Templates and Vectors

1. Generic Programming
   - function templates
   - class templates

2. the STL vector class
   - a vector of strings
   - enumerating elements with an iterator
   - inserting and erasing

3. Writing our own vector class
   - defining a namespace
   - inheriting from the STL vector class

MCS 360 Lecture 10
Introduction to Data Structures
Jan Verschelde, 20 September 2017
Templates and Vectors

1. Generic Programming
   - function templates
   - class templates

2. the STL \texttt{vector} class
   - a vector of strings
   - enumerating elements with an iterator
   - inserting and erasing

3. Writing our own vector class
   - defining a namespace
   - inheriting from the STL \texttt{vector\ class}
swapping integers

To swap two integers, we could define

```c
void IntSwap ( int& x, int& y )
{
    int z = x; x = y; y = z;
}
```

and a similar function for strings, etc...
a function template

To swap objects of any type:

template<typename T>
void MySwap ( T& x, T& y )
{
    T z = x; x = y; y = z;
}

generic programming is like abstract algebra
example: define GCD over any ring

Using the obvious name swap instead of MySwap created confusion
with the already available swap function.
using MySwap

cout << "swapping two integers..." << endl;
int a = 2; int b = 3;
cout << "before swap : a = " << a;
cout << ", b = " << b << endl;
MySwap(a,b);
cout << " after swap : a = " << a;
cout << ", b = " << b << endl;

cout << "swapping two strings..." << endl;
string s = "hello"; string t = "there";
cout << "before swap : s = " << s;
cout << ", t = " << t << endl;
MySwap(s,t);
cout << " after swap : s = " << s;
cout << ", t = " << t << endl;
Templates and Vectors

1. Generic Programming
   - function templates
   - class templates

2. the STL vector class
   - a vector of strings
   - enumerating elements with an iterator
   - inserting and erasing

3. Writing our own vector class
   - defining a namespace
   - inheriting from the STL vector class
The Standard Template Library (STL) provides

- container classes, e.g.: vectors, lists, sets, maps,
- template algorithms: searching, sorting, merging.

Features:

- reusable: adaptable and efficient;
- carefully controlled memory management.

David R. Musser, Gillmer J. Derge, Atul Saini:
Unified Modeling Language

The graphical notation for a template class:

```
Item_Type
___
vector
```
The STL vector class template definition is

template <typename T,
          typename Allocator = allocator<T>>
class vector
{
    // definition
};

The second template argument has a default, e.g.: the instantiation vector<int>
is equivalent to vector<int, allocator<int>>.
Templates and Vectors

1. Generic Programming
   - function templates
   - class templates

2. the STL vector class
   - a vector of strings
   - enumerating elements with an iterator
   - inserting and erasing

3. Writing our own vector class
   - defining a namespace
   - inheriting from the STL vector class
instantiation, push, and pop

Include the **STL vector class definition**:

```cpp
#include <vector>
```

**Instantiation**:

```cpp
vector<Item_Type> v;
```

**Adding a copy of item to end of vector**:

```cpp
void push_back(const Item_Type& i);
```

**Removing last element**:

```cpp
void pop_back();
```
```cpp
#include <string>
#include <vector>
using namespace std;

int main()
{
    vector<string> names;

    cout << "pushing names to the back..." << endl;
    do
    {
        string s;
        cout << "give a name : ";
        getline(cin,s,'\n');
        if(s == "") break;
        names.push_back(s);
    }
    while(true);
}
```
popping last element

With the `size()` method we first check if the vector is empty or not...

```cpp
if(names.size() > 0)
{
    cout << "popping last element..." << endl;
    names.pop_back();
}
```
subscripting operator

To access elements in a vector:

```cpp
Item_Type& operator[](size_t index);
```

A more secure way is

```cpp
Item_Type& at(size_t index);
```

If the index is invalid, at throws the exception `out_of_range`.
1. Generic Programming
   - function templates
   - class templates

2. The STL vector class
   - a vector of strings
   - enumerating elements with an iterator
   - inserting and erasing

3. Writing our own vector class
   - defining a namespace
   - inheriting from the STL vector class
using an iterator to write

