CSE 165/ENGR 140
Intro to Object Orient Program

Lecture 24 – Standard Library(1)
Announcement

- Lab #10 tomorrow
  - Due 4/30 at 11:59PM

- Final on Thursday (5/7)
  - Last lecture of semester
  - Review and quiz on Tuesday (5/5)

- Final project is out (in UCMCROPS)
  - Due date: 5/11 (Monday) at 11:59PM
  - Presentation date: 5/12 (Tuesday) at 3:00PM

- Reading assignment:
  - Ch. 6 and 7 (Vol. 2)
Strings (Chapter 3)

- Strings
  - Array of characters with a *null terminator* as the final element.
    ```plaintext
    Hello\0
    ```
    - Implementation techniques may vary, only the interface of the class is a standard.
      - Strings are commonly implemented as “vector<char>”,
      - but, we cannot really tell if reference counting is used in assignments:

    ```cpp
    string s2 = s1;       // deep/full copy or reference may happen
    s1[0] = '6';          // either way this statement must ONLY modify s1
    ```
**String Operations**

- **Constructors**
  ```
  string a ("text");
  string b (5, 'a'); // creates 5 'a's
  ```

- **Substrings**
  ```
  string s1 ("Anything worth doing is worth overdoing.");
  string s2 (s1, 0, 8); // Copy 8 chars starting at position 0
  ```

- **begin() / end() iterators**
  ```
  string source("a string");
  string s ( source.begin(), source.end() );
  ```
String Operations

string bigNews("I saw Elvis in a UFO. ");

// 1) get the size: num of chars
cout << "Size = " << bigNews.size() << endl;

// 2) capacity: how much we can store without reallocation:
cout << "Capacity = " << bigNews.capacity() << endl;

// 3) insert at pos 1, capacity and size will change:
bigNews.insert ( 1, " thought I" );

// 4) reserve space (capacity) in advance:
bigNews.reserve(500);

// 5) append at the end of the string:
bigNews.append("c++ is great.");
String Operations

```cpp
string s("A piece of text");

// 1) replace num chars with the new string, starting at pos:
s.replace ( pos, num, "new substring");

// 2) example of using find and then replace:
string tofind = "piece", newsubstring = "block";
size_t i = s.find ( tofind, 0 ); // start search at position 0
if ( i!=string::npos ) // Did we find the string to replace?
    s.replace ( i, tofind.size(), newsubstring );

// 3) many other manipulation methods available (see textbook)
// for ex: erase(), find_first_of(), find_first_not_of(), etc

// 4) example of operators:
string s1("This "), s2("That "), s3("The other ");
s1 = s1 + s2; // concatenation operator
s1 += s3;    // another concatenation operator
s1 += s3 + s3[4] + "ooh lala"; // combined concatenations

Output:
This That The other The other oooh lala
Strings and character traits

Declaration of String in Standard C++ header file:

```cpp
typedef basic_string<char> string;
```

String is a template!

```cpp
template< class charT,
       class traits = char_traits<charT>,
       class allocator = allocator<charT>
> class basic_string;
```

- to change the way the string class treats character comparison, you must supply a different `char_traits< > template`, because that defines the behavior of the comparison function.
using std::toupper;
using std::tolower;
using std::ostream;
using std::string;
using std::char_traits;
using std::allocator;
using std::basic_string;

struct ichar_traits : char_traits<char> {
    // We'll only change character-by-character comparison functions:

    static bool eq(char c1st, char c2nd) {
        return toupper(c1st) == toupper(c2nd);
    }

    static bool ne(char c1st, char c2nd) {
        return !eq(c1st, c2nd);
    }

    static bool lt(char c1st, char c2nd) {
        return toupper(c1st) < toupper(c2nd);
    }
}
// class continuation...:

static int compare(const char* str1, const char* str2, size_t n) {
    for(size_t i = 0; i < n; i++) {
        if (str1 == 0) return -1;
        else if (str2 == 0) return 1;
        else if (tolower(*str1)<tolower(*str2)) return -1;
        else if (tolower(*str1)>tolower(*str2)) return 1;
        assert (tolower(*str1)==tolower(*str2));
        str1++; str2++;
    } // Compare the next chars if needed
    return 0;
}

static const char* find (const char* s1, size_t n, char c) {
    while(n-- > 0)
        if(toupper(*s1) == toupper(c)) return s1;
        else ++s1;
    return 0;
};
Example: defining new traits 3/3

// Here is my new type:
typedef basic_string<char, ichar_traits> istring;

// Here is how to output my new type:
inline ostream& operator<<(ostream& os, const istring& s) {
    return os << string(s.c_str(), s.length());
}

int main() {
    // The same letters except for case:
    istring first = "tHis";
    istring second = "ThIS";
    cout << first << endl;
    cout << second << endl;
    assert(first.compare(second) == 0);
    assert(first.find('h') == 1);
    assert(first.find('I') == 2);
    assert(first.find('x') == string::npos);
}
Generic algorithms (Chapter 6)

- An algorithm that works with any type of sequence.
  - Makes your programs simpler and safer.
  - We can customize algorithms at runtime.

- Copy operations
  - Generically copy elements between two lists with a single implementation:

