Pointer/reference conventions

Pointer/references

- C++ allows several ways of passing links to objects
    - smart pointers
    - C-like pointers
    - references
- Technically, all the forms allow almost everything
    - At least using dirty tricks to bypass language rules
    - Pointers require different syntax wrt. references
- By convention, the use of a specific form signalizes some intent
    - Conventions (and language rules) limits the way how the object is used
    - Conventions help to avoid "what-if" questions
        - What if someone destroys the object I am dealing with?
        - What if someone modifies the contents of the object unexpectedly?
        - ...

Passing a pointer/reference in C++ - conventions

	What the recipient may do?	For how long?	What the others will do meanwhile?
std::unique_ptr<T>	Modify the contents and destroy the object	As required	Nothing (usually)
std::shared_ptr<T>	Modify the contents	As required	Read/modify the contents
T *	Modify the contents	Until notified to stop/by agreement	Read/modify the contents
const T *	Read the contents	Until notified to stop/by agreement	Modify the contents
T &	Modify the contents	During a call/statement	Nothing (usually)
T &&	Steal the contents		Nothing
const T &	Read the contents	During a call/statement	Nothing (usually)

Multiple values in contiguous memory

Arrays and tuples

	Homogeneous (arrays)	Polymorphic (tuples)

Fixed size	// modern: container-style
	static const std::size_t n = 3;
	std::array< T, n> a;
	
	a[0] = /*...*/;
	a[1].f();
		// structure/class
		struct S { T1 x; T2 y; T3 z; };
		S a;
		
		a.x = /*...*/;
		a.y.f();

	// native arrays (avoid!)
	static const std::size_t n = 3;
	T a[n];
	
	a[0] = /*...*/;
	a[1].f();
		// for generic access
		std::tuple< T1, T2, T3> a;
		
		std::get< 0>(a) = /*...*/;
		std::get< 1>(a).f();

Variable size	std::size_t n = /*...*/;
	std::vector< T> a(n);
	
	a[0] = /*...*/;
	a[1].f();
		std::vector< std::unique_ptr< Tbase>> a;
		a.push_back(std::make_unique< T1>());
		a.push_back(std::make_unique< T2>());
		a.push_back(std::make_unique< T3>());
		
		a[1]->f();

Array and tuple layouts

```
std::array< T, 3>
```

or

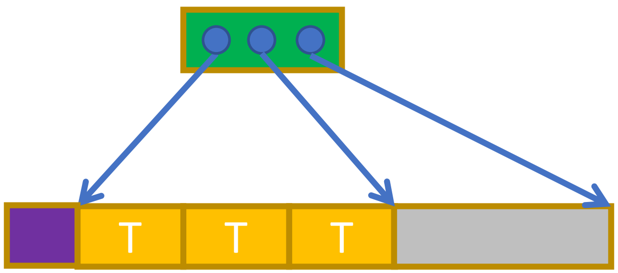
```
T[ 3]
```


```
struct { T1 x; T2 y; T3 z;}
```

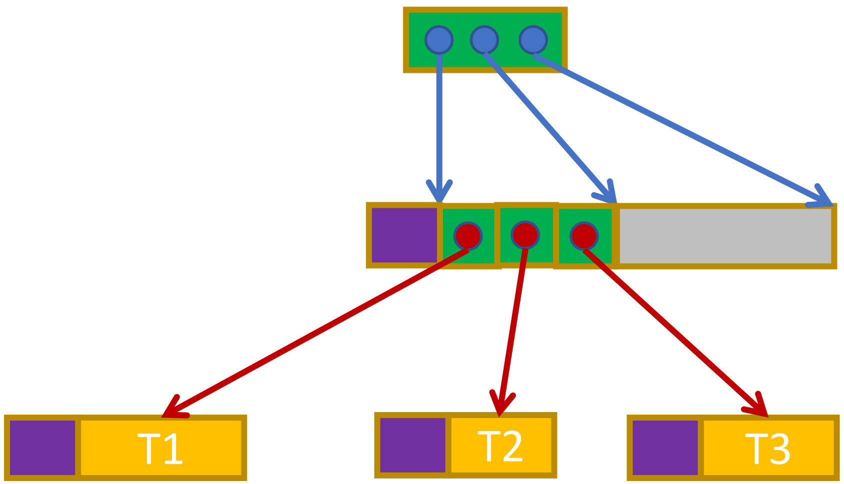
or

```
std::tuple< T1, T2, T3>
```


```
std::vector< T>
```




```
std::vector< std::unique_ptr<Tbase>>
```



