Containers

STL

- Containers
  - Generic data structures
    - Based on arrays, linked lists, trees, or hash-tables
  - Store objects of given type (template parameter)

  - The container takes care of allocation/deallocation of the stored objects
    - All objects must be of the same type (defined by the template parameter)
      - Containers can not directly store polymorphic objects with inheritance
    - New objects are inserted by copying/moving/constructing in place
      - Containers can not hold objects created outside them

  - Inserting/removing objects: Member functions of the container
  - Reading/modifying objects: Iterators

STL

- Sequential containers
  - New objects are inserted in specified location

  - array< T, N> - fixed-size array (no insertion/removal)
  - vector< T> - array, fast insertion/removal at the back end
    - stack< T> - insertion/removal only at the top (back end)
    - priority_queue< T> - priority queue (heap implemented in vector)
  - basic_string< T> - like a vector, convertible to const char *
    - string = basic_string< char>
    - u32string = basic_string< char32_t>
  - deque< T> - fast insertion/removal at both ends
    - queue< T> - FIFO (insert to back, remove from front)
  - forward_list< T> - linked list
  - list< T> - doubly-linked list

- Associative containers
  - New objects are inserted at a position defined by their properties
    - sets: type T must define ordering relation or hash function
    - maps: stored objects are of type pair< const K, T>
      - type K must define ordering or hash
    - multi-: multiple objects with the same (equivalent) key value may be inserted
  - Ordered (implemented usually by red-black trees)
    - set<T>
    - multiset<T>
    - map<K,T>
    - multimap<K,T>
  - Hashed
    - unordered_set<T>
    - unordered_multiset<T>
    - unordered_map<K,T>
    - unordered_multimap<K,T>

STL - Ordered Containers

- Ordered containers require ordering relation on the key type
  - Only < is used (no need to define >, <=, >=, ==, !=)
  - In simplest cases, the type has a built-in ordering

```
std::map< std::string, my_value> my_map;
```

  - If not built-in, ordering may be defined using a global function

```
bool operator<( const my_key & a, const my_key & b) { return /*...*/; }
std::map< my_key, my_value> mapa;
```

  - If global definition is not appropriate, ordering may be defined using a **functor**

```
struct my_functor {
  bool operator()( const my_key & a, const my_key & b) const { return /*...*/;
}
};
std::map< my_key, my_value, my_functor> my_map;
```

  - If the ordering has run-time parameters, the functor will carry them

```
struct my_functor { my_functor( bool a); /*...*/ bool ascending; };
std::map< my_key, my_value, my_functor> my_map( my_functor( true));
```

STL - Unordered containers

- Hashed containers require two functors: hash function and equality comparison

```
struct my_hash {
  std::size_t operator()( const my_key & a) const { /*...*/ }
};
struct my_equal { public:
  bool operator()( const my_key & a, const my_key & b) const { /*return a == b;*/ }
};
std::unordered_map< my_key, my_value, my_hash, my_equal> my_map;
```

- If not explicitly defined by container template parameters, hashed containers try to use generic functors defined in the library
    - std::hash< K>
    - std::equal_to< K>
  - Defined for numeric types, strings, and some other library types

```
std::unordered_map< std::string, my_value> my_map;
```

STL – Iterators

- Each container defines two member types: iterator and const_iterator

```
using my_container = std::map< my_key, my_value>;
using my_iterator = my_container::iterator;
using my_const_iterator = my_container::const_iterator;
```

- Iterators act like pointers to objects inside the container
  - objects are accessed using operators *, ->
  - const_iterator does not allow modification of the objects

- An iterator may point
  - to an object inside the container
  - to an imaginary position behind the last object: end()

STL – Iterators

```
void example( my_container & c1, const my_container & c2)
{
```

- Every container defines functions to access both ends of the container
  - begin(), cbegin() - the first object (same as end() if the container is empty)
  - end(), cend() - the imaginary position behind the last object

```
  auto i1 = begin( c1);       // also c1.begin()
```

  - c*() always returns const_iterator

```
  auto i2 = cbegin( c1);      // also c1.cbegin()
  auto i3 = cbegin( c2);      // also c2.cbegin(), begin( c2), c2.begin()
```

- Associative containers allow searching
  - find(k) - first object equal (i.e. not less and not greater) to k, end() if not found
  - lower_bound(k) - first object not less than k, end() if no such object
  - upper_bound(k) - first object greater than k, end() if no such object

```
  my_key k = /*...*/;
  auto i4 = c1.find( k);      // my_container::iterator
  auto i5 = c2.find( k);      // my_container::const_iterator
```

- Iterators may be shifted to neighbors in the container
  - all container iterators allow shifting to the right and equality comparison

```
  for ( auto i6 = c1.begin(); i6 != c1.end(); ++ i6 ) { /*...*/ }
```

  - **bidirectional** iterators (all except forward_list and unordered_*) allow shifting to the left

```
  -- i1;
```

  - **random access** iterators (vector, string, deque) allow addition/subtraction of integers, difference and comparison

```
  auto delta = i4 - c1.begin();  // number of objects to the left of i4;
                                 // my_container::difference_type === std::ptrdiff_t
  auto i7 = c1.end() - delta;    // locate the same distance from the opposite end;
                                 // my_container::iterator
  if ( i4 < i7 )
    auto v = i4[ delta].second;  // same as (*(i4 + delta)).second, (i4 + delta)->second
}
```

STL – Iterators

- Caution:
  - Shifting an iterator before begin() or after end() is **illegal**

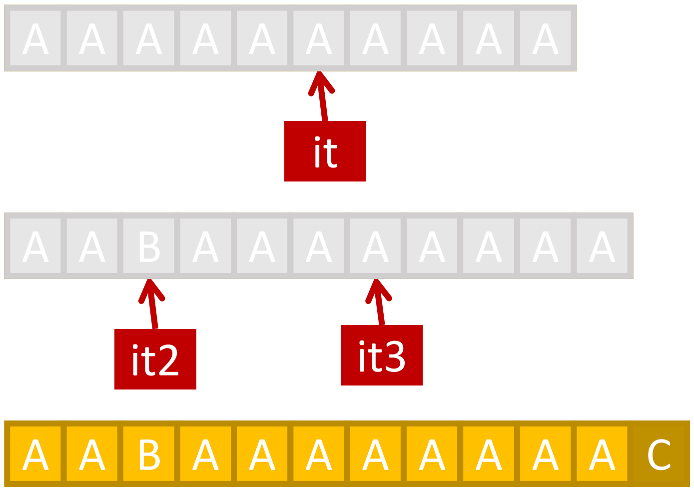
```
for (auto it=c1.end()-1; it>=c1.begin(); --it) // ERROR: underruns begin()
for (auto it=c1.rbegin(); it!=c1.rend(); ++it) // CORRECT: reverse iterator
```

  - Comparing iterators associated to different (instances of) containers is **illegal**

```
if ( c1.begin() < c2.begin() )  // ILLEGAL
```

  - Insertion/removal of objects in vector/basic_string/deque **invalidate** all associated iterators (except for some cases explicitly mentioned in the documentation)
    - The only valid iterator is the one returned from insert/erase

```
std::vector< std::string> c( 10, "A");
auto it = c.begin() + 5;  // the sixth A
std::cout << * it;
auto it2 = c.insert(c.begin() + 2, "B");
std::cout << * it;      // always ILLEGAL
                        // may CRASH if insert needed to reallocate
it3 = it2 + 4;          // the sixth A
c.push_back( "C");
std::cout << * it3;     // may CRASH
```



STL – Insertion/deletion

- Containers may be filled immediately upon construction
  - using n copies of the same object

```
std::vector< std::string> c1( 10, "dummy");
```

  - or by copying from another container

```
std::vector< std::string> c2( c1.begin() + 2, c1.end() - 2);
```

- Expanding containers - insertion
  - insert - copy or move an object into container
  - emplace - construct a new object (with given parameters) inside container
- Sequential containers
  - position specified explicitly by an iterator
    - new object(s) will be inserted before this position

```
c1.insert( c1.begin(), "front");
c1.insert( c1.begin() + 5, "middle");
c1.insert( c1.end(), "back");    // same as c1.push_back( "back");
```

STL – insertion/deletion

- insert by copy
  - slow if copy is expensive

```
std::vector< std::vector< int>> c3;
```

  - not applicable if copy is prohibited

```
std::vector< std::unique_ptr< T>> c4;
```

- insert by move
  - explicitly using std::move

```
auto p = std::make_unique< T>(/*…*/);
c4.push_back( std::move( p));
```

  - implicitly when argument is rvalue (temporal object)

```
c3.insert( begin( c3), std::vector< int>( 100, 0));
```

- emplace
  - constructs a new element from given arguments

```
c3.emplace( begin( c3), 100, 0);
```

STL – insertion/deletion

- Shrinking containers - erase/pop
  - single object

```
my_iterator it = /*...*/;
c1.erase( it);
c2.erase( c2.end() - 1);  // same as c2.pop_back();
```

  - range of objects

```
my_iterator it1 = /*...*/, it2 = /*...*/;
c1.erase( it1, it2);
c2.erase( c2.begin(), c2.end()); // same as c2.clear();
```

  - by key (associative containers only)

```
my_key k = /*...*/;
c3.erase( k);
```

Range-for loop

```
for ( type variable : range )
   statement;
```

- range is anything that has begin() and end()

- most often used with universal reference and a container:

```
for ( auto && variable : container )
   statement;
```

- may be used to modify the contents of the container by modifying the variable

- is by definition equivalent to

```
{
  auto && R = range;
  auto B = begin(R);   // or R.begin() if it exists
  auto E = end(R);     // or R.end() if it exists
  for (; B != E; ++ B)
  { type variable = * B;
    statement;
  }
}
```

Algoritmy

Algoritmy

- Sada generických funkcí pro práci s kontejnery
  - cca 90 funkcí od triviálních po sort, make_heap, set_intersection, ...

`#include <algorithm>`

  - Kontejnery se zpřístupňují nepřímo - pomocí iterátorů
    - Obvykle pomocí dvojice iterátorů - polouzavřený interval
    - Lze pracovat s výřezem kontejneru
    - Je možné použít cokoliv, co se chová jako iterátor požadované kategorie
  - Některé algoritmy kontejner pouze čtou
    - Typicky vracejí iterátor
    - Např. hledání v setříděných sekvenčních kontejnerech
  - Většina algoritmů modifikuje objekty v kontejneru
    - Kopírování, přesun (pomocí std::move), výměna (pomocí std::swap)
    - Aplikace uživatelem definované akce (funktor)
  - Iterátory neumožňují přidávání/odebírání objektů v kontejneru
    - Pro "nové" prvky musí být předem připraveno místo
    - Odebrání nepotřebných prvků musí provést uživatel dodatečně

Algoritmy

- Iterátory neumožňují přidávání/odebírání objektů v kontejneru
  - Pro "nové" prvky musí být předem připraveno místo

```
my_container c2( c1.size(), 0);
std::copy( c1.begin(), c1.end(), c2.begin());
```

  - Tento příklad lze zapsat bez algoritmů jako

```
my_container c2( c1.begin(), c1.end());
```

  - Odebrání nepotřebných prvků musí provést uživatel dodatečně

```
auto my_predicate = /*...*/; // some condition

my_container c2( c1.size(), 0);  // max size
my_iterator it2 = std::copy_if( c1.begin(), c1.end(), c2.begin(),
my_predicate);
c2.erase( it2, c2.end());    // shrink to really required size

my_iterator it1 = std::remove_if( c1.begin(), c1.end(), my_predicate);
c1.erase( it1, c1.end());    // really remove unnecessary elements
```

STL – Algoritmy

- Falešné iterátory
  - Algoritmy dovedou pracovat s čímkoliv, co napodobuje iterátor
  - Požadavky algoritmu na iterátory definovány pomocí kategorií
    - Input, Output, Forward, Bidirectional, RandomAccess
    - [C++20] tyto kategorie jsou reprezentovány v jazyce jako *Concepts*
  - Každý iterátor musí prozradit svou kategorii a další vlastnosti
    - std::iterator_traits
    - Některé algoritmy se přizpůsobují kategorii svých parametrů (std::distance)

  - Insertery

```
my_container c2;    // empty
auto my_inserter = std::back_inserter( c2);
std::copy_if( c1.begin(), c1.end(), my_inserter, my_predicate);
```

  - Textový vstup/výstup

```
auto my_inserter2 = std::ostream_iterator< int>( std::cout, " ");
std::copy( c1.begin(), c1.end(), my_inserter2);
```

C++20 - ranges

- [C++20] – dvojice iterátorů nahrazena jedním *range*
  - *range* je cokoliv, co má begin() a end()
    - Kontejner je *range* – tento druh range je vlastníkem dat!
      - kopírování takového range znamená kopírování dat
    - *view range* vznikají jako odkazy na data – nevlastní data
      - view range lze kopírovat v konstantním čase
      - all_view(k) reprezentuje odkaz na všechny prvky kontejneru
      - iota_view(10,20) reprezentuje fiktivní kontejner s obsahem [10,11,...,19]
  - *range adaptor* umožňuje upravit range filtrováním nebo transformací hodnot
    - filter_view(range, pred) vrací range reprezentující pouze prvky splňující predikát
    - adaptéry je možné aplikovat i syntaxí převzatou z unixových rour:

```
range | filter_view(pred)
```

  - Range automaticky funguje v range-based for [C++11]
  - Stávající algoritmy dostanou další interface používající range
    - nové verze jsou v namespace **std::ranges**
    - nový systém stále není kompletní ani v [C++23]

  - *ranges* využívají *concepts*; ty jsou velkým rozšířením jazyka [C++20]