Vectors, generic programming (templates), exceptions

Overview

- Vector revisited
  - How are they implemented?
- Pointers and free store
- Destructors
- Copy constructor and copy assignment
- Arrays
- Array and pointer problems
- Changing size
  - resize() and push_back()
- Templates
- Range checking and exceptions

Changing vector size

- Fundamental problem addressed
  - We (humans) want abstractions that can change size (*e.g.*, a vector where we can change the number of elements). However, in computer memory everything must have a fixed size, so how do we create the illusion of change?
- Let's we have

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

We can change its size in three ways:

- Assign to it another vector

```
vector v2(m);
v = v2;               // v is now a copy of v2
                      // now v.size() == v2.size()
```

- Add an element

```
v.push_back(7);  // add an element with the value 7 to the end of v
                 // v.size() is increased by 1
```

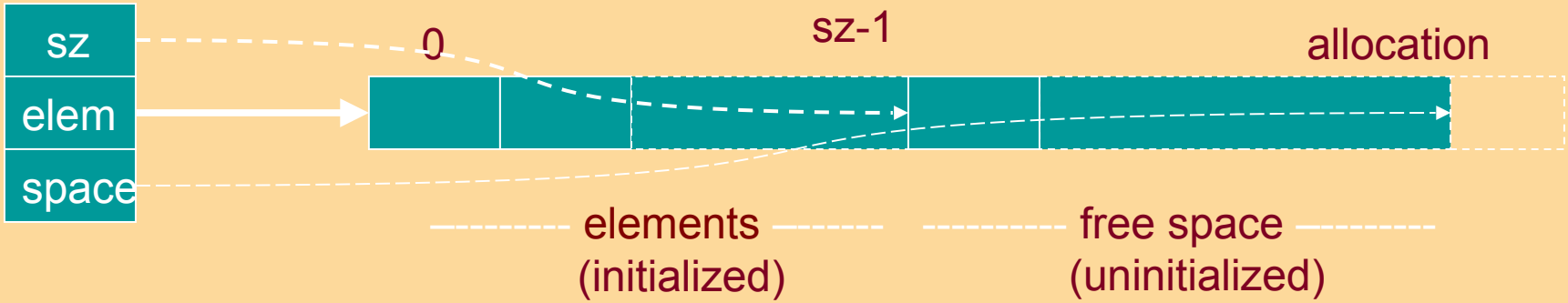
- Resize it using special function (to be implemented)

```
v.resize(m);     // v now has m elements
```

Representing vector

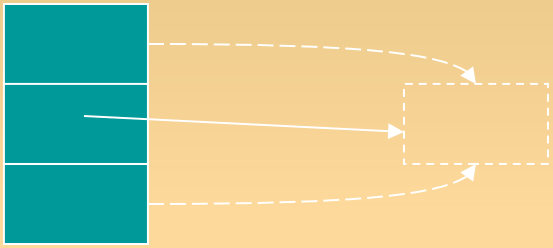
- If you **resize()** or **push_back()** once, you'll probably do it again. Let's be prepared: sometimes we will need a bit of free space for expansion:

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

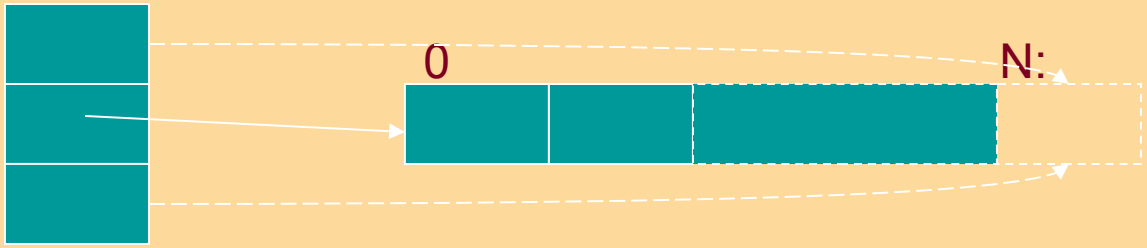


Representing vector

- An empty vector (no free store use):



- A vector(N) (no free space):



vector::reserve()

