301AA - Advanced Programming

Lecturer: **Andrea Corradini**
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

Course pages:
http://pages.di.unipi.it/corradini/Didattica/AP-18/

***AP-2018-16**: C++ Standard Template Library*
Slides freely adapted from those of Antonio Cisternino

Introduction

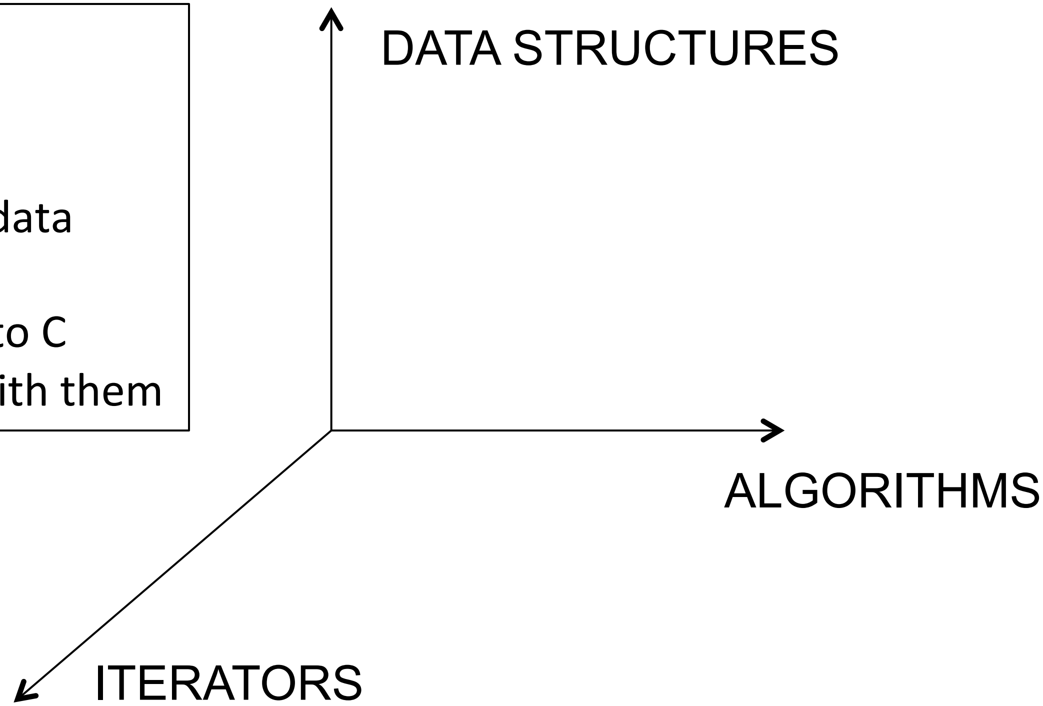
- The C++ Standard Template Library (STL) has become part of C++ standard
- The main author of STL is Alexander Stephanov
- Developed in ~1992 but based on ideas of ~1970
- He chose C++ because of templates and no requirement of using OOP!
- The library is somewhat unrelated with the rest of the standard library which is OO

The Standard Template Library

- Goal: represent algorithms in as general form as possible without compromising efficiency
- Extensive use of **templates** and **overloading**
- Only uses static binding (and inlining): not object oriented, no dynamic binding – very different from Java Collection Framework
- Use of **iterators** for decoupling algorithms from containers
- Iterators are seen as abstraction of pointers
- Many generic abstractions
  - Polymorphic abstract types and operations
- Excellent example of generic programming
  - Generated code is very efficient