<table>
<tr><th></th><th>number of elements</th><th>storage</th><th>ownership</th><th>move</th><th>copy</th></tr>
<tr><td>array<T,N></td><td>fixed N</td><td rowspan="2">inside</td><td rowspan="2">(unique)</td><td rowspan="2">by elements</td><td rowspan="2">by elements</td></tr>
<tr><td>optional<T></td><td rowspan="3">0/1</td></tr>
<tr><td>unique_ptr<T></td><td rowspan="2">individually allocated</td><td>unique</td><td rowspan="7">transfer of ownership</td><td>N.A.</td></tr>
<tr><td>shared_ptr<T></td><td>shared</td><td>sharing</td></tr>
<tr><td>unique_ptr<T[]></td><td rowspan="5">any</td><td rowspan="3">contiguous block</td><td>unique</td><td>N.A.</td></tr>
<tr><td>shared_ptr<T[]></td><td>shared</td><td>sharing</td></tr>
<tr><td>vector<T></td><td rowspan="3">unique</td><td rowspan="3">by elements</td></tr>
<tr><td>deque<T></td><td>several contiguous blocks</td></tr>
<tr><td>other containers</td><td>individually allocated</td></tr>
</table>

Smart pointers and containers

<table>
<tr><th></th><th>number of elements</th><th>storage</th><th>allocation (en masse)</th><th>insert/erase elements</th><th>random access</th></tr>
<tr><td>array<T,N></td><td>fixed, N</td><td rowspan="2">inside</td><td>(when constructed)</td><td>N.A.</td><td>[i]</td></tr>
<tr><td>optional<T></td><td rowspan="3">0/1</td><td>.emplace(...)</td><td rowspan="5">.reset()</td><td rowspan="3">N.A.</td></tr>
<tr><td>unique_ptr<T></td><td rowspan="2">individually allocated</td><td>= make_unique<T>(...)</td></tr>
<tr><td>shared_ptr<T></td><td>= make_shared<T>(...)</td></tr>
<tr><td>unique_ptr<T[]></td><td rowspan="5">any</td><td rowspan="3">contiguous block</td><td>= make_unique<T[]>(n)</td><td rowspan="4">[i]</td></tr>
<tr><td>shared_ptr<T[]></td><td>= make_shared<T[]>(n)</td></tr>
<tr><td>vector<T></td><td rowspan="3">vector<T>(n)
or
.resize(n)</td><td rowspan="2">may move elements</td></tr>
<tr><td>deque<T></td><td>several contiguous blocks</td></tr>
<tr><td>other containers</td><td>individually allocated</td><td>elements never move</td><td>no</td></tr>
</table>

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

STL

- 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> my_map;
```

  - 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**

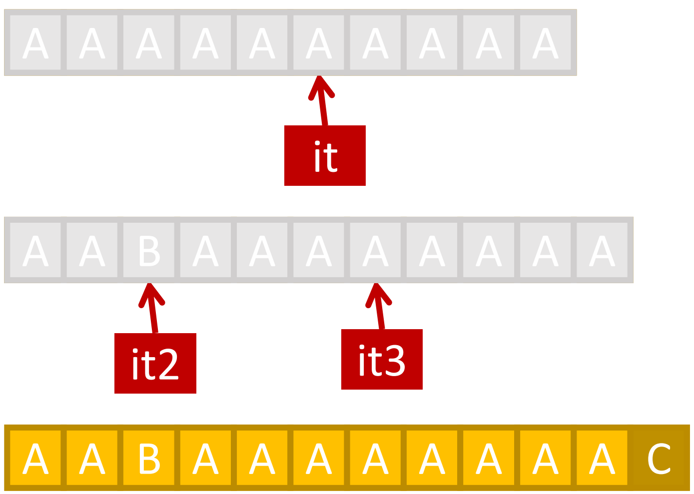
```
for (auto it=c1.end()-1; it>=c1.begin(); --it) // ERROR: underruns begin()
for (auto it=c1.rbegin(); it!=c1.rend(); ++it) // CORRECT: reverse iterator
```

  - 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;
  }
}
```