# Software Design (C++)
# 4. Templates and standard library (STL)

Juha Vihavainen
University of Helsinki

## Overview

- Introduction to templates (generics)
  - **std::vector** again
  - templates: specialization by code generation
  - pros and cons of templates
- STL:
  - containers: basic data structures
  - iterators: to access elements
  - algorithms: processing element sequences

# Templates

- But we don’t just want a vector of **double**
- We want vectors with any element types we specify
  - **std::vector <double>**
  - **std::vector <int>**
  - **std::vector <Month>** // **enum class** *type*
  - **std::vector <std::vector <Record>>** // **vector** *of* **vectors** . .
  - **std::vector <char>** // ? *why not use* **string** . .
  - **std::vector <Record*>** // ? **vector** *of pointers*
  - **std::vector <std::shared_ptr<Record>>** // *smart pointers* ☺
- We can design our own parameterized types, called *templates*
  - make the element type a parameter to a template
- A template is able to take both built-in types and user-defined types as element types (of course)



# Templates for generic programming

- Code is written in terms of yet *unknown* types that are to be later specified
- Also called “*parametric polymorphism*”
  - parameterization of types and functions by types - and by integer values, in C++
- Reduces duplication of source code
- Provides flexibility and performance
  - providing good performance is essential: real time and numeric calculations
  - providing flexibility is essential
    - e.g., C++ standard containers

## Templates for generic (cont.)

- Template definitions and specializations (instantiations)

```
template <typename T, int N>
class Buffer {  . . .
    Buffer () { buff [0] = T ();  . . . }    //  assumes N >= 1
    T buff [N];        ←   use static size for this buffer
};

template <typename T, int N>
void fill (Buffer <T, N>& buffer)  {  /*  . . .  */   }  . . .

// for a class template, specify the template arguments:
Buffer <char, 1024> buf;   //  for buf,  T is char and N is 1024

// for a function template, the compiler (usually) deduces arguments:
fill (buf);   //  here also, T is char and N is 1024: that's what buf has
              //    the same as:  fill <char, 1024> (buf);
```



## Parameterize with element type

```
// an almost real vector of Ts:
template <typename T>                    //  or (originally) "class T"
class vector {     . . .
   void push_back (T const&);
};
```

produce separate **push_back** for each instantiation: uses **T::T (T const&)** to place the new item at the end

```
std::vector <double> vd;                   //  T is double
std::vector <int> vi;                      //  T is int
std::vector <std::vector <int>> vvi;       //  T is vector <int>
std::vector <char> vc;                     //  T is char
std::vector <double*> vpd;                 //  T is double*
std::vector <std::shared_ptr <E>> spe;   //  T is shared_ptr <E>
```

## Essentially, **std::vector** is something like:

```
template <typename T>                                    // read "for all types T"
class vector {
public:
    explicit vector (int s) : sz (s), elem (new T [s]), space (s) { . . . }
    T& operator [] (int n) { return elem [n]; }  // access: return reference
    int size () const { return sz; }                // . . . etc.
    vector () : sz (0), elem (nullptr), space (0);  // zero-arg. constructor
    vector (vector const&);                          // copy ctor
    vector& operator = (vector const&);             // copy assignment
    ~ vector () { delete [] elem; }                  // destructor
private:
    int sz;                                          // the size
    T * elem;                                        // a pointer to the elements
    int space;                                       // size + free space
};
```

- This original template is analyzed only partially. The use of **T** is type checked when the template is actually instantiated (and compiled).

## Essentially, **std::vector <double>** is something like

```
// a new class is instantiated (generated) from the template and compiled:
class _vector {        // the compiler generates from: "vector <double>"
public:                // uses some internal name for vector<double>
    explicit _vector (int s) : sz (s), elem (new double [s]), space (s) { . . . }
    double& operator [] (int n) { . . . }           // access element
    int size () const { return sz; }                // . . . etc.
    _vector () : sz (0), elem (nullptr), space (0) {}  // zero-arg. ctor
    _vector (_vector const&);                        // copy ctor
    _vector& operator = (_vector const&);           // copy assignment
    ~ _vector () { delete [] elem; }                 // destructor
private:
    int sz;                                          // the size
    double * elem;                                   // a pointer to the elements
    int space;                                       // size + free space
};
```

- Member functions are instantiated <u>only if called</u>

## Templates: “no free lunch”

- Template instantiation generates custom (type-specialized) code
  - => efficient but *may* involve memory overhead (replicated binary)
  - however, only those templates used/called are actually instantiated
- Sometimes poor diagnostics (obscure messages) -- at least historically
- Delayed error messages: only when "source" actually gets generated..
- Used templates must be fully defined in each separate translation unit
  - need the template source code to be specialized by the instantiation
  - so (usually) must place template definitions in header files
  - the new **extern template** (C++11) feature suppresses multiple implicit extra instantiations of templates: a way of avoiding significant redundant work by the compiler and linker
- Usually: no problems using available template-based libraries
  - such as the C++ standard library: e.g., **std::vector**, **std::sort()**
  - initially, should probably only write simple templates yourself..



## STL background

- the STL was developed by Alex Stepanov, originally implemented for Ada (80's - 90's)
- in 1997, STL was accepted by the C++ Standards Committee as part of the standard C++
- adopting STL strongly affected various language features of C++, especially those features offered by templates
- supports basic data types such as *vectors*, *lists*, associative *maps*, *sets*, and algorithms such as sorting
  - efficient and compatible with C computation model
  - not object-oriented: uses value-copy semantics (copy ctor, assign)
  - many operations (called "algorithms") are defined as stand-**alone** functions
  - uses templates for reusability
  - provides *exception safety* for all operations (on some level)

## STL examples

```
std::vector <std::string> v; . . //  some code to initialize v
v.push_back ("123");  . .        //  can grow dynamically

if (! v.empty ())
   std::cout << v.size () << std::endl;

std::vector <std::string> v1 = v;  //  make a new copy of v  (copy ctor)

std::list <std::string> list (v.begin (), v.end ());
                                 //  makes a list copy of v using iterators
std::list <std::string> list1;     . .
std::swap (list, list1);         //  swap two lists (efficiently)
   . .                           //      actually calls: "list.swap (list1)

typedef std::shared_ptr <std::vector <int>> VectPtr;
VectPtr f (std::vector<int> v)  {  //  copy constructs local variable!
   . . v [7] = 11;  . .  return VectPtr (new std::vector<int> (v));  }
```



## Basic principles of STL

- STL containers are type-parameterized templates, rather than classes with inheritance and dynamic binding
  - e.g., no common base class for all of the containers
  - no virtual functions and late binding used
- however, containers implement a (somewhat) uniform service interface with similarly named operations (**insert**, **erase**, **size**..)
- the standard **std::string** was defined first but later extended to cover STL-like services (e.g., to provide **iterator**s)
- STL collections do not directly support I/O operations
  - **istream_iterator <T>** and **ostream_iterator <T>** can represent IO streams as STL compatible iterators
  - so IO can be achieved using STL algorithms (**std::copy**, etc.)

# Components of STL

1. *Containers*, for holding (homogeneous) collections of values: a container itself manages (owns) its elements and their memory
2. *Iterators* are syntactically and semantically similar to C-like pointers; different containers provide different iterators but with a similar pointer-like interface
3. *Algorithms* are functions that operate on containers via iterators; iterators are given as (generic) parameters; the algorithm and the container must support compatible iterators (using implicit generic constraints)

In addition, STL provides, for example

- *functors*: objects to be "called" as if they were functions ("(...)")
- various *adapters*, for adapting components to provide a different public interface (**std::stack**, **std::queue**)



```
#include <iostream>                  // std::cin, std::cout, std::cerr
#include <vector>                    // std::vector
#include <algorithm>                 // std::reverse, std::sort..

int main () {
   std::vector <double> v;           // buffer for input data
   double d;
   while (std::cin >> d) v.push_back (d);  // read elements until EOF
   if (! std::cin.eof ()) {                // check how input failed
        std::cerr << "Input error\n";   return 1; }
   std::reverse (v.begin (), v.end ());
   std::cout << "elements in reverse order:\n";
   for (const auto x  :  v)  std::cout << x << '\n';
}
```

**loop local that cannot be modified**

# STL algorithms

- STL algorithms are implemented for efficiency, having an associated time complexity (constant, linear, logarithmic)
- They are defined as function templates, parameterized by iterators to access the containers they operate on:

the same template

```
std::vector <int> v;   . . .            // initialize v
std::sort (v.begin (), v.end ());       // instantiates sort
std::deque <double> d;                  // double-ended queue
  . . .                                 // initialize d
std::sort (d.begin (), d.end ());       // instantiate, again
```

- If a general algorithm, such as sorting, is not available for a specific container (since iterators may not be compatible), it is provided as a member operation (e.g., for **std::list**)



# Introduction to STL containers

- A container holds a homogeneous collection of values

```
Container <T> c;    . . .    // initially empty
c.push_back (value);         // can grow dynamically
```

- When you insert an element into a container, you always insert a *value copy* of a given object
  - the element type **T** must provide copying of values
- Heterogeneous (polymorphic) collections are represented as containers storing pointers to **a base** class
  - brings out all memory management problems (C pointers)
  - can use **std::shared_ptr (**with reference counting)
  - can use **std::unique_ptr** (with its single-owner semantics)
- Containers support constant-time **swap**s (usually)

# Intr. to STL containers (cont.)

- in *sequence containers*, each element is placed in a certain relative position: as first, second, etc.:

| | |
|---|---|
| **std::vector <T>** | vectors, sequences of varying length |
| **std::deque <T>** | deques (with operations at either end) |
| **std::list <T>** | doubly-linked lists |
| **std::forward_list <T>** | singly-linked lists |

