第4章 列　　表

只要有可能，就应当使用列表而不是数组。本章将讲解容器，并探讨在列表中进行分组的各种方式，还会讲解如何迭代列表。

4.1 容器简介

在许多情况下都需要处理若干事物的集合。在C语言这样的编程语言中，最基本的方法就是使用数组(参见21.4节)来存储这种集合。但在C++中，数组被看成是“邪恶的”。

下面是应该避免在C++中使用数组的一些理由。

- 编译器和运行时系统都不会检查数组下标是否位于正确的范围之内。
- 使用数组的程序员有责任编写额外的范围检查代码。
- 数组的大小可以是固定不变的，或者必须使用堆中的动态内存。
- 对于堆数组，程序员有责任确保在所有可能的情况下当数组销毁时都要正确地释放内存。
- 为此，需要深入理解C++以及异常，特别是发生异常时的底层处理机制。向数组插入、预分配或追加元素都是费时的操作(在运行时和开发时都是如此)。

C++标准库和Qt都为程序员提供了在需要时安全地重新调整数组大小的能力，并会执行范围检查。`std::list`和`QList`是在各自库中最基本的泛型容器。在接口方面它们类似(接口即客户代码使用它们的方式)。二者的不同在于实现(即运行时它们的行为方式)。这两个库还提供了其他几个泛型容器，它们针对特定类型的应用进行了优化。

这里使用术语“泛型容器”，其原因有如下两个。

1. 泛型是指能够接收模板(参见11.1节)参数的类或者函数，这样它们就能够用于不同类型的数据。
2. 容器(参见6.8节)是指能够包含其他对象的对象。

为了使用泛型容器，客户代码必须包含一个能够回答如下问题的声明：什么的容器？例如

```
QList<double>  doubleList;
QList<Thing>  thingList;
```

`QList`支持许多操作。对于复用的任何类，推荐阅读API文档，以全面了解它的完整功能。对于单一的函数调用，可以有多种方式来添加、删除、交换、查询、清除、移动、定位和计数函数中的项。

4.2 迭代器

在容器中存储元素之后，迟早都要遍历容器对其中的每一个元素进行某种操作。迭代器(iterator)是一个提供对容器中的每一个元素进行间接访问的对象，它专门被设计用在循环之中。

Qt 支持如下几种风格的迭代。

- Qt 风格的 foreach 循环。类似于 Perl 和 Python 中的用法。
- Java 1.2 风格的 Iterator，它总是指向两个元素之间。
- 标准库风格的 *ContainerType*::iterator。
- 手写的、使用容器的获取函数的 while 循环或者 for 循环。
- QDirIterator，用于迭代遍历目录结构中的项。

与 Java 风格的迭代器相比，STL 风格的迭代器的行为更与指针类似。迭代器与指针的一个重要差异是：不存在与指针的空值对应的迭代器值。例如，使用指针的函数在搜索集合中的项时，如果搜索失败则可以返回一个空指针。不存在对应的、可被广泛认知的迭代器值能够表示一个失败的、基于迭代器的搜索。

4.2.1 节演示了 C++中 Qt 可用的各种风格的迭代。

迭代器模式

提供通用方法访问集合元素的类、函数或者编程元素，如果没有类型限制，则它们就是迭代器设计模式的实现[①]。C++迭代器、Java 风格的迭代器以及 foreach 循环，都是迭代器模式的例子。

Qt 中有许多类都提供了针对各种数据类型的专门化迭代，例如 QDirIterator，QSortFilterProxyModel，QTreeWidgetItemIterator 和 QDomNodeList。当引入迭代器时，其定义必须包含足够的信息，以使它能够在容器中从一个项移动到下一个项。

4.2.1 QStringList 与迭代

对于文本处理，对字符串列表进行处理是有用的。QStringList 实际上就是一个 QList<QString>，这样就能够使用 QList 的 public 接口[②]。此外，QStringList 还具有一些特别针对字符串的方便函数，例如 indexOf()，join()和 replaceInStrings()。

利用与 Perl 中类似的 split()函数和 join()函数，在列表与字符串之间进行转换相当简单。示例 4.1 中演示了列表、迭代、split()函数以及 join()函数的用法。

示例 4.1 src/containers/lists/lists-examples.cpp

```
#include <QStringList>
#include <QDebug>

/* Some simple examples using QStringList, split, and join */

int main() {

    QString winter = "December, January, February";
    QString spring = "March, April, May";
```

