CSE 4/535 Project Specific Notes on C++ [1]

Pradipto Das, Rohini Srihari

SUNY at Buffalo

September 8, 2009

[1] dedicated to my late grandfather

Part I

Important stuff first: Templates, Containers, Iterators and stuff

Notes on STL Templates

Wordweb : A model or standard for making comparisons - template

Template Parameters

1. Type Parameters
2. Nontype parameters
3. Template template parameters

```
1 template < comma-separated-list-of-parameters >
```

Template Type Parameters

1. Introduced either with keyword "typename" or "class"
2. A type parameter acts much like a typedef name

```
1 template <typename Allocator>
2 class Node {
3     class Allocator* allocator; // Error
4 ...
5 }
```

Notes on STL Templates

Wordweb : A model or standard for making comparisons - template

Template Parameters

1. Type Parameters
2. Nontype parameters
3. Template template parameters

```
1 template < comma-separated-list-of-parameters >
```

Template Type Parameters

1. Introduced either with keyword "typename" or "class"
2. A type parameter acts much like a typedef name

```
1 template <typename Allocator>
2 class Node {
3     class Allocator* allocator; // Error
4 ...
5 }
```

Notes on STL Templates

Template Non Type Parameters

stand for constant values that can be determined at compile or link time

1. An integer type or an enumeration type
2. A pointer type
3. A reference type

```
1 template <typename T,                                  // a type parameter
2           typename T::Allocator* Allocator>            // a nontype parameter
3 class aDataStructure;
```

Template Template Parameters

1. Placeholders for class templates
2. Declared much like class templates but keywords struct and union cannot be used

```
1 template <template<typename X > class C >  // OK
2 void f(C<int>* p);
3
4 template <template<typename X > struct C >  // Error
5 void g(C<int>* p);                          // not valid
```

Notes on STL Templates

Template Non Type Parameters

stand for constant values that can be determined at compile or link time

1. An integer type or an enumeration type
2. A pointer type
3. A reference type

```
1 template <typename T,                                  // a type parameter
2           typename T::Allocator* Allocator>            // a nontype parameter
3 class aDataStructure;
```

Template Template Parameters

1. Placeholders for class templates
2. Declared much like class templates but keywords struct and union cannot be used

```
1 template <template<typename X > class C >   // OK
2 void f(C<int>* p);
3
4 template <template<typename X > struct C >  // Error
5 void g(C<int>* p);                          // not valid
```

Notes on STL Templates

Default Template Arguments

- Any type of template parameter can be equipped with a default argument

```
1 template <template< typename T, typename A = Allocator<T> >
2                                                 class Container >
3 class Adapter {
4     Container<int> storage;   // implicitly equivalent to
5     ...                       // Container<int, Allocator<int> >
6 }
```

An example: the STL map declaration

1. Maps and Multimaps are implemented as a balanced tree data structure in STL
2. Each node has a key with a corresponding value. The order of elements in the tree depends on the way two keys are compared.

```
1 template < class Key, class T, class Compare = less<Key>,
2            class Allocator = allocator< pair<const Key, T> > >
3 class map;    // similarly for class std::multimap
```

Notes on STL Templates

Default Template Arguments

- Any type of template parameter can be equipped with a default argument

```
1 template <template< typename T, typename A = Allocator<T> >
2                                                 class Container >
3 class Adapter {
4     Container<int> storage;   // implicitly equivalent to
5     ...                       // Container<int, Allocator<int> >
6 }
```

An example: the STL map declaration

1. Maps and Multimaps are implemented as a balanced tree data structure in STL
2. Each node has a key with a corresponding value. The order of elements in the tree depends on the way two keys are compared.

```
1 template < class Key, class T, class Compare = less<Key>,
2            class Allocator = allocator< pair<const Key, T> > >
3 class map;    // similarly for class std::multimap
```

Notes on STL Containers

Some stl container declarations

- Some of the following declarations happen to be quite useful

```
1 vector<int> *m_aVector;
2 set<string> *m_aSetOfStrings;
3 map<long, multimap<double, Statistics*,
4                    std::greater<double> >*> *m_dataStructure;
```

STL map with key as a non-primitive type

1. What happens if you have something like map < myStructure*, long >* aMap;
2. How would stl handle insertion or lookup in this case?

