数据结构实验报告书

钟庆柱 @Apacal

January 6, 2014

目录

Chapter 1

实验概述

1.1 实验说明

实验目的：
本实验通过阅读给定 C++ 代码，完成以下任务：
1．查找出其中的错误并予以改正
2．对于正确的代码，分析其时间复杂度
3．对于各个类及其公开函数给出使用的规格说明 4．设法改进各个类及其公开函数的实现的可能的改进

实验目标：
1．编制阅读计划，包括进度表、参考资料等
2．编制测试计划，包括测试方法、测试数据准备等
3．编制勘误表
4．编制功能说明书和时间复杂度分析
5．改写代码使之自文档化
6．编制各个类的使用说明书，包括各个类及其公开函数的规格说明、测试例程、测试数据和输出结果
7．总结

要求：
1．最终必须提交包含上述各项要求的《实验报告书》一册
2．《实验报告书》必须使用 LaTeX 编制脚本并编译成 PDF 格式

最后完成日期：
2014 年 1 月 6 日

1.2 任务计划

时间	任务
2013.12.1-2013.1.2	阅读代码和测试
2013.1.3-2013.1.6	编写报告书

1.3 参考资料

本报告书中，大量参展了《The Annotated STL Sources(using SGI STL)》,《C++ Primer》,http://www.cplusplus.com/ 网站以及百度。

1.4 编者的话

本实验的研究对象为 STL 的一小部分重写，对这些重写的内容进行修正和分析。故本报告分为五部分，第一部分：实验概述，简单介绍实验和一些编者的话。第二部分：Iterator 迭代器，因为 STL 中迭代器扮演重要角色，将泛型的 algorithms 算法和泛型的 containers 容器有效的粘合在一起，所以有必要先研究 Iterator。第三部分：algorithms 算法，对 argorithm.h 进行剖析。第四部分：containers 容器，对 vector 和 list 进行剖析。第五部：为总结，对该实验进行总结。

@author Apacal
Sun Dec 29 21:02:06 HKT 2013

Chapter 2

迭代器

2.1 迭代器介绍

迭代器是一种抽象的设计概念，现实程序语言中并没有直接对应这个概念的实物。iterator 的普遍定义为：提供一种方法，使之能够依序访问某个聚合物（容器）所含的各个元素，而又无需暴利该聚合物的内部表述方式。

STL 的中心思想在于：将数据容器（containers）和算法（algorithms）分开，彼此独立设计，最后再以一粘合剂将它们粘合在一起，迭代器就是这个重要的粘合剂。

迭代器纯翠只是型别定义，承担设计相应的型别的责任，不含有任何数据成员。为了符合规格，任何迭代器都应该提供 5 个内嵌相应型别保证，以便于 iterator_traits 萃取相应的型别。这样的迭代器才能和 STL 的其它组建很好地相融，幸运的是 STL 提供一个 iteratpor class，如果每个迭代器都继承自他，那就可以保证符合 STL 所需要的规范。

2.2 random_iterator

STL 中提供的 iterator class:

```
template <class Category,
          class T,
          /* 默认Distance为ptrdiff_t 该类型与机器有关的unsigned int */
          class Distance=ptrdiff_t,
          class Pointer=T*,
          class Reference=T&>
struct iterator {
    typedef T         value_type;
    typedef Distance  difference_type;
    typedef Pointer   pointer;
    typedef Reference reference;
    typedef Category  iterator_category;
};
```

我们使用时，最少需要传递两个模板参数，一个为 Category，该参数表示 iterator 的型别，是 Input Iterator，Out Iterator，Forward Iterator，Bidirectional Iterator，Randon Access Iterator。在 std 中 Category 的定义如下。

```
/* 定义五个标记用的型别(tag_type) */
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag {};
struct bidirectional_iterator_tag {};
struct random_access_iterator_tag {};
```

这些 class 只是作为标记用，所以没有任何参数

在该实验中定义一个 random_iterator 继承自 std::iterator，它是一个双向的，具有跳跃访问的迭代器。所以在 DSAA 定义的 iterator 必须重载 +n, -n 跳跃的操作。

```
namespace DSAA {

template <typename T>
struct random_iterator :
        public std::iterator<std::forward_iterator_tag,T> {
    /* 下面 4 个typedef只是为了方便作者书写 */
    typedef std::iterator<std::forward_iterator_tag,T>  parent_t;
    typedef       random_iterator                        this_t;
    typedef       random_iterator&                       this_r;
    typedef const random_iterator&                      cthis_r;

    /* 符合stl中的规范，注意value_type的型别，我们将在稍候介绍，测试 */
    typedef typename parent_t::value_type               value_type;
    typedef typename parent_t::difference_type          difference_type;
    typedef typename parent_t::pointer                  pointer;
    typedef typename parent_t::reference                reference;
    typedef typename parent_t::iterator_category        iterator_category;
    typedef const reference                             creference;

    explicit random_iterator(T* P=0) : ptr(P) {}

    /* 重载了一系列的运算 -，++，+n，-n 等等 */
    this_r          operator++()            { ++ptr; return *this; }
    this_r          operator--()            { --ptr; return *this; }
    this_t          operator++(int)         { return this_t(ptr++); }
    this_t          operator--(int)         { return this_t(ptr--); }
    this_t          operator+ (int n)       { return this_t(ptr+n); }
    this_t          operator- (int n)       { return this_t(ptr-n); }

    difference_type diff(this_t rhs)        { return ptr - rhs.ptr; }

    bool            operator==(cthis_r rhs) { return ptr == rhs.ptr; }
    bool            operator!=(cthis_r rhs) { return ptr != rhs.ptr; }
    bool            operator<=(cthis_r rhs) { return ptr <= rhs.ptr; }
    bool            operator>=(cthis_r rhs) { return ptr >= rhs.ptr; }
    bool            operator< (cthis_r rhs) { return ptr <  rhs.ptr; }
    bool            operator> (cthis_r rhs) { return ptr >  rhs.ptr; }

     reference      operator* ()            { return *ptr; }
    creference      operator* () const      { return *ptr; }

protected:
    T* ptr;

    /* 该模板函数将iterator所指向对象的地址输出到ostream中 */
    template <typename ostream>
    friend ostream& output(ostream& os, const random_iterator& rhs) {
        return os << '|' << rhs.ptr << '|';
    }
};

};

#endif
```

通过下面的程序，我们来测试该 iterator，能不能很好的实现一些 -，++，-n，+n，diff 函数。

```
#include <iostream>
#include "iterator.h"
using namespace DSAA;

int main() {
    int numbers[]={10,20,30,40,50};
    random_iterator<int> from(numbers);
```

```
    random_iterator<int> until;

    /* 测试 +n 的操作 */
    until = from + 5;
    /* 输出 until 与 from 的距离 */
    std::cout << from.diff(until) << std::endl;
    std::cout << until.diff(from) << std::endl;

    for (random_iterator<int> it=from; it!=until; it++)
        std::cout << *it << ' ';
    std::cout << '\n';
    /* 下面将 from 和 until 对换。再输出结果, 同时测试 -n */
    /* 注意 C++ 中 [a, b) 的表示。想想为啥是 4，下面又是 6 */
    from = from + 4;
    until = until - 6;
    for (random_iterator<int> it=from; it!=until; --it)
        std::cout << *it << ' ';
    std::cout << '\n';

    return 0;
}
```

输出结果：

```
-5
5
10 20 30 40 50
50 40 30 20 10
```

在下面例子中我们将测试 output 函数, 该函数接受一个 ostream 对象，将迭代器指向对象的地址返回到 ostream 对象中。

```
#include <iostream>
#include "iterator.h"
using namespace DSAA;

int main() {
    char ch[]={'c','h','i','n','a'};
    int num[5] = {10, 20, 30, 40, 50};
    random_iterator<char> from(ch);
    random_iterator<int> until(num);
    output(std::cout, from);
    std::cout << ch << std::endl;
    output(std::cout, until);
    std::cout << &num << std::endl;
    return 0;
}
```

注意，C++ 对字符数组指针的处理, 结果如下：

```
|china|china
|0xbfa51248|0xbfa51248
```

STL 中的 iterator_traits 很巧妙地获取 iterator 的相应型别，及时是原生指针作为迭代器也可以获取到其相应的型别，我们通过下面的例子可以测试出。这让算法和容器很好地分开开发。

```
#include <iostream>
#include <typeinfo>
#include "iterator.h"
using namespace DSAA;

```

```
int main(int argc, char **argv) {
    std::iterator_traits< random_iterator<int> >::value_type j;
    std::cout << typeid(j).name() << "|" << typeid(int).name() << std::endl;;
    std::iterator_traits<int *> ::value_type jj;
    std::cout << typeid(jj).name() << "|" << typeid(int).name() << std::endl;;
    std::iterator_traits<const int *> ::value_type constj;
    std::cout << typeid(constj).name() << "|" << std::endl;;


    return 0;
}
```

不管是迭代器，还是原生指针，std::iterator_traits 都可以很好的获取其型别 value_type。结果如下

```
i|i
i|i
i|
```

Chapter 3

算法

再好的编程技巧，也无法让一个笨拙的算法起死回生，
选择了错误的算法，便注定了失败的命运。

3.1 算法概观

算法，问题之解发也。以有限的步骤，解决逻辑或数学上的问题，这以专门科目我们称之为算法（Algorithms）。当我们发现（发明）一个算法时，下一步重要步骤就是决定该算法的所消耗的资源，包括时间和空间，这个叫做算法分析（algorithm analysis)。在 algorithm.h 中有一些基本的算法。

3.2 assign

struct assign 是一个函数类，在类中重载了 operator () 等运算。其重载两个赋值的函数，具体的定义如下：

```
template <typename T>
struct assign {
    T x;

    /* 带参数的构造函数，如果没有参数，则调用默认的 */
    assign(const T& X) : x(X) {}

    /* dereference , operator*重载，定义了两个 operator*，一个返回左值，一个右值。*/
    T& operator* () { return x; }
    const T& operator* () const { return x; }

    /* 重载operator()操作符，将 y 的值改为 x 。 */
    T& operator()(T& y) { return y=x; }

    /* 重载operator()操作符，将 y,x 的值改为 X 。 */
    T& operator()(T& y, const T& X) { return y=x=X; }
};
```

下面对 assign 进行测试

```
#include <iostream>
#include <cstdio>
#include "algorithm.h"
using namespace DSAA;


int main() {
    assign<char> a('a');
    /* 测试operator* */
    std::cout << *a << " ";
    *a = 'A';
    std::cout << *a << std::endl;
    /* 测试operator() */
```

```
    char b = 'B';
    const char c = 'C';
    /* 注意下面的输出结果，*/
    printf("%c,%c,%c\n", a(b), *a, b);
    b = 'B';
    printf("%c,%c,%c\n", *a, b, a(b));
    b = 'B';
    std::cout << a(b) << " " << *a << " " << b << std::endl;
    printf("%p,%p,%p,%p\n", &(a(b)), &a, &(*a), &b);
    std::cout  << *a << " " << b << std::endl;
    std::cout << a(b, c) <<std::endl;
    std::cout << *a << " " << b << " " << c << std::endl;
    /* 仿函数临时对象履行函数功能 */
    b = 'B';
    std::cout << assign<char>('a') (b, c);
    std::cout << " " << b << " " << c << std::endl;

    return 0;
}
```

我们可以把 assign 当成函数来用，可以先构造一个 assign 对象，再履行函数功能, 比如 assign<char> a('a'); a(b); 也可以用临时对象来履行函数功能，比如 assign<char>('a') (b, c); 输出结果如下：

```
a A
A,A,B
A,A,A
A A B
0xbfcfd22c ,0xbfcfd22a ,0xbfcfd22a ,0xbfcfd22c
A A
C
C C C
C C C
```

3.3 sequence

sequence 是一个序列容器的”结点“，通过它可以较方便的建立序列容器比如 vector，定义了两个成员，action f 是一个模板类，默认是 assign<T>，int pos 是在序列的位置；其的构造函数需要两个参数，一个 action F, int P = 0, 也就是需要一个 action F 就可以构造。也重载了一些操作，比如 operator++, operator-, operator+ ,operator*, perator() 等等。其完整的定义如下：

```
template <typename T, typename action=assign<T> >
struct sequence {
    action f;
    int pos;
    sequence(action F, int P=0) : f(F),pos(P) {}

    sequence&   operator++()                          { ++pos; return *this; }
    sequence&   operator--()                          { --pos; return *this; }
    sequence    operator++(int)                       { return sequence(f,pos++); }
    sequence    operator--(int)                       { return sequence(f,pos--); }
    sequence    operator+ (int n)             const { return sequence(f,pos+n); }
    sequence    operator- (int n)             const { return sequence(f,pos-n); }
    int         operator- (const sequence& rhs) const { return pos - rhs.pos; }

    bool        operator==(const sequence& rhs) const { return pos==rhs.pos; }
    bool        operator!=(const sequence& rhs) const { return pos!=rhs.pos; }
    bool        operator<=(const sequence& rhs) const { return pos<=rhs.pos; }
    bool        operator>=(const sequence& rhs) const { return pos>=rhs.pos; }
    bool        operator< (const sequence& rhs) const { return pos< rhs.pos; }
    bool        operator> (const sequence& rhs) const { return pos> rhs.pos; }

          T&    operator* ()                          { return *f; }
    const T&    operator* ()                    const { return *f; }
```

```
24
25          T&     operator()()                          { return pos; }
26    const T&     operator()()                    const { return pos; }
27          T&     operator()(T& y)                      { return f(y); }
28 };
```

比如在 vector.h 中的 vector 构造函数的运用。

```
1 explicit vector(unsigned n=0, const T& x=T()) : vector(sequence<T>(assign<T>(x)),0,n)
    {}
```

