ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

Εισαγωγή στον Προγραμματισμό
Introduction to Programming

Διάλεξη 17: Πρότυπα και Εξαιρέσεις

Γ. Παπαγιαννάκης

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο
ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ & ΘΡΗΣΚΕΥΜΑΤΩΝ, ΠΟΛΙΤΙΣΜΟΥ & ΑΘΛΗΤΙΣΜΟΥ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
ΕΣΠΑ 2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Άδειες Χρήσης

- Το παρόν εκπαιδευτικό υλικό υπόκειται στην άδεια χρήσης **Creative Commons** και ειδικότερα

Αναφορά Δημιουργού 3.0 - Μη εισαγόμενο Ελλάδα
(Attribution 3.0– Unported GR)

- Για εκπαιδευτικό υλικό, όπως εικόνες, που υπόκειται σε άλλου τύπου άδειας χρήσης, η άδεια χρήσης αναφέρεται ρητώς.

Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Κρήτης**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο

ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ & ΘΡΗΣΚΕΥΜΑΤΩΝ, ΠΟΛΙΤΙΣΜΟΥ & ΑΘΛΗΤΙΣΜΟΥ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ

ΕΣΠΑ
2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

HY-150 Προγραμματισμός
CS-150 Programming

Lecture 17:
Templates and exceptions

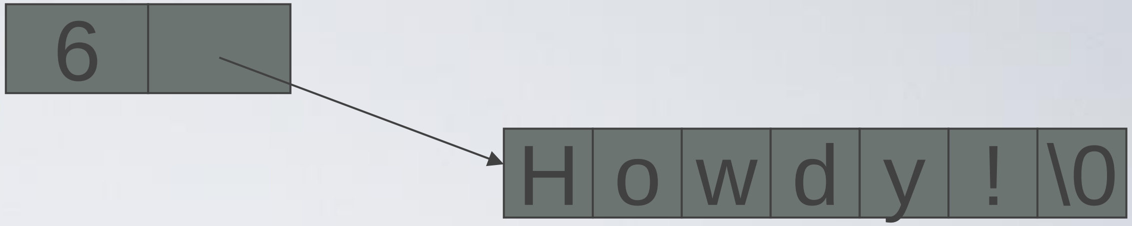
G. Papagiannakis

Abstract

- This is the third of the lectures exploring the **design** of the standard library vector and the **techniques** and **language features** used to **implement** it. Here, we deal with changing the **size** of a vector, parameterization of a vector with an element **type** (templates), and range **checking** (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?
- Given

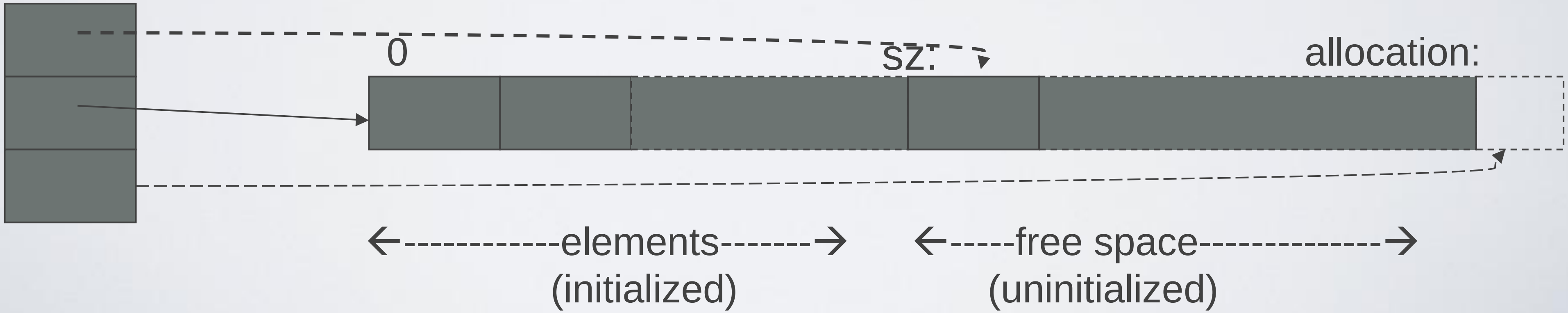
  vector v(n); *// **v.size()==n***

  we can change its size in three ways
  - Resize it
    - **v.resize(10);** *// **v** now has **10** elements*
  - Add an element
    - **v.push_back(7);** *// add an element with the value **7** to the end of **v***
*// **v.size()** increases by **1***
  - Assign to it
    - **v = v2;** *// **v** is now a copy of **v2***
*// **v.size()** now equals **v2.size()***

Representing vector

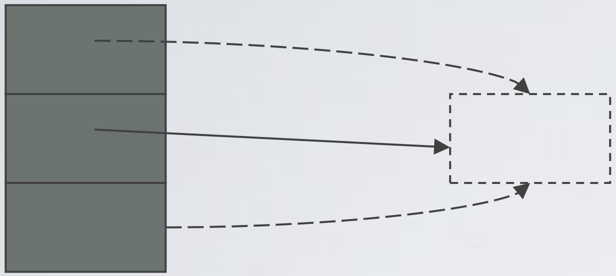
- If you **resize()** or **push_back()** once, you'll probably do it again;
  - let's prepare for that by sometimes keeping a bit of free space for future expansion