For methods defined in standard C++ library classes including STL, please check out www.cplusplus.com/reference/stl/

An example

```
1  struct voxel{  // structure declaration
2   int x; int y; double z;
3   voxel (int anX, int aY, double aZ) : x(anX), y(aY), z(aZ) {}
4  };
5  struct compareVoxel{  // a function object structure
6   bool operator() (struct voxel const& voxel_before,
7                    struct voxel const& voxel_after) const {
8       return voxel_before.z > voxel_after.z;
9     }
10 };
11 ......
12 map<voxel, string, compareVoxel> *m =
13                          new map<voxel, string, compareVoxel>();
14   map<voxel, string, compareVoxel>::const_iterator mitr;
15   m->insert(make_pair(voxel(1,2,3.0),"further"));
16   m->insert(make_pair(voxel(1,2,4.0),"nearest"));
17   m->insert(make_pair(voxel(1,2,2.0),"farthest"));
18   for (mitr = m->begin(); mitr != m->end(); mitr++)
19        cout << mitr->second << endl;
20 delete m;
```

C++ string tokenizer

A practical and useful application

How about some string tokenization?

```
1  void tokenize(string& str, vector<string>& tokens,
2                string const& delimiters = " ") {
3    // Skip delimiters at beginning.
4    size_t lastPos = str.find_first_not_of(delimiters, 0);
5    // Find the first "non-delimiter"
6    size_t pos = str.find_first_of(delimiters, lastPos);
7    while (string::npos != pos || string::npos != lastPos)
8    {
9        // If a token is found, add it to the vector
10       tokens.push_back(str.substr(lastPos, pos - lastPos));
11       // Skip delimiters.  Note: the "not_of" is useful here
12       lastPos = str.find_first_not_of(delimiters, pos);
13       // Find the next "non-delimiter"
14       pos = str.find_first_of(delimiters, lastPos);
15   }
16   return;
17 }
```

C++ Bitset

- Bitsets model fixed-sized arrays of bits or Boolean values
- Although bitsets are included under C++ STL containers, they really are not so [2]
- You cannot change the number of bits - it is a template parameter
- STL provides you with functionality manipulate individual bits in the fixed size bitset

```
1 namespace std {
2     template <size_t Bits> class bitset;
3 }
4 // Create an instance of bitset by calling:
5 bitset<7> allAsciiBitCombinations;
6 bitset<32> aFourByteDataType;
```

Million Dollar Question for Project 1 How can I use bitset to find actual Unicode character codes?

[2] http://www.ddj.com/cpp/184401382

C++ Bitset

Let's write this function

```
1  void getUnicodeCharacters(string const& hexString,
2          vector<unsigned long>& unicodeChars_vec) {
3    size_t nHalfBytesPerChar, nextHalfByteIndex = 0,
4                              endHalfByteIndex = 0;
5    // test the first byte
6    int utf8IDHalfByteVal, halfByteVal;  char c;
7    bitset<4> id;
8    bitset<32> rawUnicodeChar_bs, finalUnicodeChar_bs;
9    while ( endHalfByteIndex != hexString.length() ) {
10     id.reset();
11     c = hexString.c_str()[nextHalfByteIndex];
12     if ( (c >= (int)('a')) && (c =< (int)('f')) )
13       utf8IDHalfByteVal = c - (int)('f') + 15 ;
14     else utf8IDHalfByteVal = c - (int)('9') + 9 ;
15
16     // create the bitset id
17     id |= utf8IDHalfByteVal;
```

********** continued to next slide ...

C++ Bitset

```
1     // now check the highest half byte for this unicode char
2     if ( id[0] && .. ) {                                    // 4 bytes
3       nHalfBytesPerChar = 4*2;
4     } else if ( (id[0] == 0) && (id[1] && .. ) ) { // 3 bytes
5       nHalfBytesPerChar = 3*2;
6     } else if ( (id[1] == 0) && .. ) {                      // 2 bytes
7       nHalfBytesPerChar = 2*2;
8     } else {                                                // 1 byte
9       nHalfBytesPerChar = 1*2;
10    }
11    // update endHalfByteIndex based on nHalfBytesPerChar;
12
13    // copy the respective bytes into the unicodeChar bitset
14    for (size_t hb = nextHalfByteIndex;
15                          hb < endHalfByteIndex; hb++ ) {
16      c = (hexString.c_str())[hb];
17      // get the halfByteVal
18      // create the rawUnicodeChar_bs using
19      //                 <<= and |= operations on bitset;
20    }
```

********** continued to next slide ...

C++ Bitset

```
1       // now extract the final Unicode char bitset
2       // from the raw Unicode character bitset
3
4       switch (nHalfBytesPerChar) {
5       case 8: // do something
6       case 6: // do something
7       case 4: // do something
8       case 2: // do something
9       }
10      unicodeChars_vec.push_back
11                  (finalUnicodeChar_bs.to_ulong());
12
13      nextHalfByteIndex += nHalfBytesPerChar;
14      rawUnicodeChar_bs.reset();
15      finalUnicodeChar_bs.reset();
16    } // end outer while loop
```

Function Objects and Algorithms

Math101 : A function that takes functions as input - functional

Function Object in C++

1. is an object that has an operator () defined so that in the code below, foo() is a call of operator () for the function object foo instead of a call for a function foo()

```
1 FunctionObjectType foo;
2 ...
3 foo(...);
```

2. might be smarter since it may have a state
3. Each function object has its own type - you can pass type of a function object to a template to specify a certain behavior
4. is usually faster than a function pointer

```
1 class FunctionObjectType {
2   public: void operator() () { statements }
3           // return type depends on application
4 };
```

Function Objects and Algorithms

shuffling music in a media player library list

```
1  class MusicShuffleCriteria {
2    public: bool operator() (MusicObj const& m1,
3                             MusicObj const& m2) {
4     return  m1.random() < m2.random();
5    }
6  };
7  int main() {
8   typedef set<MusicObj, MusicShuffleCriteria>
9                           iTunesLibShuffled;
10  iTunesLib myLib;
11  ...
12  iTunesLib::iterator itr;
13  for ( itr = myLib.begin(); itr != myLib.end(); itr++ )
14    ... // play music e.g. this->play(*itr)
15 }
```

Function Objects with internal states

```
1  class FO {
2    public: int operator() () {
3     return  m_value++;   }
4     FO(int value) : m_value(value) {}
5    private: int m_value;
6  };
7  int main() {
8   list<int> collection;
9   // use generate_n() from <algorithms>
10  generate_n(back_inserter(collection), // start
11             10,                        // # of elements
12             FO(1));                    // generating sequence
13  // now replace 2nd to last-but-one elements in
14  // collection with values starting at 50
15  generate( ++collection.begin(),  // start
16            --collection.end(),    // end
17            FO(52));               // new generating sequence
18 }
19 Output: 1  2  3  4  5  6  7  8  9 10
20         1 52 53 54 55 56 57 58 59 10
```

http://www.cplusplus.com/reference/algorithm/generate_n/

Function Objects pass by value or reference

- Function objects are passed by value than reference
  ∴ we cannot use the state of the FO
- However, there is a way to pass function objects by reference

```
1  class FO {  // defined before  };
2  int main() {
3   list<int> collection;
4   FO seq(1);
5   // reuse seq later from where it leaves of
6   generate_n<back_insert_iterator<list<int> >,
7              int, FO&>
8              (back_inserter(collection), // start
9              4,                         // # of elements
10             seq);                      // pass by reference
11
12  // use seq from initial value
13  generate_n( back_inserter(collection), // start
14             4,                         // # of elements
15             seq);                      // pass by value
16 }
```

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

Function Objects pass by value or reference

Predicates vs FOs

- Predicates are functions or FOs that return a Boolean value
- Not every function that returns a Boolean value is a valid predicate for STL
- **Be warned** Do not use FOs with the algorithm remove_if(collection.begin(), collection.end(), FO) for scenarios like:

```
1  class Ith {
2    int ith, count;
3  public: Ith (int i) : ith(i), count(0) {}
4    bool operator() (int) { return ++count == ith; }
5  };
6  list<int> l; list<int>::iterator pos;
7  pos = remove_if(l.begin(), l.end(), Ith(3));
8          // dangerous - copy predicate by value
9  l.erase( pos, l.end() );
10 Output: If the list has 1 2 3 4 5 6 7 8 9
11 the output will be 1 2 4 5 7 8 9
12 // Note: Return value of remove_if() is a forward iterator
13 // pointing to the new end of the sequence, which now includes
14 // all the elements for which pred was false
```

