**Generic Programming with Templates**

- Template functions are applicable to any data type

```cpp
template <class T>
T maximum(T x, T y)
{
    if(x >= y) {
        return x;
    }
    else return y;
}
```

- We can also create template classes
- Template classes may have data members of an arbitrary type `T`
**CLASS TEMPLATES**

- Syntax (Vector.h):

```cpp
template <class T>
class Vector {
    public:
        Vector(int);
        T & operator[](int i); // return i’th element of type T
    void print();
    private:
        T * coords; // array of vector coordinates of type T
        int dimension;
};
```

- Now we can create a vector of **doubles**, or **ints**, or **booleans**, or any other type

Class name is preceded by a template preamble, **type T** is used throughout the class.
**CLASS TEMPLATES**

- **Syntax (Vector.cpp):**

  ```
  template <class T>
  Vector<T> :: Vector(int i) : dimension(i) {
      coords = new T [dimension];
  }

  template <class T>
  T & Vector<T>::operator[](int i) {
      if(i < dimension && i >= 0)
          return coords[i]; // return i’th element of type T
      else
          throw “Invalid index provided”;
  }
  ```

- Every function of a template class is a template function
CLASS TEMPLATES

Usage (main.cpp):

```cpp
#include "Vector.h"
#include "Vector.cpp"

int main() {

    Vector<double> doubleVec(3); // vector of doubles
    Vector<int> intVec(3);       // vector of ints
    Vector<bool> binaryVec(3);  // vector of booleans

    doubleVec[0] = 3.25;
    intVec[0] = 7;
    binaryVec[0] = 0;
}
```

Vector class will be compiled separately for every specific type
CLASS TEMPLATES

- Class templates allow us to create re-usable container classes

- **RPG example**: create a template class `Inventory<T>`
  - An *alchemist* would store potions in it: `Inventory<Potion>`
  - A *fighter* would store weapons in it: `Inventory<Weapon>`
  - A *wizard* would store spells in it: `Inventory<Spell>`

- You only have to write and debug your re-usable class once, and then use it everywhere!
Class templates and Inheritance

- Class templates can inherit from other class templates:

  ```cpp
template <class T>
class Inventory
  {
  };

template <class T>
class SearchableInventory : public Inventory<T>
  {
  };
```

- Derived classes must be template classes, too
- Parent class should be referred to as `Parent<T>`
- Inherited members should be accessed through `this` pointer
TEMPLATE SPECIALIZATION

Sometimes, a special kind of behaviour is required from specific types

• Suppose you write a container class;
• If characters are stored in it, you want to be able to convert them to uppercase and lowercase at will
• The rest of the functionality is shared, but only characters can be uppercase or lowercase
• Specialize the template class for char type!

How?
See example code
16.5
Introduction to the Standard Template Library
INTRODUCTION TO THE STANDARD TEMPLATE LIBRARY

- **Template functions** and **template classes** allow us to create highly re-usable code that is applicable to any data type.

- If template code is so re-usable, why didn’t C++ make some of it a part of the language?

- It did!

- **Standard Template Library (STL)**: a library containing templates for frequently used data structures and algorithms.

- Not supported by some older compilers.
STANDARD TEMPLATE LIBRARY

STL components:

- **containers**: classes that store data and impose some organization on it
- **iterators**: pointer-like objects; mechanisms for accessing elements in a container
- **algorithms**: generic operations applied to containers, such as sorting and searching
CONTAINERS

Two types of container classes in STL:

- **sequence containers**: organize and access data sequentially, as in an array. These include `vector`, `deque`, and `list`

- **associative containers**: store element in order, use keys to allow data elements to be quickly accessed. These include `set`, `multiset`, `map`, and `multimap`
**Sequence Containers**

- **vector**: An expandable array. Values are added at the end, adding values in the middle is less efficient

- **dequeue**: Like vector, but also allows to efficiently add values at the front of the expandable array

- **list**: A linked list (coming up next week). An expandable array that allows efficient insertion anywhere in the list
Including **<vector>**

```cpp
#include <string>
#include <vector>

int main() {

    vector<double> doubleVec; // vector of doubles
    vector<string> names;       // vector of strings

    doubleVec.push_back(2.5);   // infinitely expandable
    cout << doubleVec.size() << endl; // knows it's own size
    cout << doubleVec[0] << endl;   // use array notation
    doubleVec.pop_back();         // remove last element

    // ... more code ...
}
```
**Iterators**

- Given a data structure such as `vector` or `dequeue`, you would want to iterate through its elements, i.e. use the data stored there.
- Data structures are generic, but each is implemented differently.
- When you use it, you don’t want to worry about the way it is implemented – you just want to access the data.
- Solution: each STL data structure comes with a set of iterators.
- Iterators hide implementation details from you and provide a common public interface.
Iterators

Think of an iterator as a special kind of pointer:

```cpp
vector<int> vect;
for (int i = 0; i < 6; i++) vect.push_back(i);
```

```cpp
vector<int>::iterator it; // declare an iterator
it = vect.begin(); // assign it to the start of the vector
```

- `*it` – returns the value that the iterator is currently pointing to
- `it++`, `it--` – moves iterator one element forward or backward
- `it1 == it2, it1 != it2` – checks if two iterators point to the same element
- `it = vect.end();` – assign an element position to the iterator
Iterators

- Every container (vector, deque, map, etc.) provides the following functions to be used with iterators:
  - `begin()` – returns an iterator representing the beginning of the elements in the container
  - `end()` – returns an iterator representing the element just past the end of the elements
  - `cbegin()` – returns a const (read-only) iterator representing the beginning of the elements in the container
  - `cend()` – returns a const (read-only) iterator representing the element just past the end of the elements
Iterators

- All containers provide (at least) two types of iterators:
  - `container::iterator` provides a read/write iterator
  - `container::const_iterator` provides a read-only iterator
- Different types of iterators:
  - `forward` (use `++`, move in one direction only)
  - `bidirectional` (use `++` and `--`, move back and forth)
  - `random-access` (move back, forth, or jump to the middle)
  - `input` (can be used with `cin` and `iostream` objects)
  - `output` (can be used with `cout` and `ostream` objects)
#include <string>
#include <vector>
int main() {
    vector<int> vect;
    for(int i=0; i < 6; i++) vect.push_back(i);

    vector<int>::const_iterator it; // read-only iterator
    it = vect.begin(); // assign it to the start of the vector
    while (it != vect.end()) // while it hasn't reach the end
    {
        cout << *it << " "; // print the current value
        it++; // and iterate to the next element
    }
    cout << endl;
}
ALGORITHMS

- STL contains **algorithms** implemented as **function templates** to perform operations on **containers**.
- Requires `<algorithm>` header file
- **algorithm** includes:
  - `binary_search`  count
  - `for_each`       find
  - `find_if`        `max_element`
  - `min_element`   `random_shuffle`
  - `sort`         and others

- **Google is your friend!** Textbook also describes the above in some detail
#include <string>
#include <vector>
#include <algorithm>
int main() {
    vector<int> vect;
    for(int i=0; i < 6; i++) vect.push_back(i);

    vector<int>::const_iterator it; // read-only iterator
    it = min_element(li.begin(), li.end());
    cout << *it << " "; // print smallest element
    it = max_element(li.begin(), li.end());
    cout << *it << " "; // print largest element
    cout << endl;
}