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

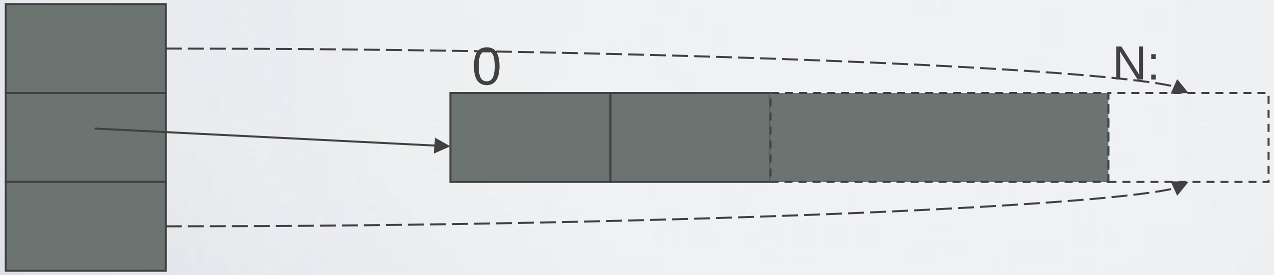


Representing vector

- An empty vector (no free store use):



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



vector::reserve()

- First deal with space (allocation);
  - Note: **reserve()** doesn't mess with size or element values

```
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()

- Given **reserve()**, **resize()** is easy
  - **reserve()** deals with space/allocation
  - **resize()** deals with element values

```
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;
}
```

vector::push_back()

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

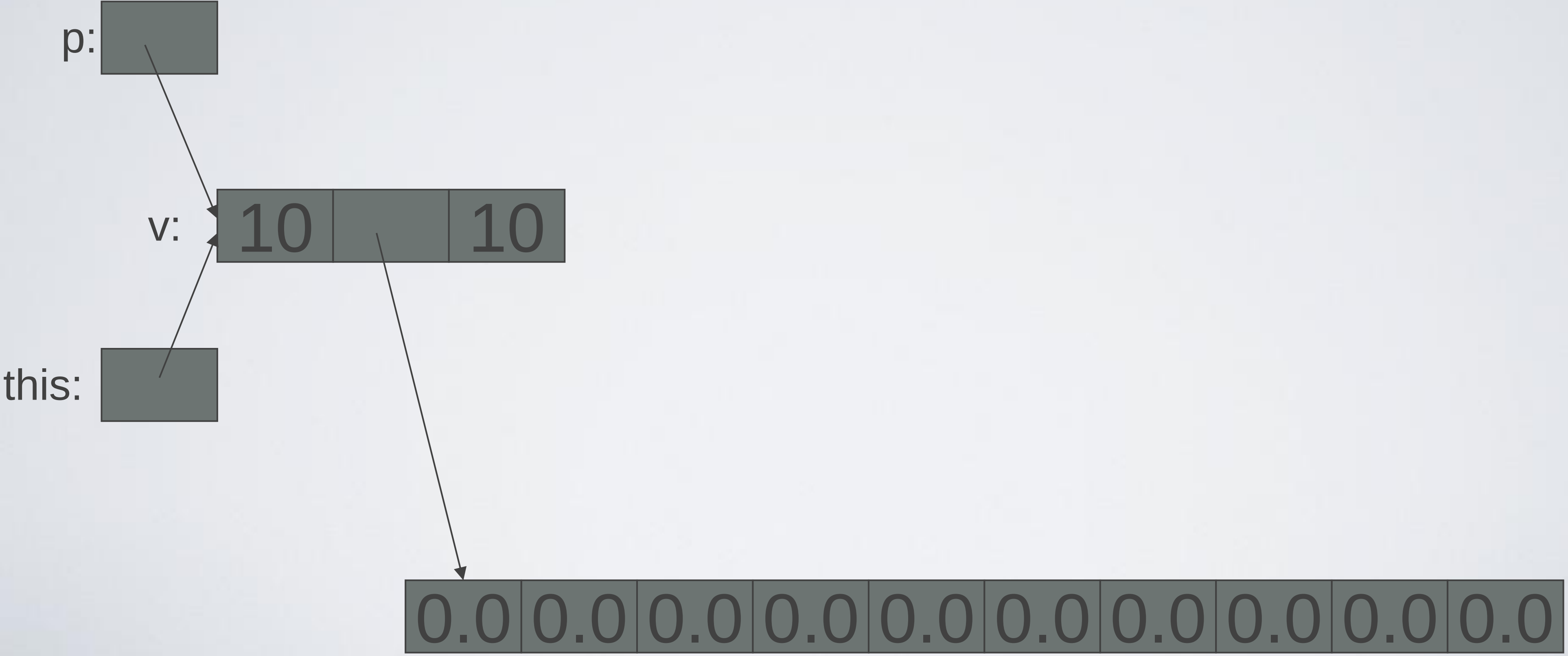
```
void vector::push_back(double d)
   // increase vector size by one
   // initialize the new element with d
{
   if (sz==0)                    // no space: grab some
            reserve(8);
   else if (sz==space)           // no more free space: get more space
            reserve(2*space);
   elem[sz]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