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

Function Adapters

- Function adapter is a function object that enables combining of different function objects with each other, with certain values or even with special functions
- Function adapters are FOs themselves

An example

```
1 find_if(collection.begin(), collection.end() // range
2         bind2nd(greater<int>(), 35) )     // criterion
```

- the expression bind2nd(greater<int>(), 35) produces a combined function object that checks whether an int value is greater than 35

This may prove very helpful during token processing

Function Adapters - Applications on strings

character replacements in strings

- Replace all non word characters

```
1  // remove all occurences of given characters
2  // within a token string
3  string::iterator itr; // itr -> reusable iterator
4  string characters = " '\":;,.?/\\~`!@%_#$%"+
5                      "^&*-+={}[]()<>|\t\r\n";
6  size_t numChars = characters.length();
7  for (size_t i = 0; i < numChars ; i++){
8    itr = remove_if( token.begin(), token.end(),
9          bind2nd(equal_to<char>(), characters.at(i)) );
10   token.erase(itr, token.end()); // token is a string
11 }
```

Function Adapters - Applications on strings

So what is bind2nd doing?

- Constructs an unary function object from the binary function object op by binding its second parameter to the fixed value x

```
1 template <class Operation, class T>
2   binder2nd<Operation> bind2nd (const Operation&
3                                 op, const T& x) {
4   return binder2nd<Operation>(op,
5       typename Operation::second_argument_type(x));
6 }
```

Parameters

- **op**: Binary function object derived from binary_function.
- **x**: Fixed value for the second parameter of op.

Return value

- An unary function object equivalent to op but with the second parameter always set to x. binder2nd is a type derived from unary_function.

Function Adapters - Applications on strings

So what is equal_to doing?

- This **struct** defines function objects for the equality comparison operation.

```
1 template <class T> struct equal_to : // derived from
2                           binary_function <T,T,bool> {
3   bool operator() (const T& x, const T& y) const
4     {return x==y;}  // this should look familiar by now
5 };
```

Uses

- Objects of this struct can be used with some standard algorithms such as mismatch, search or unique.

TAKE HOME MESSAGE: You will need function predicates for e.g. greater<double> even when storing ranked document ids in your favorite container.

Algorithms - Applications on strings

lowercase all characters in a string

- C++ style lowercasing

```
1 struct ToLower {
2 char operator() (char c) const
3              { return std::tolower(c); }
4 };
5 string token = "How about THIS?";
6 std::transform(token.begin(), token.end(),
7                  token.begin(), ToLower());
```

transform()

```
1 template < class InputIterator, class OutputIterator,
2                                  class UnaryOperator >
3 OutputIterator transform ( InputIterator first1,
4       InputIterator last1, OutputIterator result,
5       UnaryOperator op ) {
6   while (first1 != last1)
7     *result++ = op(*first1++);
8   return result;
9 }
```

Algorithms - std::transform()

transform() with binary function object

transform()

```
template < class InputIterator1 , class InputIterator2 ,
           class OutputIterator , class BinaryOperator >
OutputIterator transform (      InputIterator1 first1 ,
          InputIterator1 last1 , InputIterator2 first2 ,
     OutputIterator result , BinaryOperator binary_op );
```

```
int op_increase (int i) { return ++i; }
int op_sum (int i, int j) { return i+j; }
int main () {
  vector<int> first;  vector<int> second;
  vector<int>::iterator it;
  // set some values:
  for (int i=1; i<6; i++) first.push_back (i*10); //  first: 10 20 30 40 50
  second.resize(first.size());      // allocate space
  transform (first.begin(), first.end(), second.begin(), op_increase);
                                                  // second: 11 21 31 41 51
  transform (first.begin(), first.end(), second.begin(), first.begin(), op_sum);
                                                  //  first: 21 41 61 81 101
  cout << "first contains:";
  for (it=first.begin(); it!=first.end(); ++it)
    cout << " " << *it;
}
Output: first contains: 21 41 61 81 101
```

