USTC

Chapter 19
Vectors, templates, and exceptions

王子磊 (Zilei Wang)
Email: zlwang@ustc.edu.cn
http://vim.ustc.edu.cn

Overview

- 本章是探索标准库中 vector 实现、技术、语言特性等内容的第三部分
- 此处，我们将主要处理
  - 改变vector的大小
  - 参数化vector的元素类型 (模板)
  - 范围检查 (异常)

处理可变性

Contents

- 重新审视向量
  - 它们是如何实现的？
- 指针和自由存储
  - 内存分配 (new)
  - 元素访问
    - 数组和下标: []
    - 使用对象: *
  - 内存释放 (delete)
- 析构

- 拷贝构造函数和拷贝赋值函数
- 数组
- 数组和指针

- 改变大小
- 模板
- 范围检查和异常

改变 vector 大小

- 基本问题描述
  - 我们期望那种能够动态改变大小的抽象(如向量能够改变它元素的个数)，然而，在计算机内存中，所有的东西都是固定大小的
  - 如何创建/实现这种变化性？
- 给定

```
vector v(n);                    // v.size()==n
```

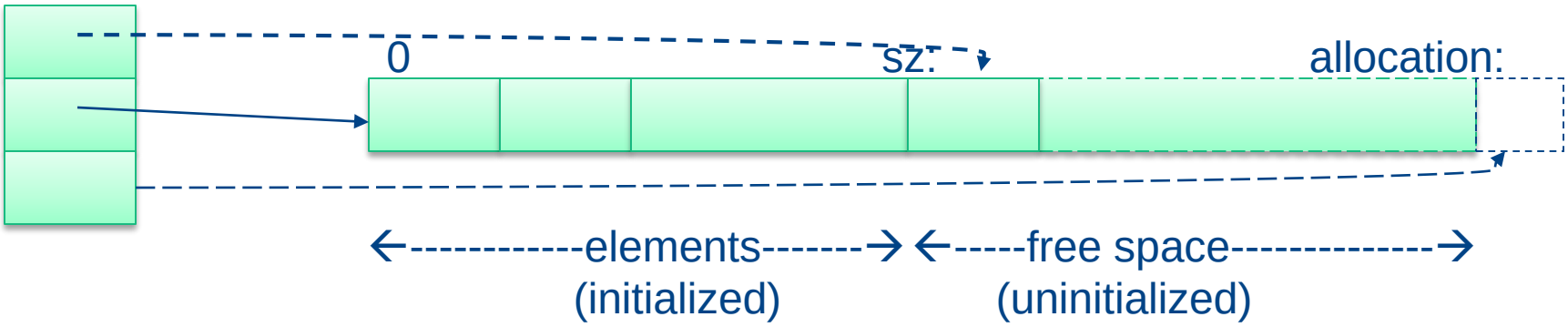
我们能够通过三种方式来改变它的大小

- Resize
  - **v.resize(10);** // *v now has **10** elements*
- 添加一个元素
  - **v.push_back(7);** // *add an element with the value **7** to the end of **v***
    // ***v.size()** increases by **1***
- 赋值
  - **v = v2;** // *v is now a copy of **v2***
    // ***v.size()** now equals **v2.size()***

vector 描述

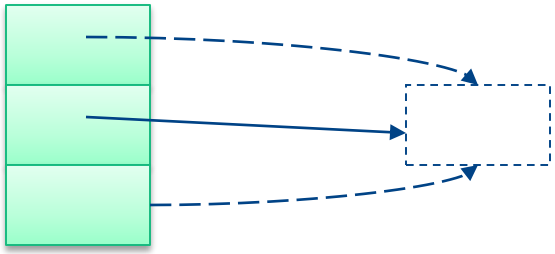
- 如果你使用 resize() 或 push_back() 一次，你很可能还会如此操作
  - 准备一些空余空间，可以用于将来的扩展使用

```
class vector {
    int sz;
    double* elem;
    int space;          // number of elements plus "free space"
                        // (the number of "slots" for new elements)
public:
    // …
};
```