3.4 find

该模板函数，根据 equality 操作符，循序查找 [it1, it2) 内的所有元素，找出第一个匹配“equality”者，则返回一 Ite(Iteraotor) 指向该元素，否则返回 it2 跌代器。其定义如下。

```
1 template <typename Iter, typename T>
2 Iter find(Iter it1, Iter it2, const T& x) { while (it1 != it2 && *it1 != x) ++it1;
    return it1; }
```

我们使用下面的例子测试

```
1 #include <iostream>      // std::cout
2 #include "algorithm.h"   // DSAA::find
3 #include <vector>        // std::vector
4
5 int main () {
6   int myints[] = { 10, 20, 30 ,40 };
7   int * p;
8
9   // pointer to array element:
10  p = DSAA::find (myints,myints+4,30);
11  ++p;
12  std::cout << "The element following 30 is " << *p << '\n';
13
14  std::vector<int> myvector (myints,myints+4);
15  std::vector<int>::iterator it;
16
17  // iterator to vector element:
18  it = DSAA::find (myvector.begin(), myvector.end(), 30);
19  ++it;
20  std::cout << "The element following 30 is " << *it << '\n';
21
22  return 0;
23 }
```

输出结果如下：

```
1 The element following 30 is 40
2 The element following 30 is 40
```

3.5 copy and rcopy

copy 和 rcopy 函数都是将 [it1, it2) 区间的值复制一份并返回一个跌代器。rcopy 是复制 (it1, it2] 区间。它们的定义如下：

```
1 template <typename Iter1, typename Iter2>
2 Iter1 copy(Iter1 d, Iter2 it1, Iter2 it2) { while (it1 != it2) *d++ = *it1++; return d
    ; }
3
4 template <typename Iter1, typename Iter2>
5 Iter1 rcopy(Iter1 d, Iter2 it1, Iter2 it2) { while (it1 != it2) *d++ = *it2--; return
    d; }
```

copy 函数我们使用会直观一点，我们大部分都是使用 [a, b) 区间，比较少使用 (a, b] 区间，所以使用 rcopy 时要特别注意其 copy 的元素是多少。如下：

```
#include <iostream>      // std::cout
#include "algorithm.h"
#include <vector>        // std::vector

int main () {
    int myints[]={10,20,30,40,50,60,70};
    std::vector<int> myvector (7);
    // copy
    DSAA::copy ( myvector.begin() , myints, myints+7 );

    std::cout << "myvector contains(copy):";
    for (std::vector<int>::iterator it = myvector.begin(); it!=myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';

    // rcopy
    std::vector<int> myvector2 (6);
    DSAA::rcopy (myvector2.begin(), myints, myints+6 );
    std::cout << "myvector contains(rcopy):";
    for (std::vector<int>::iterator it = myvector2.begin(); it!=myvector2.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';

    return 0;
}
```

输出结果如下：

```
myvector contains(copy): 10 20 30 40 50 60 70
myvector contains(rcopy): 70 60 50 40 30 20
```

3.6 for_each 遍历

for_each 可以分为两组遍历函数，一组的遍历方式为 [a, b)，另外一组遍历的方式为 (a, b]; 而每组函数，又有三个不同的，其一是三个参数的函数，[it1, it2) 是范围，act 是函数对象，也可以是函数名，act 是一个参数的函数（或者函数对象）; 其二在一的基础上增加了一个参数 d, 并且函数对象也是两个参数的。其三在二的基础上，每访问到一个元素后，d 自增 1。具体定义如下：

```
/* 第一组，遍历 [a, b)。 */
template <typename Iter, typename action>
void for_each(Iter it1, Iter it2, action act) { while (it1 != it2) act(*it1++); }

template <typename Iter1, typename Iter2, typename action>
void for_each(Iter1 d, Iter2 it1, Iter2 it2, action act) { while (it1 != it2) act(*d,
    *it1++); }

template <typename Iter1, typename Iter2, typename action>
void zip_each(Iter1 d, Iter2 it1, Iter2 it2, action act) { while (it1 != it2) act(*d,
    *it1++),++d; }

/* 第二组，遍历 (a, b]。 */
template <typename Iter, typename action>
void rfor_each(Iter it1, Iter it2, action act) { while (it1 != it2) act(*it2--); }

template <typename Iter1, typename Iter2, typename action>
void rfor_each(Iter1 d, Iter2 it1, Iter2 it2, action act) { while (it1 != it2) act(*d,
     *it2--); }

template <typename Iter1, typename Iter2, typename action>
void rzip_each(Iter1 d, Iter2 it1, Iter2 it2, action act) { while (it1 != it2) act(*d,
     *it2--),++d; }

```

```
};
```

我们只对第一组函数进行分析，实验，第二组与第一组的区别只是遍历的次序不同而已。从下面的例子可以看出，遍历函数都是逐个访问，所以时间复杂度就是 O(N) 的。

3.6.1 for_each

```
#include <iostream>      // std::cout
#include "algorithm.h"    // DSAA::for_each
#include <vector>        // std::vector

void myfunction (int i) {  // function:
    std::cout << ' ' << i;
}

struct myclass {           // function object type:
    void operator() (int i) {std::cout << ' ' << i;}
} myobject;

int main () {
    std::vector<int> myvector;
    myvector.push_back(10);
    myvector.push_back(20);
    myvector.push_back(30);

    //  for_each
    std::cout << "for_each:\n";
    std::cout << "myvector contains(function):";
    DSAA::for_each (myvector.begin(), myvector.end(), myfunction);
    std::cout << '\n';

    // or:
    std::cout << "myvector contains(class):    ";
    DSAA::for_each (myvector.begin(), myvector.end(), myobject);
    std::cout << '\n';


    // rfor_each
    std::cout << "rfor_each:\n";
    std::cout << "myvector contains(function):";
    DSAA::rfor_each (myvector.begin(), myvector.end() - 1, myfunction);
    std::cout << '\n';

    std::cout << "myvector contains(class):    ";
    DSAA::rfor_each (myvector.begin(), myvector.end() - 1, myobject);
    std::cout << '\n';
    return 0;
}
```

使用 rfor_each 时特别要注意他说遍历到的区间是 (a, b]，如果忽视这一点可能会产生严重的后果。结果如下：

```
for_each:
myvector contains(function): 10 20 30
myvector contains(class):    10 20 30
rfor_each:
myvector contains(function): 30 20
myvector contains(class):    30 20
```

3.6.2 for_each four parameters

对于四个参数的 for_each，在它调用仿函数时，会传递两个参数，一个是 *d , 一个是 *it1(就是遍历到的元素) , 而我们可以很好的定制仿函数，比如可以统计遍历到的元素。如下例所示。

```cpp
#include <iostream>      // std::cout
#include "algorithm.h"   // DSAA::for_each
#include <vector>        // std::vector


struct myclass {           // function object type:
    void operator() (int &j, int i) {
        std::cout << ' ' << i;
        ++j;
        }
} myobject;

int main () {
    int sum = 0;
    int *p = &sum;
    std::vector<int> myvector;
    myvector.push_back(10);
    myvector.push_back(20);
    myvector.push_back(30);
    //  for_each
    std::cout << "for_each:\n";
    std::cout << "myvector contains(function):";
    DSAA::for_each (p, myvector.begin(), myvector.end(), myobject);
    std::cout << '\n';
    std::cout << "the sum is:" << sum << '\n';

    *p = 0;
    // rfor_each
    std::cout << "myvector contains(class):    ";
    DSAA::rfor_each (p , myvector.begin(), myvector.end() - 1, myobject);
    std::cout << '\n';
    std::cout << "the sum is:" << sum << '\n';
    return 0;
}
```

输出结果如下所示

```
for_each:
myvector contains: 10 20 30
the sum is:3
rfor_each:
myvector contains: 30 20
the sum is:2
```

3.6.3 zip_each

zip_each 提供四个参数，并且在遍历一个后 Iter d 自增 1，给了我们更一步的定制空间。如下例所示，可以完成遍历和复制的功能。

```cpp
#include <iostream>      // std::cout
#include "algorithm.h"   // DSAA::for_each
#include <vector>        // std::vector


struct myclass {           // function object type:
    void operator() (int &j, int i) {
        std::cout << i << ' ';
        j = i;
        }
} myobject;

int main () {
    std::vector<int> myvector;
    std::vector<int> myvector2(3, 0);
    std::vector<int>::iterator p = myvector2.begin();
```

```
17
18      std::cout << "myvector2 contains:";
19      for( p = myvector2.begin(); p != myvector2.end(); ++p) {
20          std::cout << *p << " ";
21      }
22      p = myvector2.begin();
23      std::cout << '\n';
24
25      myvector.push_back(10);
26      myvector.push_back(20);
27      myvector.push_back(30);
28      //  for_each
29      std::cout << "myvector  contains:";
30      DSAA::zip_each (p, myvector.begin(), myvector.end(), myobject);
31      std::cout << '\n';
32      std::cout << "myvector2 contains:";
33      for( p = myvector2.begin(); p != myvector2.end(); ++p) {
34          std::cout << *p << " ";
35      }
36      std::cout << '\n';
37      return 0;
38  }
```

输出结果如下：

```
1 myvector2 contains:0 0 0
2 myvector  contains:10 20 30
3 myvector2 contains:10 20 30
```

Chapter 4

容器

4.1 容器的概观和分类

容器，置物之所也。研究数据的特定排列方式，以利于搜寻或排序或其它特殊目地，这一专门学科我们称之为数据结构（Data Structures）。在目前看来，几乎可以说，任何特定的数据结构都是为了实现某种特定的算法。容器可以分为序列式（sequence）和关联式（associative）两种。序列式容器，就是说其中的元素都可序（ordered），但未必就是有序（sorted）；比如 C++ 内建的 array，vector，heap，list，slist 等等。关联式容器，在观念上类似关联式数据库，每个数据（元素）都有一个键值（key）和一个实值（value），容器内部按照键值大小，以某种规律将元素放到特定的位置，比如 tree, map, set。本实验就是对 vector，dlist，slist 进行分析实验。

4.2 C++11 语法甜点

先介绍一点 c++ 11 语法上的新特性, 详细的请点击以下两个网址C++11 语法甜点或者C++11 语法甜点

4.2.1 委托构造函数

在引入 C++ 11 之前，如果某个类有多个重载的构造函数，且这些构造函数中有一些共同的初始化逻辑，通常都需要再编写一个带参数的初始化函数，然后在这些构造函数中调用这个初始化函数。在 C++ 11 中，再也不用这么麻烦了。我们可以实现一个最基础的构造函数，其他构造函数都调用这个构造函数。示例代码如下：

```
class CPerson {
    public:
        CPerson() : CPerson(0, "") { NULL; }
        CPerson(int nAge) : CPerson(nAge, "") { NULL; }
        CPerson(int nAge, const string &strName) {
        stringstream ss;
        ss << strName << "is " << nAge << "years old.";
        m_strInfo = ss.str();
        }

    private:
        string m_strInfo;
};
```

4.2.2 成员变量初始化

与 Java 和 C# 中的用法一样，可以对成员变量进行就地初始化。示例代码如下：

```
class CPerson{
    private:
        int m_nAge = 10;
        string m_strName = "Mike";
};
```

4.3 vector

vector 的数据安排和操作方式，与 array 非常相似，两者唯一的差别就是空间的运用灵活性，array 是静态空间，一旦配置就不能改变；如果需要更大的空间，那么一切都需要客户端自己来：首先配置一块新的空间，然后将元素重旧址一一往新址搬，再把原来的空间释放还给系统。而 vector 是动态空间，随着元素的加入，它内部自动扩充空间以容纳新元素。

4.3.1 vector 的数据结构

vector 所采用的数据结构很简单：线性连续空间。它的 head 指针指向连续空间的的第一个元素;siz 表示当前使用空间的大小, 也就是 head + siz - 1 就是最后一个元素的地址；cap 表示当前连续空间的大小，当新增元素时，需要检查新增后的 siz 是否大于 cap, 如果大于就需要重新配置空间;inc 表示每次增加空间的大小 (这里所说的大小是以 T 的大小为单位的)。具体定义如下：

```
template <typename T>
class vector {

private:

    const unsigned inc = 100;
    unsigned siz , cap;
    T* head;
...
```

4.3.2 vector 的迭代器

vector 维护的是一个连续的空间，所以不论其元素型别为何，普通指针都可以作为 vector 迭代器，所以其迭代器继承自 STL 中的迭代器，继承 DSAA::random_iterator<T>，两组符合 STL 规范的 iterator,iterator 和 const_iterator,reverse_iterator 和 const_reverse_iterator，其中第二组就是反过来看，头变成尾，尾变成头，但是需要注意的它 rbegin() 返回的是 –end()，rend() 返回的是 –begin()，此外它还重载了 operator() 运算符。下面来看具体定义。

4.3.2.1 iterator & const_iterator

iterator 和 const_iterator 是我们比较经常使用的一组 iterator。const_iterator 是只读的 iterator，就是不能改变元素的值，注意和 const iterator 的区别，const iterator 是没有现实意义的。具体定义如下。

```
struct iterator : public DSAA::random_iterator<T> {
    typedef DSAA::random_iterator<T>                parent_t;
    typedef       iterator                           this_t;
    typedef       iterator&                          this_r;
    typedef const iterator&                         cthis_r;

    typedef typename parent_t::value_type           value_type;
    typedef typename parent_t::difference_type      difference_type;
    typedef typename parent_t::pointer              pointer;
    typedef typename parent_t::reference            reference;
    typedef typename parent_t::iterator_category    iterator_category;
    typedef const reference&                        creference;

    explicit        iterator(T* P=0)        : parent_t(P) {}
                    iterator(parent_t itr)  : parent_t(itr) {}

    this_r          operator++()            { parent_t::operator++(); return *this; }
    this_r          operator--()            { parent_t::operator--(); return *this; }
    this_t          operator++(int)         { this_t itr(*this); operator++(); return
        itr; }
    this_t          operator--(int)         { this_t itr(*this); operator--(); return
        itr; }
    this_t          operator+ (int n)       { return this_t(parent_t::operator+(n)); }
    this_t          operator- (int n)       { return *this+(-n); }
    difference_type operator- (cthis_r rhs) { return parent_t::diff(rhs); }

     reference      operator*()             { return parent_t::operator*(); }
```

```
    creference      operator*() const       { return parent_t::operator*(); }
};
```

从定义中我们可以看出，该 iterator 继承自 iterator.h 中定义的 random_iterator。 const_iterator 定义如下，它是继承 iterator，但是他的 operator* 是不能更改元素值。(const_iterator 还是可以改变元素，具体看第五章)

```
struct const_iterator : public iterator {
    typedef       iterator                          parent_t;
    typedef       const_iterator                     this_t;
    typedef       const_iterator&                    this_r;
    typedef const const_iterator&                   cthis_r;

    typedef typename parent_t::value_type           value_type;
    typedef typename parent_t::difference_type      difference_type;
    typedef typename parent_t::pointer              pointer;
    typedef typename parent_t::reference            reference;
    typedef typename parent_t::iterator_category    iterator_category;
    typedef const reference                         creference;

    explicit    const_iterator(pointer P=0)  : iterator(P) {}
                const_iterator(parent_t itr) : parent_t(itr) {}

    creference operator*() const { return parent_t::operator*(); }
};
```

4.3.2.2 reverse_iterator & const_reverse_iterator

reverse_iterator & const_reverse_iterator 的定义和第一组的定义基本是一样的，它增加了 operator() 的重载，具体定义如下：

```
template <typename Iterator>
struct riterator {
    typedef       riterator                          this_t;
    typedef       riterator&                         this_r;
    typedef const riterator&                        cthis_r;

    /* 定义 STL 规范的型别，巧妙的定义方式。 */
    typedef typename Iterator::value_type           value_type;
    typedef typename Iterator::difference_type      difference_type;
    typedef typename Iterator::pointer              pointer;
    typedef typename Iterator::reference            reference;
    typedef typename Iterator::iterator_category    iterator_category;
    typedef const reference                         creference;

    explicit riterator(Iterator IT) : it(IT) {}

    this_r          operator++()            { --it; return *this; }
    this_r          operator--()            { ++it; return *this; }
    this_t          operator++(int)         { return this_t(it--); }
    this_t          operator--(int)         { return this_t(it++); }
    this_t          operator+ (int n)       { return this_t(it-n); }
    this_t          operator- (int n)       { return this_t(it+n); }
    difference_type operator- (cthis_r rhs) { return -it.diff(rhs.it); }

    bool            operator==(cthis_r rhs) { return it==rhs.it; }
    bool            operator!=(cthis_r rhs) { return it!=rhs.it; }
    bool            operator<=(cthis_r rhs) { return it>=rhs.it; }
    bool            operator>=(cthis_r rhs) { return it<=rhs.it; }
    bool            operator< (cthis_r rhs) { return it> rhs.it; }
    bool            operator> (cthis_r rhs) { return it< rhs.it; }

     reference      operator* ()            { return *it; }
    creference      operator* () const      { return *it; }
```

```
34
35      Iterator        operator()()            { return  it; }
36  protected:
37      Iterator it;
38  /* 输出迭代器指向元素的地址，具体用法可以参照迭代器章节 */
39  template <typename ostream>
40  friend ostream& output(ostream& os, const riterator& rhs) { return output(os,rhs.it);
        }
41  };
```

看到上面的代码可能有一些迷糊，从下面的定义我们可以看到 reverse_iterator 就是 iterator it 作为其成员，const_reverse_iterator 就是 const_iterator it 作为成员。

```
1  typedef riterator<       iterator>       reverse_iterator;
2  typedef riterator<const_iterator> const_reverse_iterator;
```

还有一些对迭代器的操作，主要就是返回开始和结束的迭代器，看 reverse_iterator cbegin() 就会对 reverse iterator 有更一步的了解，定义如下：

```
1               iterator   begin()       { return iterator(head); }
2               iterator     end()       { return iterator(head+siz); }
3
4       reverse_iterator  rbegin()       { return reverse_iterator(--end()); }
5       reverse_iterator    rend()       { return reverse_iterator(--begin()); }
6
7         const_iterator  cbegin() const { return const_iterator(head); }
8         const_iterator    cend() const { return const_iterator(head+siz); }
9
10 const_reverse_iterator crbegin() const { return const_reverse_iterator(--cend()); }
11 const_reverse_iterator   crend() const { return const_reverse_iterator(--cbegin()); }
```

在下面的例子中，我们将测试 iterator。

4.3.3 vector 的构造和内存管理

vector 提供 4 个构造函数，一个基础的 init 来分配新的内存，当需要的时候 new siz+inc 个大小的空间并调用 T 的构造函数，并把起始地址赋值给 head，返回 head。具体的定义如下：(注意委托构造函数需要 c++11 标准的支持)

```
1  ...
2  private:
3      /* base init function */
4      T* init(unsigned S) { return head = new T[cap=(siz=S)+inc]; }
5  ...
6
7  public
8
9      /* explicit construct, new array[n], all value is   x*/
10     /* 构造 siz 为 n 的 vector, 全部值赋值为 x，它委托下一个构造函数去完成*/
11     explicit vector(unsigned n=0, const T& x=T()) : vector(sequence<T>(assign<T>(x))
           ,0,n) {}
12
13     template <typename Iter>
14     vector(Iter src, int pos, unsigned len) : vector(src+pos, src+pos+len) {}
15     /* 构造 first，second 之间的元素，如果 first > second 则说明 second 是前面的元素，需要 [second, first) 区
           间，则调用 rcopy */
16     template <typename Iter>
17     vector(Iter first, Iter second) {
18         int n = second - first;
19         /* 当 n<0   时，使用 rcopy(init(-), first, second) 或者如下  */
20         if (n < 0) DSAA::copy(init(-n), second, first); else DSAA::copy(init(n), first
               , second);
21     }
22     /* 使用另外一个 vector 去构造, 直接调用 DSAA::copy */
```

```
    // 注意原始是   rhs.head+siz，因为 siz 是未知的，所以地址越界，发生错误
    vector(const vector& rhs) { DSAA::copy(init(rhs.siz), rhs.head, rhs.head+rhs.siz);
        }
...
```