① 有关设计模式的详细讨论，请参见 7.4 节。

② 事实上，QStringList 派生自 QList<QString>，所以它继承了 QList 的全部 public 接口。第 6 章中探讨了派生和继承。

```
    QString summer = "June, July, August";
    QString fall = "September, October, November";

    QStringList list;
    list << winter;                                          1
    list += spring;                                          2
    list.append(summer);                                     3
    list << fall;

    qDebug() << "The Spring months are: " << list[1] ;


    QString allmonths = list.join(", ");                     4
    qDebug() << allmonths;

    QStringList list2 = allmonths.split(", ");
    /* Split is the opposite of join. Each month will have its own element. */

    Q_ASSERT(list2.size() == 12);                            5

    foreach (const QString &str, list) {                     6
        qDebug() << QString(" [%1] ").arg(str);
    }

    for (QStringList::iterator it = list.begin();
         it != list.end(); ++it) {                           7
        QString current = *it;                               8
        qDebug() << "[[" << current << "]]";
    }

    QListIterator<QString> itr (list2);                      9
    while (itr.hasNext()) {                                  10
        QString current = itr.next();
        qDebug() << "{" <<  current << "}";
    }

    return 0;
}
```

1 追加操作数 1。
2 追加操作数 2。
3 追加成员函数。
4 从列表到字符串，以逗号为分隔符。
5 如果条件不满足，则 `Q_ASSERT` 会终止程序。
6 `Qt foreach` 循环——类似于 Perl/Python 和 Java 1.5 风格的 `for` 循环。
7 C++ STL 风格的迭代。
8 指针风格的解引用。
9 Java 1.2 风格的迭代器。
10 元素之间的 Java 迭代器指针。

以下是示例 4.1 的输出。

```
src/containers/lists> ./lists
The Spring months are:  "March, April, May"
"December, January, February, March, April, May, June, July, August, September,
October, November"
" [December, January, February] "
" [March, April, May] "
" [June, July, August] "
" [September, October, November] "
[[ "December, January, February" ]]
[[ "March, April, May" ]]
[[ "June, July, August" ]]
[[ "September, October, November" ]]
{ "December" }
{ "January" }
{ "February" }
{ "March" }
{ "April" }
{ "May" }
{ "June" }
{ "July" }
{ "August" }
{ "September" }
{ "October" }
{ "November" }
/src/containers/lists>
```

Qt 试图满足具有各种习惯和编程风格的程序员。例如，QList::Iterator 只不过是 QList::iterator 的一种 typedef 定义(别名)，它们提供了引用 STL 风格的 iterator 类的两种不同途径。QListIterator 类和 QMutableListIterator 类提供 Java 风格的迭代器，指向列表元素之间，并且可以用 previous()和 next()访问特定的元素。

4.2.2 QDir，QFileInfo 和 QDirIterator

目录(有时称为文件夹)，是文件的容器。由于目录也可以包含其他的目录，所以它天生就是一种树结构。目录也可以包含符号链接(称为 symlink)，指向另外的文件或者目录。对于处理文件或者目录的大多数操作而言，可以使用符号链接而不是文件名称或者路径名称。

Qt 提供了几种独立于平台的方法来遍历目录树。利用 QDir 类和 QFileInfo 类，可以获得目录的内容列表以及关于每一个项的信息。示例 4.2 中给出了一个递归函数，它使用这两个类来访问目录中所选择的项。它能够判断这个项是文件、目录还是符号链接，并能够根据参数的选择适当地处理这个项。第一个参数确定了要遍历的目录，第二个参数决定函数是否应递归向下地进入该目录下找到的任何子目录中，第三个参数决定函数是否应处理该目录下找到的任何符号链接。这个特定函数的主要作用是找到 MP3 文件并将它的路径添加到列表中。

示例 4.2 src/iteration/whyiterator/recurseadddir.cpp

```
[ . . . . ]
void recurseAddDir(QDir d, bool recursive=true, bool symlinks=false ) {
    d.setSorting( QDir::Name );
    QDir::Filters df = QDir::Files | QDir::NoDotAndDotDot;
```

```
    if (recursive) df |= QDir::Dirs;
    if (not symlinks) df |= QDir::NoSymLinks;
    QStringList qsl = d.entryList(df, QDir::Name | QDir::DirsFirst);

    foreach (const QString &entry, qsl) {
        QFileInfo finfo(d, entry);
        if ( finfo.isDir() ) {
            QDir sd(finfo.absoluteFilePath());
            recurseAddDir(sd);
        } else {
            if (finfo.completeSuffix()=="mp3")
                addMp3File(finfo.absoluteFilePath());          1
        }
    }
}
[ . . . . ]
```

1 非复用的部分。

示例 4.3 中列出的应用复用了 QDirIterator，以更少的行完成了示例 4.2 中同样的任务。

示例 4.3 src/iteration/diriterator/diriterator.cpp

```
[ . . . . ]
int main (int argc, char* argv[]) {
    QCoreApplication app(argc, argv);
    QDir dir = QDir::current();
    if (app.arguments().size() > 1) {
        dir = app.arguments()[1];
    }
    if (!dir.exists()) {
        cerr << dir.path() << " does not exist!" << endl;
        usage();
        return -1;
    }
    QDirIterator qdi(dir.absolutePath(),
            QStringList() << "*.mp3",
            QDir::NoSymLinks | QDir::Files,
            QDirIterator::Subdirectories );
    while (qdi.hasNext()) {
        addMp3File(qdi.next());
    }
}
[ . . . . ]
```

这两个应用有一个重要的差别。示例 4.2 中，对特定于工程的 addMp3File()函数的调用，发生在 recurseAddDir()函数的定义之内，recurseAddDir()函数管理迭代，这严重地限制了函数的复用性。示例 4.3 中，用 QDirIterator 管理迭代。对 addMp3File()函数的调用发生在 QDirIterator 的客户代码中，即 main()中，因此对这个类的复用性没有影响。在合适的地方使用迭代器模式，可以使代码更简单、更容易复用。

4.3 关系

既然某种类型的两个对象之间可以进行一对一或者一对多的通信,就可以在 UML 类框图中用各种连接线来描述对象之间的关系。

图 4.1 中,连接两个类之间的线描述了它们之间的一种特殊关系。

根据这个框图所描述的情况,一个 Employer 对象可以有许多 Person 实例。关系的两端可以用说明符指定:Employer 端的“1”和 Person 端的“*”。“*”遵从它在正则表达式中的定义,即表示“0 个或者多个”(参见 14.3 节)。

回顾图 2.6 可知,当从 Employer 的角度看待公司时,就会产生另外一组关系。图 4.2 中给出了三种关系。

图 4.1 一对多关系

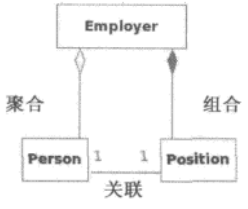

图 4.2 从 Employer 的角度看待公司时得到的关系

1. Employer 与 PositionPosition 存在组合关系(实心菱形)。它表明 Employer“拥有”或者“管理”Position,而如果没有 Employer 的话,就不应当存在 Position。
2. 从 Employer 到它的雇员存在聚合关系(空心菱形)。Employer 对应一组 Person,并让他们在工作时间进行某项工作。在聚合关系中,位于两端的对象的生命周期彼此不相关。
3. 在 Person 与 Position 之间存在一种双向关联关系。关联是一种双向关系,它不必指定对象的实现、对象之间的所属关系或者管理关系。Person 中可能并不存在指向它的 Employer 的实际指针,因为 Employer 可以通过进入 Position 并调用 Position::getEmployer()计算得到。尽管这个框图中显示了 Person 与 Position 之间存在一对一关系,但是一个 Person 可以具有多个 Position,或者一个 Employer 可以对同一个 Position 雇佣多个 Person。如果希望描述一个能够处理这种情况的系统,则它们之间必须是多对多的关联关系。

4.3.1 关系小结

前面已经讲解了三种关系:

- 关联(只用于导航性)
- 聚合(无管理的包含关系)
- 组合(带管理的包含关系)

组合关系是一种强关系,其中还描述了父-子关系和包含(托管)容器的关系。此外,每一种关系都可以具有如下的属性。

- 基数——可以是一对一、一对多或者多对多。线的一端旁边的数字((1, 1..5, *)经常用来指定这个基数。
- 导航性——可以是单向的或者双向的。单向关系在连接两个类的线上可以有箭头而没有菱形。

图 4.3 中给出了一个示例，它使用了包含关系和单向关系。

这个框图中，Book 是一个 Page 指针的(托管)容器，但是 Page 之间有自己的导航关系。也许，读者(或者浏览器)能够从这种直接导航链接中获得好处，可以进入相邻的页面或者进入目录。它们都是单向关系，所以能够加上三个自指向的箭头而框图不会出错。

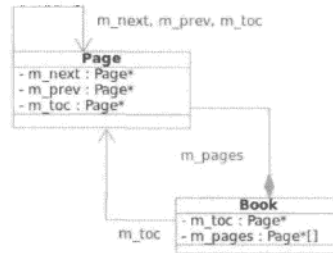

图 4.3 Book 与 Page 的关系

因为没有对 m_pages 关系给出逆向的关系，所以它描述的是一种单向关系。它也可以是一种双向包含关系。对 Page 而言，如果需要导航到它所包含的 Book 对象，则需要在框图中将这种关系描述得更清楚一些。方法可以是标记关系的另一端，或者在 Page 类中添加一个 m_Book 属性。