The **this** pointer

- 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 "pointer to self" in a member function is **this**



The **this** pointer

```
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
    // …
}
```

- The **this** pointer has uses that are less obscure
  - one of which we'll get to in two minutes

Templates

- But we don't just want vector of double
- We want vectors with element types we specify
  - **vector<double>**
  - **vector<int>**
  - **vector<Month>**
  - **vector<Record*>** *// vector of pointers*
  - **vector< vector<Record> >** *// vector of vectors*
  - **vector<char>**
- We must make the element type a parameter to **vector**
- **vector** must be able to take both built-in types and user-defined types as element types
- This is not some magic reserved for the compiler, we can define our own parameterized types, called "templates"

Templates

- The basis for generic programming in C++
  - Sometimes called “parametric polymorphism”
    - Parameterization of types (and functions) by types (and integers)
  - Unsurpassed flexibility and performance
    - Used where performance is essential (*e.g.*, hard real time and numerics)
    - Used where flexibility is essential (*e.g.*, the C++ standard library)

- Template definitions

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

- Template specializations (instantiations)

```
// 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
```

Parameterize with element type

```
// 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
```

Basically, **vector<double>** is

```
// 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

    // …
};
```

Basically, **vector<char>** is

```
// 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

    // …
};
```

Basically, **vector<T>** is

```
// 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

   // …
};
```

Templates

- Problems (“there’s no free lunch”)
  - Poor error diagnostics
    - Often spectacularly poor
  - Delayed error messages
    - Often at link time
  - All templates must be fully defined in each translation unit
    - (the facility for separate compilation of templates, called “export”, is not widely available)
    - So place template definitions in header files
- Recommendation
  - Use template-based libraries
    - Such as the C++ standard library
      - *E.g.*, **vector**, **sort()**
      - Soon to be described in some detail
  - Initially, write only very simple templates yourself
    - Until you get more experience

Range checking

```
// 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];
}
```

Range checking

```
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;
   }
}
```

Exception handling (primitive)

```
// 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
   }
}
```

Exception handling

(simpler and more structured)

```
// 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);
   // …
}
```

- if you feel that you need a try-block: think.
  - You might be able to do without it

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.

A confession

- The standard library **vector** doesn’t guarantee range checking of **[ ]**
- You have been using
  - *Either* our debug version, called **Vector**, which does check
  - *Or* a standard library version that does check (several do)

- Unless your version of the standard library checks, we “cheated”
  - In **std_lib_facilities.h**, we use the nasty trick (a macro substitution) of redefining **vector** to mean **Vector**

    #define vector Vector

    (This trick is nasty because what you see looking at the code is not what compiler sees – in real code macros are a significant source of obscure errors)
  - We did the same for **string**

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

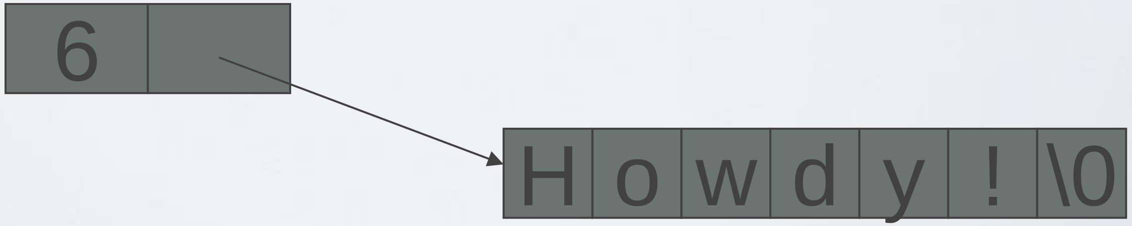
- Why doesn’t the standard guarantee checking?
  - Checking cost in speed and code size
    - Not much; don’t worry
      - No student project needs to worry
      - Few real-world projects need to worry
  - Some projects need optimal performance
    - Think huge (e.g., Google) and tiny (e.g., cell phone)
  - 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 (think airplanes)

Access to const vectors

```
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

- A **string** is rather similar to a vector<char>
  - E.g. **size(), [ ], push_back()**
  - 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



Next lecture

- An introduction to the STL, the containers and algorithms part of the C++ standard library. Here we'll meet sequences, iterators, and containers (such as **vector, list**, and **map**). The algorithms include **find()**, **find_if()**, **sort()**, **copy()**, **copy_if()**, and **accumulate()**.

Acknowledgements

Bjarne Stroustrup

Programming -- Principles and Practice Using C++

http://www.stroustrup.com/Programming/

Thank you!

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο
ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ & ΘΡΗΣΚΕΥΜΑΤΩΝ, ΠΟΛΙΤΙΣΜΟΥ & ΑΘΛΗΤΙΣΜΟΥ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
ΕΣΠΑ 2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης