第 11 章　范型和容器

本章将会更为深入地讲解范型这一主题。所谓范型(generic)，是指那些能够像操作基本类型一样轻松操作对象的类和函数。Qt 容器类是范型类，也是基于模板的范型类，同时我们还会看到列表、集合和映射的用法。本章还会讨论重载运算符(overloaded operator)、托管容器(managed container)和隐式共享(implicit sharing)。

11.1　范型与模板

C++支持四种不同的类型：

- 基本类型：int, char, float, double 等
- 指针
- 类和结构的实例
- 数组

因为这四种不同类型之间不存在公共基类型，所以如果不使用模板(template)，要编写能够对多种类型进行操作的范型函数和类将会非常困难。模板为 C++编译器提供了一个途径，能够为带有参数化类型和相同行为的类和函数生成多个版本。模板用关键字 template 以及用尖括号<>包围的模板参数进行区分。

与函数参数不同，模板参数不仅可以传递变量和值，还可以传递类型表达式。

```
template <class T > class String { ... };
template <class T, int max > Buffer { ...
    T v[max];
};
String <char> s1;
Buffer <int, 10> intBuf10;
```

11.1.1　函数模板

函数模板用来创建能够按照相同的模式进行工作的类型检查函数。示例 11.1 定义了一个模板函数，该函数通过重复地使用 operator *=来生成一个类型为 T 的值的 exp 次幂。

示例 11.1　src/templates/template-demo.cpp

```
[ . . . . ]

template <class T> T power (T a, int exp) {
  T ans = a;
  while (--exp > 0) {
    ans *= a;
  }
  return (ans);
}
```

程序员网址导航www.daocode.com

再次说明的是，编译器必须执行额外的工作以提供 C++的这一便利特性。编译器会扫描你的代码并基于函数调用时所提供的参数类型生成所需的不同版本的各个函数体，这样一来，诸如示例 11.2 所示的所有调用才能在编译时被解析。尽管模板参数中使用的关键字是 class，T 还是可以用作一个类或者基本类型。在这个例子中，对类型 T 唯一的限制是它必须是一个定义了 operator*=的类型。

示例 11.2　src/templates/template-demo.cpp

```
[ . . . . ]


int main() {
  Complex z(3,4), z1;
  Fraction f(5,6), f1;
  int n(19);
  z1 = power(z,3);                                    1
  f1 = power(f,4);                                    2
  z1 = power<Complex>(n, 4);                          3
  z1 = power(n,5);                                    4

}
```

1 第一次实例化，T 是一个复数。

2 第二次实例化，T 是一个分数。

3 如果实际参数不够“具体化”，则显式地提供模板参数，其结果是调用一个已经实例化的函数。

4 此语句会调用哪一个版本的函数呢?

每当编译器看到一个特定参数类型的组合首次用于一个模板函数时，就称此模板进行了实例化。随后用到的 power(Complex, int)和 power(Fraction, int)将会被转化为普通的函数调用。

11.1.1.1　练习：函数模板

1. 完成示例 11.2。特别是，编写一个通用的 Complex 类和 Fraction 类，并修改 main()函数，使其能够使用这两个类来正常工作。
2. 基于示例 5.13 编写一个模板函数 swap()。编写客户代码对其进行深入测试。
3. 是否存在让模板函数 swap()无法正常工作的类型?
4. 指出模板函数 swap()中参数的一些具体限制。

11.1.2　类模板

如同函数一样，类也可以使用参数化的类型。类模板主要用来生成数据的通用容器，其参数能够指明容器中的内容。所有的 Qt 容器类以及标准模板库(Standard Template Library，STL)中的所有容器类都是参数化的。参数就是问题“容器中有什么”的答案。图 11.1 给出了两个模板类的 UML 框图。

