4 C++ templates and STL
Introduction to generics and containers

Preview

- templates and template parameters
- instantiating class and function templates
- basic concepts of the C++ standard library
- containers, iterators, and algorithms
- on STL container classes

C++ templates

- a parameterized class template is a type generator that can be used to produce new types and code
- class and function templates are usually parameterized by types:

```
template <typename T> class Stack;       // declarations
template <typename T> void swap (T&, T&);
```

- a specified template name acts as such like a new type name within the program but can be given an alias name:

```
. . . Stack <std::string> . . .              // a class name
typedef Stack <std::string> StringStack;     // its alias
```


C++ templates (cont.)

- a class template is instantiated to generate particular class-types by providing the missing type parameters

```
Stack <int> intStack;         // create a stack of integers
Stack <std::string> stringStack;     // another stack
```

- template is a type generator: gives customized code
- template instances are distinct types: no inheritance or subtype relationship (type compatibility), by default
- a function template may also have type parameters that are derived by the compiler:

```
Stack <int> e1, e2;  . . .        // two stacks declared
swap (e1, e2);          // compiler instantiates for args
```

  – the compiler generates an instance of the template for the given arguments: "swap <Stack <int> > (e1, e2)"

Template parameters

- templates have three kinds of parameters:

  (1) built-in types or user-defined classes: <typename T>

  (2) integer constants: <std::size_t N>, and

  (3) pointers to objects or functions with external linkage

- multiple template arguments are allowed but they must all be compile-time constants (of course)
- note that floating-point numbers ("1.23") or string literals of type *char* * ("xyz") are not allowed

Defining class templates

```
template <typename T, std::size_t SIZE = 100> // sample
class Stack {
public:
    void push (T const& new_item);
    T top () const;                          // or: "T const&"
    void pop ();
    bool isEmpty () const;
    bool isFull () const;
    // etc. standard ctor and copy operations . . .
private:
    std::size_t top_;
    T stack_[SIZE];  // simplified version: creates T array
};
```

Defining class templates (cont.)

```
template <typename T, std::size_t SIZE>
Stack <T, SIZE >::Stack () : top_(0)  { }

template <typename T, std::size_t SIZE>
Stack <T, SIZE >::~Stack ()  { }

template <typename T, std::size_t SIZE>
void Stack <T, SIZE >::push (T const& item) {
   if (top_>= SIZE)                    // check precondition
      throw std::logic_error ("Stack::push() overflow"));
   stack_[top_++] = item;
}
```

- the member functions of a class template are function templates

Instantiating class templates

- the instantiated template names can be used wherever regular C++ class names can

```
typedef Stack <double, 50>  StackOfDouble;
void foo (Stack <int> const&);     // uses default = 100
```

- C++ templates resemble but are not macros
  - the once instantiated name identifies the same generated class-instance at all places
  - compiler typically represents the class with some generated internal name and places the instantiation into an internal repository for future use
  - any "free" (parameter-independent) names inside a template are bound *at the point of the definition* of the template (not at instantiations)

Instantiating class templates (cont.)

- a class template and its functions are instantiated only when needed, i.e., when a complete class definition or when a particular member function is really required
- consider creating objects of an instantiated template

```
Stack <int, 100> si;          //  stack of 100 integers
Stack <double, 50> sd;        //  stack of 50 doubles
```

- in order to know the size of objects, must instantiate the definition "template < . . . > class Stack { . . . }"
- but may need only to instantiate partial services: those operations that are actually called, e.g., "si.push (..)"
- when *pointers or references* to a template instance are used, no instantiation is (yet) required

Function templates

- can also define stand-alone function templates

```
template <typename T>
void swap (T& x, T& y) {
   T tmp (x);                 // use copy constructor
   x = y;                     // use assignment
   y = tmp;
}
```

- function templates are (usually) instantiated at compile time by simply calling the function template

```
int i = 2; int j = 3;   . . .
swap (i, j);      // calls: void swap <int> (int&, int&)
Integer k = 3, m = 4;   . . .
swap (k, m);      // calls: void swap <Integer> ( . . . )
```


