附录 E　链表简介

第 13 章介绍了数组并指出了什么是链表。链表是一种由小型容器组成的数据结构，这些容器被设计成能够根据需要连接在一起。基本思想是设计一个存储数据对象的类，它能够指向下一个同类型的容器。对于每个要存储的对象，创建一个容器并在需要时将它们连接起来。

这些容器被称为节点。链表中的第一个节点被称为头节点，最后一个节点被称为尾节点。

链表有 3 种基本形式，从最简单到最复杂依次为：

- 单链表。
- 双链表。
- 树。

在单链表中，每个节点都指向下一节点，但不指向前一个节点。要查找特定的节点，从链表头开始，逐个节点往下找，就像寻宝游戏（“下一个节点在沙发下面”）。双链表让你能够向前和向后移动。树是一种由节点组成的复杂数据结构，每个节点都可能指向多个节点。图 E.1 说明了这 3 种基本结构。

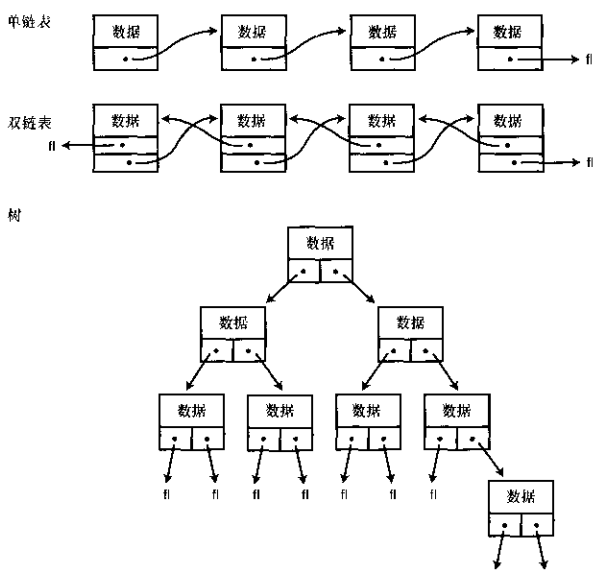

图 E.1　链表

本附录将通过一个有关如何创建（更重要的是如何使用）复杂数据结构的案例研究，一一详细讨论链表。

要创建的链表的组成部分

将创建的链表由节点组成。节点类本身是一个抽象类，将使用 3 个子类来完成工作。链表包含一个头节点（其任务是管理链表头）、一个尾节点（猜猜其任务是什么）以及零或更多个内部节点。内部节点用于记录存储在链表中的实际数据。