UML 类图把模板参数放在类框右上角的一个虚线框中。示例 11.3 中包含了这些类的定义。

示例 11.3 src/containers/stack/stack.h

```
[ . . . . ]

#include <QDebug>
template<class T> class Node {
 public:
    Node(T invalue): m_Value(invalue), m_Next(0) {}
    ~Node() ;
    T getValue() const {return m_Value;}
    void setValue(T value) {m_Value = value;}
    Node<T>* getNext() const {return m_Next;}
    void setNext(Node<T>* next) {m_Next = next;}
 private:
    T m_Value;
    Node<T>* m_Next;
};

template<class T> Node<T>::~Node() {
    qDebug() << m_Value << " deleted " << endl;
    if(m_Next) {
        delete m_Next;
    }
}

template<class T> class Stack {
 public:
    Stack(): m_Head(0), m_Count(0) {}
    ~Stack<T>() {delete m_Head;} ;
    void push(const T& t);
    T pop();
    T top() const;
    int count() const;
 private:
    Node<T>* m_Head;
    int m_Count;
};
```

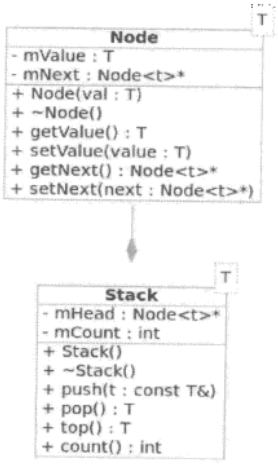

图 11.1 基于模板的栈

所有模板的定义（类和函数）都必须出现在头文件中，这是因为编译器需要用这些定义来根据模板声明生成代码。

值得注意的是，示例 11.3 和示例 11.4 中所需的模板声明代码 `template<class T>`会放在每个类或函数的定义之前，它们的名称中都有一个模板参数。

示例 11.4 src/containers/stack/stack.h

```
[ . . . . ]


template <class T> void Stack<T>::push(const T& value) {

    Node<T>* newNode = new Node<T>(value);
    newNode->setNext(m_Head);
    m_Head = newNode;
    ++m_Count;
}
```

程序员网址导航www.daocode.com

```
template <class T>  T Stack<T>::pop() {
    Node<T>* popped = m_Head;
    if (m_Head != 0) {
        m_Head = m_Head->getNext();
        T retval = popped->getValue();
        popped->setNext(0);
        delete popped;
        --m_Count;
        return retval;
    }
    return 0;
}
```

对象创建通常是在模板函数 push()中处理的。Node<T>类的析构函数会递归地删除 Node 指针，直到 m_Next 指针为零[①]。使用这种方法来控制 Node<T>对象的创建和析构使得 Stack<T>能完全地管理其动态内存。示例 11.5 中含有一些用来说明上述各个类的客户代码。

示例 11.5　src/containers/stack/main.cpp

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

int main() {
    Stack<int> intstack1, intstack2;
    int val;
    for (val = 0; val < 4; ++val) {
        intstack1.push(val);
        intstack2.push(2 * val);
    }
    while (intstack1.count()) {
        val = intstack1.pop();
        qDebug() << val;
    }
    Stack<QString> stringstack;
    stringstack.push("First on");
    stringstack.push("second on");
    stringstack.push("first off");
    QString val2;
    while (stringstack.count()) {
        val2 = stringstack.pop();
        qDebug() << val2;
    }
    qDebug() << "Now intstack2 will self destruct.";
    return 0;
}
```

运行这个程序时，应当可以看到下列输出。

```
3 deleted
3
2 deleted
2
```

① 对一个指针的 delete 操作将会自动地激活与此指针相关联的析构函数。

```
1 deleted
1
0 deleted
0
first off deleted
"first off"
second on deleted
"second on"
First on deleted
"First on"
Now intstack2 will self destruct.
6 deleted
4 deleted
2 deleted
0 deleted
```

注意

因为每个 Q_OBJECT 都需通过 moc 为其生成代码，而且 moc 并未智能到知道如何生成模板类的特化，故而不允许使一个已经标记为 Q_OBJECT 的类再次成为模板类。

11.1.3 练习：范型与模板

1. 将 Stack 的函数定义放在一个单独的文件中(stack.cpp)，适当修改工程文件，然后编译并创建此应用程序。解释该程序的输出结果。
2. 这个应用程序的普适性如何？例如，类 T 必须满足什么条件才能正常工作？
3. Stack<T>的大小有何限制？
4. 编写一个模板类 Queue<T>以及用来测试它的客户代码。

11.2 范型，算法和运算符

通过重载运算符，可以为类定义与基本类型一致的常规接口成为可能。许多范型算法充分利用了这种优势，使用常规运算符来进行基本的函数操作，例如比较等。

11.2.1 qSort

qSort()函数是一个使用堆分类算法进行实现的范型方法[①]。示例 11.6 中给出了使用此函数如何将其用于两个类似但不相同的容器的方法。

qSort()可以应用到任何 Qt 容器，只要容器内的对象拥有公共接口 operator<()和 operator==()。基本数值类型的容器也可以使用此函数进行排序。

示例 11.6 src/containers/sortlist/sortlist4.cpp