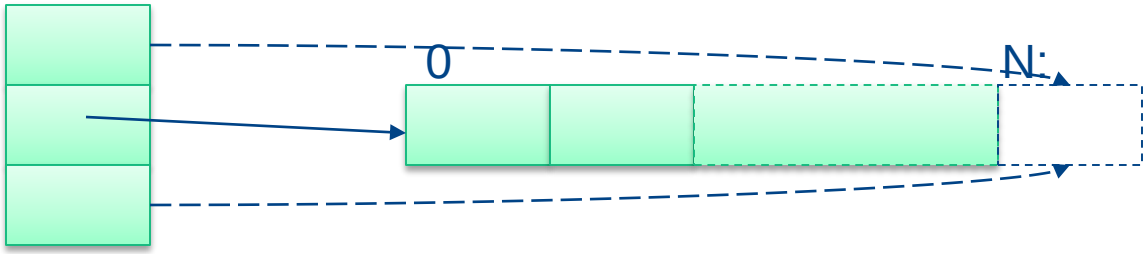


vector 描述

- 一个空 vector (没有使用自由存储):



- 一个 vector(N) (没有空余空间)



vector::reserve()

- 首先处理 space 分配，搞定space后，其余的都很容易
  - Note：**reserve()** 不处理 size 和元素值

```
void vector::reserve(int newalloc)
    // make the vector have space for newalloc elements
{
    if (newalloc<=space) return;              // never decrease allocation
    double* p = new double[newalloc];         // allocate new space
    for (int i=0; i<sz; ++i) p[i]=elem[i];    // copy old elements
    delete[ ] elem;                           // deallocate old space
    elem = p;
    space = newalloc;
}
```

vector::resize()

- 给定 reserve()后， resize() 是容易的
  - reserve() 处理 space 的分配
  - resize() 处理元素值

```
void vector::resize(int newsize)
    // make the vector have newsize elements
    // initialize each new element with the default value 0.0
{
    reserve(newsize);                    // make sure we have sufficient space
    for(int i = sz; i<newsize; ++i) elem[i] = 0;        // initialize new elements
    sz = newsize;
}
```

如果不需要重新分配，reserve会自动返回

vector::push_back()

- 给定 reserve()后， push_back()是容易的
  - reserve() 处理 space 的分配
  - push_back() 仅仅是增加一个值

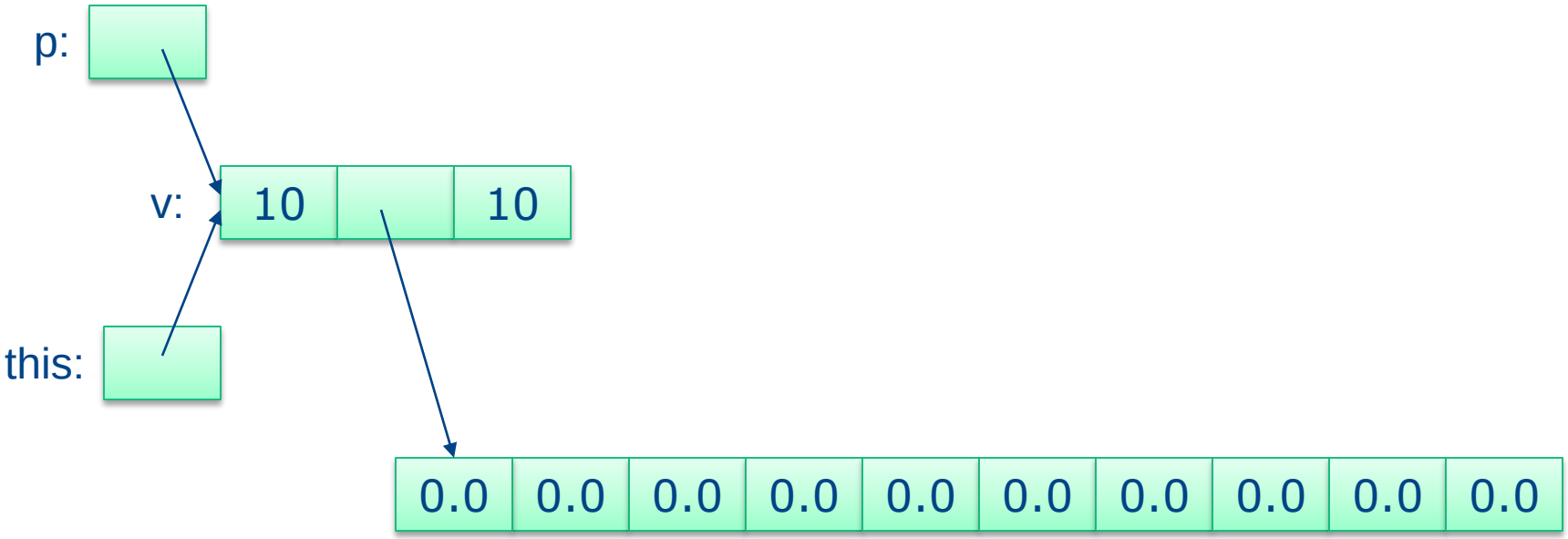
```
void vector::push_back(double d)
    // increase vector size by one
    // initialize the new element with d
{
    if (sz==0)                  // no space: grab some
        reserve(8);
    else if (sz==space)         // no more free space: get more space
        reserve(2*space);
    elem[sz] = d;               // add d at end
    ++sz;               // and increase the size (sz is the number of elements)
}
```