注意，数据和链表是不同的概念。从理论上说，可以在链表中存储任何类型的数据。被连接在一起的不是数据而是存储数据的节点。

main()函数并不知道节点，它使用链表。然而，链表完成的工作很少，它将工作委托给节点。

程序清单 E.1 列出了代码；在本附录余下的篇幅中，将详细分析这些代码。

程序清单 E.1　链表

```
// ********************************************
//    FILE:       Listing E.1
//    PURPOSE:    Demonstrate a linked list
//    NOTES:
//
// COPYRIGHT:  Copyright (C) 2000-04 Liberty Associates, Inc.
//             All Rights Reserved
//
// Demonstrates an object-oriented approach to
// linked lists. The list delegates to the node.
// The node is an abstract data type. Three types of
// nodes are used, head nodes, tail nodes and internal
// nodes. Only the internal nodes hold data.
//
// The Data class is created to serve as an object to
// hold in the linked list.
//
// ********************************************


#include <iostream>
using namespace std;

enum { kIsSmaller, kIsLarger, kIsSame};

// Data class to put into the linked list
// Any class in this linked list must support two methods:
// Show (displays the value) and
// Compare (returns relative position)
class Data
{
  public:
    Data(int val):myValue(val){}
    ~Data(){}
    int Compare(const Data &);
    void Show() { cout << myValue << endl; }
  private:
    int myValue;
};
```

```
40:
41: // Compare is used to decide where in the list
42: // a particular object belongs.
43: int Data::Compare(const Data & theOtherData)
44: {
45:   if (myValue < theOtherData.myValue)
46:     return kIsSmaller;
47:   if (myValue > theOtherData.myValue)
48:     return kIsLarger;
49:   else
50:     return kIsSame;
51: }
52:
53: // forward declarations
54: class Node;
55: class HeadNode;
56: class TailNode;
57: class InternalNode;
58:
59: // ADT representing the node object in the list
60: // Every derived class must override Insert and Show
61: class Node
62: {
63:  public:
64:    Node() {}
65:    virtual ~Node() {}
66:    virtual Node * Insert(Data * theData)=0;
67:    virtual void Show() = 0;
68:  private:
69: };
70:
71: // This is the node that holds the actual object
72: // In this case the object is of type Data
73: // We'll see how to make this more general when
74: // we cover templates
75: class InternalNode: public Node
76: {
77:  public:
78:    InternalNode(Data * theData, Node * next);
79:    ~InternalNode() { delete myNext; delete myData; }
80:    virtual Node * Insert(Data * theData);
81:    // delegate!
82:    virtual void Show() { myData->Show(); myNext->Show(); }
83:
84:  private:
85:    Data * myData;  // the data itself
86:    Node * myNext;   // points to next node in the linked list
87: };
88:
89: // All the constructor does is to initialize
90: InternalNode::InternalNode(Data * theData, Node * next):
91: myData(theData),myNext(next)
92: {
93: }
```

```
 94:
 95: // the meat of the list
 96: // When you put a new object into the list
 97: // it is passed to the node, which figures out
 98: // where it goes and inserts it into the list
 99: Node * InternalNode::Insert(Data * theData)
100: {
101:
102:    // is the new guy bigger or smaller than me?
103:    int result = myData->Compare(*theData);
104:
105:
106:    switch(result)
107:    {
108:    // by convention if it is the same as me it comes first
109:    case kIsSame:       // fall through
110:    case kIsLarger:    // new data comes before me
111:       {
112:          InternalNode * dataNode = new InternalNode(theData, this);
113:          return dataNode;
114:       }
115:
116:    // it is bigger than I am so pass it on to the next
117:    // node and let HIM handle it.
118:    case kIsSmaller:
119:       myNext = myNext->Insert(theData);
120:       return this;
121:    }
122:    return this;  // appease MSC
123: }
124:
125:
126: // Tail node is just a sentinel
127:
128: class TailNode : public Node
129: {
130:   public:
131:     TailNode(){}
132:     ~TailNode(){}
133:     virtual Node * Insert(Data * theData);
134:     virtual void Show() { }
135:
136:   private:
137:
138: };
139:
140: // If data comes to me, it must be inserted before me
141: // as I am the tail and NOTHING comes after me
142: Node * TailNode::Insert(Data * theData)
143: {
144:    InternalNode * dataNode = new InternalNode(theData, this);
145:    return dataNode;
146: }
147:
```

```
148: // Head node has no data, it just points
149: // to the very beginning of the list
150: class HeadNode : public Node
151: {
152:   public:
153:     HeadNode();
154:     ~HeadNode() { delete myNext; }
155:     virtual Node * Insert(Data * theData);
156:     virtual void Show() { myNext->Show(); }
157:   private:
158:     Node * myNext;
159: };
160:
161: // As soon as the head is created
162: // it creates the tail
163: HeadNode::HeadNode()
164: {
165:     myNext = new TailNode;
166: }
167:
168: // Nothing comes before the head so just
169: // pass the data on to the next node
170: Node * HeadNode::Insert(Data * theData)
171: {
172:    myNext = myNext->Insert(theData);
173:    return this;
174: }
175:
176: // I get all the credit and do none of the work
177: class LinkedList
178: {
179:   public:
180:     LinkedList();
181:     ~LinkedList() { delete myHead; }
182:     void Insert(Data * theData);
183:     void ShowAll() { myHead->Show(); }
184:   private:
185:     HeadNode * myHead;
186: };
187:
188: // At birth, I create the head node
189: // It creates the tail node
190: // So an empty list points to the head which
191: // points to the tail and has nothing between
192: LinkedList::LinkedList()
193: {
194:    myHead = new HeadNode;
195: }
196:
197: // Delegate, delegate, delegate
198: void LinkedList::Insert(Data * pData)
199: {
200:    myHead->Insert(pData);
201: }
```

```
202:
203: // test driver program
204: int main()
205: {
206:    Data * pData;
207:    int val;
208:    LinkedList ll;
209:
210:    // ask the user to produce some values
211:    // put them in the list
212:    for (;;)
213:    {
214:       cout << "What value? (0 to stop): ";
215:       cin >> val;
216:       if (val == 0)
217:          break;
218:       pData = new Data(val);
219:       ll.Insert(pData);
220:    }
221:
222:    // now walk the list and show the data
223:    ll.ShowAll();
224:    return 0;  // ll falls out of scope and is destroyed!
225: }
```

