Programming 2

Generic Programming

Daniel POP

Unit 1

Agenda

Generic Programming (Templates)

- Introduction
- From abstract to generic types
- Class templates and template instantiation
- Function templates
- Specialization
- Using template arguments to specify policy
- Derivation and templates

From abstract to generic types

```
// vector of integers
class vector_i {
     int *v;
     int size;
public:
     vector_i( ): v{NULL}, size{0} {
     }

     int get(int n) {
          if (0<=n && n<size)
                 return v[n];
          throw BadIndex{};
     }

     void  add(int x);
};

int main(int, char*[])  {
     vector_i v ;
     v.add(1) ;
     v.add(100);
     cout << v.get (0);
     return 0;
 }
```

From abstract to generic types

```
// vector of floats
class vector_f {
     float *v;
     int size;
public:
     vector_f( ): v{NULL}, size{0} {
     }

     float get(int n) {
          if (0<=n && n<size)
                 return v[n];
          throw BadIndex{};
     }

     void  add(float x);
};

int main(int, char*[])  {
     vector_f v ;
     v.add(1.5) ;
     v.add(100.5);
     cout << v.get (0);
     return 0;
 }
```

From abstract to generic types

```
// vector of integers
class vector_i {
     int *v;
     int size;
public:
     vector_i( ): v{NULL}, size{0} {
     }

     int get(int n) {
          if (0<=n && n<size)
                 return v[n];
          throw BadIndex{};
     }

     void  add(int x);
};

int main(int, char*[])  {
     vector_i v ;
     v.add(1) ;
     v.add(100);
     cout << v.get (0);
     return 0;
 }
```

```
// vector of floats
class vector_f {
     float *v;
     int size;
public:
     vector_f( ): v{NULL}, size{0} {
     }

     float get(int n) {
          if (0<=n && n<size)
                 return v[n];
          throw BadIndex{};
     }

     void  add(float x);
};

int main(int, char*[])  {
     vector_f v ;
     v.add(1) ;
     v.add(100);
     cout << v.get (0);
     return 0;
 }
```

From abstract to generic types

Declaration of template class

```
template <class TElement> class vector {
     TElement *v;
     int size;
public:
     vector( ): v{NULL}, size{0} {
     }

     TElement get(int n) {
          if (0<=n && n<size)
                 return v[n];
          throw BadIndex{};
     }

     void  add(TElement x);
};
```

Usage of template class (template instantiation)

```
int main(int, char*[] )  {
     vector<int> v;
     v.add(1) ;
     v.add(100);
     cout << v.get (0);

      vector<float> fv;
      return 0;
}
```

Implementation

```
template<class TElement> void vector< TElement >::add(TElement x) {
     // add e to the vector v
}
```

Definitions

Express algorithms independently of representation details.

Generic Programming = Decide which algorithms you want; parameterize them so that they work for a variety of suitable types and data structures.

DEFINITION [Template, Parameterized data type] A template represents a <u>family of types or functions</u>, parameterized by a *generic* type.

Benefits

- Increased code reusability (e.g. generic lists/containers, generic sorting algorithms etc.)
- Allow design and implementation of general purpose libraries (e.g. math libraries, data structures etc.)

Template declaration

```
Syntax
template<list-of-parameters> Declaration;
```

1) **List-of-parameters** – a comma separated list of template’s parameters:
– type parameters (class T): T may be initialized with a primitive type (int, char, float), a user-defined type (MyClass, complex etc.), a pointer type (void *, etc), a reference type (int&, MyClass&) or a template instantiation (vector<int>)
– non-type parameters (int i): non-type parameters can be instantiate only with constant values and are constant in type declaration/definition.

2) **Declaration**
– declaration/definition of function or class
– definition of a static member of a template
– definition of a class/function member of a template
– definition of a template member of an ‘ordinary’ class

```
template<class Type1, class Type2, int I, float FL> class MyClass {
     /* class declaration */
};
```

Template class instantiation

DEFINITION [Template instantiation] The process of generating a class definition from a template class is called template instantiation.

