11.1 Overview

The standard template library (STL) provides templates for data structures and algorithms. Each data structure is in its own file. For example, there is `vector`, `stack`, and `set`. These are implemented as templates (which we will discuss more later). For now, it suffices to know that things like `vector` and `stack` are created to store a specific data type. This data type is specified in angle brackets at declaration:

```cpp
vector<int> A;
```

Thereafter we can just treat `A` as before. For example, the `push_back` method adds an item at the end of the vector. Another useful method is `emplace(val)`; this adds an object to the structure treating `val` as the input to the constructor for that object.

11.2 Iterators and Range-for Loops

The idea of iterators is simply wonderful. They allow one to do the same operation on all the elements of a collection. Creating your own requires learning more (which we skip), but using iterators is standardized. This is especially useful in avoiding working out how many elements there are: the basic idea is element access and element traversal.

In the Standard Template Library (STL), structures have iterators. While the paradigm is the same for each, each iterator is a different type. A C++ iterator behaves like a pointer; it can be incremented and it can be tested for completion by comparing with a companion iterator. The actual value is obtained by dereferencing.

Note that `begin()` returns the first element in the collection, while `end()` returns a value beyond the last element: it must not be dereferenced!

```cpp
int addup ( vector<int> & A ) {
    int sum = 0;
    vector<int>::const_iterator start = A.begin();
    vector<int>::const_iterator stop = A.end();
    for( ; start!=stop; ++start )
        sum += *start;
    return sum;
}
```

You can leave out the `const_` part. Or even replace it with `auto`: this “typename” can be used in places to help the reader where the compiler can infer the type. In the above case, the `vector` template class also implements subscripting; so one could write:
int addup ( vector<int> & A ) {
    int sum = 0;
    for(int i=0; i<A.size(); i++ )
        sum += A[i];
    return sum;
}

A **range-for** loop can be used to process all the entries in some data structure. E.g.

```cpp
int addup ( vector<int> & A ) {
    int sum = 0;
    for(int val : A )
        sum += val;
    return sum;
}
```

### 11.3 Adding Templates

Certain code needs the data-type to support certain operations. For example, a set needs a way to test for equality. Actually, the **set** from the STL assumes there is an equivalent to the `<` command. (Two objects $A$ and $B$ are considered equal if both $A < B$ and $B < A$ are false.)

Note that if you need to create your own method for use in an STL data structure, you should use a two-argument version as friend, not the one-argument method described earlier:

```cpp
class Foo {
    friend bool operator<(Foo & A, Foo & B);
};
bool operator<(Foo & A, Foo &B) { ... }
```

**Sample Code**

```
MyInteger.h
TestMyInteger.cpp
```