- First, deal with space allocation

```
// make the vector have memory space for newalloc elements
void vector::reserve(int newalloc) {
  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 elements
  delete[ ] elem;                       // deallocate old space
  elem = p;                  // elem now points to new space
  space = newalloc;          // redefine space
}
```

  - Note: **reserve()** has nothing to do with size or element values

vector::resize()

- Having **reserve()**, it is easy to write **resize()**
  - **reserve()** deals with space allocation
  - **resize()** deals with number of element

```
// change the vector to have newsize elements,
// initialize each new element with the default value 0.0
// if newsize > sz

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

vector::push_back()

- Having **reserve()**, is easy to write **push_back()**
  - **reserve()** deals with space/allocation
  - **push_back()** just adds a value

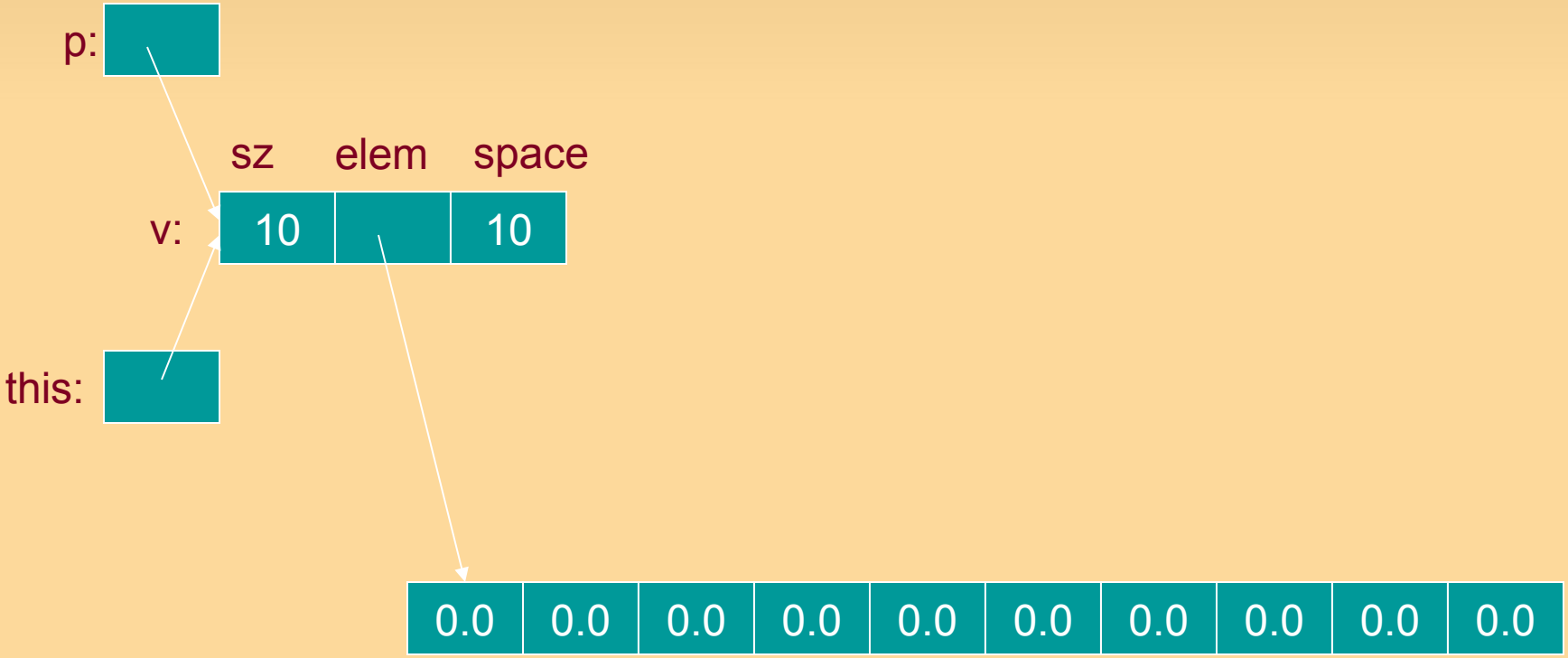
```
// increase vector size by one
// initialize the new element with d
void vector::push_back(double d)
{
    if (sz == 0)                // empty vector: grab some space (here is 8)
        reserve(8);
    else if ( sz == space )     // no more free space: get more space
        reserve(2*space); // ask 2 times more

    elem[sz] = d;   // add d at end
    ++sz;           // and increase the size
}
```

resize() and push_back()

