目录

[线性表 1](#_Toc40183349)

[线性表介绍 1](#_Toc40183350)

[线性表功能 1](#_Toc40183351)

[线性表代码实例 2](#_Toc40183352)

[基于数组的线性表：顺序表 3](#_Toc40183353)

[顺序表介绍 3](#_Toc40183354)

[顺序表的功能与属性 3](#_Toc40183355)

[顺序表的功能实现 3](#_Toc40183356)

[顺序表的代码实例 4](#_Toc40183357)

[使用STL实现顺序表，vector向量的应用 7](#_Toc40183358)

[链式描述的顺序表：链表 9](#_Toc40183359)

[链表介绍 9](#_Toc40183360)

[链表的功能与属性 10](#_Toc40183361)

[链表的代码实例 10](#_Toc40183362)

[对于线性表我想讲的一些瞎扯的地方 17](#_Toc40183363)

[正式开始瞎扯之前想瞎扯的 17](#_Toc40183364)

[准备开始瞎扯 18](#_Toc40183365)

[然后，就来实现线性表吧。 20](#_Toc40183366)

[然后，实现顺序表 22](#_Toc40183367)

# 线性表

## 线性表介绍

线性表就是一串数字，对，就是一串数字。

我们熟悉的数组就是线性表的属性。

比如：a[5]={ 0, 1, 2, 3, 4 }

但只有数字，不足以构成线性表。

还需要各种可调用的方法修改这组数。

此时就要各种各样的方法。

## 线性表功能

各种方法，就是线性表的功能，一个完整的线性表，应该需要有以下功能：

* **创建：**创建一个表
* **空？：**判断表是否为空
* **元素个数：**返回表的元素个数
* **获取：**获取表中的数
* **索引：**给予一个数，寻找它在表中的索引
* **删除：**删除表中的数
* **插入：**在表中插入一个数

## 线性表代码实例

template<class T>

class linearList

{

public:

// 析构函数

virtual ~linearList() {};

// 是否为空

virtual bool empty() const = 0;

// 返回表的元素个数

virtual int size() const = 0;

// 返回索引位置的数

virtual T& get(int theIndex) const = 0;

// 返回该数在表中第一次出现的位置

virtual int indexOf(const T& theElement) const = 0;

// 删除索引位置处的数

virtual void erase(int theIndex) = 0;

// 在索引中插入一个数

virtual void insert(int theIndex, const T& theElement) = 0;

// 输出表

virtual void output(ostream& out) const = 0;

};

# 基于数组的线性表：顺序表

## 顺序表介绍

线性表可以有好几种组成方式，最简单的自然就是基于数组的顺序表。

一个顺表有好几个不同的属性。

## 顺序表的功能与属性

顺序表的功能基本基于线性表，所以只要继承线性表即可。

但顺序表需要有自己的属性：

* **数组：**用来储存元素
* **元素个数：**数组中的元素个数
* **容量：**顺序表的容量

## 顺序表的功能实现

* **构造函数：创建一个数组**
  + 给予一个数x
  + 检查数的可行性（x<1）
  + 将x赋值给容量，为数组分配内存，数组长度为0
* **复制构造函数：复制一个顺序表**
  + 给予一个顺序表
  + 将原顺序表的值一一赋值给新顺序表
* **析构函数：删除一个顺序表**
  + delete就完事了。
* **空？：判断顺序表是否为空**
  + 判断顺序表长度就行
* **元素个数：顺序表有元素的个数**
  + 返回顺序表长度就行
* **获取：获取表中的数**
  + 给予一个索引
  + 检测索引可行性
  + 返回索引的值
* **索引：返回一个数在表中的索引**
  + 给予一个数
  + 寻找该数是否存在
  + 返回该数的位置
* **删除：删除表中的数**
  + 给予一个索引
  + 判断索引可行性
  + 后面的数向前覆盖一格
  + 减少长度
* **插入：在表中插入一个数**
  + 给予一个索引
  + 判断空间是否充足
  + 判断索引可行性
  + 从索引位置开始，把所有数向后覆盖一格
  + 在索引位置处插入该数
  + 增加长度

## 顺序表的代码实例

template<class T>

class arrayList : public linearList<T>

{

public:

// 构造（创建）

arrayList(int initialCapacity = 10);

// 构造（复制）

arrayList(const arrayList<T>&);

// 析构（删除）

~arrayList() {delete [] element;}

/\*ADT 方法\*/

// 空？

bool empty() const {return listSize == 0;}

// 数量

int size() const {return listSize;}

T& get(int theIndex) const;

int indexOf(const T& theElement) const;

void erase(int theIndex);

void insert(int theIndex, const T& theElement);

void output(ostream& out) const;

/\*额外的方法\*/

// 容量

int capacity() const {return arrayLength;}

// 迭代器

class iterator;

iterator begin() {return iterator(element);}

iterator end() {return iterator(element + listSize);}

protected:

// arrayList的其他成员

// 检查索引

void checkIndex(int theIndex) const;

