C++: Templates, STL, Strings

CIT595
Spring 2010

Templates

- A template allows a class or function to be parameterized by one or more data types
  - re-usable/flexible

- All templates start with the keyword `template` followed by a list of template parameters

- Example:
  - `template <typename T>
    class MyT;
  
  - `template <typename V, typename B>

Example of a Template class

```c++
template <typename T> class MyT{
    public:
    MyT() {}  
    MyT(T d): data(d) {}  
    MyT(MyT<T> & m): data(m.data) {}  // copy constructor
    T value() const {return data;}
    void operator=(T r){ data = r;}
    void operator=(MyT <T> & r) { data = r.data; }

    private:
    T data;
};
```

- Template parameter is used as a data type throughout the template definition

Using a Template Class

- To create instances of a template class, data must be passed as parameters:
  - `MyT<int> imt(5);`
  - `MyT<double> dmt (3.14);`
  - `MyT<int> * imtPointer = new MyT<int>(0);`

- Once an object is created from a template class works like any object
  - `imt.data = 7;
  - cout << imt.value() endl;
  - imtPointer->value() endl;`
Using a Template Class contd..

- Anytime user defined type can be also be used as template parameter
- Suppose there are classes A and B

```cpp
A someA;
MyT<A> amt(someA);
MyT<B> bmt;
```

Advantages of Templates

- Templates are put together at compile time
  - Allows static type checking
  - Compile error examples:
    ```cpp
    bmt = amt;
    A a = imt.value();
    ```
  - Compile warning example:
    ```cpp
    int val = dmt.value();
    ```
- Templates avoid all the dynamic casting that would involved with using polymorphism

Standard Template Library (STL)

- Data Structures/Containers
  - Template classes with common data structures such as lists and stacks
- Iterators
  - A generalization of a pointer used to access containers (ADT) without knowing anything about the internal structure of the container
- Generic Algorithms
  - Functions that can work with many different data structures
- Functional Objects
  - Objects that overload the operator() (function call) so that can be used like a function

STL Data Structures/Containers

- There are several groups of data structures/containers:
  - Sequence structures
    - List or array type things
    - Include vectors, lists, and deques
  - Sequence Adaptors
    - Built on top of sequence structures
    - Includes stacks and queues
  - Associative structures
    - Store key-value pairs
    - Includes maps and sets
Example of STL data structure

- `vector<T>`: similar to java vector
  - has [] overloaded: `myVect[1]`
  - has `push_back()` to add to end
  - has `pop_back()` to remove from end
  - Supports iterators
  - `#include <vector>`
  - Constructors Example
    - `vector<T> v`;
    - `vector<T> v(value)`;
    - `vector<T> v(size, initial_value)`;

Example code using Vector

```cpp
vector<int> v; //vector of ints
int i;
while(!cin.eof()){
    cin >> i;
    v.push_back(i);
}
for(i = 0; i < v.size(); i++){
    cout << v[i] << " ";
}
```

Iterator

- Iterator = container + position

```cpp
vector<int>::iterator it = v.begin();
while (it != v.end()) {
    int x = *it;
    cout << "Current thing is " << x << endl;
    ++it;
}
```

Iterator contd..

- `vector<T>::iterator` is a type- right?
  - Yes, but we may have to give a hint…Why?
  - What does this mean:
    - `x * y`
  - `x` times `y` you say?
    - Always?
    - Are you sure?…
Iterator contd..