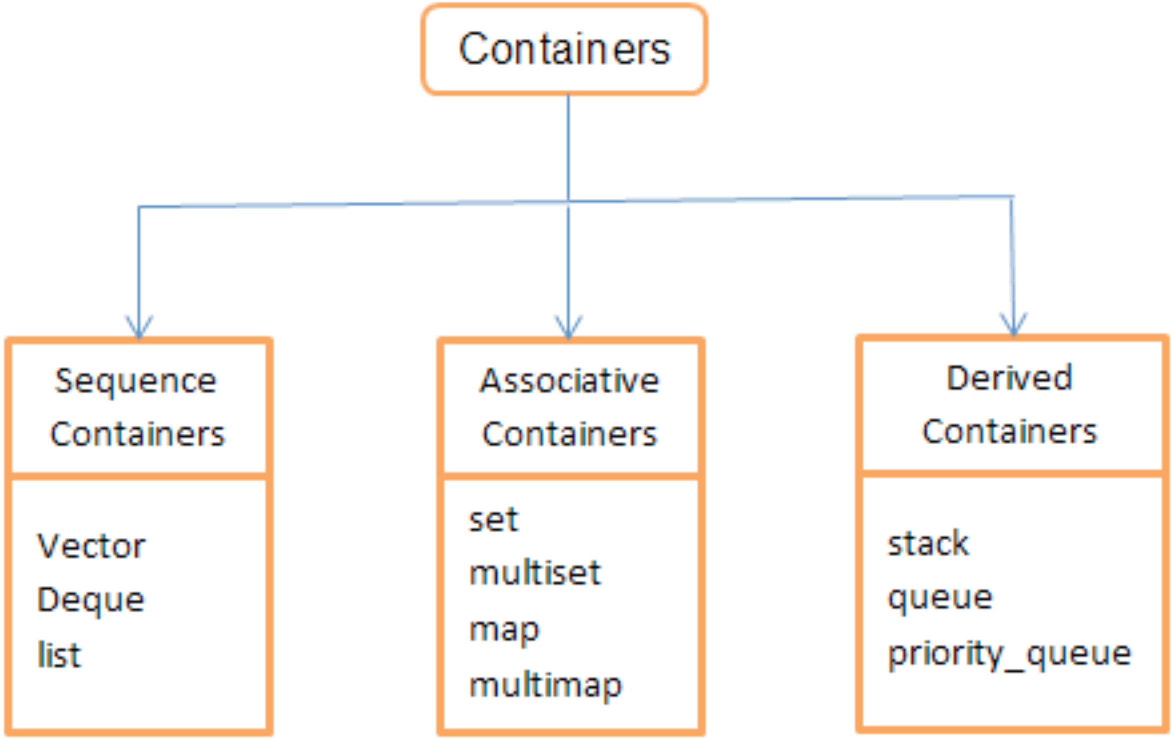
3D generic world

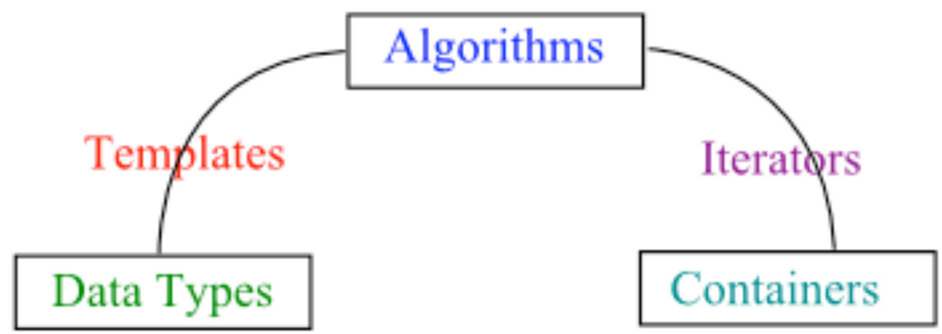
Stephanov observed three orthogonal dimensions in algorithms: iterators allow algorithms to iterate over data structures.
Iterators are very similare to C pointers and compatible with them



Main entities in STL

- **Container**: Collection of typed objects
  - Examples: array, vector, deque, list, set, map ...
- **Iterator**: Generalization of pointer or address. used to step through the elements of collections
  - forward_iterator, reverse_iterator, istream_iterator, …
  - pointer arithmetic supported
- **Algorithm:** initialization, sorting, searching, and transforming of the contents of containers,
  - for_each, find, transform, sort
- **Adaptor**: Convert from one form to another
  - Example: produce iterator from updatable container; or stack from list
- **Function object**: Form of closure (class with "operator()" defined)
  - plus, equal, logical_and
- **Allocator**: encapsulation of a memory pool
  - Example: GC memory, ref count memory, ...