Template constraints

- the operations performed within the body of a template *implicitly constrain* the parameter types
- this is called "constraints through use":

```
template <typename T>
 . . . //  some code within a class template . . .
 . . . T t1, t2;          //  implies existence of default ctor
 . . . t1 + t2            //  implies a plus operator
 . . .
```

- the above code *implies* that T should provide +:
  - true for all built-in numerical types
  - can be defined for user-defined classes
- if missing, generates a *compile-time error* => supports early and secured error checking

Compilation of templates

- the current way of organizing template code is to avoid separate compilation of declarations and definitions, and put all of the template definitions into header files
- the header files are then included into each translation unit that instantiates the templates
- the C++ standard library is totally based on templates and provides examples of their extensive use
  - containers, algorithms, strings, IO streams, etc.
  - parameterization of classes and functions contributes to reusability and adaptability of software components
  - note that inheritance and late binding are required to provide polymorphism *at run time* (sometimes needed and sometimes not)

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
  - many operations (called "algorithms") are defined as stand-alone functions
  - uses templates for reusability
  - provides *exception safety* for all operations

STL examples

```
std::vector <std::string> v;   // empty vector of strings
 . . .                          // 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 copy of v
std::list <std::string> s (v.begin (), v.end ());
                    // makes a list copy of v using iterators
std::list <std::string> s1;              . . . // initialize s1
std::swap (s, s1);          // swap two lists (efficiently)
              // actually calls: "s.swap (s1)"
```


Basic principles of STL

- STL containers are type-parameterized templates, rather than classes with inheritance and dynamic binding
  - 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
- the standard std::string was defined first but later extended to cover STL-like services (e.g., iterators)
- STL collections do not generally support I/O operations
  - istream_iterator <T> and ostream_iterator <T> can represent IO streams as STL compatible iterators
  - IO can also be achieved using STL algorithms (copy, etc.)

Components of STL

(1) *containers*, for holding (homogeneous) collections of values: a container itself manages (owns) its elements

(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 used as if they were functions ("()")
- various adapters, for adapting components to provide a different interface


```
#include <iostream>                // get std::cin, std::cout
#include <vector>                           // get std::vector
#include <algorithm>   // get std::reverse, std::sort, etc.
int main () {
   std::vector <double> v;           // buffer for input data
   double d;
   while (std::cin >> d)                      // read elements
      v.push_back (d);
   if (!std::cin.eof ()) {          // check how input failed
      std::cerr << "Format error\n";   return 1; }
   std::reverse (v.begin (), v.end ());
   std::cout << "elements in reverse order:\n";
   for (std::size_t i = 0; i < v.size (); ++i)
      std::cout << v [i] << '\n';
}
```


Basic concepts of STL

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

```
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 are not compatible), it is provided as a special member operation (e.g., for std::list)

Introduction to 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 actually 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 pointer memory management problems
  - cannot use std::auto_ptr (with its odd copy semantics)
  - *smart pointers* with reference counting work are OK
- containers support *constant-time* swaps - if use the same mem. manager - and *usually do*; if in doubt, can check:
  assert(x.get_allocator()==y.get_allocator()); [see Stroustrup]

Intr. to containers (cont.)

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

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

- *associative containers* are used to represent sorted collections (the key type must provide operator <)

set <KeyType>	sets with unique keys
map <KeyType, ValueType>	maps with unique keys
multiset <KeyType>	sets with duplicate keys
multimap <KeyType, ValueType>	- the same

- hash_map <KeyType, ValueType> is provided by many libraries but not (yet) by the standard

Intr. to containers (cont.)

- standard containers are somewhat interchangeable - in principle, you could choose the one that is the most efficient for your needs
  - however, interfaces and services are not identical
  - changing a container may well involve changes to the client code
- different kinds of algorithms require different kinds of iterators
  - once you choose a container, you can apply 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 container is a *protected* member)

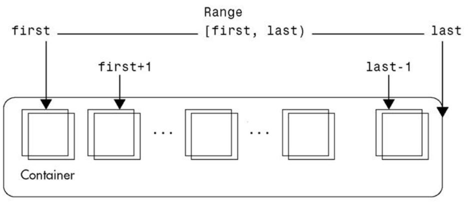
Iterators

- an iterator provides access to elements in a container; every iterator it has to support

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

- container classes provides 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_iterators are required to handle *const* containers

Iterators (cont.)



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

- a container holds a set of values, of type value_type
- an iterator points to an element of this container, or just beyond the last (is 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;
for (it = c.begin (); it != c.end (); ++it) {
   . . . it->op (); . . . std::cout << *it; . . .
}
```