resize() and push_back()

```
class vector {        // an almost real vector of doubles
    int sz;                   // the size
    double* elem;             // a pointer to the elements
    int space;                // size+free_space
public:
    vector() : sz(0), elem(0), space(0) { }                    // default constructor
    explicit vector(int s) :sz(s), elem(new double[s]) , space(s) { }    // constructor
    vector(const vector&);                                     // copy constructor
    vector& operator=(const vector&);                          // copy assignment
    ~vector() { delete[ ] elem; }                              // destructor

    double& operator[ ](int n) { return elem[n]; }             // access: return reference
    int size() const { return sz; }                            // current size

    void resize(int newsize);                                  // grow (or shrink)
    void push_back(double d);                                  // add element

    void reserve(int newalloc);                                // get more space
    int capacity() const { return space; }                     // current available space
};
```

this 指针

- 一个 vector 是一个对象
  - **vector v(10);**
  - **vector* p = &v;** *// we can point to a **vector** object*
- 有时，**vector** 的成员函数需要引用对象本身
  - 在成员函数中，“指向自身的指针”是 **this**



this 指针

```
vector& vector::operator=(const vector& a)
    // like copy constructor, but we must deal with old elements
{
    // …
    return *this;    // by convention,
                     // assignment returns a reference to its object: *this
}

void f(vector v1, vector v2, vector v3)
{
    // …
    v1 = v2 = v3;    // rare use made possible by operator=() returning *this
    // …
}
```