1. Templates
make algorithms independent of the data types
2. Iterators
make algorithms independent of the containters

A digression: Iterators in Java

- Iterators are supported in the Java Collection Framework: interface **`Iterator<T>`**
- They exploit generics (as collections do)
- Iterators are usually defined as *nested classes* (*non-static private member classes*): each iterator instance is associated with an instance of the collection class
- Collections equipped with iterators have to implement the **`Iterable<T>`** interface

```
class BinTree<T> implements Iterable<T> {
    BinTree<T> left;
    BinTree<T> right;
    T val;
    ...
    // other methods: insert, delete, lookup, ...
    public Iterator<T> iterator() {
        return new TreeIterator(this);
    }
```

Iterators in Java (cont'd)

```
class BinTree<T> implements Iterable<T> {
    …
    private class TreeIterator implements Iterator<T> {
        private Stack<BinTree<T>> s = new Stack<BinTree<T>>();
        TreeIterator(BinTree<T> n) {
            if (n.val != null) s.push(n);
        }
        public boolean hasNext() {
            return !s.empty();
        }
        public T next() {     //preorder traversal
            if (!hasNext()) throw new NoSuchElementException();
            BinTree<T> n = s.pop();
            if (n.right != null) s.push(n.right);
            if (n.left != null) s.push(n.left);
            return n.val;
        }
        public void remove() {
            throw new UnsupportedOperationException();
  } }
}
```

Iterators in Java (cont'd)

- Use of the iterator to print all the nodes of a BinTree:

```
for (Iterator<Integer> it = myBinTree.iterator();
                                     it.hasNext();   )
    {   Integer i = it.next();
        System.out.println(i);
    }
```

- Java provides (since Java 5.0) an *enhanced for* statement (*foreach*) which exploits iterators. The above loop can be written:

```
for (Integer i : myBinTree)
        System.out.println(i);
```

- In the *enhanced for*, **`myBinTree`** must either be an array of integers, or it has to implement **`Iterable<Integer>`**
- The enhanced for on arrays is a **bounded iteration.** On an arbitrary iterator it depends on the way it is implemented.

Example of use: **Vector** and **Forward Iterator**

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
   vector<int> vec; // create a vector to store int
   int i;
   // display the original size of vec
   cout << "vector size = " << vec.size() << endl;
   // push 5 values into the vector
   for(i = 0; i < 5; i++) {
      vec.push_back(i);
   }
   // display extended size of vec
   cout << "extended vector size = " << vec.size() << endl;
   // access 5 values from the vector
   for(i = 0; i < 5; i++) {
      cout << "value of vec [" << i << "] = " << vec[i] << endl;
   }
   // use iterator to access the values
   vector<int>::iterator v = vec.begin();
   while( v != vec.end()) {
      cout << "value of v = " << *v << endl;
      v++;
   }
   return 0;
}
```

Example: using algorithm inner_product

```
template< class InputIt1, class InputIt2, class T >
T inner_product( InputIt1 first1, InputIt1 last1,
                 InputIt2 first2, T value );
```

the signature

```
#include <iostream>
#include <numeric>

int main() {
  int A1[] = {1, 2, 3};
  int A2[] = {4, 1, -2};
  const int N1 = sizeof(A1) / sizeof(A1[0]);

  std::cout << inner_product(A1, A1 + N1, A2, 0)
            << std::endl;
  return 0;
}
```

It will print 0:

0 = 0 + 1 * 4 + 2 * 1 + 3 * -2

Initial value for the accumulator

Start of A1

End of A1

Start of A2

With strings?

- We have strings in two vectors: labels and values to display
- Can we exploit inner product algorithm?
- It would be enough to use string concatenation with a tab instead of ‘*’ and with a new line instead of ‘+’
- Note that overloading of ‘+’ and ‘*’ operators for strings make no sense: we don't want just string cat and we may interfere with already defined overloads
- Fortunately there is another version of inner_product that allows specifying function objects to use instead of ‘*’ and ‘+’

inner_product: more general definition