输出：

```
What value? (0 to stop): 5
What value? (0 to stop): 8
What value? (0 to stop): 3
What value? (0 to stop): 9
What value? (0 to stop): 2
What value? (0 to stop): 10
What value? (0 to stop): 0
2
3
5
8
9
10
```

分析：

首先需要注意的是第 24 行定义的枚举常量，它提供了 3 个常量：kIsSmaller、kIsLarger 和 kIsSame。可能存储在该链表中的每个对象都必须支持 Compare( )方法。这些常量将用作 Compare( )方法的返回值。

出于演示的目的，第 30～39 行创建了 Data 类，Compare( )方法是在第 43～51 行实现的。Data 对象存储一个值，并能够将自己同其他 Data 对象进行比较；它还支持 Show( )方法，用于显示 Data 对象的值。

理解链表工作原理的最简单方式是，逐步剖析一个使用链表的例子。main( )函数头位于第 204 行；第 206 行声明了一个 Data 指针；第 208 行定义了一个局部链表。

LinkList 类是在第 177～186 行声明的。当创建链表时将调用第 192 行的构造函数。构造函数所做的惟一工作是创建一个 HeadNode 对象，并将该对象的地址赋给第 185 行声明的链表中的指针。

创建 HeadNode 对象时，将调用 HeadNode 的构造函数（第 163～166 行）。该构造函数创建一个 TailNode 对象，并将其地址赋给头节点的 myNext 指针。创建 TailNode 对象时将调用第 131 行的 TailNode 构造函数，后者是一个内联函数，没有执行任何操作。

这样，通过在堆栈中创建一个链表，创建了一个头节点和一个尾节点，并确定了链表、头节点和尾节点之间的关系，如图 E.2 所示。

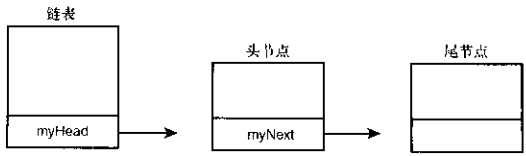

图 E.2 创建后的链表

回到 main( )函数。从第 212 行开始是一个无限循环，该循环提示用户输入要添加到链表中的值。用户可以添加任意数目的值，添加完后输入 0。第 216 行检测输入的值，如果为 0，则结束循环。

如果输入的值不为 0，第 218 行创建一个新的 Data 对象。第 219 行将其插入到链表中。出于演示的目的，假设用户输入 15，将调用第 198 行的 Insert 方法。

链表将插入该对象的工作委托给头节点：调用第 170 行 Insert 方法。头节点将这项任务交给其 myNext 指向的节点去完成。此时（第一次插入时），myNext 指向尾节点（别忘了，创建头节点时将创建一个指向尾节点的链接）。因此，将调用第 142 行的 Insert 方法。

TailNode::Insert 知道，必须将传来的对象插入到自己前面，即新对象将位于尾节点的前面。因此第 144 行创建一个新的 InternalNode 对象，并将数据和指向自己的指针作为参数。这将调用位于第 90 行的 InternalNode 的构造函数。