- this 指针我们已经使用过，其意义是显而易见的

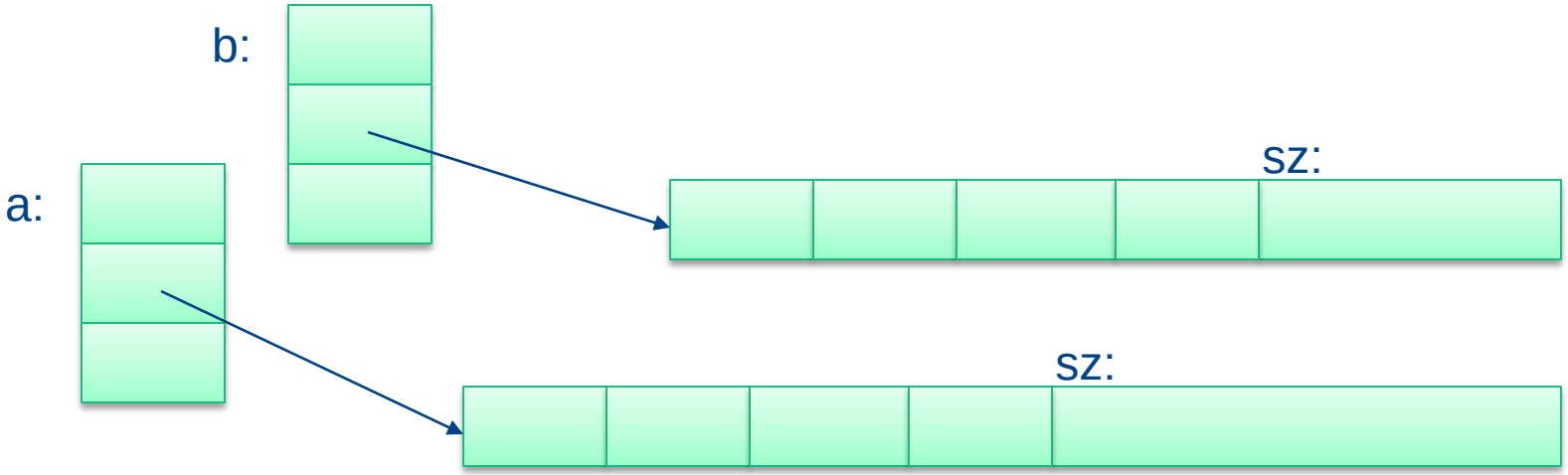
赋值

- “Copy and swap”是一种强大的通用思路

```
vector& vector::operator=(const vector& a)
    // like copy constructor, but we must deal with old elements
    // make a copy of a then replace the current sz and elem with a's
{
    double* p = new double[a.sz];                    // allocate new space
    for (int i = 0; i<a.sz; ++i) p[i] = a.elem[i];   // copy elements
    delete[ ] elem;                                  // deallocate old space
    sz = a.sz;                                       // set new size
    elem = p;                                        // set new elements
    return *this;              //  return a self-reference
}
```

赋值优化

- “Copy and swap” 是最一般化的思想
  - 但它并不总是最有效的
  - 如果目标 vector 已经有足够的存储空间，我们怎么做？
    - 仅仅拷贝
    - 例如：**a = b;**



赋值优化

所有赋值函数的固定语句

```
vector& vector::operator=(const vector& a)
{
    if (this==&a) return *this;          // self-assignment, no work needed

    if (a.sz<=space) {                   // enough space, no need for new allocation
        for (int i = 0; i<a.sz; ++i) elem[i] = a.elem[i];    // copy elements
        space += sz-a.sz;                                    // increase free space
        sz = a.sz;
        return *this;
    }

    double* p = new double[a.sz];                            // copy and swap
    for (int i = 0; i<a.sz; ++i) p[i] = a.elem[i];
    delete[ ] elem;
    sz = a.sz;
    space = a.sz;
    elem = p;
    return *this;
}
```

模板

- 如果我们想让 vector 不仅仅是double的
- 我们期望 vector 能够处理我们指定的类型
  - **vector<double>**
  - **vector<int>**
  - **vector<Month>**
  - **vector<Record*>** *// vector of pointers*
  - **vector< vector<Record> >** *// vector of vectors*
  - **vector<char>**
- 我们必须将元素类型当做参数传递给 **vector**
- **vector** 必须能够处理内建类型和用户自定义类型
- 对编译器，这并不是需要保留的魔法，我们能够定义自己的参数化类型，称作为“模板”

这些是 vector 部分的设计需求

模板

- 它是 C++ 中泛型编程的基础
  - 有时称为“参数化多态”
    - 通过类型和整数指定模板类型的参数化
  - 无以伦比的灵活性和性能
    - 当性能要求高时使用它 (*e.g.*, 硬实时和数值计算)
    - 当灵活性要求高时使用它 (*e.g.*, C++ 标准库)

- 模板定义 (语法)

```
template<class T, int N> class Buffer { /* … */ };
template<class T, int N> void fill(Buffer<T,N>& b) { /* … */ }
```

- 模板特例化 (实例化)

```
// for a class template, you specify the template arguments:
Buffer<char,1024> buf;          // for buf, T is char and N is 1024

// for a function template, the compiler deduces the template arguments:
fill(buf);      // for fill(), T is char and N is 1024; that's what buf has
```

元素类型的参数化

```
// an almost real vector of Ts:
template<class T> class vector {
    // ...
};

vector<double> vd;                  // T is double
vector<int> vi;                     // T is int
vector< vector<int> > vvi;          // T is vector<int>
                                    //                  in which T is int
vector<char> vc;                    // T is char
vector<double*> vpd;                // T is double*
vector< vector<double>* > vvpd;     // T is vector<double>*
                                    //                  in which T is double
```

