C++ Vectors, Lists and Language Features

2501ICT Nathan

René Hexel and Joel Fenwick

School of Information and Communication Technology
Griffith University

Semester 1, 2011
Outline

1. Linear Collection Introduction
   - Linear Collections: Lists and Arrays

2. C++ Language Features
   - Templates
   - Namespaces and Operator Overloading
Lists and Arrays

Linear Collections in C++
C++ Arrays

- **std::vector**
  - array class
  - locating an element at a given position takes constant time

- **std::list**
  - faster insertions and deletions
  - but slower random access

- Iterators
  - enumerate all elements
  - *similar to NSEnumerator in Objective-C*
Example (prints: l3 has 1 element starting with Hello)

```cpp
std::string s("Hello");
std::vector<int> v1; // an empty vector
    v1.assign(3, 0); // 3 zero elements
std::vector<std::string> v2(1, s); // a vector with one string
std::list<std::string> l1(1, s); // a list with one string
std::list<std::string> l2(l1); // copy l1 into l2
12.merge(l1); // merge l1 into l2

if (l1 == l2) // same content?
    printf("l1 is equal to l2 -- how come?\n");

std::list<std::string> l3(l2); // copy l2 into l3
l3.unique(); // remove duplicates
int count3 = l3.size(); // number of elements

const char *first = l3.front().c_str(); // first element as char *
printf("l3 has %d element starting with %s\n", count3, first);
```
Other Useful Methods

- **front()**
  - returns the first element of a list or vector

- **back()**
  - returns the last element of a list or vector

- **empty()**
  - removes all elements from a list or vector

- **reverse()**
  - reverses a list

- **splice(iterator pos, list &source)**
  - moves elements from source to the list, starting at pos

→ See list and vector in the C++ Reference
Enumerating Array Example

Example (prints: 1 2 3)

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

int main(int argc, char *argv[])
{
    std::vector<int> vec;

    for (int i = 1; i <= 3; i++)
        vec.push_back(i);

    std::vector<int>::iterator enumerator = vec.begin(); // iterator
    while (enumerator != vec.end()) // loop through array
        printf("%d ", *enumerator++); // print each element

    printf("\n");

    return EXIT_SUCCESS;
}
```
C++ Templates
The same Problem: how to store different types in lists, arrays, and other collection classes?

An additional Problem: C++ has no reflection capabilities
- types must be known at compile time
- a generic list would not be able to know which types of objects it stores

Templates
- allow to specify what data type is put in a collection
- they look like Java generics
  - e.g. `vector<int>` denotes an array of integers
  - e.g. `list<string>` denotes a list of strings
Namespaces

Using C++ Namespaces
Namespaces

- The Problem: two types, variables, or functions have the same name
  - Objective-C uses a prefix such as NS (e.g. NSString for the string class)
  - C++ uses namespaces
- The std Namespace
  - used for the standard C++ classes
  - std::string, std::vector, std::list, etc.
- using namespace std;
  - should come right after the #include part
  - avoids having to write std:: all the time
  - makes code more readable
  - use only in .cc (not .h) files!
  - always write full names in header files!
**Iterator with and without Namespace**

**without** `using namespace`  
```cpp
#include <cstdlib>
#include <vector>

int main(int argc, char *argv[]) {
    std::vector<int> vec;
    for (int i = 1; i <= 3; i++)
        vec.push_back(i);
    std::vector<int>::iterator e = vec.begin();
    while (e != vec.end())
        printf("%d ", *e++);
    printf("\n");
    return EXIT_SUCCESS;
}
```

**with** `using namespace`  
```cpp
#include <cstdlib>
#include <vector>

using namespace std;

int main(int argc, char *argv[]) {
    vector<int> vec;
    for (int i = 1; i <= 3; i++)
        vec.push_back(i);
    vector<int>::iterator e = vec.begin();
    while (e != vec.end())
        printf("%d ", *e++);
    printf("\n");
    return EXIT_SUCCESS;
}
```
Operator Overloading in C++

René Hexel and Joel Fenwick

C++ Vectors, Lists and Language Features
Operator Overloading

- C++ allows class methods to override standard operators
  - allows usage of `enumerator++` instead of `enumerator.nextObject()`
  - powerful, but dangerous feature
  - needs to be used with care!

- Method name is `operator` followed by the actual operator
  - `operator+()` redefines the `+` binary operator
  - `operator-()` redefines the `−` binary operator
  - etc.

- Used a lot in the C++ `std` classes
  - `cout` in `<iostream>` for standard output
  - `cin` in `<iostream>` for standard input
  - `operator+` to concatenate strings
  - `operator[]` to index a vector
  - `operator*` to dereference an iterator
  - etc.
#include <cstdlib>
#include <cstdio>
#include <string>

using namespace std;

int main(int argc, char *argv[]) {
    string s1("Hello");
    string s2(" void");

    /*
     * concatenate both strings
     */
    string s3(s1.append(s2));

    printf("%s\n", s3.c_str());
    return EXIT_SUCCESS;
}

---

#include <cstdlib>
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char *argv[]) {
    string s1 = "Hello";
    string s2 = " void";

    /*
     * concatenate both strings
     */
    string s3 = s1 + s2;

    cout << s3 << endl;
    return EXIT_SUCCESS;
}