```
#include <QList>
#include <QtAlgorithms>   // for qSort()
#include <QStringList>
#include <QDebug>
```

① Wikipedia 中有一篇非常好的介绍堆分类算法的文章，参见 http://en.wikipedia.org/wiki/Heapsort。

程序员网址导航www.daocode.com

```
class CaseIgnoreString : public QString {
public:
    CaseIgnoreString(const QString& other = QString())
    : QString(other) {}
    bool operator<(const QString & other) const {
        return toLower() < other.toLower();
    }
    bool operator==(const QString& other) const {
        return toLower() == other.toLower();
    }
};

int main() {
    CaseIgnoreString s1("Apple"), s2("bear"),
                     s3 ("CaT"), s4("dog"), s5 ("Dog");

    Q_ASSERT(s4 == s5);
    Q_ASSERT(s2 < s3);
    Q_ASSERT(s3 < s4);

    QList<CaseIgnoreString> namelist;

    namelist << s5 << s1 << s3 << s4 << s2;                    1

    qSort(namelist.begin(), namelist.end());
    int i=0;
    foreach (const QString &stritr, namelist) {
        qDebug() << QString("namelist[%1] = %2")
                        .arg(i++).arg(stritr) ;
    }

    QStringList strlist;
    strlist << s5 << s1 << s3 << s4 << s2;                     2

    qSort(strlist.begin(), strlist.end());
    qDebug() << "StringList sorted: " + strlist.join(", ");
    return 0;
}
```

1 毫无规律地插入所有项。

2 值集合保存的是 `QString`，但是向其中添加了 `CaseIgnoreString`，因此需要进行转换。

`operator<<()` 在 C 语言中是左移运算符，但在 `QList` 类中对其进行了重载，现在它的作用是向列表中添加一个项。

示例 11.7 给出了这个程序的输出结果。

示例 11.7　src/containers/sortlist/sortlist-output.txt

```
namelist[0] = Apple
namelist[1] = bear
namelist[2] = CaT
namelist[3] = dog
namelist[4] = Dog
StringList sorted: Apple, CaT, Dog, bear, dog
```

需要注意的是，CaseIgnoreString 对象添加到 QStringList 之后，其排序是区分大小写的。这是因为 CaseIgnoreString 在添加到 strlist 中时必须转换为 QString，因此当对 strlist 的项进行比较时，它们是作为 QString 比较并且排序是大小写敏感的。

11.2.1 练习：范型，算法和运算符

1. QStringList 是“写时复制”对象的值容器，从某种意义上来说，它像是一个指针容器，但是相对来说更加灵活。示例 11.6 中，CaseIgnoreString 会被添加到 QStringList，该过程需要进行(类型)转换。此时需要实际进行字符串数据的复制操作吗？为什么？
2. 向如图 11.2 所示的 ContactList 类中分别添加更多的函数 operator+=, operator-=, add()和 remove()。编写客户代码对这些函数进行测试。

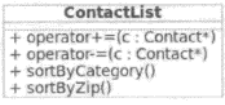

图 11.2 ContactList UML

11.3 有序映射示例

前面曾经提到，QMap 是一个关联数组，能够在添加和删除项时维护键的分类顺序。基于键的插入和删除操作非常快速，效率可达 $\lg n$，且迭代也是按照键的顺序进行的。

QMap 是一个值容器，但指针是简单值，因此可以使用 QMap 来存储分配在堆中的 QObject 的指针。尽管如此，在默认情况下，值容器不会管理堆对象，因此为了避免内存泄漏，必须确保这些对象在合适的时候进行了销毁操作。图 11.3 描述了一个类，它对 QMap 进行了扩展，以包含有关课本的信息。继承了 QMap 之后，QMap 的所有公共接口都变成了 TextbookMap 类公共接口的一部分，而我们仅仅添加了一个析构函数和两个便于添加与显示容器中课本(Textbook)的函数。

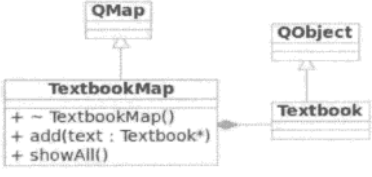

图 11.3 TextbookMap

TextbookMap 由以 ISBN 为键和 Textbook 指针为值的键值对（<key, value>）构成。示例 11.8 给出了该类的定义。

示例 11.8 src/containers/qmap/textbook.h