```
template<class InputIt1, class InputIt2, class T,
  class BinaryOperation1, class BinaryOperation2>
T inner_product( InputIt1 first1, InputIt1 last1,
  InputIt2 first2, T init, BinaryOperation1 op1,

  BinaryOperation2 op2 );
```

- Ordered map/reduce
- Initializes result to `init`
- For each `i1` in `[first1, last1)`,
  and `i2 = first2 + (i1 - first1))`
  updates result as follows:

  `result = op1(result, op2(*i1, *i2))`
- Let us show the generality of such algorithm

Column printing with C strings

Cat function object: operator() will be invoked by inner_product

This function object is a closure: operator() behaves differently depending on sep!

```
#include <iostream>
#include <numeric>
#include <string.h>
struct Cat {
  const char* sep;
  Cat(const char* s) : sep(s) {}
  char* operator()(const char* t, const char* s) {
    char* ret = new char[strlen(t) + strlen(sep) + strlen(s) + 1];
    strcpy(ret, t); strcat(ret, sep); strcat(ret, s);
    return ret;
  }};
int main() {
  char *A1[] = { "Name", "Organization", "Country" };
  char *A2[] = { "Antonio Cisternino", "Università di Pisa", "Italy" };
  const int N1 = sizeof(A1) / sizeof(A1[0]);

  std::cout << inner_product(A1, A1 + N1, A2, "", Cat("\n"), Cat("\t"))  << std::endl;
  return 0;
}
```

Using pointers to access the elements of the arrays

Two more arguments: the function objects to use instead of + and *

…and with C++ std::string

```
#include <iostream>
#include <numeric>
#include <string.h>
#include <string>
#include <vector>
struct CatS {
  std::string sep;
  CatS(std::string s) : sep(s) {}
  std::string operator()(std::string t, std::string s) { return t + sep + s; }
};
int main() {
  std::vector<std::string> s, v;
  s.push_back(std::string("Hello")); s.push_back(std::string("Antonio"));
  v.push_back(std::string("World")); v.push_back(std::string("Cisternino"));

  std::vector<std::string>::const_iterator A1 = s.begin(), A2 = v.begin();
  int N1 = s.size();
  std::cout << inner_product(A1, A1 + N1, A2, std::string(""), CatS(std::string("\n")),
    CatS(std::string("\t")))  << std::endl;
  return 0;
}
```

Much easier than before

Using vector<T> instead of arrays

A1 and A2 now are *iterators* to vector<string>

The three calls

```
std::cout << inner_product(A1, A1 + N1, A2, 0)
           << std::endl;

std::cout <<
  inner_product(A1, A1 + N1, A2, "",
              Cat("\n"), Cat("\t")) << std::endl;

std::cout <<
  inner_product(A1, A1 + N1, A2,
           std::string(""), CatS(std::string("\n")),
           CatS(std::string("\t"))) << std::endl;
```

The same syntax…

- Though we have used different data types and containers the invocation of inner_product has been essentially the same
- And we are not using inheritance…
- How is this possible? On what language mechanisms do rely STL?
- What really are iterators? Why can be interchanged with pointers?
- STL seems to be really effective and generic but what happens to the code generated?

C++ namespaces!

- STL relies on C++ namespaces
- Containers expose a type named *iterator* in the container's namespace
- Example: *std::vector*<*std::string*>*::iterator*
- Each class implicitly introduces a new namespace
- The *iterator* type name assumes its meaning depending on the context!

Complexity of operations on containers

- It is *guaranteed* that inserting and erasing at the end of the vector takes *amortized* constant time whereas inserting and erasing in the middle takes linear time.

Container	***insert/erase overhead at the beginning***	***in the middle***	***at the end***
Vector	linear	linear	amortized constant
List	constant	constant	constant
Deque	amortized constant	linear	amortized constant

Complexity of use of Iterators

- Consider the following code:

```
std::list<std::string> l;
…
quick_sort(l.begin(), l.end());
```

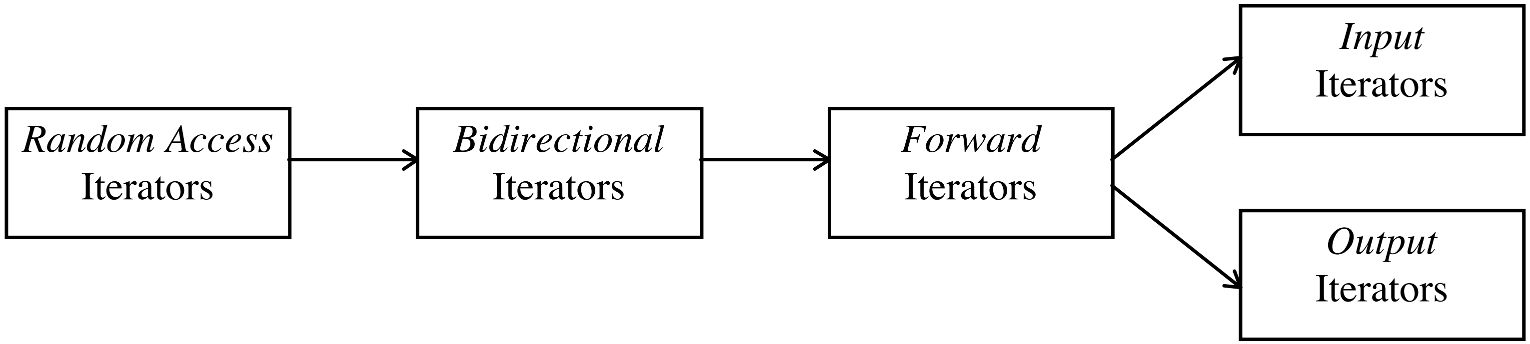
- This is not reasonable: quick_sort assumes random access to container's elements!
- How can we control complexity of algorithms and guarantee that code behaves as expected?

Classifying iterators

- The solution proposed by STL is assume that iterators implement all operations in constant time
- Containers may support different iterators depending on their structure:
  - **Forward iterators**: only dereference (operator*), and pre/post-increment operators (operator++)
  - **Input and Output iterators**: like forward iterators but with possible issues in dereferencing the iterator (due to I/O operations)
  - **Bidirectional iterators**: like forward iterators with pre/post-decrement (operator--)
  - **Random access iterators**: like bidirectional iterators but with integer sum ($p + n$) and difference ($p - q$)
- Iterators heavily rely on operator overloading provided by C++

Categories of iterators

- Five categories, with decreasing requirements



- Each category has only those functions defined that are realizable in constant time. [Efficiency concern of STL!]
- Not all iterators are defined for all categories: since random access takes but linear time, random access iterators cannot be used with lists.

Container	*Iterator Category*
vector	random access iterators
list	bidirectional iterators
deque	random access iterators

C++ operators and iterators

- **Forward iterators** provide for one-directional traversal of a sequence, expressed with ++:
  - Operator ==, !=, *, ++
- **input iterators and output iterators** are like forward iterators but do not guaratee these properties of forward iterators:
  - that an input or output iterator can be saved and used to start advancing from the position it holds a second time
  - That it is possible to assign to the object obtained by applying * to an input iterator
  - That it is possible to read from the object obtained by applying * to an output iterator
  - That it is possible to test two output iterators for equality or inequality (== and != may not be defined)
- **Bidirectional iterators** provide for traversal in both directions, expressed with ++ and --:
  - Same operators as forward iterator
  - Operator --
- **Random access iterators** provide for bidirectional traversal, plus bidirectional "long jumps":
  - Same operators as bidirectional iterator
  - Operator += *n* and -= *n* with *n* of type *int*
  - Addition and subtraction of an integer through operator + and operator –
  - Comparisons through operator <, operator >, operator <=, operator >=
- **Any C++ pointer type, T*, obeys all the laws of the random access iterator category.**

Iterator validity

- When a container is modified, iterators to it can become **invalid**: the result of operations on them is not defined
- Which iterators become invalid depends on the operation and on the container type

Container	***operation***	***iterator validity***
vector	inserting	reallocation necessary - all iterators get invalid
		no reallocation - all iterators before insert point remain valid
	erasing	all iterators after erasee point get invalid
list	inserting	all iterators remain valid
	erasing	only iterators to erased elements get invalid
deque	inserting	all iterators get invalid
	erasing	all iterators get invalid

Limits of the model

- Iterators provide a linear view of a container
- Thus we can define only algorithms operating on single dimension containers
- If it is needed to access the organization of the container (i.e. to visit a tree in a custom fashion) the only way is to define a new iterator
- Nonetheless the model is expressive enough to define a large number of algorithms!

Under the hood…

- To really understand the philosophy behind STL it is necessary to dig into its implementation
- In particular it is useful to understand on which language mechanisms it is based upon:
  - Type aliases (typedefs)
  - Template functions and classes
  - Operator overloading
  - Namespaces

Iterators: small struct

- Iterators are implemented by containers
- Usually are implemented as struct (classes with only public members)
- An iterator implements a visit of the container
- An iterator retains inside information about the state of the visit (i.e. in the vector the pointer to the current element and the number of remaining elements)
- The state may be complex in the case of non linear structures such as graphs

A simple forward iterator for vectors