- Maybe above that I wrote
  typedef int x;
  ....
- Now what does this mean:
  x * y;
- Declaration of an int pointer called x.
- You can’t locally determine the type

Silly example, but real consequences

- In templates: need to be explicit what is a type
  sometimes
  template<class T> void foo(int x) {
    T::x (y);
    ....
  }
- Is T::x a type => declare variable y
- Is T::x a function => call it with y as argument

Rule

- If you have a qualified (with ::) name that depends on a template parameter
  - The default is to assume its not a type
  - If you want a type, use typename
type
name vector<T>::iterator it;

Iterator can be used with Many STL classes

- list<T>::iterator
- map<key,data>::iterator
- set<T>::iterator

- All iterators “look” the same
  - ++: go to next
  - * : current item (deference the pointer)

- Decouples element access from structure
STL has various algorithms
- `#include <algorithm>`
- `sort`
- `min_element`
- `count`
- `find`
- `etc …`
- Almost all act on iterators
  - Allows to act on any type of container

STL Algorithms contd..
- Generic algorithms are just template functions
  - E.g. `find` algorithm

```cpp
template<class InputIterator, class T>
InputIterator find ( InputIterator first, InputIterator last, const T& value ) {
    for ( ; first!=last; first++) {
        if ( *first==value ) {
            break;
        }
    }
    return first;
}
```

Containers Iterator Algorithms

STL algorithms contd..
- Suppose you want to remove from an item from a vector
  - `myVect.remove(x)`  right?
  - No
- STL algorithms have `remove()` which works on iterators
  - `remove(myVect.begin(), myVect.end(), x);
- None of the STL containers have a remove
  - Use the `remove` algorithm instead
- `remove()` returns a forward iterator pointing to the new end of the sequence
  - which includes all the elements with a value other than `value`
Example with remove (without containers)

```cpp
int myints[] = {10,20,30,30,20,10,10,20};
// bounds of range:
int* pbegin = myints;
int* pend = myints+sizeof(myints)/sizeof(int);

pend = remove(pbegin, pend, 20);

cout << "range contains:";
for (int* p=pbegin; p!=pend; p++)
    cout << *p << " ";
cout << endl;
```

Containers and Memory Management

- Containers can hold just about anything
  - Primitives (such as int, float, and char)
  - User defined types
  - Pointers

- All values placed into a container are copied there using the assignment operator (=)
  - The objects are still copied, but how you code the class controls whether the copy is deep or shallow
    - A deep copy is accomplished via a overloaded copy constructor & assignment operator, and should be used if your object contains pointers to anything (other objects, character buffers, etc)

- Removing a primitive or a pointer causes it to be discarded by the container
- Removing an object causes it to be discarded by the container as well, but the object's destructor is called in the process
- If you tell a container to remove a pointer (whether it's a character buffer or an object), that memory is lost (leaked)
  - Unless you've retained a local copy of that pointer with which you can later call delete

STL conclusion

- An ISO C++ standard framework of about 10 containers and about 60 algorithms connected by iterators
  - Other organizations provide more containers and algorithms in the style of the STL
    - Boost.org, Microsoft, SGI, …

- Probably the currently best known and most widely used example of generic programming
Functional Objects

■ Any object overloading the parenthesis operator can be used as a function

```cpp
class Bigger {
public:
    Bigger(int v): val(v) {}
    bool operator()(int test)
    { return test > val; }
private:
    int val;
};
Bigger byte(8);
if(byte (3)) { ..
```

C++ Strings

■ C++ has a new String type:
```cpp
#include <string>
using std::string;

string a;
string b = “Initial string”;
string c(“Another string”);
string d(b); //copy constructor
a = “A different string”;
```

C++ String functions

■ The string has many functions:
  ■ Number of chars: s.length()
  ■ Assign: s1 = s2
  ■ Append: s1 += s2
  ■ Concatenate: s1 + s2
  ■ Character access: s[index]
  ■ Comparison: s1 == s2, s1 != s2
  ■ Substring: s.substr(start, length)
  ■ Input: cin >> s

Strings

■ String concatenation and assignment only works with other strings
  int a = 5;
  string text = “a=” + a; //This won’t work

■ For other data types, a string stream can be used:
  ```cpp
  #include <sstream>
  using namespace std;
  ...
  int x = 5, y = 6;
  ostringstream formatter;
  formatter << “sum of” << x << “ and” << y << “ is” << (x+y);
  string s = formatter.str();
  ```