```
#ifndef _TEXTBOOK_H_
#define _TEXTBOOK_H_

#include <QObject>
#include <QString>
#include <QMap>

class Textbook : public QObject {
    Q_OBJECT
  public:
    Textbook(QString title, QString author, QString isbn, uint year);
[ . . . . ]
private:
    uint m_YearPub;
    QString m_Title, m_Author, m_Isbn;
```

程序员网址导航www.daocode.com

```
};

/* Managed collection of pointers */
class TextbookMap : public QMap<QString, Textbook*> {
  public:
    ~TextbookMap();
    void add(Textbook* text);
    QString toString() const;

};
#endif
```

示例 11.9 中，析构函数使用对 QMap 的 values() 使用 qDeleteAll()，以删除所有的指针。这对于一个值容器来管理其对象是有必要的。

示例 11.9　src/containers/qmap/qmap-example.cpp

```
[ . . . . ]

TextbookMap::~TextbookMap() {
    qDebug() << "Destroying TextbookMap ..." << endl;
    qDeleteAll(values());
    clear();
}

void TextbookMap::add(Textbook* text) {
    insert(text->getIsbn(), text);
}

QString TextbookMap::toString() const {
    QString retval;
    QTextStream os(&retval);
    ConstIterator itr = constBegin();
    for ( ; itr != constEnd(); ++itr)
        os << '[' << itr.key() << ']' << ": "
        << itr.value()->toString() << endl;
    return retval;
}
```

不止一种的遍历映射方法

遍历映射中的每个元素并获得其值，可以有很多种方法。示例 11.9 中用到了 C 语言风格的循环和 STL 风格的迭代器。在单核计算机上，这一做法会非常有效。此处给出的代码是使用 Qt 的 foreach 循环对这一方法的重写，看起来也很不错，但是在运行时，它会创建一个临时列表并需要 $n \log_2 n$ 次的树查找运算，因而在空间或者时间上并不会非常高效。

```
void TextbookMap::showAll() const {
    foreach (QString key, keys()) {
        Textbook* tb = value(key);
        cout << '[' << key << ']' << ":"
             << tb->toString() << endl;
    }
}
```

重要的是要明白，正如示例 11.10 中看到的客户代码那样，当从 TextbookMap 中通过 remove()移除一个指针时，会同时移除对该指针的管理责任。一旦将其移除，你就要承担删除其指针的职责!换句话说,客户代码很容易产生一些内存泄漏。同样的问题也存在于 QMap 的其他成员函数中，例如 QMap::erase()和 QMap::take()。通过使用 TextbookMap 的成员函数隐藏这些危险的 QMap 函数可以减少这些问题,TextbookMap 版本能够移除和删除或者重父化(reparent)那些不再需要 Textbook 的指针。另外一种(可能较为安全的)做法是使用 private 派生而不是使用 public 派生。这样，TextbookMap 的公有接口将可能会只包含那些由你仔细定义过的、安全的公有成员函数。

示例 11.10 src/containers/qmap/qmap-example.cpp

```
[ . . . . ]

int main() {
    Textbook* t1 = new Textbook("The C++ Programming Language",
        "Stroustrup", "0201700735", 1997);
    Textbook* t2 = new Textbook("XML in a Nutshell",
        "Harold","0596002920", 2002);
    Textbook* t3 = new Textbook("UML Distilled",
        "Fowler", "0321193687", 2004);
    Textbook* t4 = new Textbook("Design Patterns", "Gamma",
        "0201633612",1995);
    {                                                          1
        TextbookMap m;
        m.add(t1);
        m.add(t2);
        m.add(t3);
        m.add(t4);
        qDebug() << m.toString();
        m.remove (t3->getIsbn());                              2
    }                                                          3
    qDebug() << "After m has been destroyed we still have:\n"
        << t3->toString();
    return 0;
}
```

1 出于演示目的而引入的内部块。

2 移除但未删除。

3 块的结束——局部变量都被销毁了。

从示例 11.11 的输出结果可以看出，当 TextbookMap::ShowAll()函数对容器进行迭代时，各个 Textbook 已经按照 ISBN(键)的顺序进行了放置。

示例 11.11 src/containers/qmap/qmap-example-output.txt

```
src/containers/qmap> ./qmap
[0201633612]:Title: Design Patterns; Author: Gamma; ISBN: 0201633612;
Year: 1995
[0201700735]:Title: The C++ Programming Language; Author: Stroustrup;
ISBN: 0201700735; Year: 1997
[0321193687]:Title: UML Distilled; Author: Fowler; ISBN: 0321193687;
```

程序员网址导航www.daocode.com

```
Year: 2004
[0596002920]:Title: XML in a Nutshell; Author: Harold; ISBN:
0596002920; Year: 2002
Destroying TextbookMap ...
After m has been destroyed we still have:
Title: UML Distilled; Author: Fowler; ISBN: 0321193687; Year: 2004
src/containers/qmap>
```

11.4　函数指针和仿函数

仿函数(functor)是可被广泛调用的结构。那些可较容易地转换为函数指针(function pointer)的常规 C 和 C++函数，都可以归纳为这一类型。泛型算法通常会通过重载来接受各类可调用结构作为参数。示例 11.12 给出了如何通过函数指针来间接地调用函数。

示例 11.12　src/functors/pointers/main.cpp

```
#include <QtGui>

QString name() {
    return QString("Alan");
}

typedef QString (*funcPtr)();                                    1
Q_DECLARE_METATYPE(funcPtr);                                     2

int main() {
    qRegisterMetaType<funcPtr>("funcPtr");                       3
    funcPtr ptr = name;                                          4

    QString v = (*ptr)();                                        5
    qDebug() << v << endl;                                       6
}
```

1 一个可以返回 `QString` 且不带参数的函数。
2 声明，因而可以用于 `QVariant` 中。
3 注册器，因而可以用于队列中的信号参数。
4 赋值给指向函数的指针的函数名称。
5 通过解引用函数 `ptr` 调用一个方法。
6 输出“`Alan`”。

指向函数的指针经常用在 C 的回调函数中，或者用于对特定事件进行响应时调用的函数中。在 C++中使用面向对象的、基于模板的各种机制也是可能的。使用这种方式，在编译时就可以指定参数的类型和返回值的类型，从而让它们成为类型安全的。

从它可被解引用并可像函数一样调用的意义上讲，C++中的仿函数是一个行为类似于指针的可调用对象。遵守 RT1[①]或后续版本的 C++的标准库，在头文件<functional>中为这些仿函数提供基本的类类型(class type)。C++的函数调用运算符提供了部分让对象可以像函数一样工作的

① C++技术报告 1(C++ Technical Report 1，TR1)是一份草稿性质的文档，含有对 C++标准函式库建议追加的项目，这些项目很有可能包含在下一个官方标准中。详细信息请参阅 Wikipedia 文章：http://en.wikipedia.org/wiki/C%2B%2B_Technical_ Report_1。

语法糖(syntactic sugar)[①]std::unary_function 和 std::binary_function 类型会为 C++ 的仿函数提供额外的类型信息。它们都是一些可扩展的参数化基类类型，可用于 QtAlgorithms、C++ STL 以及 Qt 的 Concurrent 库中。示例 11.13 展示了如何定义可用于替代函数指针的仿函数。

示例 11.13 src/functors/operators/functors.h

```
[ . . . . ]
class Load : public std::unary_function<QString, QImage> {          1
public:
    QImage operator() (const QString& imageFileName) const {
        return QImage(imageFileName);
    }
};
class Scale {
    public:
    typedef QImage result_type;                                      2
    QImage operator() (QImage image) const {
        for (int i=0; i<10; ++i) {
            QImage copy = image.copy();
            copy.scaled(100, 100, Qt::KeepAspectRatio);
        }
        if (image.width() < image.height()) {
            return image.scaledToHeight(imageSize,
                                        Qt::SmoothTransformation);
        }
        else {
            return image.scaledToWidth(imageSize,
                                       Qt::SmoothTransformation);
        }
    }
};
class LoadAndScale : public std::unary_function<QString, QImage> { 3
public:
    Scale scale;
    QImage operator() (const QString& imageFileName) const {
        QImage image(imageFileName);
        return scale(image);
    }
};
[ . . . . ]
```

1 定义 result_type。

2 仿函数对象所需的一个属性(trait)。

3 还是定义 result_type。

示例 11.14 中，创建了 LoadAndScale 的一个临时实例并将其传递给 QtConcurrent 算法，该算法的映射函数进行了重载，以便可以接受函数指针和它的映射函数的 std::unary_function 对象。17.2 节中将会详细讨论 QtConcurrent。

① 语法糖表示“语法中的糖分”。该术语由 Peter J. Landin 创造，是指向一门计算机语言语法中添加附加物或附加成分，它不会影响语言的功能，但能使人类在使用该语言时显得“更甜美”一些。语法糖为程序设计人员提供了一种编写程序的替代方式，这一方式更具实用性和更有助于形成较好的程序设计风格，或者可使代码读起来更自然。但是，语法糖不会影响形式上的可表达性，也不会让语言拥有某种新功能。——译者注

程序员网址导航www.daocode.com

示例 11.14　src/functors/operators/imagefunctor.cpp

```
[ . . . . ]