Instead of

```cpp
void write ( vector<string> v )
{
    for(int i=0; i < v.size(); i++)
        cout << "v[" << i << "] = " << v[i] << endl;
}
```

there is a more general way:

```cpp
void write_with_iterator ( vector<string> v )
{
    for(vector<string>::const_iterator i=v.begin(); i != v.end(); i++)
        cout << *i << endl;
}
```
the const_iterator

An iterator is a pointer-like object, it refers to a position in a vector. To get the beginning and ending position:

const_iterator begin();
const_iterator end();

The operator * returns a reference to the object at the current position of the iterator:

Item_Type& operator*

The postfix operator increments the current position:

const_iterator& operator++()
1 Generic Programming
   - function templates
   - class templates

2 the STL vector class
   - a vector of strings
   - enumerating elements with an iterator
   - inserting and erasing

3 Writing our own vector class
   - defining a namespace
   - inheriting from the STL vector class
To insert a name at any position:

```cpp
cout << "give a name : ";

string n;
getline(cin,n,\n');

cout << "give an index : ";
size_t i;
cin >> i;

names.insert(names.begin()+i,n);
```
erasing an element

To erase a name at any position:

```cpp
    cout << "give an index to delete : ";

    size_t k;
    cin >> k;

    names.erase(names.begin()+k);
```

Note that `erase()` does not check if `k` is less than the size of the vector.
shallow or deep copy?

If we assign two vectors, do we copy references (shallow) or copy entire content (deep)?

To check what the STL vector class does:

```cpp
vector<string> w = names;

w[0] = "check for shallow copy";
cout << names[0] << endl;
cout << w[0] << endl;
```

An assignment to `w[0]` does not change `names[0]`. 
Templates and Vectors

1. Generic Programming
   - function templates
   - class templates

2. the STL vector class
   - a vector of strings
   - enumerating elements with an iterator
   - inserting and erasing

3. Writing our own vector class
   - defining a namespace
   - inheriting from the STL vector class
implementation of a vector class

namespace OurVector
{
    template <typename Item_Type>
    class vector
    {
        private:
            Item_Type* data;
            size_t number;
            static const size_t capacity = 10;

        public:
            //
    }
}

A static variable is shared by all objects.
We proceed in the following steps:

1. Define a constructor, `size()`, subscripting operator, and `push_back()` for a fixed capacity constant.
   → test on writing sequences of numbers

2. Write `pop_back()`, `insert()` and `erase()`.
   → throw exceptions if wrong index (no iterator)

3. A `reserve()` method doubles the capacity.
   → applied if needed in `push_back` and `insert()`
**Templates and Vectors**

1. **Generic Programming**
   - function templates
   - class templates

2. **the STL `vector` class**
   - a vector of strings
   - enumerating elements with an iterator
   - inserting and erasing

3. **Writing our own vector class**
   - defining a namespace
   - inheriting from the STL `vector` class
inheriting from vector

Goal: replace subscripting operator with \texttt{at}.

\begin{verbatim}
$ ourvector
give a number (0 to exit) : 9
give a number (0 to exit) : 8
give a number (0 to exit) : 0
v[0] = 9
v[1] = 8
give an index : 7
terminate called after throwing an instance of ‘std::out_of_range’
  what(): wrong index
Abort trap
\end{verbatim}
our own vector

```cpp
#include <iostream>
#include <vector>
#include <stdexcept>
using namespace std;

class OurVector : public vector<double>
{
    public:

        const double& operator[](size_t index)
        {
            if(index < 0 || index >= this->size())
                throw out_of_range("wrong index");
            return (*this).at(index);
        }
};
```
The definition of `OurVector` overrides the subscripting operator of the STL `vector` class.

```cpp
OurVector numbers;

// omitted code

cout << "give an index : ";
size_t k; cin >> k;
cout << "number[" << k << "] : "
     << numbers[k] << endl;
```
Summary + Exercises

Started Chapter 4: *Sequential Containers*, covered: an introduction to the STL `vector` class.

Exercises:

1. Give code for a templated function that reverses the order of the elements in any vector. Show that the same code works for vectors of integers and strings.

2. Give a program to generate a vector of integers, randomly generated between $-100$ and $+100$.

3. Extend the code of the previous exercise to remove all negative numbers from a vector of numbers.

4. Define a constructor of the vector class in the namespace `OurVector` that takes a constant array as one input argument. The first input argument of the constructor is the number of elements in the array. Provide a test program for your constructor.