Syntax

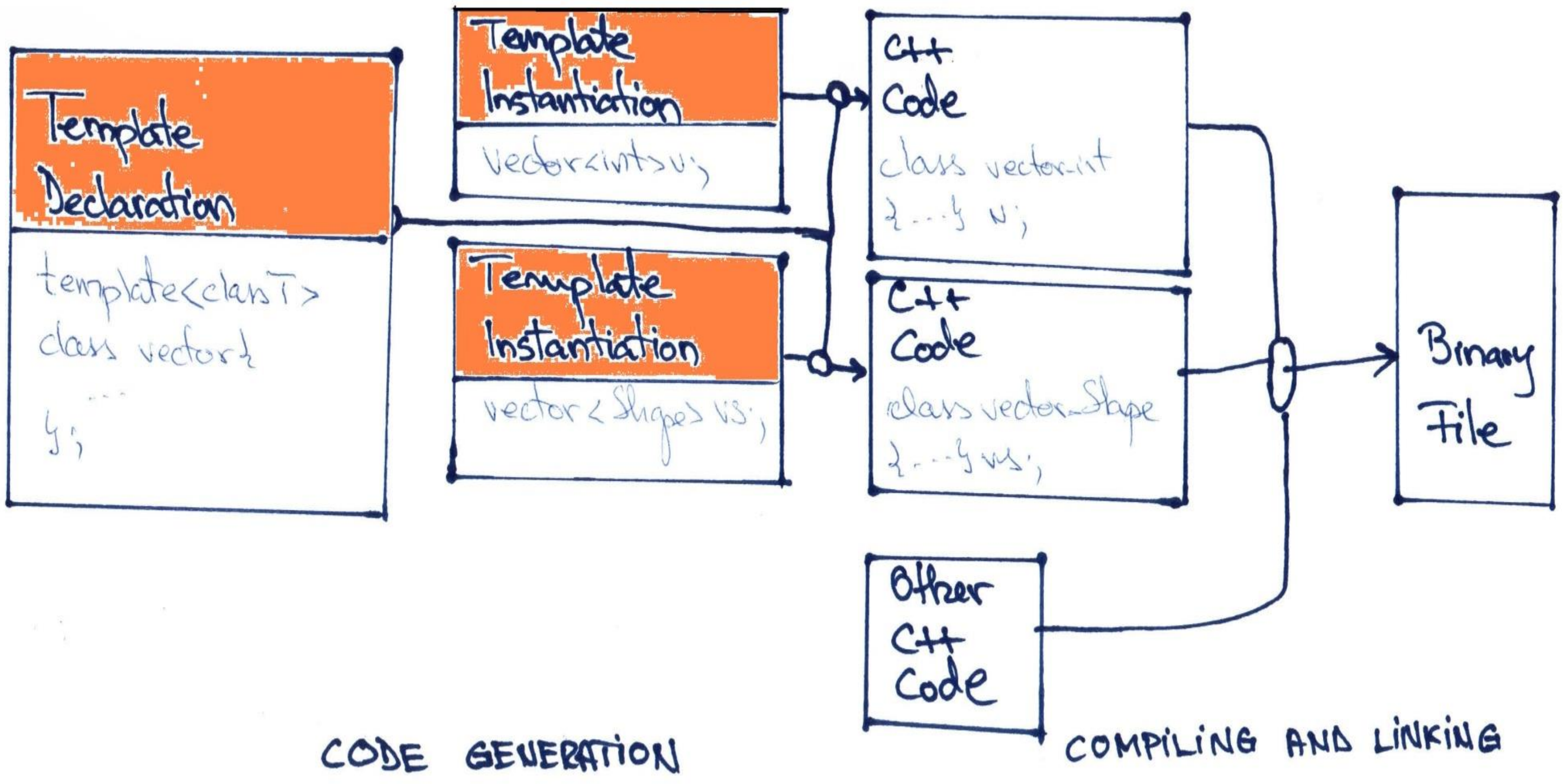
```
TemplateClassDefinition<arguments> decl;
```

```
vector<int> d1;
vector<float> d2;
Buffer<char, 10> d3;

d1.add(10);
d1 = d2;
```

```
MyClass<int, Employee, 10, 0.5> x;
MyClass<Employee&, Manager*, 20-1, 103/7> y;
```

Code generation



An important challenge for the C++ compiler is to reduce the amount of generated code at template instantiation:

Generate code only for used classes/functions

Type equivalence

‘On-demand’ code generation

The compiler will generate class declarations only for instantiated templates.

Ordinary C++ functions are generated only for used member functions of instantiated templates.

If no template instantiation is made then nothing is generated.

```
int main(int, char*[] )  {
     vector<int> v0, v1;
     v0.add(1) ;
     v0.add(100);
     cout << v0.get (0);
     v1 = v0;

     vector<float> fv;
     return 0;
}
```

- for class vector<int> => class declaration (incl. inline functions), assignment operator, function add
- for class vector<float> => class declaration (incl. inline functions)

Type equivalence

```
template <class T, int i> class Buffer {
    T buffer [i];
    int size;
public:
    Buffer() : size{i} { }
    // error: being constant, i cannot be modified
    void f( ) { i=10;}
};
```

The following types are equivalent:

```
typedef  unsigned char UCHAR ;
Buffer<unsigned char ,10> ⇔ Buffer <UCHAR , 10> ⇔ Buffer <UCHAR , 40-30>
Buffer<char, 100> b1; // char buffer[100];
Buffer<void*, 200-10> b2; // void* buffer[190];

int N = 10;
Buffer<char, N> b; // => error; N is not constant
```

The following types are <u>not</u> equivalent:

`Buffer<unsigned char, 10>` **`is not equivalent to`** `Buffer <char , 10>` because ‘unsigned char’ and ‘char’ are not the same type!

Type checking

Errors in template definition:

- that can be detected at compilation time, e.g. missing semicolon or misspelled keyword;
- that can be detected only at template instantiation (see example below)
- that can be detected only at runtime

DEFINITION [Point of instantiation] The first instantiation of a template is called point of instantiation and it has an important role in finding out and fixing template errors.

The arguments passed in for a template instantiation have to support the expected public interface of that type.

```
template<class T> void vector<T>::add(T x) {
     // add e to the vector v
     cout << "Added element " << x;
}

class X { };
```

For debugging purpose, use the most frequent types as arguments for type parameters.

```
void f1() {
     vector<int>  vi; // instantiation
     vi.add(100); // =>  OK!
}

void f2() {
     vector<X>  vX; // instantiation
     vX.add(X()); // =>  error! Why?
}
```

Remarks

A template class may have three types of *friends*:

- Non-template class/function
- Template class/function
- Template class/function with a specified type (specialization template).

