list 列表中添加一项，内容为界面中单行文本框中输入的文字；同时将该项内容追加显示在左侧 listWidget 中。

(3) 单击“清空左侧内容”按钮时，清空左侧的显示(列表中的项不修改)。

(4) 单击“重新显示已有列表”按钮时，首先清空左侧的显示，然后将数据成员 list 中的内容显示在左侧。

4. 定义一个 BaseClass 类，包括成员函数 fn1()和 fn2()，将 fn1()声明为虚函数；由 BaseClass 类派生出 DerivedClass 类，也有同原型成员函数 fn1()和 fn2()；在主函数中定义

DerivedClass 类型的对象，由 BaseClass 类型的指针来指向，通过对象名和指针分别调用 fn1() 和 fn2()函数，观察运行结果。

5. 定义一个基类 BaseClass，包含虚析构函数；由它派生出 DerivedClass 类，包含析构函数；在主程序中定义一个 BaseClass 的指针 pa，并将其指向一个由 new 运算符申请的 DerivedClass 对象空间，然后通过 delete 运算符释放该 pa 指针指向的内容，观察虚析构函数是如何执行的(并试着将 BaseClass 的虚析构函数改为普通析构函数，观察运行结果有何不同)。