        connect(m_futureWatcher, SIGNAL(progressValueChanged(int)),
                this, SIGNAL(progressCurrent(int)));
        emit statusMessage("Loading and Transforming in parallel");
        m_futureWatcher->setFuture(QtConcurrent::mapped(files,
                                                        LoadAndScale()));
```

11.5　享元模式：隐式共享类

与 Java 不同，C++没有垃圾收集机制(garbage collection)。所谓的垃圾收集，实际是一个对不再引用的堆内存进行恢复的线程。垃圾收集会在 CPU 相对空闲或者内存不足时开始运行。当一个对象不再被引用后，它会被删除掉，这样，它所占用的内存就可用于其他对象。它具有减少开发人员工作，使其无须担心内存泄漏的好处[①]。但是，它也肯定会加重 CPU 的工作负担。

接下来的例子会给出一种通过引用计数(reference counting)将垃圾收集构建到类的设计中去的方法。引用计数是一个资源共享的例子。无论是从开发人员的角度还是从 CPU 时间的角度看，它都被认为是要比垃圾收集管理堆更为有效的方法。

注意

如果打算修改一个对象(例如，调用了一个非 const 成员函数)，且其引用计数值要比 1 大，那么首先需要的是将其分离，以便它不再共享。

对基于引用计数方式工作的隐式共享类，要避免删除那些共享的已管理对象。使用这个类的客户无须关注其引用计数或者内存的指针。

QString，QVariant 和 QStringList 都是采用这种方式实现的，也就是说，通过值进行参数传递和函数返回是相当快的。如果需要在一个函数内部修改存储在容器内部的值，可以通过引用来传递，而不是通过指针进行传递。

如果通过 const 引用来传递，速度可能还要更快一些，这样会允许 C++把整个复制操作都进行优化。使用 const 引用时，函数无法对引用进行任何修改，也就不会发生自动转换。

享元模式[②]

为了避免对同一对象的多个副本进行存储，在很多情况下，都可以在实际对象出现的地方用一个轻量级封装器(wrapper)来进行代替。封装器类会包含一个指向共享数据的指

① 实际上，Java 开发人员难免有些担心并总是会试图用各种技巧来尽可能少地创建堆对象。他们还会用多种方法来尽可能经常地强制运行垃圾收集程序(并且，这必然会消耗更多的 CPU 周期)。

② Flyweight 一词在拳击比赛中是指最轻量级，即“蝇量级”或者“羽量级”的意思。本书将其译作“享元模式”是为了能够更确切地反映该模式本意：享元模式以共享的方式高效地支持大量的细粒度对象。享元对象之所以可以实现共享，关键是要能够区分内蕴状态(Internal State)和外蕴状态(External State)。内蕴状态存储在享元对象内部，并且不会随环境的改变而改变，故而可以共享内蕴状态；外蕴状态则会随环境的改变而改变，因此也就无法共享。享元对象的外蕴状态必须由客户保存，并在享元对象创建之后，在需要使用时再传入到享元对象的内部。外蕴状态与内蕴状态是相互独立的。详情可参阅 http://tech.it168.com/KnowledgeBase/Articles/8/d/4/8d42715fe1b5939224f3823fcce731da.htm 一文。——译者注

针，而不是对数据的副本进行维护。通过这种方式工作的那些类就是对享元模式(Flyweight Pattern)的实现，有时也称为桥接模式(Bridge Pattern)或者私有实现模式(Private Implementation Pattern)。

在查看 Qt 源代码时，你无疑会注意到，file_p.cpp 和 file_p.h 文件中包含了大多数 Qt 类的实现细节。在改变一个类的实现时，这种模式可有助于确保源代码及其二进制之间的兼容性。此外，这种模式还可用来实现隐式共享，其中的数据可被多个实现共享。

桥接模式的灵活性是有代价的：其代码会更复杂(需要管理的类要两倍以上)，而且必须要实现一个和原始类有同样接口的封装器。

为了能够获得自己的轻量级隐式共享，你可以编写自己的引用计数代码，或者复用这两个 Qt 类：QSharedData 和 QSharedDataPointer。

QSharedData 提供了一个公有的 QAtomicInt ref 成员，可用作引用计数值。QAtomicInt 则提供一个 deref()操作，由 QSharedDataPointer 用来减少和测试引用计数值，以确定它是否可以安全地删除共享数据。QSharedDataPointer 会根据共享数据的复制或者分离来更新共享数据的引用计数值。

下一个例子，如图 11.4 所描绘，从一个相对常规的非共享的 MyString 类开始，它是利用动态字符数组实现的字符串，如示例 11.15 所示。

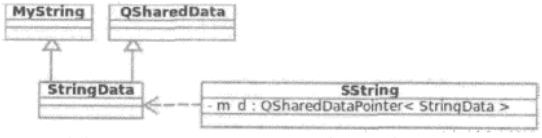

图 11.4 QSharedData 的私有实现示例

示例 11.15 src/mystring/shareddata/mystring.h

```
#ifndef MYSTRING_H
#define MYSTRING_H
#include <iostream>
class MyString {
public:
    MyString(const MyString& str);
    MyString& operator=(const MyString& a);
    MyString();
    MyString(const char* chptr);
    explicit MyString(int size);
    virtual ~MyString();
    friend std::ostream& operator<<(std::ostream& os, const MyString& s);
    int length() const;
    MyString& operator+= (const MyString& other);
    friend MyString operator+(const MyString&, const MyString&);
protected:
    int   m_Len;
    char* m_Buffer;                                          1
    void  copy(const char* chptr);
};
#endif        //  #ifndef MYSTRING_H
```

1 指向动态数组起始点的指针。

程序员网址导航www.daocode.com

示例 11.16 通过增加引用计数功能来扩展 MyString 类，这是私有实现类。

示例 11.16　src/mystring/shareddata/stringdata.h

```
[ . . . . ]
class StringData : public QSharedData, public MyString {
public:
    friend class SString;
    StringData() {}
    StringData(const char* d) : MyString(d) {}
    explicit StringData(int len) : MyString(len) {}
    StringData(const StringData& other)
            : QSharedData(other), MyString(other) {}
};
[ . . . . ]
```

如示例 11.17 所示，隐式共享类 SString 是使用 QSharedDataPointer 来实现“写时复制”功能的一个类示例。

示例 11.17　src/mystring/shareddata/sstring.h

```
[ . . . . ]
class SString {
public:
    SString();
    explicit SString(int len);
    SString(const char* ptr);
    SString& operator+= (const SString& other);
    int length() const;
    int refcount() const {return m_d->ref;}
    friend SString operator+(SString, SString);
    friend std::ostream& operator<< (std::ostream&, const SString&);
[ . . . . ]
private:
    // Private Implementation Pattern
    QSharedDataPointer<StringData> m_d;

