Chapter 2 CONTAINER CLASSES

存储结构

- 自动变量放在栈 无法用代码释放 只能在作用于完结后自动删除
- 动态变量放在堆HEAP 必须人为释放
- new的作用 开辟内存 作用域结束后要删除对应的动态存储空间

  *注意 指向那片动态内存的指针没办法人为删除
- 一定要有strPTR = new string
  - sPtr -> length()等价于(*sPtr).length()

数组 arrays

- 随机访问性random-access：通过给出元素下标index实现
- 通用规则：a[i]等价于*(a+i)
- 数组容量固定且无法更改
- 扩容

```
int capacity;
cin >> capacity
string*words = new string[capacity]

capacity *= 2; // 扩容words数组
string*temp = new string[capacity] // 创建新的数组temp

for(int i = 0; i < capacity/2; i++)
    temp[i] = words[i];

words = temp; // 指针指向扩容后数组的地址
```

- 释放内存：delete[] words;

容器类 container classes

- 容器container是一个变量，由很多项的集合组成
- 容器类container classes是一个类，其每个实例都是一个包含很多项的容器
- 常用的容器类：数组array和链式结构linked structure
  - 数组：允许随机访问元素；插入、删除、调整大小的操作复杂繁琐

链式结构 linked structure

- link：相当于指针指向容器内的下一项
- Linked类
  - 需要定义模板 - 使得用户每次可以自定义容器对象的元素类型
  - Linked类的基本功能：

    1. 能够构造出Linked对象 **Linked();**
    2. 能够返回Linked对象中项的数量 **long size();**
    3. 能够在Linked对象的前面插入新的项 **void push_front(constT& newItem);**

- Linked类的定义

```
template<class T>
class Linked
{
      protected:
            struct Node  { T item; Node* next; }
            Node* head;
            long length;
      public:
            class Iterator
            {
                  friend class Linked<T>;
                  protected:
                  public:
            } // class Iterator
            …
} // class Linked
```

```
template<class T>
class Linked
{
    struct Node
    {
        T item;
        Node* next;
    };

    Node* head; // 指向第一个节点
    long length; // 容器中的元素数量

    long size()
    {
        return length;
    }

    void push_front(constT& newItem)
    {
        Node* newHead = new Node;
        newHead -> item = newItem;
        newHead -> next = head; // 新节点的next指向未插入前
的head
        head = newHead; // 头节点head指向新节点
        length++;
    }

    bool empty(); // 判断此容器是否有元素

    Iterator begin()
    {
        return Iterator(head);
    }

    Iterator end()
    {
        return Iterator(NULL);
    }
```

```
    void pop_front()
    {
        Node* oldHead = head;
        head = head -> next; //头节点指向被删除节点的下一节点
        delete oldHead;
        --length;
    }

    // 析构器
    ~Linked()
    {
        while(head != NULL)
            pop_front(); // 循环调用pop_front删除元素
    }

} // class Linked
```

- 迭代器iterator
  - 允许在不违背数据抽象原理（即不允许用户代码访问Linked类的实现细节）的同时循环通过容器
  - begin()方法返回位于容器对象第一项的迭代器
  - end()方法返回位于容器对象最后一项之外（后一项）的迭代器
  - operator++将调用的迭代器对象前进到容器的下一项
    - 若迭代器位于最后一项，operator++将把迭代器定义到最后一项之后的位置上
  - 定义iterator类
    - 简化的唯一字段：Node* nodePtr;
    - **隐式复制构造：temp = *this （并非直接赋值）**
    - Iterator**通过友元声明使得迭代器能够访问容器类对象**

```
// 通过newPtr初始化迭代器
Iterator(Node* newPtr)
{
    nodePtr = newPtr;
```

```
}

// 显式定义公有缺省构造器
Iterator()
{
}

// 后加运算符:Iterator对象在Linked对象中前进，并返回调用前(!)迭代器位置上的项
Iterator++(int)
{
    Iterator temp = *this;  // 为临时对象temp赋调用对象的值
    nodePtr = (*nodePtr).next; // 令nodePtr指向下一个Node对象
    return temp; //返回temp
}
```

- 其他定义

THE DEFINITIONS OF ==, !=, AND * ARE SIMPLE:

```
bool operator== (const Iterator& itr) const
{
    return nodePtr == itr.nodePtr;
} // operator==

bool operator!= (const Iterator& itr) const
{
    return nodePtr != itr.nodePtr;
} // operator!=

T& operator*( ) const
{
    return nodePtr -> item;
} // operator*
```

- **注意实参和形参**

Now a user can iterate through a Linked container:

```
Linked<string> words;

// Read in the values for words from the input:
…
int count = 0;
Linked<string>::Iterator itr;
for (itr = words.begin( ); itr != words.end( ); itr++)
    if ((*itr).length( ) == 4)
        count++;
cout << “There are “ << count << “ 4-letter words.”;
```

实参

```
template<class T>
class Linked
{
        protected:
                struct Node  { T item; Node* next; }
                Node* head;
                long length;
        public:
                class Iterator
                {
                        friend class Linked<T>;
                        protected:
                        public:
                } // class Iterator
                …
} // class Linked
```