vector<double> 的基本形式

```
// an almost real vector of doubles:
class vector {
    int sz;                     // the size
    double* elem;  // a pointer to the elements
    int space;                  // size+free_space
public:
    vector() : sz(0), elem(0), space(0) { }                 // default constructor
    explicit vector(int s) :sz(s), elem(new double[s]), space(s) { } // constructor
    vector(const vector&);                          // copy constructor
    vector& operator=(const vector&);                       // copy assignment
    ~vector() { delete[ ] elem; }                           // destructor

    double& operator[ ] (int n) { return elem[n]; }         // access: return reference
    int size() const { return sz; }                         // the current size

    // …
};
```

vector<char> 的基本形式

```
// an almost real vector of chars:
class vector {
    int sz;                     // the size
    char* elem;                 // a pointer to the elements
    int space;                  // size+free_space
public:
    vector() : sz(0), elem(0), space(0) { }                 // default constructor
    explicit vector(int s) :sz(s), elem(new char[s]), space(s) { } // constructor
    vector(const vector&);                                  // copy constructor
    vector& operator=(const vector&);                       // copy assignment
    ~vector() { delete[ ] elem; }                           // destructor

    char& operator[ ] (int n) { return elem[n]; }           // access: return reference
    int size() const { return sz; }                         // the current size

    // ...
};
```

vector<T> 的基本形式

```
// an almost real vector of Ts:
template<class T> class vector {        // read "for all types T" (just like in math)
    int sz;                             // the size
    T* elem;                            // a pointer to the elements
    int space;                          // size+free_space
public:
    vector() : sz(0), elem(0), space(0);                // default constructor
    explicit vector(int s) :sz(s), elem(new T[s]), space(s) { }     // constructor
    vector(const vector&);                              // copy constructor
    vector& operator=(const vector&);                   // copy assignment
    ~vector() { delete[ ] elem; }                       // destructor

    T& operator[ ] (int n) { return elem[n]; }  // access: return reference
    int size() const { return sz; }                     // the current size

    // ...
};
```

模板

- 问题(“没有免费的午餐”)
  - 拙劣的出错诊断
    - 程序可能会显示出大量拙劣的出错信息
  - 延迟错误消息
    - 通常在 连接时
  - 所有的模板必须在引用单元(头文件)中完整定义
    - 模板分别编译功能(称为“export”),并不是广泛支持
    - 因此放置模板定义在头文件中
- 建议
  - 使用基于模板的库
    - 例如 C++ 标准库
      - *e.g.*, **vector**, **sort()**
      - 稍后会描述相关细节
  - 开始时,先写非常简单的模板并逐步改进以获得经验
    - 直到你已经拥有较多的经验

范圍檢查

```
// an almost real vector of Ts:

struct out_of_range { /* ... */ };

template<class T> class vector {
    // ...
    T& operator[ ](int n);                  // access
    // ...
};

template<class T>  T& vector<T>::operator[ ](int n)
{
    if (n<0 || sz<=n) throw out_of_range();
    return elem[n];
}
```

范围检查

```
void fill_vec(vector<int>& v, int n)          // initialize v with factorials
{
    for (int i=0; i<n; ++i) v.push_back(factorial(i));
}

int main()
{
    vector<int> v;
    try {
        fill_vec(v,10);
        for (int i=0; i<=v.size(); ++i)
            cout << "v[" << i << "]==" << v[i] << '\n';
    }
    catch (out_of_range) {                    // we'll get here (why?)
        cout << "out of range error";
        return 1;
    }
}
```

原始的异常处理

```
// sometimes we cannot do a complete cleanup

vector<int>* some_function()          // make a filled vector
{
    vector<int>* p = new vector<int>;          // we allocate on free store,
                                               // someone must deallocate
    try {
        fill_vec(*p,10);
        // …
        return p;// all's well; return the filled vector
    }
    catch (…) {
        delete p;  // do our local cleanup
        throw;              // re-throw to allow our caller to deal
    }
}
```

糟糕的程序块，没有完成任何有效功能

异常处理 (更简单但更结构化)

```
// When we use scoped variables cleanup is automatic

vector<int> glob;

void some_other_function()          // make a filled vector
{
    vector<int> v;                  // note: vector handles the deallocation of elements

    fill_vec(v,10);
    // use v
    fill_vec(glob,10);
    // ...
}
```

- 当你认为自己需要一个 try 块时，请仔细考虑一下
  - 不需要它你可能也能处理好

RAII (资源获取即初始化)