};
[ . . . . ]
```

SString 的公有方法委托给了 StringData。每当 m_d 以非 const 的方式解引用时，实际的共享数据就会被自动复制。示例 11.18 给出了修改共享元时 refcount()会减小引用值的一个范例。

示例 11.18　src/mystring/shareddata/sharedmain.cpp

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

void passByValue(SString v) {
    cout << v << v.refcount() << endl;
    v = "somethingelse";
```

1

```
    cout << v << v.refcount() << endl;                    2
}

int main (int argc, char* argv[]) {
    SString s = "Hello";
    passByValue(s);
    cout << s << s.refcount() << endl;                    3
}
```

1 引用数值等于 2。

2 引用数值等于 1。

3 引用数值等于 1。

QExplicitlySharedDataPointer 与 QSharedDataPointer 一样，但每当需要一个副本时都必须对 QSharedData 进行显式调用 detach()。

无论是出于隐式共享的原因还是出于其他原因，大多数的 Qt 类都实现了享元模式。只有当复制真正修改时，才会复制那些收集的对象，并和原来的容器相脱离。这时才产生和付出时间/内存方面的代价。

11.6 练习：范型

本练习要求给出一些数据结构来跟踪符号之间的关系。一个关系是元素满足下列两个属性的符号集合 S 上的一个布尔运算符 frop：

1. 自反性——对于 S 内的任意符号 s，s *frop* s 总是真（true）。
2. 对称性——对于 S 内的任意符号 s 和 t，如果 s *frop* t 为真，那么 t *frop* s 也为真。

frop 类似于社交网站上的布尔运算符 is-friends-with。is-friends-with 不具有传递性，一个人不会自动和自己的所有朋友的朋友成为朋友[①]。

在这个问题中，可构建一个数据结构来对符号之间的关系进行排序。每一个符号都是一个随机的字母和数字构成的字符串。这个数据结构应当使用一个或者多个 QSet，QMap 或者 QMultiMap。

1. 编写一个程序，重复让用户输入关系或者命令，并不断跟踪所能见到的字符对之间的关系。接口应该相当简单：用户输入要分析的字符串并由分配函数 processLine() 加以处理。换句话说，processLine()应当希望字符串具有下列形式：
   a. 要在两个字符串之间加一个 frop：string1 = string2。
   b. 列出 string1 的朋友：string1(该行中没有符号 =)。
2. 添加另外一个函数 takeback(int n)，其中 n 代表第 *n* 个断言。如果第 *n* 个断言添加了 frop，那么该函数应当“撤销”(undo)那个断言，应确保关系的完整性。在

① 传递性(transitivity)是没有给出的布尔运算符的第三个属性——对于 S 内的任意符号 s，t 和 u，如果 s *op* t 且 t *op* u 都是真，那么 s *op* u 也为真。如果布尔运算符 *op* 具有自反性、对称性和传递性，那么它就是一个等价关系(Equivalence Relation)。

程序员网址导航www.daocode.com

运行此函数之后，给出涉及的两个符号(string1 和 string2)更新后的朋友清单。让 processLine()函数扫描形如“**takeback** *n*”的信息，并随之调用 takeback()函数作为响应。

11.7　复习题

1. 解释模板参数和函数参数之间的一个重要不同之处。
2. 实例化某个模板函数意味着什么？描述一种实例化的方法。
3. 通常来说，需要将模板定义放在头文件中。为什么？
4. 有些编译器支持 export。这有什么作用？
5. Qt 的容器类用来存储值类型。哪些东西不适于使用值集合中的值进行存储？
6. 哪些容器提供从键到值的映射？至少列出并描述两种，然后说明它们之间的不同之处。
7. 一个容器“管理”其堆对象的含义是什么？一个包含堆对象指针的容器怎样才能变成“托管容器”？
8. 至少举出三个实现了享元模式的 Qt 类示例。
9. 在定义一个类模板时，应当怎样将其代码分别放到头文件(.h)和实现文件(.cpp)中？解释你的答案。