```
template <class T>
struct v_iterator {
  T *v;
  int sz;
  v_iterator(T* v, int sz) : v(v), sz(sz) {}
  // != implicitly defined
  bool operator==(v_iterator& p) { return v == p->v; }
  T operator*() { return *v; }
  v_iterator& operator++() { // Pre-increment
    if (sz) ++v, --sz; else v = NULL;
    return *this;
  }
  v_iterator operator++(int) { // Post-increment!
    v_iterator ret = *this;
    ++(*this); // call pre-increment
    return ret;
  }
};
```

Where is used v_iterator?

```
template <class T>
class vector {
private:
  T v[];
  int sz;
  struct v_iterator { … };
public:
  typedef v_iterator iterator;
  typedef v_iterator const const_iterator;
  typedef T element;
  …
  iterator begin() { return v_iterator(v, sz); }
  iterator end() { return v_iterator(NULL, 0); }
};
```

Inheritance? No thanks!

- STL relies on typedefs combined with namespaces to implement genericity
- The programmer always refers to *container::iterator* to know the type of the iterator
- *There is no relation among iterators for different containers*!
- The reason for this is **PERFORMANCE**
- Without inheritance types are resolved at compile time and the compiler may produce better code!
- This is an extreme position: sacrificing inheritance may lead to lower expressivity and lack of type-checking
- STL relies only on coding conventions: when the programmer uses a wrong iterator the compiler complains of a bug in the library!

Inlining

- STL relies also on the compiler
- C++ standard has the notion of **inlining** which is a form of semantic macros
- A method invocation is type-checked then it is replaced by the method body
- Inline methods should be available in header files and can be labelled *inline* or defined within class definition
- Inlining isn't always used: the compiler tends to inline methods with small bodies and without iteration
- The compiler is able to determine types at compile time and usually does inlining of function objects

Memory management

- STL abstracts from the specific memory model used by a concept named ***allocators***.
- All the information about the memory model is encapsulated in the **Allocator** class.
- Each container is parametrized by such an *allocator* to let the implementation be unchanged when switching memory models.

```
template <class T,
   template <class U> class Allocator = allocator>
   class vector {
... };
```

- The second template argument is a default argument that uses the pre-defined allocator "allocator" (implementing STL's own memory management strategies), when no other allocator is specified by the user.

Potential problems

- The main problem with STL is error checking
- Almost all facilities of the compiler fail with STL resulting in lengthy error messages that ends with error within the library
- The generative approach taken by C++ compiler also leads to possible code bloat
- Code bloat can be a problem if the working set of a process becomes too large!