- non-const iterators support overwrite semantics: modify or overwrite the elements *already stored* in the container
- in addition, there are iterator adapters that support insertion semantics (i.e., while writing through an iterator, adds a new element at that point)
- *for* can be replaced by an algorithm: for_each, copy
  - generic algorithms are not written for a particular container class in STL but use iterators instead

Using iterators within function templates

```
template <typename InputIterator, typename T>
bool contains (InputIterator first, InputIterator beyond,
               T const& value) {
    while (first != beyond && *first != value)
        ++first;        // note implicit constraints on first and T
    return first != beyond;
}
// can operate on primitive arrays:
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");
```


Syntax: using "typename" keyword

- for generic programming, STL provides "standard" types

```
template <typename T> class vector {
public:
    typedef T   value_type;   // in every container
    typedef T * iterator; . . .   // "T *" depends on impl.
    typedef std::size_t size_type; . . .   // or whatever . .
```

- "typename" is also a way of telling a compiler that a name is meant to identify a type; for example

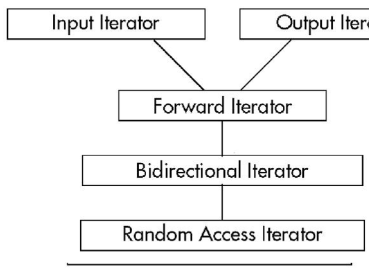
```
template <typename T> void fun (T& v) {
    typename T::iterator it = v.begin ();  . . .
}
```

- often required when a type name depends on a template parameter; see, e.g., Appendix C 13.5. *Typename and template* [Stroustrup] - *Warning*: enforcement varies

More on iterators

- a sequence of consecutive values in the container is determined by an iterator range, defined by two iterators: [first, last)
  - last is *assumed* reachable from first by using the ++ operator, and all iterator values, including first but excluding last can be dereferenced ("*")
- iterators can be compared for equality and inequality
  - they are equal if they point to the same element of the container (or both just beyond the last value)
- the compiler does not normally check the validity of ranges, e.g.,
  - that iterators really even refer to the same container
  - but checked container libraries are available..

Iterator categories



- input iterator: . . =*it ++
- output iterator: *it= . . ++
- forward iterator: allows multipass traversals
- bidirectional iterator: --
- random access: [ ] it+i it-i

More on iterators (cont.)

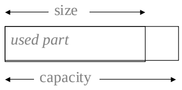
- an empty range is specified as [first, first)
- can add and subtract integers from random iterators
- random iterators can be subtracted from each other, so last - first is the distance between these two iterators, equal to the number of elements in this range
  - there is a special type called difference_type for this purpose

Sequence examples

```
std::deque <double> d (10, 1.0);        // with 10 values (1.0)
std::vector <Integer> v (10);      // same as: v (10, Integer ())
          // vector with 10 items; each with the default value
std::list <Integer> s1;                            // empty list
// store some elements:
s1.push_front (6);  . . .
s1.insert (s1.end (), 13);   . . .                // push_back
// create list s2 that is a copy of s1
std::list <Integer> s2 (s1.begin (), s1.end ());
// reinitialize all elements to Integer (2)
s2.assign (s2.size () - 2, 2);        // has two fewer elements
```