```
class vector {  // an almost real vector of doubles. Class declaration.
    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
    vector(int s) :sz(s), elem(new double[s]) , space(s) { } // constructor
    vector(const vector&);                                  // copy constructor
    vector& operator=(const vector&);                       // assignment operator
    ~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
};
```

The *this* pointer
(once more)

- A vector is an object
  - **vector v(10);**
  - **vector* p = &v;** *// we can point to a **vector** object*
- Sometimes, **vector**'s member functions need to refer to that object
  - The name of that is "pointer to itself".
    Accessible in any member function as **this**



The *this* pointer

```
// assignment operator
vector& vector::operator = (const vector& a)
{
    // …
    return *this; // operator = () must returns a reference to its object:
                  // *this (self-reference)
}

void f(vector v1, vector v2, vector v3)
{
    // …
    v1 = v2 = v3;          // possible because operator = ()
                           // returning *this (self-reference)
    // …
}
```

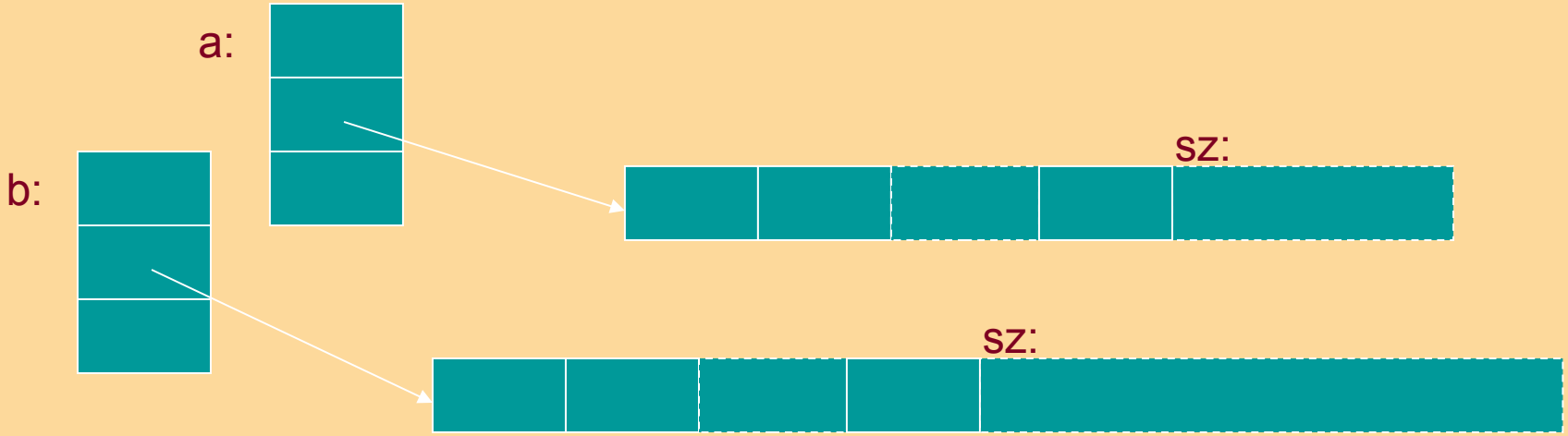
Assignment
(again)

- Copy and swap is a powerful general idea

```
// again assignment operator
vector& vector::operator = (const vector& a)
{
    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 elem (swap)
    return *this;      //  return a self-reference
}
```

Optimize assignment

- "Copy and swap" is the most general idea
  - but not always the most efficient
  - What if there is already sufficient space in the target vector?
    - Then just copy!
    - For example: **b = a;**



Optimized assignment

```
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 (if a vector was smaller than this)
                             // or decrease (if sz < a.sz) free space
        sz = a.sz;           // change size of this vector
        return *this;        // exit
    }