从定义中我们可以看到前两个构造函数都是委托第三个来完成的，下面我们来测试这些构造函数。注意第一个构造函数的歧义（详情可看第五章）

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

int main ()
{

  DSAA::vector<int> first;                                // empty vector of ints
  for(int i =0; i < 4; ++i)
      first.push_back(i);

  int myints[] = {16,2,77,29};
  DSAA::vector<int> second (myints, 0, 3); //指定元素，第二个参数为位置，第三个为长度
  DSAA::vector<int> third (second.begin() , second.end());
  DSAA::vector<int> third2 (third.end() , third.begin());  // rcopy
  DSAA::vector<int> fourth (third2);                       // a copy of third


  std::cout << "The contents of first are:";
  for (DSAA::vector<int>::iterator it = first.begin(); it != first.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';


  std::cout << "The contents of second are:";
  for (DSAA::vector<int>::iterator it = second.begin(); it != second.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  std::cout << "The contents of third are:";
  for (DSAA::vector<int>::iterator it = third.begin(); it != third.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  std::cout << "The contents of third2 are:";
  for (DSAA::vector<int>::iterator it = third2.begin(); it != third2.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  std::cout << "The contents of forth are:";
  for (DSAA::vector<int>::iterator it = fourth.begin(); it != fourth.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```

输出结果如下：

```
The contents of first are: 0 1 2 3
The contents of second are: 16 2 77
The contents of third are: 16 2 77
The contents of third2 are: 16 2 77
The contents of forth are: 16 2 77
```

4.3.4 vector 的元素操作

vector 所提供的元素操作太多了，我们主要讲解插入和删除的操作。

4.3.4.1 front() & back() & operator[]

front() 返回第一个元素，back() 返回最后一个元素，都是 cosnt 引用。operator[] 有两个版本，一个返回引用，一个返回常引用。下下面例中

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


template <typename T>
void mycopy(const DSAA::vector<T> &from, DSAA::vector<T> &to) {
    for(int i = 0; i < to.size(); ++i) {
     //    to[i] = from[i];
        from[i] = to[i];
    }
}


int main(int argc, char **argv) {
    DSAA::vector<int> to(5);
    DSAA::vector<int> from((unsigned)6, 10);
    std::cout << "from: \n";
    for(int i = 0; i < 5; ++i)
        std::cout << from[i] << " ";
    std::cout << '\n';


    std::cout << "to: \n";
    for(int i = 0; i < 5; ++i)
        std::cout << to[i] << " ";
    std::cout << '\n';

    mycopy(from, to);

    std::cout << "after copy: \n";
    std::cout << "from: \n";
    for(int i = 0; i < 5; ++i)
        std::cout << from[i] << " ";
    std::cout << '\n';


    std::cout << "to: \n";
    for(int i = 0; i < 5; ++i)
        std::cout << to[i] << " ";
    std::cout << '\n';
    return 0;
}
```

我们一不小心将 from 和 to 的顺序调换，编译的时候就会报错，如下：

```
./testDSAA/ext/vectorOperator[].cc: In instantiation of
    'void mycopy(const DSAA::vector<T>&, DSAA::vector<T>&) [with T = int']:
./testDSAA/ext/vectorOperator[].cc:28:20:   required from here
./testDSAA/ext/vectorOperator[].cc:9:17: error: assignment of read-only location
    '(& from)->DSAA::vector<T>::operator[]<int>(i)'
         from[i] = to[i];
                 ^
```

具体定义如下：

```
      T& operator[](int i)       { return head[i]; }
const T& operator[](int i) const { return head[i]; }


const T& front() const { return *head; }
const T& back () const { return *(head+siz-1); }
```

4.3.4.2 insert & erase

vector 重要的就是对插入和删除操作之后的空间配置和元素储存。对于 insert 我们分析基础的 inset_before，insert_after 类似。

```
/* 重新设置 vector 的 siz     */
void resize(unsigned len) {
    /* 如果已经没有空间了，new T[len + inc]，并将元素 copy   */
    if (len > cap) {
        cap = len + inc;
        T* t = new T[cap];
        DSAA::copy(t, head, head+siz);
        delete[] head;
        head = t;
    }
    siz = len;
}
/* 在 pos 位置插入 x   */
bool insert_before(int pos, const T& x) {
    /* 判断位置是否错误 */
    if (pos < 0 || pos > siz) return 0;
    unsigned size = siz;

    /* 改变 size, 如果不够，则会重新申请空间并释放空间   */
    resize(size+1);
    T *p = head+pos-1, *q = head+size-1;
    /*注意 reverse_iterator 重载了 -和 ++ 的运算，看一下源码就懂了*/
    reverse_iterator r = rbegin();
    *DSAA::rcopy(r, p, q) = x;
    return 1;
}

/* 在 pos 位置 (head[pos]) 插入 [first, second), 并把元素后移。  */
template <typename Iter>
bool insert_before(int pos, Iter first, Iter second) {
    if (pos < 0 || pos > siz) return 0;
    /* 确保 m 的值更大   */
    int n = second - first, m = n < 0 ? -n : n;
    if (n == 0) return 1;
    unsigned size = siz;
    resize(size+m);
    T *p = head+pos-1, *q = head+size-1;
    reverse_iterator r = rbegin();
    r = DSAA::rcopy(r, p, q);
    /* 巧妙的运用, 如果 first    > second, (second, first] 从右到左放到 vector 中   */
    if (n < 0) DSAA::copy(r, ++second, ++first); else DSAA::rcopy(r, --first, --second
        );
    return 1;
}
```

其他的一些操作都是基于上面两个函数 (还有 insert_after 类似的操作和原理就不叙述了)，它的插入时间是线性的，通过下面的例子来简单的测试一下。

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

int main ()
{
    DSAA::vector<int> myvector ((unsigned)1,100);
    DSAA::vector<int>::iterator it;

    it = myvector.begin();
    myvector.insert ( it , 200 );
    myvector.insert ( 2 , 600 );
    myvector.insert ( 2 , 700 );

    int myarray [] = { 501,502,503 };
```

```
//    std::cout << *myarray << " " << *(myarray+1) << '\n';
//    std::cout << *(myvector.begin() + myvector.size()-1) << "ssss";
    myvector.insert (myvector.begin(), myarray+3, myarray);

    myvector.insert ( 1 , 800 );
    myvector.insert ( 0 , 8 );
    std::cout << "myvector contains:";
    for (it=myvector.begin(); it != myvector.end(); it++)
        std::cout << ' ' << *it;
    std::cout << '\n';

    return 0;
}
```

输出结果如下：

```
myvector contains: 8 100 800 503 502 200 100 700 600
```

erase(pos, unsigned len=1) 删除 head[pos] 的元素，默认长度为 1，注意是第 pos+1 个元素。其它的删除操作都是基于这个函数的。不用说，删除也是线性时间 O(N)。

```
bool erase(int pos, unsigned len=1) {
    if (pos < 0 || pos >= siz) return 0;
    if (len == 0 || pos+len > siz) len = siz-pos;
    copy(head+pos, head+pos+len, head+siz);
    siz -= len;
    return 1;
}
```

通过下面的例子来测试一下：

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

int main ()
{
    DSAA::vector<int> myvector ((unsigned)1,100);
    DSAA::vector<int>::iterator it;

    it = myvector.begin();
    myvector.insert ( it , 200 );
    myvector.insert ( 2 , 600 );
    myvector.insert ( 2 , 700 );

    int myarray [] = { 501,502,503 };
//    std::cout << *myarray << " " << *(myarray+1) << '\n';
//    std::cout << *(myvector.begin() + myvector.size()-1) << "ssss";
    myvector.insert (myvector.begin(), myarray+3, myarray);