On STL vectors

- represent resizable (flexible) arrays (as std::strings)
- capacity is maximum size before reallocation
  - copying elements can be prevented by reserve
- size is the current number of elements actually stored in the vector (less than or equal to the capacity)



- insertions at the end of a vector are amortized constant time (while an single insertion might be linear in size)
- on reallocation, any iterators or references are invalidated
- overwriting operations through iterators do not reallocate vectors, so the programmer must prevent any overflow and memory corruption


```
std::vector <int> v;   v.reserve (100);
int i = 0;
while (std::cin >> i)              // read from the standard input
    v.push_back (i);               // will expand vector if needed
for (std::size_t i = 0; i < v.size (); ++i)
    std::cout << v[i] << " ";
try {                              // use checked access
    std::cout << v.at (100);       // at () may throw
} catch (std::out_of_range const&) {   // invalid index
    std::cout << "doesn't have 101 elements" << std::endl;
}
// peculiar pop_back loop (explanation left as an exercise)
for (std::size_t i = 0; i < v.size () / 2; ++i) v.pop_back ();  //?
std::vector <int> v1 (v);          // copy to v1
v1.insert (v1.begin () + 1, 117);  // insert as second
```


Deques

- deques are similar to vectors (*random access*)
- additionally operations to insert and remove elements in front (in O (1) *amortized* time)
  - push_front () — add new first element
  - pop_front () — remove the first element
- removals and inserts into middle take linear time (O (n))
- deques don't provide operations capacity and reserve
- usually implemented as an array of arrays: one end "grows from 0 to x" and the other "grows from x to 0"

```
. . . . <===     allocates memory in blocks
========
===> . . . .
```

- indexing requires determination of memory block => is little slower than for vectors (but constant time)

Linked lists

```
std::list <char> s;                   // empty list
s.insert (s.end (), 'a');             // or push_back
s.insert (s.end (), 'b');             // s contains 'a' and 'b'
std::list <char> s1;                  // new empty list
// copy s to s1:
s1.insert (s1.end (), s.begin (), s.end ());
s.clear ();                           // remove all elements
assert (s1.front () == 'a');
s1.erase (s1.begin ());               // remove first element
assert (s1.front () == 'b');
```


Choosing correct containers

- choose *vectors* when there are
  - random access operations
  - most insertions and removals are at the rear end
- choose *deques* when there are
  - random access operations
  - frequent insertions and deletions at either end
- choose *lists* when there are
  - few random access operations
  - frequent insertions and deletions at inside positions
  - want to guarantee that iterators and references are valid after structural modifications (can remember positions)

Templates: summary

- a template is partially checked at the point of definition
- template parameter-dependent code uses *implicit constraints* that are checked when the template becomes specified at its instantiation
- the code may compile for some type arguments, and fail for some other type arguments (reported at compile time)
- the implicit constraints of a class and function templates are required only if a template becomes instantiated
- and templates are instantiated only when really needed: an object is created or a particular function is called
- all type parameters need not satisfy *all* requirements implied by the *full* template definition - since only some member functions may be actually needed and called for a given type parameter in a given context (system)

STL: summary

- *containers* are parameterized class templates; they try to make *minimal* assumptions about the type of elements that they hold - but of course need some operations, e.g., for constructing and copying elements
- *iterators* are similar to pointers and provide access to elements within a particular container
  - iterators can be used for either reading or modifying the elements of the container
- *algorithms* are parameterized function templates; they are purposely decoupled from the containers
  - do not need to know the actual type of the containers
  - they always use the iterators to access elements in the container

Iterators: summary

- validity of iterators is not guaranteed (as usual in C/C++)
  - especially, modifying the organization of a container often invalidates all existing iterators and references (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 ptr arithmetics
  - have same security problems as other native pointers
  - some libraries can provide special checked iterators
- for other containers (e.g., std::lists), iterators are provided as abstractions defined as classes
  - with properly overloaded operators ++, *, ->, etc.
  - but traverse links between nodes instead of address calculations