T\* element; // 顺序表中的元素

int arrayLength; // 顺序表的长度

int listSize; // 顺序表的元素个数

};// 构造（创建）

template<class T>

arrayList<T>::arrayList(int initialCapacity)

{

if (initialCapacity < 1)

{ostringstream s;

s << "Initial capacity = " << initialCapacity << " Must be > 0";

throw illegalParameterValue(s.str());

}

arrayLength = initialCapacity;

element = new T[arrayLength];

listSize = 0;

}

// 构造（复制）

template<class T>

arrayList<T>::arrayList(const arrayList<T>& theList)

{

arrayLength = theList.arrayLength;

listSize = theList.listSize;

element = new T[arrayLength];

copy(theList.element, theList.element + listSize, element);

}

// 检查索引可行性

template<class T>

void arrayList<T>::checkIndex(int theIndex) const

{

if (theIndex < 0 || theIndex >= listSize)

{ostringstream s;

s << "index = " << theIndex << " size = " << listSize;

throw illegalIndex(s.str());

}

}

// 获取

template<class T>

T& arrayList<T>::get(int theIndex) const

{

checkIndex(theIndex);

return element[theIndex];

}

// 索引

template<class T>

int arrayList<T>::indexOf(const T& theElement) const

{

int theIndex = (int)(find(element, element + listSize, theElement) - element);

// 查找

if (theIndex == listSize)

// 没有找到则返回-1

return -1;

else return theIndex;

}

// 删除

template<class T>

void arrayList<T>::erase(int theIndex)

{

// 检查索引

checkIndex(theIndex);

// 将后面的数全部向前移动一位

copy(element + theIndex + 1, element + listSize, element + theIndex);

// 这个地方的调用析构函数我觉得很诡异……不是很明白

element[--listSize].~T();

}

// 插入

template<class T>

void arrayList<T>::insert(int theIndex, const T& theElement)

{

// 检查索引可行性

if (theIndex < 0 || theIndex > listSize)

{

ostringstream s;

s << "index = " << theIndex << " size = " << listSize;

throw illegalIndex(s.str());

}

// 检查空间是否充足

if (listSize == arrayLength)

{

// 扩容

changeLength1D(element, arrayLength, 2 \* arrayLength);

arrayLength \*= 2;

}

// 全部向后移动一格

copy\_backward(element + theIndex, element + listSize, element + listSize + 1);

element[theIndex] = theElement;

listSize++;