InternalNode 构造函数只是将其 Data 指针初始化为传入的 Data 对象的地址，并将其 myNext 指针初始化为传入的节点的地址。此时，它指向的节点为尾节点（别忘了，尾节点传入自己的 this 指针）。

创建 InternalNode 后，第 144 行将该中间节点的地址赋给指针 dataNode，然后 TailNode::Insert( )方法返回该地址。这将回到 HeadNode::Insert( )，它将 InternalNode 的地址赋给 HeadNode 的 myNext 指针（第 172 行）。最后 HeadNode 的地址被返回给链表，第 200 行将其丢弃（由于链表已经知道头节点的地址，因此没有使用 HeadNode 的地址来做任何工作）。

如果不使用该地址，为什么要返回它呢？Insert 是在基类 Node 中声明的，其他实现需要该返回值。如果修改 HeadNode::Insert( )的返回值，将导致编译错误。返回 HeadNode 并让链表将其丢弃实简单。

那么发生了什么事情呢？数据被插入到了链表中。链表将数据传递给头节点，头节点不管三七二十一，将数据传给自己指向的节点。在首次插入数据时，头节点指向尾节点。尾节点创建一个新的内部节点，并将其初始化为指向尾节点。然后尾节点将新节点的地址返回给头节点，后者将其 myNext 指针重新设置为指向新节点。就像变魔术一样，数据被插入到了链表中正确的位置，如图 E.3 所示。

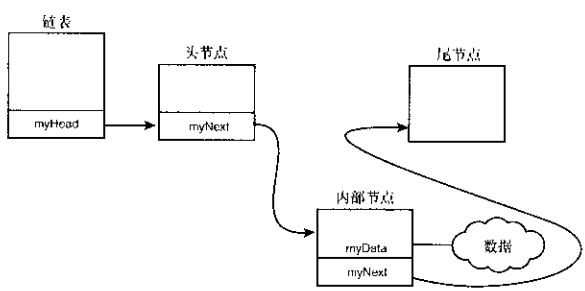

图 E.3 插入第一个节点后的链表

插入第一个节点后，程序回到第 214 行继续执行。再次检查输入的值。假设输入的值为 3。这将导致第 218 行创建一个新的 Data 对象，第 219 行将其插入链表。

在第 200 行，链表再次将数据传递给其 HeadNode。HeadNode::Insert( )方法将新 Data 对象传给其 myNext 指针指向的节点。正如读者所知，它现在指向的是包含值为 15 的 Data 对象的节点。这将调用第 99 行的 InternalNode::Insert( )方法。

在第 103 行，InternalNode 使用其 myData 指针命令其 Data 对象（值为 15 的那个）调用它的 Compare( )方法，并将新的 Data 对象（值为 3）作为参数传入。这将调用第 43 行的 Compare( )方法。

该方法对这两个值进行比较。由于 myValue 的值为 15，而 theOtherData.myValue 的值为 3，因此返回值为 kIsLarger。这使程序跳到第 110 行执行。

为新的 Data 对象创建了一个新的 InternalNode。新节点指向当前的 InternalNode 对象，同时 InternalNode::Insert( )方法将新节点的地址返回给 HeadNode。这样新节点（其对象值小于当前节点的对象值）被插入到链表中，现在的链表如图 E.4 所示。

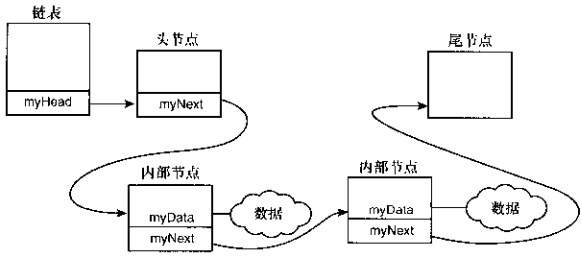

图 E.4　插入第一个节点后的链表

在第二次循环中，用户输入 8。它比 3 大但比 15 小，因此应被插入到两个已有的内部节点之间。插入过程与前面相同，只是对象值为 3 的节点进行比较时，返回值为 kIsSmaller 而不是 kIsLarger（这意味着值为 3 的对象小于值为 8 的新对象）。这导致 InternalNode::Insert( )方法跳到第 118 行执行。该内部节点不是创建一个新节点并将其插入到链表中，而是将新 Data 对象传递给其 myNext 指针指向的节点的 Insert 方法。在这里，它对 Data 对象值为 15 的 InternalNode 调用 InsertNode。

