Brief Introduction to STL

Slides by G. Lipari (Scuola Superiore S. Anna, Pisa),
selected and modified by A. Bechini (Dept. Information Engineering, Univ. of Pisa)

Introduction to STL

"Don't reinvent the wheel:
use libraries"
- B. Stroustrup

Introduction

- Here we introduce the basic object of the **C++ std library**
- You will need them when writing your programs and exercise
- Don't panic:
  you don't need to understand
  how these objects are implemented,
  but only how they can be used.

3

A few words on namespaces

- In C, there is the *name-clashing* problem
  - cannot declare two entities with the same name
- One way to solve this problem in C++ is to use **namespaces**
  - A **namespace** is a collection of declarations
  - We can declare two entities with the same name in different namespaces
  - All the standard library declarations are inside **namespace std**;

4

Namespaces: Example

```
namespace sportgames {
  const int howManyGames = 23;
  class marathon { … };
  class soccer { … };
…
}
```

- Namespaces can also nest

```
namespace sportgames {
  class marathon { … };
  namespace ballsportgames {
    class soccer { … };
  …
  }
}
```

5

Using entities inside namespaces

- There are two ways:
  - Using the scope resolution operator ::
  - the *using namespace xx* directive

```
std::string a;        // declaring an object of type
                      // string from the std namespace

mylib::string b;      // declaring an object of type
                      // string from the mylib namespace
```

```
using namespace std;    // from now on use std

string a;               // declaring an object of type
                        // string from the std namespace
```

6

Namespace std and basic I/O (I)

```
#include <iostream>
int main()
{
  std::cout << “Hello world!”;
}
```

- Basic I/O function are included with *iostream*
- *cout* is the standard output stream
- *std::cout* means that the *cout* object is contained in a namespace called *std::*
- all the *std* library is contained in *std*
- we can also use the *using* directive

7

Namespace std and basic I/O (II)

```
#include <iostream>
using namespace std;
int main()
{
  cout << “Hello world!\n”;
}
```

- operator << sends its right part to the stream to the left
- it can send every kind of variable or constant:

```
int age = 30;

cout << “I am “ << age << “ years old\n”;
```

8

Introducing STL containers

- sometimes we do not know how many elements an array will contain

```
struct Entry {
    string name;
    int number;
};

Entry phone_book[1000];

void print_entry(int i) {
    cout << phone_book[i].name << ' ' << phone_book[i].number << "\n";
}
```

- what if phone_book overflows?

9

Containers: vector (I)

- we can use the vector<Entry> container

```
struct Entry {
    string name;
    int number;
};

vector<Entry> phone_book(10);      // initially, only 10 elements

void print_entry(int i) {
    cout << phone_book[i].name << ' ' << phone_book[i].number << "\n";
}

void add_entry(const Entry &e) {
    phone_book.push_back(e);       // after 10 elements, expands automatically
}
```

10

Containers: vector (II)

- What is the push_back() function?
  - insert a new element at the end of the vector.
    If there is not enough space, the vector is enlarged
- How can we know the actual number of elements?
  - using the size() function

```
void add_entry(const Entry &e) {
   phone_book.push_back(e);        // expands automatically
   cout << "Now the numer of elements is " << phone_book.size() << "\n";
}
```

11

Containers: vector (III)

- for efficiency reasons, operator [] is not checked for out-of-range
- however, we can use the function at() instead of []

```
// this causes a segmentation fault if i is out of range
void print_entry(int i) {
   cout << phone_book[i].name << ' ' << phone_book[i].number << "\n";
}

// this throws an out_of_range exception
void print_entry_with_exc(int i) {
   cout << phone_book.at(i).name << ' ' << phone_book.at(i).number << "\n";
}
```

12

First example

- We will write a program that:
  - reads a file line by line
  - stores each line in a vector;
  - outputs the file upside/down (from the last line to the first) into another file

13

Reading the command line

- A program can read the command line through its main function

```
int main(int argc, char* argv[])
{
  cout << "Num of args: " << argc << "\n";
  for (int i =0; i<argc; ++i)
    cout << argv[i] << "\n";
}
```

```
$> ./args joe 5.0 12 india
Num of args: 5
./args
joe
5.0
12
india
```

- argc contains the number of args+ 1
- argv[i] contains the i-th argument
- argv[0] is always equal to the name of the program

14

Now the code...

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>

using namespace std;

int main(int argc, char *argv[])
{
  if (argc < 3) {
    cout << "Usage: ";
    cout << argv[0] << " <input file> <output_file>" << endl;
    exit(-1);
  }
  ifstream in(argv[1]);
  ofstream out(argv[2]);
  …
```

15

Now the code...

```
  …
 vector<string> lines;

  string str;
  while (getline(in, str)) lines.push_back(str);

  int n = lines.size();
  cout << "The size of the input file is " << n << " lines\n";
  for (int i=n; i > 0; --i)
    out << lines[i-1] << endl;

  cout << "Done!!" << endl;

}
```

16

Containers: map (I)

- what if we want to search the phone_book by name?
- we have to perform a linear search

```
int get_number(const string &name)
{
  for (int i=0; i<phone_book.size(); ++i)
     if (phone_book[i].name == name) break;

  if (i== phone_book.size()) {
    cout << "not found!!\n";
    return 0;
  }
  else return phone_book[i].number;
}
```

17

Containers: map (II)

- Another (more optimized) way is to use map<string, int>