形参

查找方法find

```
template<class InputIterator, class T>
InputIterator find(INputIterator first, InputIterator last,
constT& value)
{
    while (first != last && *first != value)
        ++first;
    return first;
}
```

- 即可用于Linked结构也可用于数组array查找

EXERCISE: provide a definition of the following generic algorithm:

```
// Postcondition: true has been returned if for each
//                iterator itr in the range from first1
//                (inclusive) to last1 (exclusive),
//                *itr = *(first2 + (itr – first1)).
//                Otherwise, false has been returned.
template <class InputIterator1, class InputIterator2>
bool equal (InputIterator1 first1, InputIterator1 last1,
            InputIterator2 first2);
```

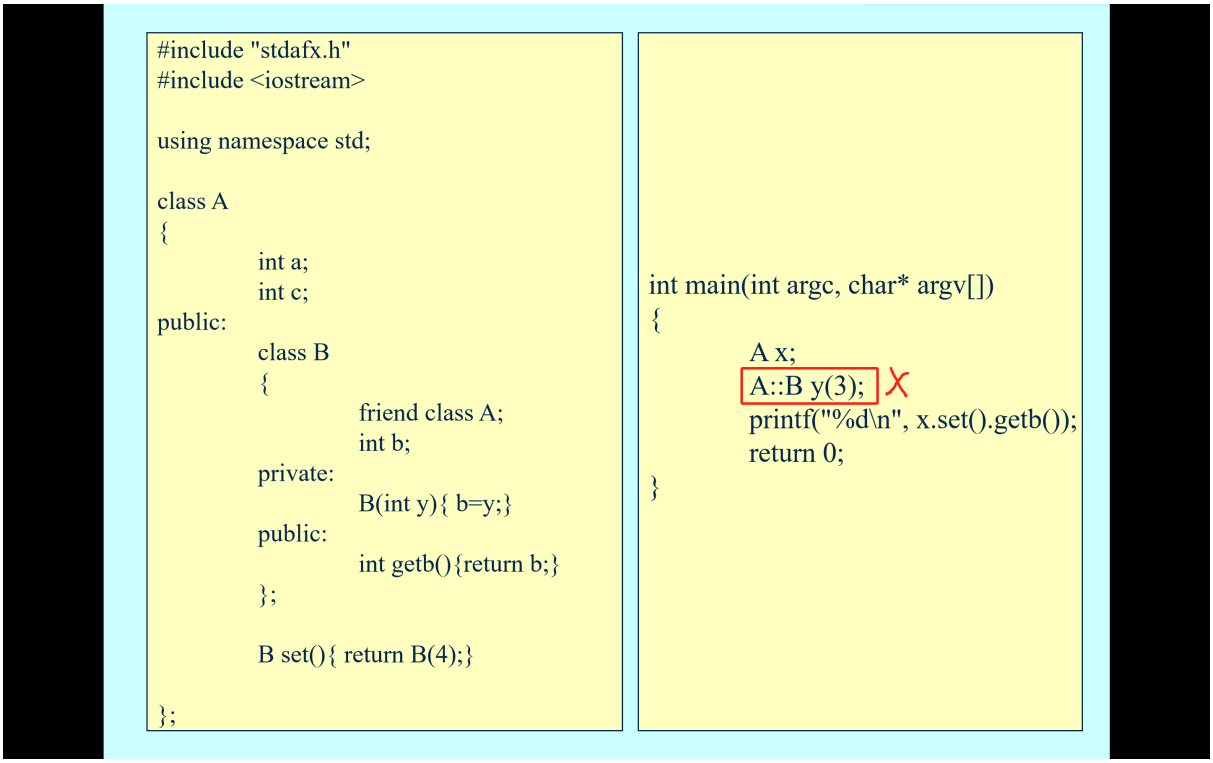
HINT: DEFINE

```
InputIterator itr1 = first1,
              itr2 = first2;
```

```
template <class InputIterator1, class InputIterator2>
bool equal (InputIterator1 first1, InputIterator1 last1,
            InputIterator2 first2)
{
        InputIterator1 itr1;
        InputIterator2 itr2;
        for(itr1=first1,itr2=first2;
                itr1!=last1&&*itr1==*itr2; itr1++,itr2++) ;
        return itr1==last1;
}
```

补充内容：

嵌入interate



```
#include "stdafx.h"
#include <iostream>

using namespace std;

class A
{
        int a;
        int c;
public:
        class B
        {
                friend class A;
                int b;
        private:
                B(int y){ b=y;}
        public:
                int getb(){return b;}
        };

        B set(){ return B(4);}

};
```

```
int main(int argc, char* argv[])
{
        A x;
        A::B y(3);
        printf("%d\n", x.set().getb());
        return 0;
}
```

X

待整理笔记

运算符重载

删除迭代器

动态存储方式

删除时把结束位置都定义在最后一个元素后的一个位置 这样才能完成循环彻底删除 [first, last)

搜索#ifndefine

三种：输入 双向 随机访问迭代器

[first,last) 一般last对应null

Chapter 3 从3.4.3开始