- Vector
  - 在构造函数中获取元素的内存
  - 管理它 (改变大小、控制访问等)
  - 在析构函数中返还 (释放) 该内存
- 这是一般化资源管理策略 (称之为 RAII) 的一个特例
  - 也称作为 “作用域化资源管理”
  - 尽可能的使用这种方式
  - 这比其他方式更简单和廉价
  - 它优美地与使用异常的错误处理进行交互
  - 资源的例子有：
    - 内存、文件句柄、套接字、I/O连接、锁、widgets、线程

一点疑惑

- 标准库 vector 没有对 [ ] 操作进行范围检查
- 你现在可以使用
  - 我们的的debug版本 **Vector**，它进行检查
  - 或者标准库版本，它不进行检查(部分实现可能会检查)
- 除非你使用的标准库版本是检查的，否则
  - 在 **std_lib_facilities.h** 中，我们采用一种技巧(宏定义)以重新定义 **vector** 使之代表**Vector**

    #define vector Vector

    (这种技术是恶劣的，因为你看到的代码和编译器看到的不同——在世界的代码中，宏是产生晦涩难懂的错误的一个重要来源)
  - 我们对 **string** 做了同样的事情

标准是怎么样保证的？

```
// the standard library vector doesn't guarantee a range check in operator[ ]:
template<class T> class vector {
    // …
    T& at(int n);                   // checked access
    T& operator[ ](int n);          // unchecked access
};

template<class T>  T& vector<T>::at (int n)
{
    if (n<0 || sz<=n) throw out_of_range();
    return elem[n];
}

template<class T>  T& vector<T>::operator[ ](int n)
{
    return elem[n];
}
```

标准是怎么样保证的？

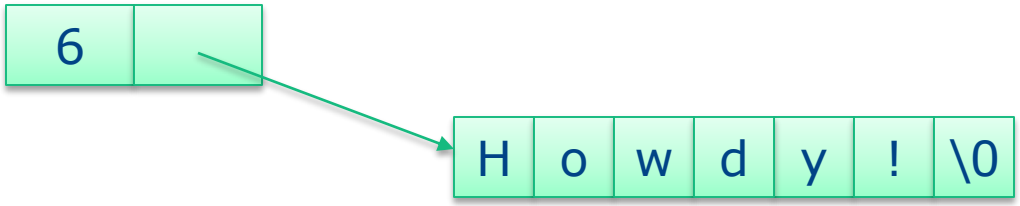
- 为什么标准中不保证检查呢？
  - 速度和代码大小上的检查代价
    - 不严重，不用担心
      - 大家作业中无需担心这一问题
      - 非常少的实际项目需要考虑这一问题
  - 一些项目需要最优的性能
    - 考虑非常大的 (e.g., Google) 或非常小的(e.g., 手机)
  - 标准必须为每个人服务
    - 你能够在优化上层进行检查
    - 你无法在检查之上进行优化
  - 一些项目不允许使用异常
    - 在异常引入之前的老项目
    - 高可靠性、硬实时的代码

vector 的 const 访问

```
template<class T> class vector {
    // …
    T& at(int n);                                   // checked access
    const T& at(int n) const;                       // checked access

    T& operator[ ](int n);                          // unchecked access
    const T& operator[ ](int n) const;              // unchecked access
    // …
};

void f(const vector<double> cvd, vector<double> vd)
{
    // …
    double d1 = cvd[7];         // call the const version of [ ]
    double d2 = vd[7];          // call the non-const version of [ ]
    cvd[7] = 9;                 // error: call the const version of [ ]
    vd[7] = 9;                  // call the non-const version of [ ]: ok
}
```

string

- string 与 vector<char> 非常相像
  - e.g. size(), [ ], push_back()
  - 它们是采用相同的语言特性和技术建立的
- string 为字符操作进行了优化
  - 连接操作 (+)
  - 能够产生 C 风格的字符串 (c_str())
  - >> 通过分隔符进行分割中断



Next

- 介绍 STL，以及C++标准库的容器和算法部分
  - 会看到序列、迭代器、容器 (如：**vector, list, 和 map**)
  - 算法包括 **find(), find_if(), sort(), copy(), copy_if() 和 accumulate()**