Setting Locales in C++

Locales

- A locale is a collection of functions and parameters used to support national or cultural conventions
- Depending on this locale, different format for floats, dates, monetary values and so on are used
- The format of a string defining a locale is usually: *language[_area[.code]]* as in de_DE.88591 (German in Germany with ISO Latin-1 encoding)

```
1 char * setlocale ( int category, const char * locale );
2    // declared in <clocale> - C style global function
```

setlocale() influences the results of character classification and manipulation functions as well as the I/O functions

```
1  locale = setlocale(LC_ALL, ""); cout << "locale = " << locale;
2  // produces: locale = en_US.UTF-8
```

Setting Locales in C++

Locales

- By changing a stream's locale a programmer can change the way the stream behaves to suit a different set of conventions

```
1 class locale {
2   // ....
3   explicit locale(const char* name);    // construct locale using
4                                         // C locale name
5   static locale global(const locale&); // set the global locale
6                                         // (get the previous locale)
7   // ....
8 };
```

Setting the global locale doesn't change the behavior of existing streams that are using the previous value of the global *locale*.

```
1 void f() {
2   std::locale loc("POSIX");   // standard locale for POSIX
3   cin.imbue(loc);             // let cin use loc
4   // ...
5   cin.imbue(std::locale::global()); // reset cin to use the
6                                     // default locale
7 }
```

C++ Unicode File I/O

Lets look at a concrete example

```
1   ifstream uifs;
2   uifs.open( inFilename.c_str(), ios::binary );
3   string delim = "\n\r";  vector<string> utokens;
4   vector<unsigned long> unicodeChars_vec;
5
6   if ( uifs.is_open() ) {
7     ofstream uofs;
8     uofs.open( outFilename.c_str(), ios::out|ios::binary );
9     if ( !uofs.is_open() ) { // exit }
10    string uline, hexString; char hexByte[3];
11    unsigned char val;
12
13    while ( !uifs.eof() ) {
14     getline(uifs, uline);      uofs << uline;
15     utokens.clear();
16     CUtilities::tokenize(uline, utokens, delim);
17     for ( size_t i = 0; i < utokens.size(); i++ ) {
18       hexString = "";
19       for ( size_t j = 0; j < utokens.at(i).length(); j++ ) {
20         val = (unsigned char)(utokens.at(i).c_str())[j];
21         sprintf(hexByte, "%x", val);   hexString += hexByte;
22       }
23       unicodeChars_vec.clear();
24       getUnicodeCharacters(hexString, unicodeChars_vec);
25     }
26    }
27    uifs.close();   uofs.close();
28  }
```

An unique design pattern

Something to think about...

Let's say your search engine framework have many different modules that use the same token processing rules on a string whenever needed.
The token processor class has a member function void process(string& token) that changes the token being passed as a reference.

- How would you write a single tokenizer class that is instantiated only once in the entire execution life cycle of the program so that whenever any module needs to process a token no separate token processor object is instantiated?

An unique design pattern

```
1  class CTokenProcessor
2  {
3  private:
4   static CTokenProcessor *m_cTokenProcessor;   // NOTE!
5
6   set<string> *m_stopWordSet;
7   // TODO: probably not needed (other than fun)
8   int m_isTokenMonth;
9   int m_isTokenSlashSeparatedDate;
10
11 protected:
12  CTokenProcessor(void);   // protected constructor
13
14 public:
15  static CTokenProcessor* getInstance();
16
17  // service functions
18  void processToken(string& token);
19  void readStopWordsFile(void);
20  int isNumber(const string& str);
21 private:
22  ~CTokenProcessor(void);
23 };
```

An unique design pattern

So how would you write getInstance()?

```
1 CTokenProcessor* CTokenProcessor::getInstance() {
2   if ( m_cTokenProcessor == 0 ) {
3     m_cTokenProcessor = new CTokenProcessor();
4   }
5   return m_cTokenProcessor;
6 }
```

The destructor is also private and that ensures persistence of a single instance of this class in memory througout the entire lifecycle of the program

Part II

The End

That's all folks - Thanks!

Questions?