# The Standard Template Library (STL)

- provides
  - containers: to manage collections of objects
  - iterators: represent a position in a collection
    - use `operator++()` to go to the next position
    - use `operator*()` to access the element at the position
  - algorithms: to process the elements in a collection; e.g searching, sorting
  - function objects: to customize algorithms
- separates data from operation
  - containers to manage data
  - algorithms to provide operations on data
  - iterators to act as the glue between the two
- all components are templates
- an example of **generic programming**
- 2 types of containers
  - sequence containers: ordered collections; `vector`, `deque` & `list`
  - associative containers: sorted collections (ordered by a sorting criterion); `set`, `multiset`, `map` & `multimap`

- STL containers provide *value semantics*
  - they create internal copies of their elements
  - hence they contain the values of the objects inserted rather than the objects themselves

- basic requirements for container elements
  - must be copyable (by a copy ctor); the copy must be equivalent to the original
  - must be assignable (by the assignment operator)
  - must be destructible (by the dtor)

- the STL was designed for performance; it performs almost no error checking
  - any use that violates preconditions results in undefined behaviour
  - as example, user needs to make sure that indexes, iterators & ranges are valid

# Vectors

- provide random access to a sequence of varying length with constant time insertion & deletion at the end

- like dynamic arrays

```
#include <iostream>
#include <vector>
using namespace std;
void print(const vector<int>& v);

int main() {
  vector<int>  vec;
  vec.reserve(20);  // recommended to minimize realloc

  for (int i = 0; i < 5; i++)  // append 5 ints
    vec.push_back(i);
  print(vec);                  // 0 1 2 3 4

  cout << vec.front() << ' '
       << vec.back() << endl;  // 0 4
  vec.pop_back();              // delete last element
  print(vec);                  // 0 1 2 3

  // use iterator to traverse vector
  vector<int>::iterator  it;
  for (it = vec.begin(); it != vec.end(); ++it)
    *it *= 2;                  // double each element

  int  a[] = { 4, 5, 6, 7, 8 };
  vec.insert(vec.begin(), a, a+4); // range [a, a+4)
  print(vec);                  // 4 5 6 7 0 2 4 6
```

```
  it = vec.begin();
  vec.erase(it + 2, it + 5);   // 4 5 2 4 6
  print(vec);

  vec.assign(a+2, a+5);
  print(vec);                  // 6 7 8

  vector<int>  vec2(5, 1);     // 5 copies of 1
  print(vec2);                 // 1 1 1 1 1
}

void print(const vector<int>& v) {
  for (vector<int>::size_type i = 0; i < v.size(); i++)
    cout << v[i] << ' ';
  cout << endl;
}
```

- note that we don't really need to use an iterator because we have random access (using `[]`)
- the `begin()` member function returns an iterator to the first element of the container
- the `end()` member function returns an iterator to 1 past the last element of the container
- there is also a `const_iterator` for use with constant containers (e.g. when a container is passed as a constant reference to a function)
- there are also `reverse_iterator` & `const_reverse_iterator` (see `list` for an example)
- the iterators associated with a `vector` are models of `Random Access Iterator`: given such an iterator `it`, we can use, e.g., `++it`, `--it`, `it + 2`

- note that `pop_back()` returns nothing & there is no `pop_front()` method
- we can construct a vector from an array as follows:

```
int a[] = { 0, 1, 2, 3, 4};
vector<int>  vec(a, a + 5);
// vec now contains 0 1 2 3 4
```

  Actually, we can construct any standard container from a range specified by 2 iterators; note that the STL uses half-open ranges: `[a, a+5)` in the example above

- all standard sequence containers have `insert` & `assign` methods

```
void insert(iterator pos, InputIterator begin,
                          InputIterator end);
void assign(InputIterator begin, InputIterator end);
```

  where `begin` & `end` are iterators that specify a range & `pos` is an iterator specifying the position to insert.

- all standard sequence containers have `erase` methods

```
iterator erase(iterator it);
iterator erase(iterator begin, iterator end);
```

  where `it` specifies the element to erase & `begin` & `end` specify the range to erase; the methods return an iterator to the element immediately after the one(s) removed, or `c.end()` if no such element exists (where `c` is the container on which `erase` is invoked)

- avoid using `vector<bool>`

# Deques

- "double-ended" queues
- provide random access to a sequence of varying length with constant time insertion & deletion at both ends
- like dynamic arrays that can grow at both ends

```
#include <iostream>
#include <deque>
using namespace std;
void print(const deque<int>& d);

int main() {
  deque<int>  deq;

  for (int i = 0; i < 5; i++)
    if (i % 2 == 0)
      deq.push_front(i);  // insert at front
    else
      deq.push_back(i);   // insert at back

  print(deq);             // 4 2 0 1 3
  deq.pop_back();         // delete last element
  deq.pop_front();        // delete first element

  // using a const iterator to print the deque
  deque<int>::const_iterator  it;
  for (it = deq.begin(); it != deq.end(); ++it)
    cout << *it << ' ';
  cout << endl;
}
```

```
void print(const deque<int>& d) {
  for (deque<int>::size_type i = 0; i < d.size(); i++)
    cout << d[i] << ' ';
  cout << endl;
}
```

- there are also `reverse_iterator` & `const_reverse_iterator`
- the iterators associated with `deque` are also models of `Random Access Iterator`

## Lists

- no random access
- provide linear-time access to a sequence of varying length with constant-time insertion & deletion at any position
- like doubly-linked lists

```
#include <iostream>
#include <list>
using namespace std;
void print(const list<int>& lst);

int main() {
  list<int>  lst;

  for (int i = 0; i < 5; i++)
    if (i % 2 == 0)
      lst.push_front(i);  // insert at front
    else
      lst.push_back(i);   // insert at back
  print(lst);             // 4 2 0 1 3

  list<int>::iterator  it;
  for (it = lst.begin(); it != lst.end(); ++it)
    *it *= 2;
  print(lst); // 8 4 0 2 6

  it = lst.begin();
  ++it;
  lst.insert(it, 7);  // insert 7 before position 'it'
  print(lst);         // 8 7 4 0 2 6
```

```
  lst.remove(8);      // remove all occurrences of 8
  print(lst);         // 7 4 0 2 6

  it = lst.begin();
  ++it;
  lst.erase(it);  // remove element at position 'it'
  print(lst);     // 7 0 2 6

  // print in "reverse" order
  list<int>::reverse_iterator  rit;
  for (rit = lst.rbegin(); rit != lst.rend(); ++rit)
    cout << *rit << ' ';
  cout << endl;
}

void print(const list<int>& lst) {
  for (list<int>::const_iterator  it = lst.begin();
       it != lst.end(); ++it)
    cout << *it << ' ';
  cout << endl;
}
```

- note that `pop_back` & `pop_front` are also available
- there is also a `const_reverse_iterator`
- the iterators associated with `list` are models of `Bidirectional Iterator`