Static members: each instance has its own static data.

Use *typedef* for frequently used templates.

```
typedef vector<char> string;

typedef Buffer<char, 100> Buffer100;

string name;
Buffer100 buf;
```

Function templates

```
int min(int x, int y) {
     return x<y ? x : y;
}

float min(float x, float y) {
     return x<y ? x : y;
}

Complex& min(complex& x,
             complex& y) {
     return x<y ? x : y;
}

void f() {
     complex c1(1,1), c2(2,3);
     min(0,1);
     min(6.5, 3);
     min(c1, c2);
}
```

Function templates

```
int min(int x, int y) {
     return x<y ? x : y;
}

float min(float x, float y) {
     return x<y ? x : y;
}

Complex& min(complex& x,
             complex& y) {
     return x<y ? x : y;
}

void f() {
     complex c1(1,1), c2(2,3);
     min(0,1);
     min(6.5, 3);
     min(c1, c2);
}
```

$\Rightarrow$

```
template<class T> T min(T x, T y) {
     return x<y ? x : y;
}

void f() {
     complex c1(1,1), c2(2,3);
     min(0,1); // ⇔ min<int>(0,1);
     min<float>(6.5, 3);
     min(c1, c2); // ⇔ min<complex>(c1,c2);
}
```

Remark: The operator < has to be overloaded for complex class.

Function templates: declaration and impl.

```
template<class T> T min(T x, T y);

template<class T> T min(T x, T y) {
     return x<y ? x : y;
}

template<class T> void sort(T[] array,
                            int size) {
  // sort the array
}

void f() {
     complex c1(1,1), c2(2,3);
     min(0,1); // ⇔ min<int>(0,1);
     min<float>(6.5, 3);
     min(c1, c2); // ⇔ min<complex>(c1,c2);

     complex carray[12];
     int iarray[] = {4, 1, 2, 3};

     sort(iarray, 4);
     sort(carray, 12);
}
```

Syntax

```
template<list-of-parameters> func-decl;
```

Function templates: instantiation and ambiguities

Template function instantiation is similar to class instantiation. Examples:

- min<int>(0, 1);
- min<complex>(…);

If the arguments type uniquely identify the parameter type then the compiler will automatically do the instantiation so that the programmer doesn't have to explicitly instantiate the function.

```
min(0,1); // okay
min(2.5, 3); // ambiguous; need explicit qualification min<float>(2.5, 3);
min<float>(2.5, 3); // okay
```

To resolve **ambiguities**:

- Use explicit qualification
- User overloading/specialization to add new functions that match the call (e.g. min(double, int))

Specialization

DEFINITION [Specialization] Alternative definitions of a template for a particular type of template parameters are called user specializations.

Specialization is a way of specifying alternative implementations for different uses of a common interface.

- Template Class Specialization
- Template Function Specialization

Template class specialization

Example: Template vector. A lot of vector<type*> instances will be created => code bloat (unnecessarily long, slow and wasteful of resources)

Solution: Containers of pointers can share the same implementation expressed through specialization.

```
// specialization for void*
template<> class vector<void*> {
     void** p;
     void*& operator[] (int i);
     // ...
};

// partial specialization for T*
template<class T> class vector<T*>
                     : private vector<void*> {
public:
     typedef vector<void*> Base;
     vector() : Base() {}
     explicit vector(int i) : Base(i) { }
     T*& elem(int i) {
          return static_cast<T*&> (Base::elem(i));
     }
     T*& operator[](int i) {
          return static_cast<T*&> (Base::operator[](i));
};
```

```
void f () {
    // specialized version
    vector<void*> v1;

    // partial specializations
    vector<MyClass*> v2;
    vector<int*> v3;
}
```

Only derivation and inline functions are defined in the partial specialization template; the rest of functionalities are implemented in the base class.

Order of specialization for class templates

Order of specialization

1. General template must be declared before any specialization.
2. The specialization must be in the scope for every use of the template with the type for which it was specialized.
3. The most specialized version will be preferred over the others.

```
// breaking remark 1
// specialization for T* defined before template definition
template<class T> class List<T*> { } ;
template<class T> class List {} ;


// breaking remark 2
// li will be instantiated using List template instead of its specialization List<T*>
// because the latter is not yet declared
template<class T> class List {} ;
List<int*> li;
template<class T> class List<T*> {};
```

Template function specialization

Template function specialization is appropriate when there is a more efficient alternative to a general algorithm for a set of template arguments.

Example: sorting vectors

```
template<class T> bool less(T a, T b) { return a < b; }

template<class T> void sort(vector<T>& v) {
     //...
    if(!less(v[j], v[k]) swap(v[j], v[k]);
}

vector<char*> vc;
sort(vc); // will not sort correctly the vector; why?
```

Template function specialization

Template function specialization is appropriate when there is a more efficient alternative to a general algorithm for a set of template arguments.

Example: sorting vectors

```
template<class T> bool less(T a, T b) { return a < b; }

template<class T> void sort(vector<T>& v) {
    //...
    if(!less(v[j], v[k]) swap(v[j], v[k]);
}

vector<char*> vc;
sort(vc); // will not sort correctly the vector; why?
```

Solution: specialize less function for **char***

```
template<> bool less<char*>(char* a, char* b) {
    return strcmp(a, b) < 0;
}

// or equivalently with

template<> bool less(char* a, char* b) {
    return strcmp(a, b) < 0;
}
```

Function template overloading

If someone needs a custom version of a template function or of a member function of a template class, then that function can be overloaded for that particular type.

```
template<class T> class complex : public number<T>
{
    // other declarations
};

template<class T> T sqrt(T);
template<class T> number<T> sqrt(number<T>);
template<class T> complex<T> sqrt(complex<T>);
double sqrt(double);

void f(complex<double> c) {
     c = sqrt(c );
     double d = sqrt(49.0);
     int x = sqrt(49);
}
```

Remark: If the template function has parameters of a type that is not defined as template parameter, then this parameters are handled as any other ‘ordinary’ C++ function parameter (for example, various possible conversion are applied).

Function overload resolution rules:

1. Find the set of function templates specializations.
2. If two template functions can be called then **select the most specialized** one for next step.
3. Do overload resolution for this set of functions, plus any ordinary functions as for ordinary functions.
4. If a function and a specialization are equally good matches, the function is preferred.
5. If no match is found then the call is an error.

Exercise

EXERCISE: Write a template function that compares two strings (implemented using `vector` template) and returns 0 if they are identical, -1 if the first is lexicographically lower than the second one and 1 otherwise. Its prototype is:

```
template<class T> int compare(const vector<T>&
s1, const vector<T>& s2);
```

Using template arguments to specify policy (I)

```
template<class T, class C> int compare(const vector<T>& s1, const vector<T>& s2) {

     for(int i=0; i<s1.length() && i<s2.length(); i++)
          if(!C::eq(s1[i], s2[i])) return C::lt(s1[i], s2[i]) ? -1 : 1;

     return s1.length()-s2.length();
}

template<class T>  class Cmp {
public:
    static int eq(T a, T b) { return a==b; }
    static int lt(T a, T b) { return a<b; }
}

class NoCaseSensitiveComparator {
public:
    static int eq(char a, char b) { return ???; }
    static int lt(char a, char b) { return ???; }
}

void f(vector<char> str1, vector<char> str2) {
     compare<char, Cmp<char>>(str1, str2);

     compare<char, NoCaseSensitiveComparator>(str1, str2);
}
```

Using template arguments to specify policy (II)

DEFINITION [Policy] = a definite course or method of action selected from among alternatives and in light of given conditions to guide and determine present and future decisions. [Webster dictionary]

Example 1: method to compare elements of a vector in a vector compare algorithms; different criteria can be used for the same type (char), for example: case/non-case sensitive, for English/Romanian etc.

Example 2: method to compare elements in a vector in a sorting algorithm.

Benefits of passing policies as template arguments over passing pointers to functions:

– several operations can be passed a single argument with no run-time cost

– easier to inline, while inlining a call through a pointer to a function requires special attention from compiler.

Read more about **Strategy** design patterns at:

https://popdaniel.wordpress.com/2012/10/31/design-patterns-week-4/

Default template parameters

```
template<class T, class C = Cmp<T>> int compare(const vector<T>& s1,
                                                 const vector<T>& s2) {
     for(int i=0; i<s1.length() && i<s2.length(); i++)
          if(!C::eq(s1[i], s2[i])) return C::lt(s1[i], s2[i]) ? -1 : 1;

     return s1.length()-s2.length();
}

void f(const vector<char> str1, const vector<char> str2) {
     compare(str1, str2); // equivalent to compare<char, Cmp<char>>(str1, str2);
     compare<char, NoCaseSensitiveComparator>(str1, str2);
}
```

Derivation and templates

- Deriving a template from a non-template class provides a common implementation for a set of templates; in other words, template provides a safe interface to a non-safe type.

  Example: see class `vector<T*>` declaration in slide 8.

- Passing the derived type to base class => definition of the basic operations on containers only once and separate from the container definition itself.

  Example:

```
template<class C> class BasicOp {
      bool operator==(const C&) const;
      bool operator!=(const C&) const;
};

// passing the derived type to base class
template<class T> class MathContainer : public BasicOp<MathContainer<T>> {
public:
       T& operator[](size_t);
};

template<class C> bool BasicOp<C>::operator==(const C& a) const {
     if(size()!=a.size()) return false;
     //...
}
```

Derivation vs. templates

Parameterisation vs. inheritance:

- two techniques to implement polymorphic behaviour
- two ways to define new types based on existent ones
- *run-time polymorphism* – usage of virtual functions and inheritance
- *compile-time (parametric) polymorphism* – usage of templates

When choose one vs. another approach?

If hierarchical relationship exist between concepts, then use run-time polymorphism (inheritance); otherwise use templates.

If run-time efficiency is a must, use templates.

Member templates (I)

A class (or a template class) can have members that are themselves templates.

```
class X { // This is an example of a class with template function member.
public:
     template<class T> f(vector<T> array);

     int f1(int a, int b, int c);
} ;
```

Illegal to have virtual and template:

```
                        template<class T> virtual bool intersect();
```

Member templates (II)

A template class that has members that are themselves templates.

```
template<class S> class complex {
private:
      S re, im;
public:
     template<class T> complex(const complex<T>& c)
         : re(c.re), im(c.im) { }
};

int main() {
     complex<float> cf;
     complex<double> cd = cf; // use float -> double conv.
     class Quad { } ; // without conversion to int

     complex<Quad> cq;
     complex<int> ci = cq; // error: no conv. Quad -> int exist

     return 0;
}
```

Use **checked_cast** operator to avoid unreasonable conversions.

Invariance

Let’s say that `Shape` is a base class of `Circle`.

Treating `vector<Circle*>` as `vector<Shape*>` is an error! We say that template classes are **invariant** in parameter type.

```
class Circle : public Shape { };

void f(vector<Shape*>& s) {
    s.add(new Triangle());
}

void g(vector<Circle*>& c) {
    f(c); // error: no vector<Circle*> to vector<Shape*> conversion
}
```

Any two unrelated user-defined types in C++ can't be assigned to each-other by default. You have to provide a copy-constructor or an assignment operator.

Template conversion

Use member templates to specify relationships between templates when necessary.

```
template<class T> class Ptr {
     T* p;

public:
     Ptr(T* a) : p(a) {
     }

     // convert Ptr<T> to Ptr<T2>
     template<class T2> operator Ptr<T2>();
};

template<class T> template<class T2> Ptr<T>::operator Ptr<T2>() {
     return Ptr<T2>(p); // T2* has to be a T*
}

void f(Ptr<Circle> pc) {
     Ptr<Shape> ps = pc; // ok: Circle* is a Shape*
     Ptr<Circle> p2 = ps; // error: Shape* is not a Circle*
}
```

Further Reading

[Stroustrup, 1997] Bjarne Stroustrup – The C++ Programming Language 3rd Edition, Addison Wesley, 1997 [Chapter 13]

[Stroustrup, 2013] Bjarne Stroustrup – The C++ Programming Language 4th Edition, Addison Wesley, 2013 [Chapter 23]

Unit 2

Agenda

Introduction to C++ Standard Template Library (STL)

- Introduction
- Containers
- Iterators and allocators
- Algorithms and function objects
- Strings
- Streams
- Numerics

Library Organization

DEFINITION [Library] In computer science, a library is a collection of subroutines or classes used to develop software. Libraries contain code and data that provide services to independent programs.

The main criterion for including a class in the library was that it would somehow be *used by almost every C++ programmer* (both novices and experts), that it *could be provided in a general form* that did not add significant overhead compared to a simpler version of the same facility, and that simple uses should be *easy to learn*.

Essentially, the C++ standard library provides the most common fundamental data structures together with the fundamental algorithms used on them.

Organization of the library:

- Containers
- Algorithms and Function Objects
- Iterators and Allocators
- Strings
- Streams
- Numerics

What is provided by the C++ standard library?

The C++ standard library:

- provides support for language features (memory management, RTTI)
- supplies info about implementation-dependent aspects (largest float value etc)
- supplies functions that cannot be implemented optimally in the language itself for every system (sqrt, memmove etc.)
- provides strings and I/O streams (with internationalization and localization support)
- provides a framework of containers (vectors, maps etc.) and generic algorithms (traversals, sorting, merging etc.)
- supports numerical computations (complex numbers, BLAS=Basic Linear Algebra Subprograms etc.)
- provides a framework for extending the provided facilities
- provides a common foundation for other libraries

Framework of containers, algorithms and iterators is commonly referred as STL = Standard Template Library [Stepanov, 1994]

Containers (I)

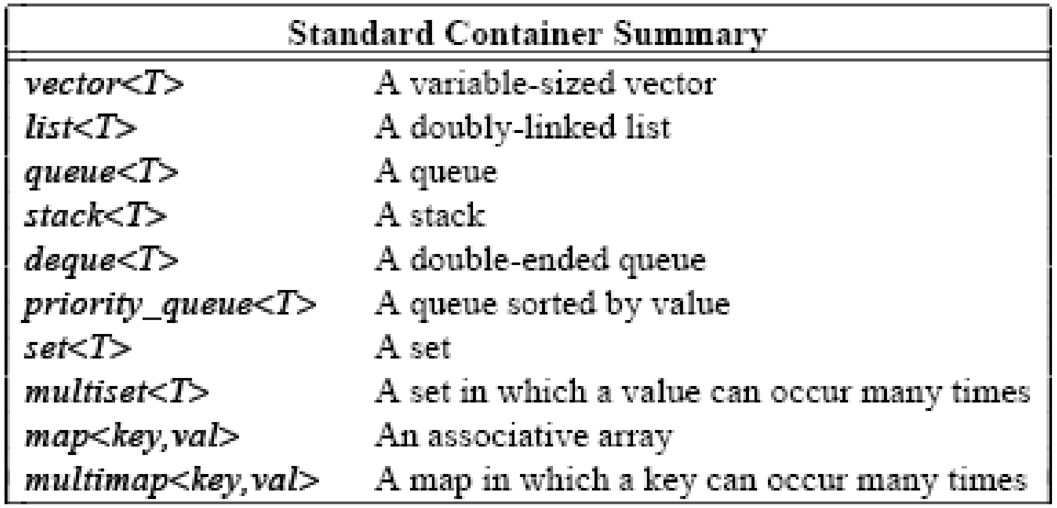
DEFINITON [Container] A container is an object that holds other objects.

Examples: list, vectors, stacks etc.

Advantages of containers:

- simple and efficient
- each container provides a set of common operations; they can be used interchangeably wherever reasonable
- standard iterators are available for each container
- type-safe and homogeneous
- are non-intrusive (i.e. an object doesn't need to have a special base class to be a member of the container)
- each container takes an argument called allocator, which can be used as a handle for implementing services for every container (persistence, I/O)

Containers (II)



Standard Container Summary	
vector<T>	A variable-sized vector
list<T>	A doubly-linked list
queue<T>	A queue
stack<T>	A stack
deque<T>	A double-ended queue
priority_queue<T>	A queue sorted by value
set<T>	A set
multiset<T>	A set in which a value can occur many times
map<key,val>	An associative array
multimap<key,val>	A map in which a key can occur many times

Basic containers: vector, list, deque.

Other containers: stack, queue, priority queue (implemented using basic containers).

Associative containers: map, multimap

‘Almost containers’: bitset, String, valarray

```
vector<Point> citites;

void addPoints(Point sentinel) {
     Point point;
     while(cin>>point) {
          if(point==sentinel) return;

          cities.push_back(point); // add the point to the end of the vector
     }
}
```

Allocators

DEFINITION [Allocator] The allocator is an abstraction that insulates the container implementation from memory access/management details.

Key concepts of an allocator:

- provides standard ways to allocate/deallocate memory
- provides standard names of types used as pointers or references

Use allocators to offer garbage-collected memory management.

Standard allocator is implemented in *class<template T> class allocator { }* (included in <memory>) and it uses operator new to allocate memory.

User-defined allocators usually override *allocate* and *deallocate* member functions.

```
template<class T> class  my_alloc { /* . . . */} ;

vector<int, my_alloc> v;

vector<int> v2 = v; // ERROR: different allocators
```

Iterators

DEFINITION [Iterator] Iterators are abstractions that provide an abstract view of data so that the writer of an algorithm doesn't have to know the representation details of the data structure.

DEFINITION [Sequence] A sequence is a collection of elements that can be traversed from the "beginning" to the "end" by using "next-element" operation.

Key concepts of iterator:

- the element the iterator is pointing to: operator * or ->
- the next element in sequence: operator ++
- equality: operator ==

Examples: int* is an iterator for int[], list<int>::iterator is an iterator for list<int>

- Declared in *std* namespace; include <iterator>
- const or modifiable
- Operations: read, write, access, iteration, comparison
- Iterator types: input, output, forward, bidirectional, random access
- Doesn't check the range of the container

Vector

```
template<class T, class A = allocator<T>> class std::vector {
     // types
     typedef T value_type;
     typedef … iterator; // T*
     // iterators
     iterator begin();
     reverse_iterator rbegin();
     // element access
     reference operator[] (size_type n);
     // constructors
     explicit vector(size_type n, const T& val=T(), const A&=A());
     // operations
     void push_back(const T& x);
     // size and capacity
     void resize(size_type sz, T val=T());
};
template<class T, class A> bool std::operator==(const vector<T, A>& v1,
                                                const vector<T, A>& v2);
```

- Allocator – supplies functions that a container uses to allocate/deallocate memory for its elements
- Iterator – pointers to elements of a container used to navigate the container
- Element access – operator [], at (checked access), front, back
- Constructors – constructors, copy-constructors, destructor
- Operations – push back, pop back, insert, erase, clear, push front, pop front, operator =
- Size and capacity – size, empty, resize,capacity, reserve
- Others – swap two vectors, get_allocator
- Helper functions – operator == and <
- There is a specialization for vector<bool>

Vector - Example

```
#include <vector>

template<class C> typename C::value_type sum(const C& container) {
     typename C::value_type s = 0;
     typename C::const_iterator p = container.begin();
     while(p!=container.end()) {
          s += *p;
          p++;
     }
     return s;
};

int main() {
     vector<int> v{32};

     for(int i=0; i<32; i++)
          v.push_back(i);

     int x = sum(v);

     cout << “Element 3-th” << v[3] << “ of total “ << v.size();

     v.clear();

     return  0;
}
```

Iterators – special types

Reverse iterator - A reverse iterator provides functions to iterate through elements in reverse order, from “end” to “beginning”.

- Implemented in reverse_iterator class;
- Containers provide **rbegin** and **rend** member functions to create reverse iterators.

Checked iterators - provide range-checked access to sequence’s elements

- Checked_iter – the interface for they

Stream iterators – present I/O streams as collections

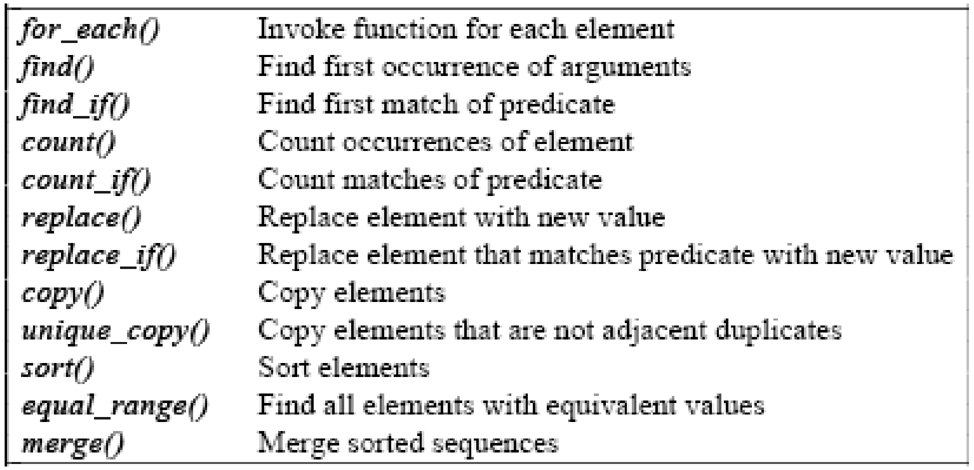
- Available stream iterators: ostream_iterator, istream_iterator, ostreambuf_iterator, istreambuf_iterator.
- See example below:

```
void f() {
     ostream_iterator<int> os{cout};
     *os = 7; // output 7, cout << 7
     ++os; // get ready for next output
     *os = 70; // output 70, cout << 70
}
```

Algorithms

- There are 60 algorithms in standard library;
- Defined in *std* namespace; include <algorithm>
- Can be applied to standard containers, strings and built-in arrays
- Expressed as template function

Standard Algorithms (a selection):



for_each()	Invoke function for each element
find()	Find first occurrence of arguments
find_if()	Find first match of predicate
count()	Count occurrences of element
count_if()	Count matches of predicate
replace()	Replace element with new value
replace_if()	Replace element that matches predicate with new value
copy()	Copy elements
unique_copy()	Copy elements that are not adjacent duplicates
sort()	Sort elements
equal_range()	Find all elements with equivalent values
merge()	Merge sorted sequences

Classification:
- non modifying sequence
- modifying sequence
- sorted sequence
- others: set, heap, minimum, maximum, permutations

```
void f(list<string>& ls) {
     list<string>::const_iterator p = find(ls.begin(), ls.end(), “Timisoara”);
     if(p==ls.end()) cout << “Didn’t find Timisoara”;
}
```

Function objects (I)

DEFINITION [Function object] An instance of a class with an application operator overloaded is called function-like object / functor / function object.

Advantages:

- support better complex operations than ordinary functions
- easier to inline

Mechanism to customize standard algorithms behaviour

In std namespace; include <functional>

```
template<class T> class SumMe {
public:
     SumMe(T i=0) : sum(i) { }
     void operator() (T x) { sum += x; }
     T result() const { return sum; }
private:
     T sum;
};

void f(vector<int> v) {
     SumMe<int> s;
     for_each(v.begin(), v.end(), s);
     cout << "Sum is " << s.result();
}
```

Function objects (II)

Function object base – unary_function and binary_function

Predicate – is a function object that returns a bool value;
Ex: `bool operator() (...);`

Examples of predicates supplied in standard library: equal_to, greater, less etc.

Arithmetic function objects: plus, minus, etc.

```
class Club {
public:
     string name;
      // other members
};

class Club_eq : public unary_function<Club, bool> {
     string s;
public:
     explicit Club_eq(const string& ss) : s(ss) { }
     bool operator() (const Club& c) const {
           return s==c.name;
      }
};

void f(list<Club> lc) {
     list<Club>::iterator p = find_if(lc.begin(), lc.end(), Club_eq("Bayern"));
     // if p==lc.end() -> club Bayern not found
     // if p!=lc.end() -> club Bayern found
}
```

Strings

DEFINITON [String] A string is a sequence of characters.

Offered by class **std::string** in the header <string>.

Support common string operations such as: concatenation, insertion, subscripting, assignment, comparison, appending, searching for substrings, extract substrings.

Supports any character set (using char_traits template).

```
string foo() {
     string s1 = “First string”;
     string s2 = s1, s3{s1, 6, 3};
     wstring ws{s1.begin(), s1.end()}; // string of wchar_t
     s3 = s2;
     s3[0] = ‘A’;
     const char* p = s3.data(); // conversion to C-style string
     delete p; // ERROR: the array is owned by string object
     if(s1==s2) cout << “Strings have same content”;
     s1 += “ and some more.”;
     s2.insert(5, “smth”);
     string::size_type i1 = s1.find(“string”); // i1=6
     string::size_type i2 = s1.find_first_of(“string”); // i2=3
     s1.replace(s1.find(“string”), 3, “STR”);
     cout << s.substr(0, 5);
     cin >> s3;
     return s1 + ‘ ‘ + s2; // concatenation
}
```

Streams

The challenge is to design an easy, convenient, safe to use, efficient, flexible I/O system able to handle user-defined types (on top of handling built-in types). The stream I/O facilities is the result of the effort of meet this challenge.

- standard input/output streams: cin, cout, cerr.
- overloaded operators: <<, >>
- support a variety of formatting and buffering facilities
- support any character set (ostream, wostream)

```
#include <string> // make standard strings available
#include <iostream> // make standard I/O available

int main() {
     using namespace std ;
     string name;
     cout << “Enter your name:"; // prompt the user
     cin >> name; // read a name
     cout << "Hello, " << name << ´\ n´;
     cout.setf(ios_base::hex, ios_base::basefield);
     cout << 123; // outputed in hex format
     cout.width(4);
     cout << 12; // “  12” (preceded by two blanks)
     ofstream ofs;
     ofs.seekp(10);
     ofs<<‘#’;
```

```
     ofs.seekp(-1, ios_base::cur);
     ofs << ‘*’;
     return 0;
}
```

Numerics - complex

`complex` numbers: supports a family of complex numbers, using different scalar to represent real/imaginary part.

```
template<class scalar> class complex {
public:
     complex(scalar re, scalar im);
     // ….
}
```

Arithmetic operations and common mathematical functions are supported

```
void f(complex<float> fl, complex<double> db) {
     complex<long double> ld = fl+sqrt(db);
     db += fl*3;
     fl = pow(1/fl, 2);
}
```

Numerics - valarray

Vector arithmetic is provided in **valarray** class

Different from `vector` class because they have different aims:

- `vector` – general, flexible, efficient mechanism for holding values;
- `valarray` – added math operations, less general and optimized for numerical computation)

```
template<class T> class valarray {
public:
     T& operator[](size_t); // size_t = unsigned int
     //. . .
}
```

Supports BLAS-style and generalized slicing.

Matrix can be constructed from `valarray`.

```
void f(const valarray<double>& a1, const valarray<double>& a2) {
     valarray<double> a=a1*3 + a2/a1, aa;
     a += a2*6.7;
     aa = abs(a1);
     double d = a2[6];
}
```

Numerics - valarray

Vector arithmetic is provided in **valarray** class

Different from `vector` class because they have different aims:

- `vector` – general, flexible, efficient mechanism for holding values;
- `valarray` – added math operations, less general and optimized for numerical computation)

```
template<class T> class valarray {
public:
     T& operator[](size_t); // size_t = unsigned int
     //. . .
}
```

Supports BLAS-style and generalized slicing.

Matrix can be constructed from `valarray`.

```
void f(const valarray<double>& a1, const valarray<double>& a2) {
     valarray<double> a=a1*3 + a2/a1, aa;
     a += a2*6.7;
     aa = abs(a1);
     double d = a2[6];
}
```

Covariance

Reminder: C++ template are invariant!

There are examples of covariance in C++ STL. But, what covariance means?

With covariance, the templated type maintains the relationship between argument types; if argument types are unrelated, the templated types shall be unrelated. Hence, if Derived is a sub-type of Base then TEMPLATE<Derived> shall be sub-type of TEMPLATE<Base>, i.e., any place where TEMPLATE<Base> is expected, TEMPLATE<Derived> can be substituted and everything will work just fine. The other way around is not allowed.

Example: std::shared_ptr, std::unique_ptr, std::pair, std::tuple, std::function

Covariance. Examples

```
struct Vehicle {};
struct Car : Vehicle {};

// Invariance
std::vector<Vehicle *> vehicles;
std::vector<Car *> cars;

vehicles = cars; // Invariance: does not compile

// Covariance
std::shared_ptr<Vehicle> shptr_vehicle;
std::shared_ptr<Car> shptr_car;
shptr_vehicle = shptr_car; // Covariance: works
shptr_car = shptr_vehicle' // Covariance: does not work

std::unique_ptr<Vehicle> unique_vehicle;
std::unique_ptr<Car> unique_car;
unique_vehicle = std::move(unique_car); // Covariance: works
unique_car = std::move(unique_vehicle); // Covariance: does not work
```

Covariance. How to

```
// SmartPointer class is covariant
template <class T> class SmartPointer
{
public:
    template <typename U>
    SmartPointer(U* p) : p_(p) {}

    template <typename U>
    SmartPointer(const SmartPointer<U>& sp,
                 typename std::enable_if<std::is_convertible<U*, T*>::value, void>::type * = 0
      : p_(sp.p_) {}

    template <typename U>
    typename std::enable_if<std::is_convertible<U*, T*>::value, SmartPointer<T>&>::type
    operator=(const SmartPointer<U> & sp)
    {
        p_ = sp.p_;
        return *this;
    }

   T* p_;
};
```

Contravariance

Reminder: C++ does not support contravariance!

Covariant Return Types and Contravariant Argument Types in std::function

```
template <class T>
using Sink = std::function<void (T *)>;

// Sink is contravariant in its argument type.
Sink<Vehicle> vehicle_sink =
    [](Vehicle *){ std::cout << "Got some vehicle\n"; };
Sink<Car> car_sink = vehicle_sink; // Works!
car_sink(new Car());
vehicle_sink = car_sink; // Fails to compile


// Sink is covariant in return type
std::function<Car * (Metal *)> car_factory =
    [](Metal *){ std::cout << "Got some Metal\n"; return new Car(); };

std::function<Vehicle * (Iron *)> vehicle_factory = car_factory;

Vehicle * some_vehicle = vehicle_factory(new Iron()); // Works
```

Variance chart in STL

Type	Covariant	Contravariant
STL containers	No	No
std::initializer_list<T *>	No	No
std::future<T>	No	No
std::shared_ptr<T>	Yes	No
std::unique_ptr<T>	Yes	No
std::pair<T *, U *>	Yes	No
std::tuple<T *, U *>	Yes	No
std::atomic<T *>	Yes	No
std::function<R *(T *)>	Yes (in return)	Yes (in arguments)

Source: http://cpptruths.blogspot.com/2015/11/covariance-and-contravariance-in-c.html

Further Reading

[Stroustrup, 1997] Bjarne Stroustrup – The C++ Programming Language 3rd Edition, Addison Wesley, 1997 [Chapter 16, 17, 18, 19, 20, 21, 22]

[Stroustrup, 2013] Bjarne Stroustrup – The C++ Programming Language 4th Edition, Addison Wesley, 2013 [Chapter 30]