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

Templates

- But we don't want just a vector of double
- We want vectors with elements of any type
  - **vector<double>**
  - **vector<int>**
  - **vector<Month>**
  - **vector<Record*>** *// vector of pointers*
  - **vector<vector<Record> >** *// vector of vectors*
  - **vector<char>**
- I.e. we want to pass to the **vector** the element type as parameter
- **vector** must be able to take both built-in types and user-defined types as element types
- In C++ we can define our own parameterized types, called "templates"

Templates

- Templates is the basis for generic programming in C++ (sometimes called "parametric polymorphism")
  - Could be used for parameterization of classes, functions, literals
  - Used where performance is essential (*e.g.,* real time software, numerics)
  - Used where flexibility is essential (*e.g.,* C++ standard library)

Templates

- Template definitions for classes

```
// class Buffer to store N elements of some type T
template<class T, int N>  class Buffer  {
    T* pBuffer;                              // data member
    Buffer() { pBuffer = new T[N]; }   // constructor
  ~Buffer() { delete[] pBuffer; }        // destructor
    // ...
};

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

Here we parametrize class (T) and literal (N)

Templates

- Template definitions for functions

// function which uses parametrized type
template<class T> void foo(T& a) {
  // do something with object of type **T**
}

// function which uses object of templated class
template<class T, int N> void fill (Buffer<char, 1024>& b) { …. };

// for a function template, compiler deduce the template arguments
// out of function arguments

Buffer<char,1024> buf;
fill(buf); *//* **T** = **char** *and* ***N*** = ***1024*** *will be used inside*

Parameterize vector with elements type

```
// template of vector for type T:
template<class T> class vector {
    // …
    T* elem;
};
```

Examples of use:

```
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> is

```
// 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
   // constructor
    explicit vector(int s) :sz(s), elem(new double[s]), space(s) { }
    vector(const vector&);                           // copy constructor
    vector& operator=(const vector&);                // assignment operator
    ~vector() { delete[ ] elem; }                    // destructor

    double& operator[ ] (int n) { return elem[n]; } // accessor to element n
    int size() const { return sz; }                    // the current size

    // …
};
```

Remark: keyword *explicit*

In C++ by default a single argument constructor also defines an implicit type conversion. For some types it's nice, for some cases isn't not what we want:

```
class vector {
  // ...
  public:
     explicit vector(int s) : { ... }
    // …
};

void f(vector v) {…}  // a function
```

Without **explicit** keyword **:**

```
vector v = 37;              // OK: calls vector(37)
vector v = vector(32);      // OK
vector v (64);              // OK
v = 18;                     // OK
f(11);                      // OK
```

With **explicit** keyword:

```
vector v = 37;              // illegal
vector v = vector(32);      // OK
vector v (64);              // OK
v = 18;                     // illegal
f(11);                      // illegal
```

vector<char> is

```
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
    // constructor with parameter
    explicite vector(int s) :sz(s), elem(new char[s]), space(s) { }
    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
    // …
};          // Identical to vector of double (if to substitute char to double)
```

vector<T> is

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

    T& operator[ ] (int n) { return elem[n]; }      // return reference to T
    int size() const { return sz; }                 // the current size
    // …
};   // compiler "duplicates" code for every T instance
```

Templates

- Problems ("there's no free lunch")
  - Poor error diagnostic
  - Delayed error messages (often at link stage)
  - All templates must be fully defined in each compilation unit (file)
    - So, place template definitions in header files
- Recommendation
  - Use template-based libraries
    - Such as the C++ standard libraries for **vector, list, map** etc...
    - **BOOST** (not a part of C++) matrix, complex variables operations
  - Initially, write only very simple templates yourself
    - Until you get more experience

Range checking

*// an almost real **vector** of **T**s:*

```
class out_of_range { };  // define an exception class
template<class T> class vector {
    // …
    T& operator[ ](int n);          // access
    // …
};
// definition of operator[] outside the class
template<class T>  T& vector<T>::operator[ ](int n)
{
    if (n<0 || sz<=n) throw out_of_range();
    return elem[n];
}
```

Note: in real std::vector **operator[]** do not throw an exceptions. Use **at(int)**

Range checking
(example with use of exceptions)