再次进行比较，并创建一个新 InternalNode。新 InternalNode 指向 Data 对象值为 15 的 InternalNode，其地址被传回给 Data 对象值为 3 的 InternalNode，如第 119 行所示。

最终结果是，新节点被插入到链表中正确的位置。

如果可能，应在调试器中以步进方式执行插入一系列节点的过程，你将观察到这些方法相互调用，指针被正确地调整。

读者学到了什么

在设计良好的面向对象程序中，不存在主管。每个对象都完成自己的一小部分工作，而最终效果是平稳运行的机器。

链表的惟一工作是维护头节点。头节点将新数据传递给它指向的节点，而不考虑它指向的是哪个节点。

每当收到数据后，尾节点都创建一个新节点并将其插入到链表中。尾节点只知道一点：只要有数据传来，便将其插入到我的前面。

内部节点要复杂些，它们命令自己包含的对象同新对象进行比较。根据比较结果决定是插入还是往下传递。

注意，内部节点（在前面的程序清单中为 InternalNode）并不知道如何进行比较，这项工作被留给对象自己去完成。内部节点只知道让对象去比较，并期望得到 3 种结果之一。给定一种结果时，它执行插入操作；否则往下传，不知道也不关心将传递到哪里结束。

21天学通C++（第五版）

[美] Jesse Liberty　Bradley Jones　著

李佩乾　杨小珂　译

人民邮电出版社

SAMS Teach Yourself C++ in 21 Days

经典C++教程
全美销量超过25万册

21天学通C++

（第五版）

[美] Jesse Liberty Bradley Jones 著
李佩乾 杨小珂 译

人民邮电出版社
POSTS & TELECOM PRESS

SAMS

只需21天便可具备开始使用C++进行编程所需的全部技能。通过阅读这本内容全面的教程，读者可快速掌握基本知识并学习更高级的特性和概念。

- 了解有关 C++ 和面向对象编程的基本知识；
- 掌握 C++ 提供的所有全新和高级的特性；
- 通过完成实际范例，学习如何使用 C++ 编写高效的程序；
- 向 C++ 权威人士学习专家级技巧。

读者可根据自己的需要阅读本书，循序渐进地逐章阅读或选择最感兴趣的章节阅读。

本书英文版站点

可从www.samspublishing.com下载本书中所讨论的所有源代码。

- 不需要任何编程经验；
- 学习 C++ 以及面向对象设计、编程和分析；
- 编写快速、功能强大的 C++ 程序，编译源代码，创建可执行文件；
- 了解新的 ANSI 标准以及如何从标准修订中获益；
- 使用函数、数组、变量和指针完成复杂的编程工作；
- 学习使用继承和多态扩展程序的功能；
- 通过向编程专家学习，掌握 C++ 特性；
- 适用于任何遵循 ANSI C++ 标准的编译器。

ISBN 7-115-13692-0

9 787115 136923 >

ISBN7-115-13692-0/TP·4812
定价：59.00 元

人民邮电出版社网址 www.ptpress.com.cn

作者简介

Jesse Liberty 编著了大量有关软件开发的图书，其中包括 C++ 和 .NET 方面的畅销书。他是 Liberty Associates 公司（http://www.LibertyAssociates.com）的总裁，该公司致力于为客户提供编程、咨询和培训方面的服务。

Bradley Jones 是 Microsoft Visual C++ MVP，他身兼网站管理员、经理、编码大师、执行编辑等职，其主要精力放在众多软件开发网站和频道上，其中包括 Developer.com、CodeGuru.com、DevX、VBForums、Gamelan 以及 Jupitermedia 的其他网站。这些影响力在不断扩大的网站每月为 250 万开发人员提供信息。