//myvector contains: 8 100 800 503 502 200 100 700 600
    myvector.insert ( 1 , 800 );
    myvector.insert ( 0 , 8 );
    std::cout << "myvector contains:";
    for (it=myvector.begin(); it != myvector.end(); it++)
        std::cout << ' ' << *it;
    std::cout << '\n';
    myvector.erase(4);
    std::cout << "myvector contains(erase(4)):";
    for (it=myvector.begin(); it != myvector.end(); it++)
        std::cout << ' ' << *it;
    std::cout << '\n';

    return 0;
}
```

输出结果如下：

```
myvector contains: 8 100 800 503 502 200 100 700 600
myvector contains(erase(4)): 8 100 800 503 200 100 700 600
```

4.3.4.3 clear()

我们来看一下 clear() 的定义

```
void clear() { siz = 0; }
```

就是把 siz 赋值为零，没有把空间释放给堆栈，看看以下测试会出现什么情况呢？

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

int main() {
    DSAA::vector<char> myvector(5, 'a');
    myvector.clear();
    std::cout << myvector.size() << " " << myvector.empty() << std::endl;
    DSAA::vector<char>::iterator ite = myvector.begin();
    for(int i = 0; i < 5; ++i, ++ite)
    std::cout << myvector[i] << ' ' << *ite << ' ';
    return 0;
}
```

输出结果如下：

```
0 1
a a a a a a a a a a
```

其实这就像 C++ 一样，我可以你给你使用指针的权利，但是不能保证你使用指针都能访问到你想要的东西，有可能你越界，开发容器我也给你随意访问的权利，但是也不保证就是你想要的。

4.4 list

相对于 vector 的连续线性空间，list 就显得复杂许多，它的好处在于每次插入和删除都只是配置或释放一个元素的空间，充分利用空间, 并且这些都是常量时间完成的。该实验有 dlist (双向链表) 和 slist (单向链表)。这里我们就一起来分析下，这样对比着更容易理解。

4.4.1 list 的节点 (node)

list 本身和 list 节点是不同的结构需要分开设计，下面是 slist 和 dlist 的 node 的定义。

```
/* slist node */
struct node {
    union { T data, element; };
    node* next;
    node(const T& x=T(), node* N=0) : data(x),next(N) {}
};

/* dlist node */
struct node {
    union { T data, element; };
    node *prev, *next;
    node(const T& x=T(), node* P=0, node* N=0) : data(x),prev(P),next(N) {}
};
```

从上面定一我们知道，slist 就是一个结构体，里面一个储存数据 data (注意 union 的特性，所有不会有空间的浪费), 还有一个节点指针指向下一个节点，而 dlist 对于 slist 有两个结点指针分别指向上一个和下一个节点。

4.4.2 list 迭代器

```
template <typename T>
...
public:
    /* Forward Iterator */
    class iterator {
        node* ptr;
    public:
        iterator(node* P=0) : ptr(P) {}

        iterator& operator++()    { ptr = ptr->next; return *this; }
        iterator  operator++(int) { iterator it(ptr); ptr = ptr->next; return it; }
        T&        operator* ()                    { return ptr->next->data; }
        bool operator==(const iterator& rhs) const { return ptr->next==rhs.ptr->next;
            }
        bool operator!=(const iterator& rhs) const { return ptr->next!=rhs.ptr->next;
            }

        friend class slist;
    };

    class const_iterator {
        node* ptr;
    public:
        const_iterator(node* P=0) : ptr(P) {}

        const_iterator& operator++()        { ptr = ptr->next; return *this; }
        const_iterator  operator++(int)     { const_iterator it(ptr); ptr = ptr->next;
             return it; }
        const T&        operator*()         const { return ptr->next->data; }
        bool operator==(const_iterator rhs) const { return ptr->next==rhs.ptr->next; }
        bool operator!=(const_iterator rhs) const { return ptr->next!=rhs.ptr->next; }

        friend class slist;
    };
...
```

```
33
34 template <typename T>
35 class dlist{
36 ...
37 /* Bidirectional Iterator */
38     class iterator {
39         node* ptr;
40     public:
41         iterator(node* P) : ptr(P) {}
42
43         iterator& operator++()    { ptr = ptr->next; return *this; }
44         iterator  operator++(int) { iterator it(ptr); ptr = ptr->next; return it; }
45         iterator& operator--()    { ptr = ptr->prev; return *this; }
46         iterator  operator--(int) { iterator it(ptr); ptr = ptr->prev; return it; }
47         T& operator*()            { return ptr->data; }
48         bool operator==(const iterator& rhs) const { return ptr==rhs.ptr; }
49         bool operator!=(const iterator& rhs) const { return ptr!=rhs.ptr; }
50
51         friend class dlist;
52     };
53
54     class const_iterator {
55         node* ptr;
56     public:
57         const_iterator(node* P) : ptr(P) {}
58
59         const_iterator& operator++()      { ptr = ptr->next; return *this; }
60         const_iterator  operator++(int)   { const_iterator it(ptr); ptr = ptr->next;
               return it; }
61         const_iterator& operator--()      { ptr = ptr->prev; return *this; }
62         const_iterator  operator--(int)   { const_iterator it(ptr); ptr = ptr->prev;
               return it; }
63         const T&        operator*() const { return ptr->data; }
64         bool operator==(const const_iterator& rhs) const { return ptr==rhs.ptr; }
65         bool operator!=(const const_iterator& rhs) const { return ptr!=rhs.ptr; }
66
67         friend class dlist;
68     };
69
70     class reverse_iterator {
71         node* ptr;
72     public:
73         reverse_iterator(node* P) : ptr(P) {}
74
75         reverse_iterator& operator++()    { ptr = ptr->prev; return *this; }
76         reverse_iterator  operator++(int) { reverse_iterator it(ptr); ptr = ptr->prev;
                return it; }
77         reverse_iterator& operator--()    { ptr = ptr->next; return *this; }
78         reverse_iterator  operator--(int) { reverse_iterator it(ptr); ptr = ptr->next;
                return it; }
79         T& operator*()                    { return ptr->data; }
80         bool operator==(const reverse_iterator& rhs) const { return ptr==rhs.ptr; }
81         bool operator!=(const reverse_iterator& rhs) const { return ptr!=rhs.ptr; }
82
83         friend class dlist;
84     };
85
86     class const_reverse_iterator {
87         node* ptr;
88     public:
89         const_reverse_iterator(node* P) : ptr(P) {}
90
91         const_reverse_iterator& operator++()      { ptr = ptr->prev; return *this; }
92         const_reverse_iterator  operator++(int)   { const_reverse_iterator p(ptr); ptr
               =ptr->prev; return p; }
93         const_reverse_iterator& operator--()      { ptr = ptr->next; return *this; }
```

```
94          const_reverse_iterator  operator--(int)   { const_reverse_iterator p(ptr); ptr
                =ptr->next; return p; }
95          const T&                operator*() const { return ptr->data; }
96          bool operator==(const const_reverse_iterator& rhs) const { return ptr==rhs.ptr
                ; }
97          bool operator!=(const const_reverse_iterator& rhs) const { return ptr!=rhs.ptr
                ; }
98
99          friend class dlist;
100     };
101 ...
102 };
```