4.4 练习：关系

1. 这些练习中，需根据图 4.2 实现某些关系。图 4.2 中的框图仅仅是个开始。为了完成这项任务，需向类添加一些成员。
   a. 实现 Employer::findJobs()函数，返回全部开放职位(Position)的列表。
   b. 实现调用 Employer::hire()函数的 Person::apply(Position*)函数，并且如果成功的话返回与 hire()函数相同的结果。
   c. 为了增加趣味性，让 Employer::hire()函数有一半的次数随机返回 false。
   d. 对于返回有关雇佣信息的那些 Person 方法，一定要处理 Person 还没有被雇佣的情况，并返回一些有意义的信息。
   e. 在客户代码中创建更多用于测试的 Employer 对象(Galactic Empire 和 Rebel Forces)、Person 对象(Darth Vader, C3PO 和 Data)以及 Position 对象(Tie Fighter Pilot, Protocol Android 和 Captain)。
   f. 在 Person 类中定义一个 QList<Person*> s_unemploymentLine 对象，确保所有还没有工作的人都位于其中。
   g. 编造一些有趣的工作情景，运行这个应用以判断它们是否成功。
2. a. 图 4.4 中描述了一个 Contact 系统的模型[①]。ContactList 可以派生自或者复用任何 Qt 容器，只要该容器支持下面列出的操作。
   - getPhoneList(int category)接收的值将与 Contact 的类别成员比较，以此来达到选择的目的。该函数返回一个 QStringList，其中包含选中的 Contact、姓名以及电话号码，中间使用制表符\t 分隔。

① 模型与视图将在第 13 章探讨。这里，将管理应用信息(不包括信息的获取、显示和传送)的数据结构称为模型。

- getMailingList()选择机制与 getPhoneList 相似，它返回一个包含地址标号数据的 QStringList。

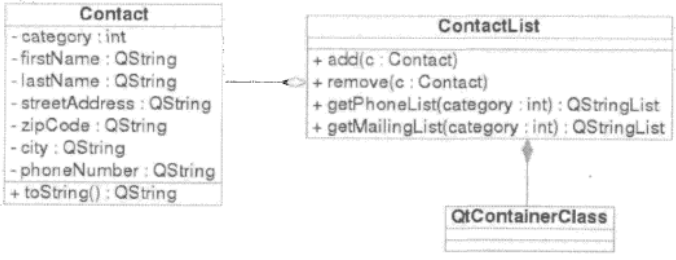

图 4.4 Contact 的 UML 框图

b. 编写一个 ContactFactory 类，产生随机的 Contact 对象。
示例 4.4 中给出了许多提示。

示例 4.4 src/containers/contact/testdriver.cpp

```
[ . . . . ]

void createRandomContacts(ContactList& cl, int n=10) {
    static ContactFactory cf;
    for (int i=0; i<n; ++i) {
        cf >> cl;                                          1
    }
}
```

1 将联系人信息添加到联系人列表中。

有许多种途径能够产生随机的名称/地址。一种方法是使用 ContactFactory 创建常见的名、姓、街道名、城市名等[①]。在需要产生 Contact 对象时，可以从列表中随机选取一个元素，然后添加随机生成的地址编号、邮编等。1.13.1 节中演示过 random()函数的用法。

c. 编写客户代码，测试这些类。特别地，客户代码应当随机生成某些联系人信息。之后，应测试两种查询方法，即 getPhoneList()和 getMailingList()，确保它们返回正确的子列表。在标准输出终端输出原始列表以及查询结果。通过列举原始 ContactList 中与查询结果对应元素的数量来总结查询的结果。

4.5 复习题

1. 命名三个可能需要使用 QStringList 的应用程序。
2. 如何将 QStringList 转换成 QString？为什么要这样做？
3. 如何将 QString 转换成 QStringList？为什么要这样做？
4. 什么是迭代器模式？给出三个示例。

① 垃圾邮件是联系人信息的一个绝佳来源。只需查看垃圾邮件文件夹，即可从邮件标题或者消息体中获得这些信息。在 Dist 目录下已经提供了一个垃圾邮件联系人信息的初始列表。

5. 画一个 UML 框图，它具有三个或者多个类，并确保框图体现了如下这些不同类型的关系：

   - 聚合和组合
   - 一对一和一对多
   - 单向和双向

   这些类应反映现实世界的情况，且关系应是真实的。用几段话解释框图中每种关系存在的理由。
6. 组合与聚合有何不同?
7. 阅读 `QList` 的 API 文档，找出向列表添加元素的三种不同方式。
8. 给出 `QStringList` 中具有而 `QList` 中没有的三个方法。
9. `QList` 中为什么具有 `iterator` 和 `Iterator`? 二者有什么不同?
10. 将 Qt 或者 STL 容器类与数组进行比较，探讨它们的优缺点。
11. 分析 `QList` 声明的语法。位于尖括号中的是什么？为什么它是必须的?
12. 为什么 Qt 支持这么多不同类型的迭代器?
13. `QFile` 与 `QFileInfo` 有何不同?