他最擅长的是大 C 语言领域：C、C++和 C#，但在 PowerBuilder、VB、Java、ASP 和 COBOL I/II 开发以及各种现在没有人提起的古老技术方面也拥有丰富的经验。他还从事过咨询人员、分析员、项目负责人、联合发行人和作者等工作，他最近参与编著的图书包括《21 天学通 C#》和《21 天学通 C 语言（第六版）》等。他还是 Indianapolis .NET Developers Association（成员超过 700 的 INETA 集团的一家连锁公司）的联合创始人兼总裁。你经常可以在 CodeGuru.com 和 VBForums.com 论坛上看到他发表的贴子，他还每周在 CodeGuru 上发表一篇通信稿，其读者为数以万计的开发人员。

前　言

本书旨在帮助读者学习如何使用 C++进行编程。没有人仅在三个星期内就能学好一种严谨的编程语言，但本书每章的内容都可以在几小时内阅读完毕。

只需 21 天，读者就能学习诸如控制输入/输出、循环和数组、面向对象编程、模板和创建 C++应用程序等基本知识，所有这些内容都被组织成结构合理、易于理解的章节。每章都提供范例程序清单，并辅以范例输出和代码分析以演示该章介绍的主题。

为加深读者对所学内容的理解，每章最后都提供了常见问题及其答案以及测验和练习。读者可对照附录 D 提供的测验和练习答案，了解自己对所学内容的掌握程度。

本书针对的读者

通过阅读本书来学习 C++时，读者不需要有任何编程经验。本书从入门开始，既介绍 C++语言，又讨论使用 C++进行编程涉及的概念。本书提供了大量语法实例和详细的代码分析，它们是引导读者完成 C++编程之旅的优秀向导。无论读者是刚开始学习编程还是已经有一些编程经验，书中精心安排的内容都将让你的 C++学习变得既快速又轻松。

本书约定

提示：提供使读者进行 C++编程时更高效、更有效的信息。

注意：提供与读者阅读的内容相关的信息。

FAQ：对 C++语言的用法进行了深入剖析，澄清一些容易混淆的问题。

警告：提醒读者注意在特定情况下可能出现的问题或副作用。

应该：提供当前章介绍的基本原理的摘要。

不应该：提供一些有用的信息。

在程序清单中，在每行代码中都加上了行号；没有行号的代码行是前一行的续行（有些代码行太长，无法在一行中列出）。这种情况下，应将两行作为一行输入，不能将它们分开。

本书的范例代码

本书正文及附录 D 中的范例代码可从 Sams 网站下载，其网址为 http://www.samspublishing.com。在文本框“Search”中输入本书英文版的 ISBN（0672327112），单击 Search 按钮，然后单击原版书名（Sams Teach Yourself C++ in 21 Days, 5th Edition）便可链接到可下载范例代码的页面，点击 Downloads 即可下载。

图书在版编目（CIP）数据

21 天学通 C++：第五版 /（美）利伯帝（Liberty, J.），（美）琼斯（Jones, B.）著；李佩乾，杨小珂译．—北京：人民邮电出版社，2005.9

ISBN 7-115-13692-0

Ⅰ．2… Ⅱ．①利…②琼…③李…④杨… Ⅲ．C 语言—程序设计 Ⅳ．TP312

中国版本图书馆 CIP 数据核字（2005）第 097601 号

版 权 声 明

21 天学通 C++（第五版）

◆ 著 [美] Jesse Liberty Bradley Jones

译 李佩乾 杨小珂

责任编辑 李 际

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号

邮编 100061 电子函件 315@ptpress.com.cn

网址 http://www.ptpress.com.cn

北京顺义振华印刷厂印刷

新华书店总店北京发行所经销

◆ 开本：787×1092 1/16

印张：40.5

字数：1 317 千字 2005 年 9 月第 1 版

印数：1－5 000 册 2005 年 9 月北京第 1 次印刷

著作权合同登记号 图字：01-2004-5456 号

ISBN 7-115-13692-0/TP・4812

定价：59.00 元

读者服务热线：(010)67132705 印装质量热线：(010)67129223