# Introduction to C/C++

## Data structures in the STL

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

March 18, 2010

## Outline

## Standard libraries

- Until now we have seen
  - The basic of the C/C++ programming language
  - A few data structures
- During the years, programmers have built sophisticated libraries implementing data structures in a generic way
- All the data structures we have seen are available in standard libraries of C and C++
- In the next slides, we will have a quick look at the Standard Template Library (STL) of C++

## Templates

- Templates are a C++ language construct to make a data structure or an algorithm *generic* (i.e. independent of the data type)
  - Templates are an advanced programming topic! We will see today only how to use templates in the STL
- Template are expressed using angular parenthesis
- An example:

```
vector<int> array;      // a vector of integers

class Entry {
...
};

vector<Entry> array;  // a vector of Entry
```

# Strings

- In C/C++, strings are just array of characters
- Direct manipulation of array of characters is not easy
  - we have to allocate memory,
  - pay attention to length (to avoid overflow)
  - compare or make simple operation (like concatenation) through functions
- In the STL, a class `string` is provided that hides all this complexity

```
string a;         // creates an empty string
string b("zzz");  // creates a string with content zzz

a = "Pippo ";     // assigns characters to string a
```

- The class string (defined in the STL) automatically takes care of allocating memory (opportunely resizing the string allocation)

# Examples of usage of string

- You can find a reference to the string class in
  `http://www.cplusplus.com/reference/string/string/`

exstring.cpp

```
int main()
{
    string a, c;
    string b("Lipari");

    a = "Giuseppe";
    c = b + " " + a;
    cout << c << endl;

    int w = c.find("p");
    cout << "p at position " << w << endl;

    int w2 = c.find("sep");
    cout << "sep at position " << w2 << endl;

    string sub = c.substr(w, w2-w);
    cout << "Substring between w and w2 is ["
         << sub << "]" << endl;
    if (sub < "Alberto") cout << "Before" << endl;
    else cout << "After" << endl;
}
```

# Variable sized arrays

- Another problem with plain C is the use of arrays
  - Arrays must have fixed size
  - If we want variable size, we must deal with dynamic memory allocation
  - This may be annoying, so the STL has a class for dynamic sized arrays, called `vector`

```
vector<int> vec;              // this is an empty array
for (int i=0; i<10; i++)      // allocates memory for 10 integers,
    vec.push_back(i);         // and initialize them

vec[5] = vec[6] + 10;         // read and assignment
```

# Vector example

- The vector interface can be seen at

  `http://www.cplusplus.com/reference/stl/vector/`

exvector.cpp

```
int main()
{
    vector<string> names;
    names.push_back("Giuseppe");
    names.push_back("Eleonora");
    names.push_back("Edoardo");
    names.push_back("Margherita");

    for (unsigned i=0; i<names.size(); i++)
        cout << names[i] << endl;

    cout << endl;
    names.pop_back();
    names.pop_back();

    for (unsigned i=0; i<names.size(); i++)
        cout << names[i] << endl;
}
```

# Iterators

- Sometimes it is necessary to go through a *data structure* step by step
  - However, not all data structures identify every element with an integer index like the array (or the vector)
- Therefore, the STL provides a generalization of an index, called *iterator*

```
vector<string> names;
vector<string>::iterator i;
...
i = names.begin();          // i now "points" to the first element
while (i != names.end()) {  // until the end of the vector
    ...
    i++;                    // go to next element
}
```

# Example

exvectorit.cpp

```
int main()
{
    vector<string> names;
    vector<string>::iterator i;
    names.push_back("Giuseppe");
    names.push_back("Eleonora");
    names.push_back("Edoardo");
    names.push_back("Margherita");

    for (i=names.begin(); i!=names.end(); i++)
        cout << *i << endl;

    cout << endl;
    names.pop_back();
    names.pop_back();

    names.insert(names.begin(), "Edoardo");
    for (i=names.begin(); i!=names.end(); i++)
        cout << *i << endl;
}
```

## Vector internal implementation

- The vector is internally implemented as a variable size array
  - Therefore, internally it allocates and deallocated memory depending on the current number of element inside
  - However, all elements are sequential in memory
  - In the previous example the `insert()` simply moves all element one step ahead to make space for the additional element to be inserted in the first place
  - Similarly, a `push_back()` may imply a copy of all elements!
  - Therefore, insertion in a vector is a costly operation which takes O(n).

## Lists

- The STL also provides the simple linked list we have seen in the course
- In the STL, the template parameter indicates the data type

```
list<int> lst;
for (int i=0; i<10; i++)
    lst.push_back(i);

// going through all elements
list<int>::iterator i = lst.begin();
int sum = 0;
while (i!=lst.end()) {
    sum += *i;
    i++;
}
```

# A complete example

exlist.cpp

```
int main()
{
    list< vector<int> > lst;

    for (int i=0; i<10; i++) {
        vector<int> vec;
        for (int j=0; j<5; j++)
            vec.push_back((i+1)*j);

        lst.push_back(vec);
    }
    list< vector<int> >::iterator k;

    int count = 0;
    // looks for number 18
    for (k=lst.begin(); k!=lst.end(); k++)
        for (unsigned i=0; i < (*k).size(); i++)
            if ((*k)[i] == 18) count++;

    cout << "18 has been found " << count << " times" << endl;
}
```

# Complexity

- As explained in the previous lecture the complexity of inserting in a ordered list is O(n)
- However, inserting at the head or at the tail is O(1)
- Tipically sorting takes less time on a `list` than on a `vector`, because in list we only have to swap the pointers, while in a vector we have to swap the elements

## Queue

- A FIFO queue can be implemented by using a deque (Double ended queue)
- The main operations on a deque are `push_back`, `pop_back`, `push_front` and `pop_back`
- They all have complexity O(1)
- The deque is also the standard underlying implementation for a `stack`

```
deque<int> q;

q.push_front(1);
q.push_back(10);
q.push_front(2);
unsigned n = q.size();
int sum = 0;
for (int i=0; i<n; i++) {
    sum += q.back();
    q.pop_back();
}
```

## Trees

- In the STL, a `map` is an *associative array*
  - An associative array contains pairs `<key, value>`
  - It can be treated as an array where the index may be anything (the key)
  - Internally it is implemented as a balanced binary tree (more specifically, a red-black tree, which is similar to an AVL)

```
map<string, string> names;
names["Lipari"] = "Giuseppe";
names["Ancilotti"] = "Paolo";
names["Buttazzo"] = "Giorgio";
names["Di Natale"] = "Marco";

if (names["Ancilotti"] == "Paolo") ...
```

# Example

# Complexity Table

- Which container to use?
  - It depends on the typical use in our program
  - The following table can help deciding ...

| container | insert | ins head | ins back | search | sort |
|---|---|---|---|---|---|
| vector | O(n) | O(n) | O(n) | O(n) | O(n log(n)) |
| list | O(n) | O(1) | O(1) | O(n) | O(n log(n)) |
| map | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n log(n)) |

- Also look here:
  - `http://www.cplusplus.com/reference/stl/`