从定义中我们可以看出，slist 的迭代器是一种单向移动的迭代器，而 dlist 是一种双向移动的迭代器。注意 iterator 和 const_iterator 的区别，dlist 还提供 reverse_iterator。

4.4.3 list 的数据结构

在该 list 中，都有一个 *head 和 *rear 分别标识开始和结束，它们都不存放数据。具体定义如下：

```
1  template <typename T>
2  class slist{
3  ...
4  private:
5      node *head, *rear;
6  }
7
8  template <typename T>
9  class dlist{
10 ...
11 private:
12     node *head, *rear;
13 ...
14 }
```

4.4.4 list 的构造

```
1  template <typename T>
2  class slist{
3  ...
4      slist() : head(new node) { rear = head; }
5      slist(const slist& rhs) : head(new node) {
6          rear = head;
7          node *p = head, *q = rhs.head;
8          while (q != rhs.rear) { insert(p, new node(*(q=q->next))); p = p->next; }
9      }
10     ~slist() { clear(); delete head; }
11 ...
12 };
13 template <typename T>
14 class dlist{
15 ...
16     dlist() : head(new node),rear(new node(T(),head)) { head->next = rear; }
17     dlist(const dlist& rhs) : head(new node),rear(new node(T(),head)) {
18         head->next = rear;
19         for (node *p=rhs.head->next; p!=rhs.rear; p=p->next) insert(rear, p->data);
20     }
21     ~dlist() { clear(); delete head; delete rear; }
22 ...
23 };
```

注意 head 和 rear 都是不储存数据的，所以 head=rear 就是表示是空链表。

4.4.5 list 的 insert 和 erase

插入和删除是 list 最为关键的一部分，先对 insert 进行分析。

```
/* slist */
    node* insert(node* p, node* q) { q->next = p->next; p->next = q; if (rear == p)
        rear = q; return q; }
    node* insert(node* p, const T& x) { return insert(p, new node(x)); }

/* dlist */
    node* insert_before(node* p, node* r) { r->prev = p->prev; r->next = p; return p->
        prev=p->prev->next=r; }
    node* insert_after (node* p, node* r) { r->next = p->next; r->prev = p; return p->
        next=p->next->prev=r; }
    node* insert       (node* p, node* r)     { return insert_before(p, r); }
    node* insert_before(node* p, const T& x) { return insert(p, new node(x)); }
    node* insert       (node* p, const T& x) { return insert_before(p, x); }
    node* insert_after (node* p, const T& x) { return insert_after(p, new node(x)); }
```

这些操作理解都不会很难，注意 dlist 还提过一个 insert_after 的操作。erase 操作也就是把该结点删除，插入和删除的操作的时间都是常量的。

```
/* dlist */
    void erase(node* r) { r->prev->next = r->next; r->next->prev = r->prev; delete r;
        }
/* slist */
    void  erase (node* p) { node* q = p->next; p->next = q->next; if (q == rear) rear
        = p; delete q; }
```

注意 slist 的 erase 删除的是 p->next，insert 和 erase 都是基础，供 remove，push_back 等等调用的。通过下面的例子来测试一下。

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

int main ()
{
    DSAA::dlist<int> mylist;
    DSAA::dlist<int>::iterator it = mylist.begin();

    mylist.push_back (100);
    mylist.push_back (200);
    mylist.push_back (300);

    std::cout << "mylist contains:";
    for (it=mylist.begin(); it!=mylist.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';

    mylist.clear();
    mylist.push_back (1101);
    mylist.push_back (2202);

    std::cout << "mylist contains:";
    for (it=mylist.begin(); it!=mylist.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';

    return 0;
}
```

输出结果如下：

```
mylist contains: 100 200 300
mylist contains: 1101 2202
```

Chapter 5

总结

通过这个实验，让我们知道 STL 中的一些组建是如何运行的，虽然并不能完全了解。也让自己更加认识到自己的不足，必须的把一些经典的计算机的书籍看完。

5.1 待解决的问题

5.1.1 vector 中普通构造函数与模板构造函数二义性

这个问题按照函数选择规则应该是不存在，似乎可用模板特例化解决。问题具体如下：

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

int main() {
    DSAA::vector<int>myvector(4, 4);
}
```

编译出错，我们可以看到它调用的模板构造函数。如下：

```
In file included from ./testDSAA/vector.h:26:0,
                 from ./testDSAA/endVector1.cc:2:
./testDSAA/algorithm.h: In instantiation of
    'Iter1 DSAA::copy(Iter1, Iter2, Iter2) [with Iter1 = int*; Iter2 = int']:
./testDSAA/vector.h:139:54:   required from
    'DSAA::vector<T>::vector(Iter, Iter) [with Iter = int; T = int]'
./testDSAA/endVector1.cc:5:35:   required from here
./testDSAA/algorithm.h:69:71: error: invalid type argument of unary '*' (have 'int')
 Iter1 copy(Iter1 d, Iter2 it1, Iter2 it2) { while (it1 != it2) *d++ = *it1++; return
     d; }
                                                                       ^
```

我们在对以下测试可以进一步知道，确实是调用二义性。

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

int main() {
    DSAA::vector<int>myvector((unsigned)4, 10);
    DSAA::vector<int>::iterator ite = myvector.begin();
    for(ite = myvector.begin(); ite != myvector.end(); ++ite)
        std::cout << *ite << ' ';
    std::cout << '\n';
    return 0;
}
```

输出结果如下：

```
10 10 10 10
```

5.1.2 vector 中的 const_iterator

在 vector 中使用 const_iterator it，对 it 进行解引用，返回的并不是 const 引用，如下例所示，并不会编译错误。

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

int main() {
    DSAA::vector<int> myvector;
    for(int i = 1; i < 6; ++i)
        myvector.push_back(i);
    DSAA::vector<int>::const_iterator it1 = myvector.cbegin();
    *it1 = 7;
    std::cout << "myvecctor is:";
    std::cout << '\n';
    for(it1 = myvector.begin(); it1 != myvector.end(); ++it1)
        std::cout << ' ' << *it1;
    std::cout << '\n';
    return 0;
}
```

并不会产生编译错误，输出结果如下：

```
myvecctor is:
 7 2 3 4 5
```

分析 const_iterator 的定义，发现它是继承自 iteraor, 而 iterator 有返回不是常引用的 operator* 操作符重载，所以不能达到目地，解决方法应该是在先定义 const_iterator，iterator 继承 const_ierator。等考完试，再重写看看。