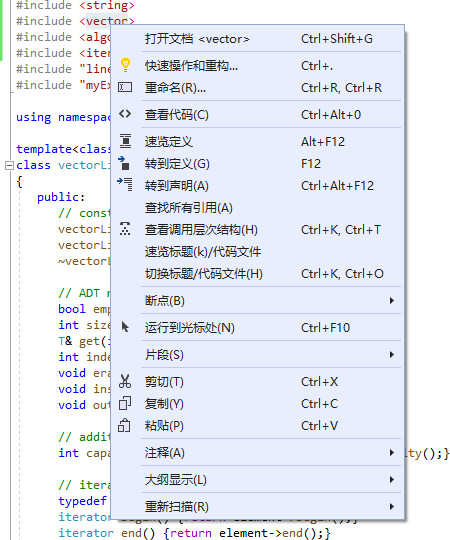
}

## 使用STL实现顺序表，vector向量的应用

如果每次都要自己写一个顺序表，那太麻烦了。

C++的标准模板库本身就有各种各样的容器。

vector就相当于线性表，本身就有各种各样强大的功能。



我们可以通过这种方式查看vector的源代码。

但源码谁也看不懂，这里就简单举例几个vector的用法。

# 链式描述的顺序表：链表

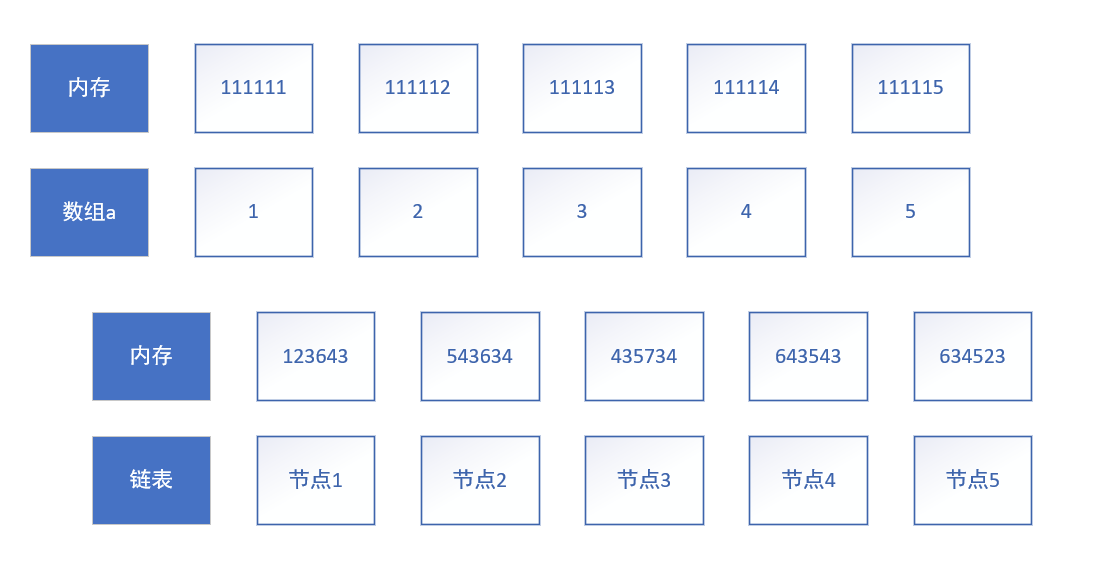
## 链表介绍

链表是一串链起来的表

Emmm……字面意思

不过还是与数组有区别的

这还是要从内存讲起



可以见到数组在内存中是有一串有顺序的数，所以可以直接通过索引来读取一个数。

但链表的内存却不同，是各种各样的

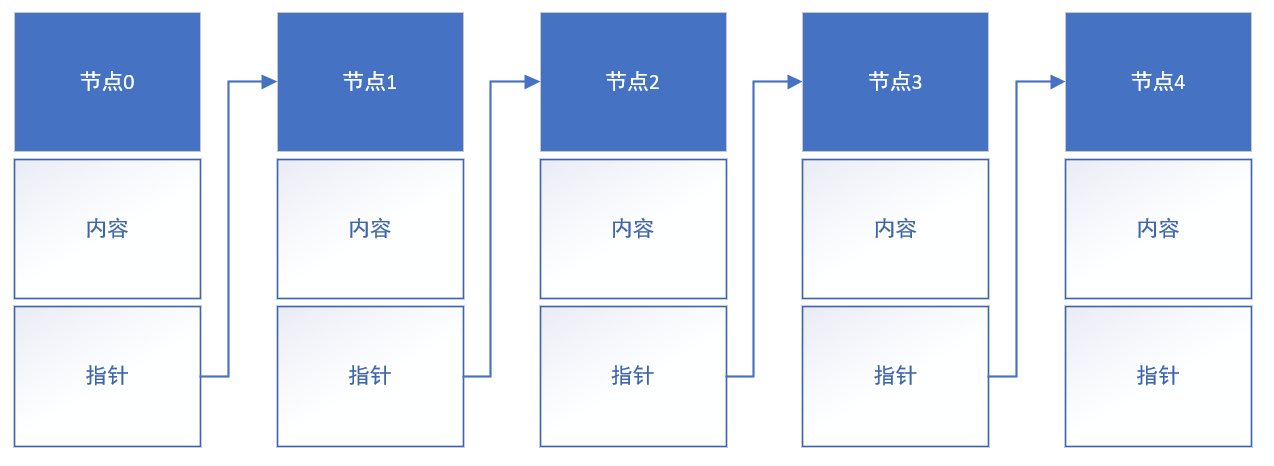
（这里需要通过代码实例）

所以链表不能通过索引来直接获得值。

那链表怎么读取数？

就顺着链子一点一点往下读呗

具体情况看下图



那链表有什么好处？

就是如果我们要删除节点2

只需要将节点1与节点3连接起来，再将节点2删除就行

与数组相比，少了移动后面所有数据这个步骤

在处理大量数据中占一些优势

但读取又不如数组。

所以使用要看具体情况

## 链表的功能与属性

链表功能继承顺序表

链表的基础就是节点

节点中包含以下属性：

* **元素：**用来保存元素
* **节点：**用来指向下一个节点

## 链表的代码实例

/\*

头文件：

链表节点

\*/

#ifndef chainNode\_

#define chainNode\_

template <class T>

struct chainNode

{

// 数据

T element;

chainNode<T>\* next;

// 方法

chainNode() {};

chainNode(const T& element)

{

this->element = element;

}

chainNode(const T& element, chainNode<T>\* next)

{

this->element = element;

this->next = next;

}

};

#endif

/\*

头文件：

线性列表，链表

\*/

#ifndef chain\_

#define chain\_

#include <iostream>

#include <sstream>

#include <string>

#include "linearList.h"

#include "chainNode.h"

#include "myExceptions.h"

using namespace std;

class linkedDigraph;

template <class T> class linkedWDigraph;

template<class T>

class chain : public linearList<T>

{

friend linkedDigraph;

friend linkedWDigraph<int>;

friend linkedWDigraph<float>;

friend linkedWDigraph<double>;

public:

// 构造（创建）

chain(int initialCapacity = 10);

// 构造（复制）

chain(const chain<T>&);

//析构

~chain();

/\*ADT方法\*/

// 空？

bool empty() const { return listSize == 0; }

// 位置

int size() const { return listSize; }

T& get(int theIndex) const;

int indexOf(const T& theElement) const;

void erase(int theIndex);

void insert(int theIndex, const T& theElement);

void output(ostream& out) const;

protected:

// 检查索引可行性

void checkIndex(int theIndex) const;

// 头结点

chainNode<T>\* firstNode;

// 链表长度

int listSize;

};

// 构造函数(创建)

template<class T>

chain<T>::chain(int initialCapacity)

{

if (initialCapacity < 1)

{

ostringstream s;

s << "初始容量=" << initialCapacity << " 必须 > 0";

throw illegalParameterValue(s.str());

}

firstNode = NULL;

listSize = 0;

}

// 构造函数(复制)

template<class T>

chain<T>::chain(const chain<T>& theList)

{

listSize = theList.listSize;

// 如果是一张空表

if (listSize == 0)

{

firstNode = NULL;

return;

}

// 获取目标头节点

chainNode<T>\* sourceNode = theList.firstNode;

// 复制头节点

firstNode = new chainNode<T>(sourceNode->element);

// 准备复制剩余元素

sourceNode = sourceNode->next;

chainNode<T>\* targetNode = firstNode;

// 复制剩余元素

while (sourceNode != NULL)

{

targetNode->next = new chainNode<T>(sourceNode->element);

targetNode = targetNode->next;

sourceNode = sourceNode->next;

}

// 结束链表

targetNode->next = NULL;

}

// 析构函数

template<class T>

chain<T>::~chain()

{

chainNode<T>\* nextNode;

// 摧毁所有元素

while (firstNode != NULL)

{

nextNode = firstNode->next;

delete firstNode;

firstNode = nextNode;

}

}

// 检查索引可行性

template<class T>

void chain<T>::checkIndex(int theIndex) const

{

if (theIndex < 0 || theIndex >= listSize)

{

ostringstream s;

s << "index = " << theIndex << " size = " << listSize;

throw illegalIndex(s.str());

}

}

// 根据索引获取一个数

template<class T>

T& chain<T>::get(int theIndex) const

{

//检查

checkIndex(theIndex);

// 顺着链表移动，直到该索引

chainNode<T>\* currentNode = firstNode;

for (int i = 0; i < theIndex; i++)

currentNode = currentNode->next;

return currentNode->element;

}

// 元素首次出现的返回索引

template<class T>

int chain<T>::indexOf(const T& theElement) const

{

chainNode<T>\* currentNode = firstNode;

int index = 0;

while (currentNode != NULL && currentNode->element != theElement)

{

// 顺着链表移动，直到找到该数

currentNode = currentNode->next;

index++;

}

// 如果不存在，则返回-1。否则返回索引

if (currentNode == NULL)

return -1;

else

return index;

}

// 删除链表中的一个数

template<class T>

void chain<T>::erase(int theIndex)

{

// 检查索引

checkIndex(theIndex);

// 寻找要删除的索引

chainNode<T>\* deleteNode;

if (theIndex == 0)

{

// 删除头节点

deleteNode = firstNode;

firstNode = firstNode->next;

}

else

{

// 顺着链表移动，直到该索引的位置

chainNode<T>\* p = firstNode;

for (int i = 0; i < theIndex - 1; i++)

p = p->next;

deleteNode = p->next;

// 注意这里是指向下下个节点，方便删除

p->next = p->next->next;

}

listSize--;

delete deleteNode;

}

// 插入

template<class T>

void chain<T>::insert(int theIndex, const T& theElement)

{

if (theIndex < 0 || theIndex > listSize)

{

// 非法索引

ostringstream s;

s << "index = " << theIndex << " size = " << listSize;

throw illegalIndex(s.str());

}

if (theIndex == 0)

// 直接插入

firstNode = new chainNode<T>(theElement, firstNode);

else

{

// 找到索引所在位置

chainNode<T>\* p = firstNode;

for (int i = 0; i < theIndex - 1; i++)

p = p->next;

// insert after p

p->next = new chainNode<T>(theElement, p->next);

}

listSize++;

}

// 依旧是诡异的输出

template<class T>

void chain<T>::output(ostream& out) const

{

for (chainNode<T>\* currentNode = firstNode;

currentNode != NULL;

currentNode = currentNode->next)

out << currentNode->element << " ";

}

// 重载 <<

template <class T>

ostream& operator<<(ostream& out, const chain<T>& x)

{

x.output(out);

return out;

}

#endif

# 对于线性表我想讲的一些瞎扯的地方

## 正式开始瞎扯之前想瞎扯的

线性表我真的拖了好久好久了。

从上学期开学，我就准备开始学习数据结构，然后又不知道被什么奇怪的东西拖延了，然后又没有继续学习，然后上学期只有四级的表现稍微好点，别的地方不论是学习成绩还是计算机学习都开花了。