```
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);
        cout<<v[10]<<endl;      // exception thrown here
    } catch (out_of_range) {    // we'll get here
        cout << "out of range error"<<endl;
        return 1;
    }
}
```

Exception handling (primitive)

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

vector<int>* some_function()    // make a filled vector on heap
{
    vector<int>* p = new vector<int>;   // we allocate on free store,
                                        // someone must deallocate
    try {
        fill_vec((*p),10);              // fill vector
        int j =  (*p)[10];              // exception is thrown here
        return p;           // OK; return  pointer to filled vector
    } catch (…) {           // something wrong. Exception was catched
        delete p;           // do our local cleanup (delete vector on heap)
        throw;              // re-throw to allow our caller to treat error
    }
}
```

Exception handling

```
// When we use scoped variables
// ( = automatic variables, objects "on stack")
// cleanup is automatic and exception
// handling probably is not needed

vector<int> glob;  // global scope vector
void some_other_function() {
    vector<int> v;  // vector itself handles deallocation of elements

    fill_vec(v, 10);
    // ...
    fill_vec(glob, 10);
    // …
}
// v will be destroyed here (destructor function will be called)
// globe will be destroyed after the end of your program
```

RAII (Resource Acquisition Is Initialization)

- Vector
  - Acquires memory for elements in its constructor
  - Manage it (changing size, controlling access, etc.)
  - Gives back (releases) the memory in the destructor
- This is a special case of the general resource management strategy called RAII
  - Also called "scoped resource management"
  - Use it wherever you can
  - It is simpler and cheaper than anything else
  - It interacts beautifully with error handling using exceptions
  - Examples of "resources":
    - Memory, file handles, sockets, I/O connections (iostreams handle those using RAII), locks, widgets, threads.

What the standard guarantees

// 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];
}
```

What the standard guarantees

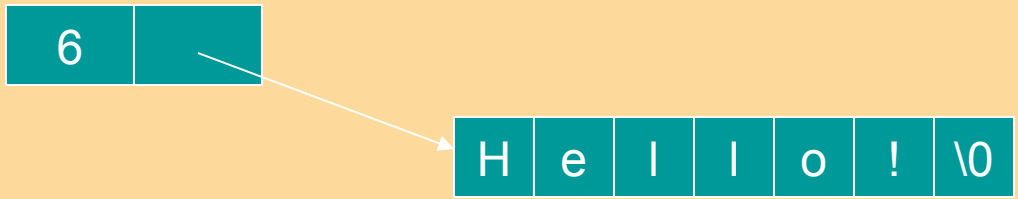
- The standard library **vector** doesn't guarantee range checking of **[ ]**. Why?
  - Checking cost in speed and code size
    - Not much; don't worry
      - No student project needs to worry
      - Few real-world projects need to worry
  - Some projects need optimal performance
  - The standard must serve everybody
    - You can build checked on top of optimal
    - You can't build optimal on top of checked
  - Some projects are not allowed to use exceptions
    - Old projects with pre-exception parts
    - High reliability, hard-real-time code (e.g. programs for avionics)

Access to *const* vector

```
template<class T> class vector {
    // non-const and const versions of accessors are provided
    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: attempt to change const object
    vd[7]  = 9;         // call the non-const version of [ ]: ok
}
```

String

- A **string** is rather similar to a vector<char>
  - E.g. **size(), [ ], push_back(), find()**
  - Built with the same language features and techniques
- A **string** is optimized for character string manipulation
  - Concatenation (**+**)
  - Can produce C-style string (**c_str()**)
  - **>>** input terminated by whitespace and output **<<**



Next talk

- STL vector, STL list
- Algorithms
- Iterators

Practical part for today

- Create some class (or use what you have already) with overloaded streaming operators “<<” and “>>”
- Demonstrate in main() how to read information from keyboard keyboard into your class object using overloaded “>>” operator.
- Demonstrate in main() how to print content of your class objects or save it in a file using “<<” operator.

Practical part for today

- Try to write some example of templates function
- Show how it works for few built-in types