```
map<string, int> phone_book;

void add_entry(const string &name, int number)
{
   phone_book[name] = number;
}

int get_number(const string &name)
{
   int n = phone_book[name];
   if (n == 0) cout << "not found!\n";

   return n;
}
```

18

Containers: map (III)

- You can think of map<> as an **associative array**
  - in our example, the index is a string, the content is an integer
- How map is implemented is not our business!
  - Usually implemented as hash tree, or red-black tree
  - linear search in a vector is O(n)
  - searching a map is O(log(n))
- Very useful!!

19

Iterators

- What if we want to print all elements of a map?
- we need an **iterator**...

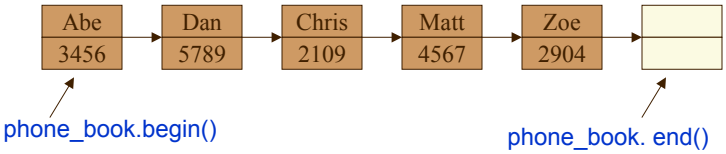
```
map<string, int> phone_book;

void print_all()
{
   map<string, int>::iterator i;

   for (i = phone_book.begin(); i != phone_book.end(); ++i);
      cout << "Name : " << (*i).first << " ";
      cout << "Number : " << (*i).second << "\n";
   }
}
```

20

What the ?@#$ is an iterator?

- An iterator is an object for dealing with a sequence of objects inside containers
- You can think of it as a special pointer

```
phone_book.begin();        // the beginning of the sequence
phone_book.end();          // the end of the sequence
```

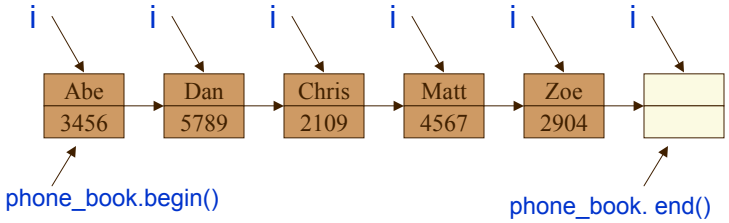


21

Iterators

- Here is how the for() works:

```
void print_all() {
    map<string, int>::iterator i;
    for (i = phone_book.begin(); i != phone_book.end(); ++i);
        cout << "Name : " << (*i).first << " ";
        cout << "Number : " << (*i).second << "\n";
    }
}
```



22

A brief explanation of iterators

- When you build a container, you need special functions to access the members and traverse the container
  - One solution could be to write special member functions, like getFirst(), getNext(), and so on
  - This solution is inflexible, for many reasons
    - internal state for the container
    - you can traverse in one way only
    - generic functions, like sort, could not work in general
  - A better solution is to provide an additional std class, called iterator, to access the members with a unified interface

23

Iterator

- An iterator is like a pointer to an object, with restrictions
  - it can only point to element of a container
  - operator++() returns the next element in the container
  - operator*() returns the member itself
- Every container defines its own iterator class, the interfaces are the same
  - in this way, it is possible to write generic functions that use only iterators

24

Iterators

- There are iterators for all containers
  - vector, string, list, map, set, etc.
  - all support begin() and end()
- Iterators are also used for **generic algorithms** on containers
  - find, foreach, sort, etc.

25

Introducing generic algorithms

- Let's get back to the vector example

```
struct Entry {
   string name;
   int number;
};

vector<Entry> phone_book(10);      // initially, only 10 elements
```

- what if we want to order the entries alphabetically ?
  - In the old C / C++ programming, we would take a good book of algorithms (like "The art of computer programming" D. Knuth) and write perhaps a shell-sort
  - With the standard library, this has already been done by someone else and it is fast and optimized; all we have to do is to customize the algorithm for our purposes.

26

sort()

- We have to specify an ordering function
  - the algorithm needs to know if a < b
  - we **re-use operator <** on strings

```
bool operator <(const Entry &a, const Entry &b)
{
   return a.name < b.name;
}
```

- Now we can use the sort algorithm:

```
template<class Iter> void sort(Iter first, Iter last);
```

```
sort(phone_book.begin(), phone_book.end());
```

27

The complete program

```
bool operator < (const Entry &a, const Entry &b) { return a.name < b.name;}

void add_entry(const string &n, int num) {
  Entry tmp;
  tmp.name = n; tmp.number = num;
  phone_book.push_back(tmp);
}

int main() {
  add_entry("Lipari Giuseppe", 1234);
  add_entry("Ancilotti Paolo", 2345);
  add_entry("Cecchetti Gabriele", 3456);
  add_entry("Domenici Andrea", 4567);
  add_entry("Di Natale Marco", 5678);
  sort(phone_book.begin(), phone_book.end());

}
```

28

Generic algorithms

- sort is an example of **generic algorithm**
  - to order objects, you don't really need to know what kind of objects they are, nor where they are contained
  - **all you need is how they can be compared**
  - (the < operator)
- So, to customize the sort algorithm, you have to specify what does it mean A < B
- You will learn later how to write a generic algorithm, that does not rely on the type of objects

29

Generic algorithms

- Another example: for_each()

```
void print_entry(const Entry &e)
{
   cout << e.name << " \t " << e.number << "\n";
}

int main(){
  …
  for_each(phone_book.begin(),phone_book.end(),print_entry);
}
```

- Try to change the container from vector<> to map<>. The for_each does not need to be changed!
- for_each() works **as long as it has a couple of iterators**

30

Another example

- Suppose we want to print
  only the first 5 elements of the sequence:

```
for_each(phone_book.begin(),
         phone_book.begin()+min(3,phone_book.size()),
         print_entry);
```

- It is all that simple!
- We will show in the next lessons how it is possible to combine these objects to do almost everything.

31