ALGORITHMS AND DATA STRUCTURES

# ITERATORS

# ITERATORS

- Consider some element in a range of elements
- An iterator is any object that, by pointing to such element, has the ability to iterate through the elements of that range
- It typically uses the operators: increment (++) and dereference (∗)
- Most obvious, but not simplest, form of iterators are *pointers*

http://www.cplusplus.com/reference/iterator/

# ITERATORS

- Similarities with pointers
  - iterators give us indirect access to an object
  - an iterator may be valid or invalid
  - can use an iterator to fetch an element
  - iterators can move from one element to another
  - can be dereferenced to obtain the element

# ITERATORS

- All of the container libraries include its own `iterators`
    - Are a data type defined inside each container class
- `iterators` have predefined `begin()` and `end()` methods
    - **`begin()`**: returns `iterator` to 1st element of the container
    - **`end()`**: returns `iterator` to “one past last” element of the container
- Dereferencing an invalid iterator or an off-the-end iterator has undefined behavior

# ITERATOR OPERATIONS

- Some basic operations are
  - `*iter`: Returns a reference to the element denoted by the iterator `iter`
  - `++iter`: Increments `iter` to refer to next element of the container
  - `--iter`: decrements `iter` to refer to previous element of the container

# ITERATOR OPERATIONS

- Some basic operations are
  - **`iter->mem`**: Dereferences iter and fetches the member named mem from the underlying element. Equivalent to `(*iter).mem`
  - **`iter1 OP iter2`**: Compares two iterators for equality **`==`** or inequality **`!=`**
    - Two iterators are equal if they denote the same element or are the off-the-end iterator for the same container

# EXAMPLE: STRING

- Boolean operations

```
string s("some string content");
if (s.begin() != s.end()) {
    string::iterator it = s.begin();
    *it = toupper(*it);
}
```

- Increment iterators

```
string s("some string content”);
string::iterator it;
for (it = s.begin(); it != s.end(); ++it) {
    *it = toupper(*it);
}
```

# ITERATOR TYPES

- We generally do not need to know the precise type of an `iterator`
- Instead, libraries that have iterators define types named `iterator` and `const_iterator` that represent actual iterator types
- Similar to a `const` pointer, a `const_iterator` can read but not write the element it denotes
- An `iterator` can both read and write

# ITERATOR TYPES

- Constant `iterator`: can only read not write elements

```
vector<int>::iterator it; // r&w vector<int> elements
string::iterator it2; // r&w chars in a string

vector<int>::const_iterator it3; // r not w elements
string::const_iterator it4; // r not w characters
```

  - `cbegin()` and `cend()` are analogues of `begin()` and `end()` for constant `iterators` (C++11)

- **Important**: do not use `iterators` in loops if you are adding elements to the container

# ITERATOR ARITHMETIC

- `iter + n`: moves `iter` n elements forward in the container
- `iter += n`: same as before with extra assignment to `iter`
- `iter1 - iter2`: subtracts two `iterators` referring to the same container

```
list<double> li;
// ...
// fill list somehow
// …
// calculate midpoint iterator
list<double>::iterator mid = li.begin() + li.size() / 2;
```

# MORE EXAMPLES

- `iterators` for `set` are naturally constant

```
set<int> iset = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
set<int>::iterator set_it = iset.begin();

if (set_it != iset.end()) {
    // error: keys in a set are read-only
    *set_it = 42;
    // ok: can read the key
    cout << *set_it << endl;
}
```

# MORE EXAMPLES

- With map

```
map<string, size_t> word_count;

// operate on word_count
map<string, size_t>::iterator map_it = word_count.begin();

while (map_it != word_count.end()) {
    cout << (*map_it).first << " "
         << (*map_it).second << endl;
    map_it++;
}
```

# TIPS

- If container is empty, `begin()` and `end()` are the same
- The `iterator` returned by `end()` cannot be incremented
- When only reading elements use `cbegin()` and `cend()`
- Do not use `iterators` in loops if you are adding elements to the container
- When using `iterators` with `map` and `set`, they give elements in ascending order