炸裂

然后，又遇到这个诡异的疫情，又躺在家里，躺了好久好了

话说回来躺着是真的舒服

如果可以我希望天天躺着嘿嘿

然后么，在“寒假”，我躺尸的时候，弄懂了链表，稍微了解下栈与队列，映射，还有……树……

唉，躺得舒服了，也要付出点代价，别人都能写游戏了我还只能写写贪吃蛇，害。

找个机会把我那个破游戏也写出来。

然后么，开学了，不是，是到学校了，我突然自我意识觉醒，我是不是该学习了？？？于是，我翻开了带回家吃了万年灰的书本。



翻了好久好久

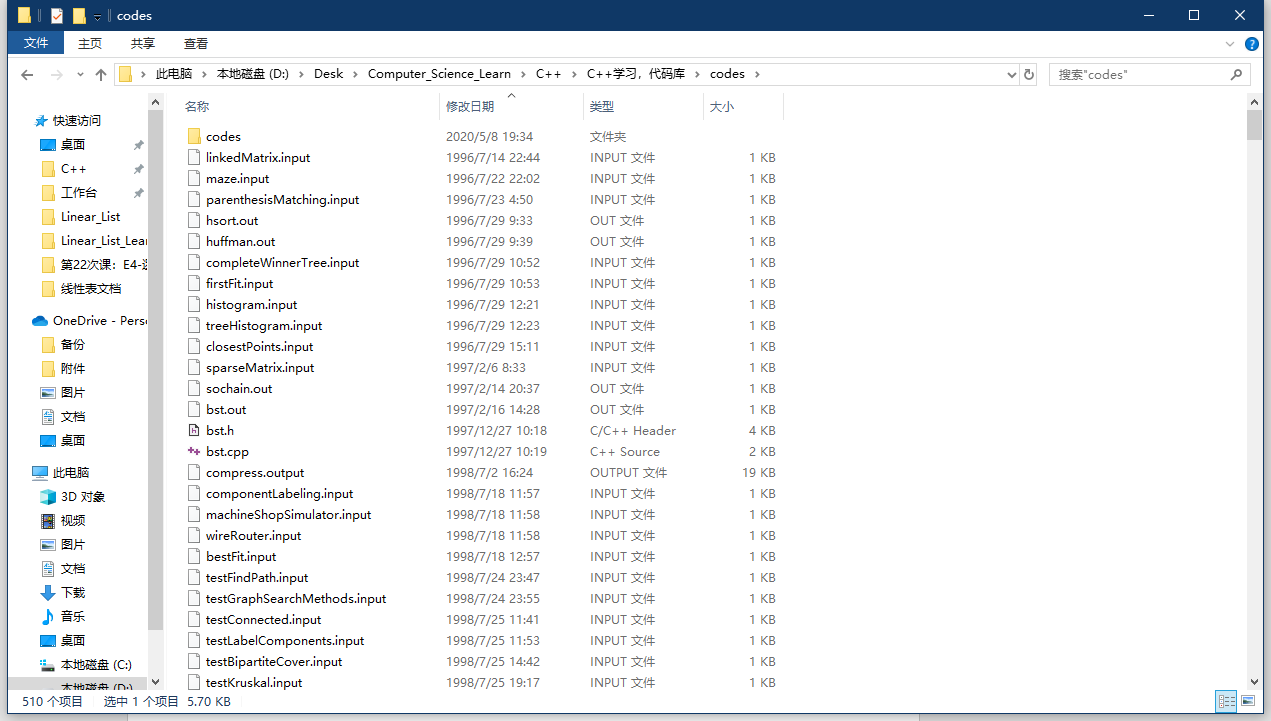
看不懂

害，看不懂就看不懂吧

那抄代码

结果代码抄到一半，就发现了这些代码网上可以批发下载

？？？？？？？



看看这些年龄比我还大的代码，我就害怕

最后编辑时间2004年，这作者怕不是写了10年这本书。

太恐怖了

于是我开始看代码看代码

顺便跑一跑

值得一提的是，这些代码我看了挺久

但没怎么跑过哈哈哈

完全靠作者写的程序运行的

原因嘛，就是懒~

那么，今天就给大家讲讲，我从上年9月，到今天，计算机到底进步了多少

就一个线性表，嗯！

## 准备开始瞎扯

线性表，顾名思义，就是线性表

唉这不废话吗……

就是字面意思，一串，串起来的东西

可以是数，可以是字符串，函数对象结构体等等等等

但我真的没尝试过后面这些东西什么函数对象结构体

Emmmm，第一次写的学籍管理系统，和，数学建模大赛研究的疫情模拟算是用到了吧

线性表分为两种，一种为顺序表，另一种为链表。

顺序表就是一组数，一串串起来的数，串起来，葫芦串那种



后者链表，就是一条链子，emmm，这么说不知道准不准确

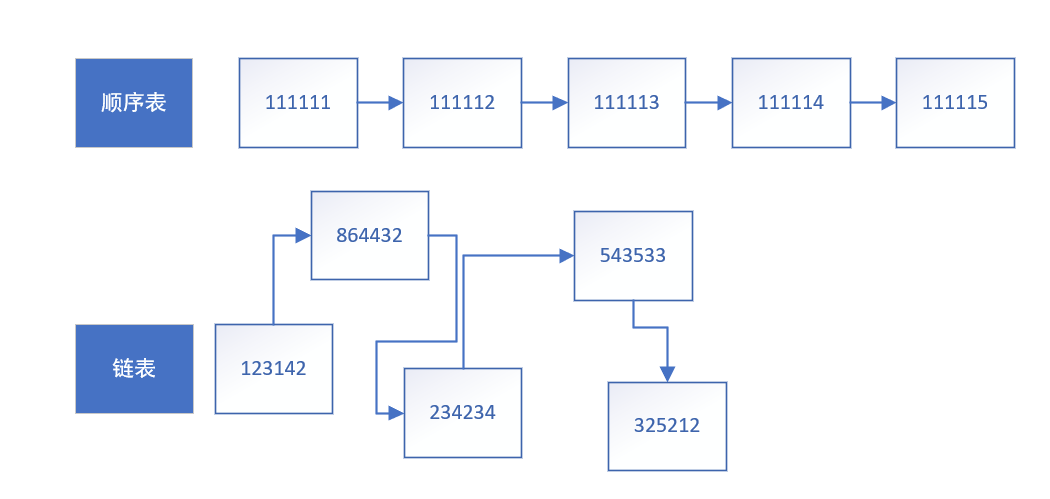


就是这样。



如果转换到计算机内存中的样子

那就是



这样，就是顺序表就是一串储存起来的内存，而链表就是将分散在各处的内存串起来。

不知道这么理解对不对……

自己先持个怀疑态度

没准明天的自己看今天的自己就像个傻X

## 然后，就来实现线性表吧。

注：代码基本来自《数据结构、算法与应用，C++语言描述》/（美）萨特吉·萨尼 著；王立柱，刘志红 译——北京：机械工业出版社。当然存在自己的魔改与翻译部分代码。

唉不是说线性表分为顺序表与链表吗？怎么就直接实现线性表了。

害，看书看多了，作者这么干的，我也这么干了。总不能直接魔改吧。

写到这里我又去重新看了一遍继承……晕乎乎的

顺序表与链表虽然结构不同，但是基本功能是相同的，基本功能就由线性表来描述咯。

那线性表应该要有什么功能呢？

* **创建：**创建一个表
* **空？：**判断表是否为空
* **元素个数：**返回表的元素个数
* **获取：**获取表中的数
* **索引：**给予一个数，寻找它在表中的索引
* **删除：**删除表中的数
* **插入：**在表中插入一个数

如果使用代码描述：

template<class T>

class linearList

{

public:

// 析构函数

virtual ~linearList() {};

// 是否为空

virtual bool empty() const = 0;

// 返回表的元素个数

virtual int size() const = 0;

// 返回索引位置的数

virtual T& get(int theIndex) const = 0;

// 返回该数在表中第一次出现的位置

virtual int indexOf(const T& theElement) const = 0;

// 删除索引位置处的数

virtual void erase(int theIndex) = 0;

// 在索引中插入一个数

virtual void insert(int theIndex, const T& theElement) = 0;

// 输出表

virtual void output(ostream& out) const = 0;

};

看到这里我又忍不住去看了一眼虚函数

又有点晕乎乎的，总是不知道这些东西有什么用

但不加又会出莫名其妙的错误，可以不用继承，但又觉得怪怪的

哦，想起来了，是抽象类。

好吧扯远了

一个线性表就扯完了。

如果你跟我刚开始一样看到这些代码一脸懵逼，不知道他在扯什么东西为什么要这么扯，可以往下看看

对以上代码一些奇怪的地方的一些解释：

1. **template<class T>这行代码是什么意思，文中的T又是写什么东西？**

这个是C++的模板函数，可以使线性表更加灵活。

比如在创建对象时linearList<int> linearListA，T就会全部被代替为int，double，string，char同理。

更详细的建议百度。

1. **virtual是什么？**

在这里是虚函数的意思，作用就是为了防止发生奇怪的冲突。建议去看看C++的动态。

1. **~linearList()**

……………析构函数………

1. **那堆函数名后面为什么要加一个const？？**

表明他们不会修改类对象，这么加了编译器就知道了好提醒你

1. **const = 0是个什么鬼？**

那个不是const = 0，是虚函数=0，表明它是个纯虚函数，什么东西都没有的意思，完全靠继承实现函数功能。

1. **ostream又是个啥？**

我也不知道，我也不清楚，就是用来自定义输出的吧。

写累了，休息会

## 然后，实现顺序表

顺序表继承于线性表，但是多了元素（数组）、顺序长度、顺序表元素个数三个属性。

所以顺序表的功能与线性表是一样的，只是多了几个属性罢了。

接下来，就要将他们的功能一一实现。

代码请看文档末尾

一开始看源码真的挺懵逼的，我觉得他的异常处理与输出手法，有点非常，个性，嗯，可能是为了全书代码能够正常运行？

如果只看书，没有源代码，我甚至完全看不懂他的异常处理。

虽然这种异常处理的方法很巧妙。

唉，总有他的理由吧，但为了能看明白了……

if (initialCapacity < 1)

{

ostringstream s;

s << "Initial capacity = " << initialCapacity << " Must be > 0";

throw illegalParameterValue(s.str());

}

illegalParameterValue当时我抄了好久这个代码，结果结果结果一直输出失败。

在翻看源代码的时候。



里面有一串

// illegal parameter value

class illegalParameterValue

{

public:

illegalParameterValue(string theMessage = "Illegal parameter value")

{message = theMessage;}

void outputMessage() {cout << message << endl;}

private:

string message;

};

我发现了这个

我的例外？？

我的例外？？？？

我的…………？

template<class T>

void arrayList<T>::output(ostream& out) const

{

copy(element, element + listSize, ostream\_iterator<T>(cout, " "));

}

template <class T>

ostream& operator<<(ostream& out, const arrayList<T>& x)

{

x.output(out); return out;

}

然后就看到了这串代码

写书的时候完全没有这玩意，导致我半天看不懂，我的，天……

这一串我一直认为没啥用的代码

特别诡异……

诡异的是他写了output()但他的输出全靠迭代器输出……明明没用到output啊

…………好吧整明白了

还有就是他对output的重载，我翻遍代码也没有找到相关语句调用了这个重载。

所以，我将其修改为：

不是，我刚说完，咋就打我脸了，这两行代码删了就报错……

原来还是调用了

由于我没有学过迭代器，然后作者又以神奇的方法用了迭代器，所以，我就只能自己写一份来测试这些功能了

不过我会将完整的代码放在文档末尾，或者是，我的GitTub上？

没兴趣看我瞎扯淡可以拉到文章末尾。

当我又继续翻代码文档时候，看到作者一句：

cout << "Incremented by 1 list is " << y << endl;

一句话就实现了所有内容的输出。



为什么y能直接被你输出啊，y不是个对象吗？

这就很离谱，于是顺着重载标识，我找到了……

此时，问题来了

好吧好吧好吧我承认是我基础不扎实，如果无聊就跳过吧……

1. 什么是illegalParameterValue，你说了半天也没解释这个？

这是个对象，如果遇到异常就抛出这个对象，说实话我也没看明白一个对象怎么被抛出……写到这里我又要吐槽了！抛出异常就抛出异常，结果什么都没发生，这在之后怎么处理？？？

难道是我没做过项目，抛出异常都要有规定的操作？

1. 什么是ostream？

ostream是一个输出流，在C++中，常用的输出流是cout。但一般cout后跟的<<只能输出一些数字啊，什么的，一旦类型多了就麻烦，所以需要一个专门的输出流，来正常输出一些东西。

比如我要输出对象中的几个数。

class Cpoint

{

public:

// 说实话我又看不懂这个构造函数是个什么操作……

Cpoint(int x\_, int y\_):x(x\_),y(y\_){}

int x, y;

friend ostream& operator <<(ostream& os, const Cpoint& p)

{

return os << "x = " << p.x << " y = " << p.y << endl;

}

};

如果使用ostream&类型对<<进行重载，就能使<<自定义输出一个对象的值。

Cpoint C(1, 2);

cout << C << endl;

这样既可。

1. 第2问你说了自定义，那原来的代码直接一个output，又是一个ostream\_iterator，这谁看得懂啊？

这个，只能说是，一个用法，真的要剖析真的，有点，复杂。又是迭代器又是输出流又是拷贝的。

只要记住这个用法：

copy(element, element + listSize, ostream\_iterator<T>(cout, " "));

就是将一组数拷贝……拷贝到输出流……我自己都有点不信了，他到底怎么做到拷贝进去的……



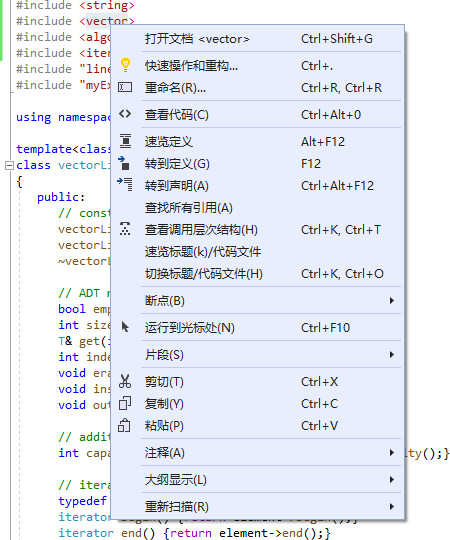
算了算了放弃了，这算是个坑，挖个坑。

## 那有没有更加简单的方法实现？

有！

C++的标准模板库提供了一个东西：vector向量。

有个简单的方法能看到vector的源码



不知道又是哪个古老大神写的

就这样被保存了下来，加入了C++……

向量的功能很强大，很多大神在网上也整理了它的具体用法，这里就不再赘述了（主要是之前整理的资料突然失去了一半心态有点爆炸……）

## 一些乱七八糟的源码

template<class T>

class arrayList : public linearList<T>

{

public:

// 构造（创建）

arrayList(int initialCapacity = 10);

// 构造（复制）

arrayList(const arrayList<T>&);

// 析构（删除）

~arrayList() { delete[] element; }

/\*ADT 方法\*/

// 空？

bool empty() const { return listSize == 0; }

// 数量

int size() const { return listSize; }

T& get(int theIndex) const;

int indexOf(const T& theElement) const;

void erase(int theIndex);

void insert(int theIndex, const T& theElement);

void output(ostream& out) const;

/\*额外的方法\*/

// 容量

int capacity() const { return arrayLength; }

// 迭代器

class iterator;

iterator begin() { return iterator(element); }

iterator end() { return iterator(element + listSize); }

// 顺序表的迭代器，没学过迭代器的我看得我一脸懵逼

class iterator

{

public:

// 双向迭代器

typedef bidirectional\_iterator\_tag iterator\_category;

typedef T value\_type;

typedef ptrdiff\_t difference\_type;

typedef T\* pointer;

typedef T& reference;

// 构造

iterator(T\* thePosition = 0) { position = thePosition; }

// 析构

T& operator\*() const { return \*position; }

T\* operator->() const { return &\*position; }

// 前增

iterator& operator++()

{

++position;

return \*this;

}

// 后增

iterator operator++(int)

{

iterator old = \*this;

++position;

return old;

}

// 前减

iterator& operator--()

{

--position; return \*this;

}

// 后减

iterator operator--(int)

{

iterator old = \*this;

--position;

return old;

}

// 相等检测

bool operator!=(const iterator right) const

{

return position != right.position;

}

bool operator==(const iterator right) const

{

return position == right.position;

}

protected:

T\* position;

};

protected:

/\*arrayList的其他成员\*/

// 检查索引

void checkIndex(int theIndex) const;

T\* element; // 顺序表中的元素

int arrayLength; // 顺序表的长度

int listSize; // 顺序表的元素个数

};

// 构造(创建)

template<class T>

arrayList<T>::arrayList(int initialCapacity)

{

// 检测大小可行性

if (initialCapacity < 1)

{

ostringstream s;

s << "Initial capacity = " << initialCapacity << " Must be > 0";

throw illegalParameterValue(s.str());

}

// 开始创建

arrayLength = initialCapacity;

element = new T[arrayLength];

listSize = 0;

}

// 构造（复制）

template<class T>

arrayList<T>::arrayList(const arrayList<T>& theList)

{

// 先复制容量

arrayLength = theList.arrayLength;

// 再复制长度

listSize = theList.listSize;

// 再分配空间

element = new T[arrayLength];

// 再拷贝

copy(theList.element, theList.element + listSize, element);

}

// 检查索引可行性

template<class T>

void arrayList<T>::checkIndex(int theIndex) const

{

if (theIndex < 0 || theIndex >= listSize)

{

ostringstream s;

s << "index = " << theIndex << " size = " << listSize;

throw illegalIndex(s.str());

}

}

// 获取

template<class T>

T& arrayList<T>::get(int theIndex) const

{

// 检查

checkIndex(theIndex);

// 返回

return element[theIndex];

}

// 索引

template<class T>

int arrayList<T>::indexOf(const T& theElement) const

{

int theIndex = (int)(find(element, element + listSize, theElement) - element);

// 查找

if (theIndex == listSize)

// 没有找到则返回-1

return -1;

else

return theIndex;

}

// 删除

template<class T>

void arrayList<T>::erase(int theIndex)

{

// 检查索引

checkIndex(theIndex);

// 将后面的数全部向前移动一位

copy(element + theIndex + 1, element + listSize, element + theIndex);

// 这个地方的调用析构函数我觉得很诡异……不是很明白

element[--listSize].~T();

}

// 插入

template<class T>

void arrayList<T>::insert(int theIndex, const T& theElement)

{

// 检查索引可行性

if (theIndex < 0 || theIndex > listSize)

{

ostringstream s;

s << "index = " << theIndex << " size = " << listSize;

throw illegalIndex(s.str());

}

// 检查空间是否充足

if (listSize == arrayLength)

{

// 扩容

changeLength1D(element, arrayLength, 2 \* arrayLength);

arrayLength \*= 2;

}

// 全部向后移动一格

copy\_backward(element + theIndex, element + listSize, element + listSize + 1);

element[theIndex] = theElement;

listSize++;

}

// 输出，他的输出的意思大概就是将数字导入输入流，然后……用迭代器输出？

template<class T>

void arrayList<T>::output(ostream& out) const

{

copy(element, element + listSize, ostream\_iterator<T>(cout, " "));

}

// 重载 <<

template <class T>

ostream& operator<<(ostream& out, const arrayList<T>& x)

{

x.output(out); return out;

}