- *associative containers* are used to represent sorted collections

| | |
|---|---|
| **std::map <KeyType, ValueType>** | (ordered search tree) |
| **std::unordered_map <KeyType, ValueType>** | (hash map) |

  - for a **map**, provide **operator** < for the key type
  - for a hash map, provide **std::hash<Key>** for the key type
  - also *set*s and *multi*-key/value versions



# Intr. to STL containers (cont.)

- Standard containers are somewhat interchangeable - in principle, you can choose the one that is the most efficient for your needs
  - however, interfaces and services are not *exactly* identical
  - changing a container may well involve changes to the client source code (that calls the services of a container)
- Different kinds of algorithms require different kinds of iterators
  - once you choose a container, you can apply only those algorithms that accept a compatible iterator
- Container adapters are used to adapt containers for the use of specific interfaces (e.g., **push ( . . . )**, **pop ()**, etc.)
  - for example, **std::stack** and **std::queue** are adapters of sequences; the actual container (**deque**) is a protected member

# Iterators (again)

- an iterator provides access to elements in a container; every iterator it has to support (at least)

| | |
|---|---|
| ***it it->** | to access the current element or its member |
| **++it** | to move to the next element |
| **it == it1** | "pointer" equality |
| **it != it1** | "pointer" inequality |

- container classes provide iterators in a uniform way as *standardized typedef names* within the class definition

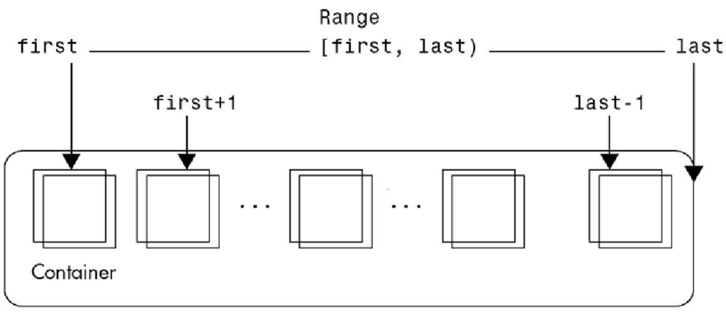
**std::vector<std::string>::iterator** // *is a typedef*
**std::vector<std::string>::const_iterator**

| | |
|---|---|
| **begin ()** | points to the first element (if any) |
| **end ()** | points beyond the last (end marker) |

- **const_iterator**s are required for **const-**qualified containers



# Iterators (cont.)




**C::iterator first = c.begin (), last = c.end ();**

- a container holds a set of values, of type **C::value_type** (typedef)
- an iterator points to an element of this container, or just beyond the last proper element (a special *past-the-end value*)
- it can be dereferenced by using the operator ***** (e.g., "***it**"), and the operator **->** (e.g., "**it->op ()**")

## Iterators (cont.)

- Iterators are syntactically compatible with *C* pointers

```
Container c;      . . .
Container::iterator it;                          // a shorter form:
for (it = c.begin (); it != c.end (); ++it)  {   // for (auto& x : c) . .
    . . . it->op (); . . . std::cout << *it; . . .
}
```

**to allow modifications to the container**

- Could use a range-**for** statement, or an algorithm: **for_each**, **copy**
- Non-**const** iterators support overwrite semantics: can modify or overwrite the elements already stored in the container
- *Generic algorithms* are not written for a particular container class in STL but use iterators instead
- There are *iterator adapters* that support insertion semantics (i.e., while writing through an iterator, *inserts* a new element at that point)



## Using iterators within function templates

```
template <typename It, typename T>   // a sample function template
bool contains (It first, It beyond, T const& value) {
    while (first != beyond && *first != value) ++first;
    // note implicit constraints on It first and T value
    return first != beyond;
}
// can operate on any primitive array:
int a [100];    . . .                     // initialize elements of a
bool b = contains (a, a+100, 42);
// can operate on any STL sequence:
std::vector <std::string> v;  . . .       // initialize v
b = contains (v.begin (), v.end (), "42");
```

Why not '<' comparison?

the same source code

## Iterators: summary

- *Validity of iterators* is not guaranteed (as usual in C/C++)
  - especially, modifying the organization of a container may *invalidate* any existing iterators and references (this depends on the kind of container and modification)
- For array-like structures, iterators are (usually) native C-style pointers to elements of the array (e.g., **std::vector**)
  - efficient: uses direct addresses and pointer arithmetics
  - may have the same security problems as other native pointers
  - some libraries may provide optional *special checked* iterators
- For other containers (e.g., **std::list**), iterators are provided as abstractions defined as classes
  - with properly overloaded operators `++`, `*`, `->`, etc.
  - but traverse *links* between nodes instead of address calculations



## Summary

- C++ containers are based on generic *templates*
- Templates provide compile-time polymorphism via type parametrization
  - STL templates don't use such *object-oriented* features as inheritance or late binding of methods
- Templates are instantiated, and these instantiations are then compiled (in a selected manner)
- STL provides *containers*, *iterators*, and *algorithms*