```cpp
// Implementation of the “copy algorithm”:
template<typename T>
void copy ( T* begin, T* end, T* dest ) {
    while (begin != end)
        *dest++ = *begin++;
}
```
Example of copy operations:

```
// 1) copy applied to int arrays:
int a[] = {10, 20, 30};
int b[3];
copy ( a, a + 3, b );

// 2) copy applied to string arrays:
string a[] = {"read", "my", "lips"};
string b[3];
copy ( a, a + 3, b );

// 3) copy applied to vector<int>:
int a[] = {10, 20, 30};
const size_t SIZE = sizeof a / sizeof a[0];
vector<int> v1(a, a + SIZE);
vector<int> v2(SIZE);
copy ( v1.begin(), v1.end(), v2.begin() );
assert( equal(v1.begin(), v1.end(), v2.begin()) );
```
Generic algorithms

- Selective copy example:

```cpp
#include <algorithm>
#include <cstddef>
#include <iostream>

// You supply this predicate:
bool gt15(int x) { return 15 < x; }

int main() {
    int a[] = {10, 20, 30};
    const size_t SIZE = sizeof a / sizeof a[0];
    int b[SIZE];
    int* endb = remove_copy_if (a, a+SIZE, b, gt15); // selective copy
    int* beginb = b;
    while (beginb != endb)
        cout << *beginb++ << endl; // Prints: 10
}

// other functions available: replace_if(), replace_copy_if(), etc
```
Function objects

- Provide a way to better define user functions, just have them inside classes as operator():

```cpp
class gt_n {
  int value;
public:
  gt_n(int val) : value(val) {}
  bool operator()(int n) {
    return n > value;
  }
};

int main() {
  gt_n f(4); // Constructor to create an instance, val = 4
  cout << f(3) << endl; // Function call, prints 0 (for false)
  cout << f(5) << endl; // Function call, prints 1 (for true)
}
```
Function objects

- Generator: takes no arguments and returns a value
  - The standard library provides one generator, the function `rand()` declared in `<cstdlib>`, and has some algorithms, such as `generate_n()`, which apply generators to a sequence.

- Unary Function: takes a single argument of any type and optionally returns a value

- Binary Function: takes two arguments of any two types and optionally returns a value

- Unary Predicate: Unary Function that returns a bool.

- Binary Predicate: Binary Function that returns a bool.
Function objects

- **Strict Weak Ordering**
  - A binary predicate that allows for a more general interpretation of “equality.” Some of the standard containers consider two elements equivalent if neither is less than the other (using operator<( )). For example, data records (structs) may be sorted on a subset of the struct’s fields, where two records with equal keys are not really “equal”.

- **LessThanComparable**: A class that has a less-than operator <

- **Assignable**: A class that has a copy-assignment operator = for its own type

- **EqualityComparable**: A class that has an equivalence operator == for its own type
Function objects

- The generic algorithms in the standard library use function objects.
- The `<functional>` header defines a number of useful generic function objects.

```cpp
#include <algorithm>
#include <cassert>
#include <functional>
#include <iostream>
#include <iterator>

using namespace std;

int main() {
    int a[] = {10, 20, 30};
    const size_t SIZE = sizeof a / sizeof a[0];
    remove_copy_if(a, a + SIZE,
                   ostream_iterator<int>(cout, "\n"),
                   bind2nd(greater<int>(), 15));
}
```
**Function objects**

Examples of function objects:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Result produced</th>
</tr>
</thead>
<tbody>
<tr>
<td>plus</td>
<td>BinaryFunction</td>
<td>arg1 + arg2</td>
</tr>
<tr>
<td>minus</td>
<td>BinaryFunction</td>
<td>arg1 - arg2</td>
</tr>
<tr>
<td>multiplies</td>
<td>BinaryFunction</td>
<td>arg1 * arg2</td>
</tr>
<tr>
<td>divides</td>
<td>BinaryFunction</td>
<td>arg1 / arg2</td>
</tr>
<tr>
<td>modulus</td>
<td>BinaryFunction</td>
<td>arg1 % arg2</td>
</tr>
<tr>
<td>negate</td>
<td>UnaryFunction</td>
<td>- arg1</td>
</tr>
<tr>
<td>equal_to</td>
<td>BinaryPredicate</td>
<td>arg1 == arg2</td>
</tr>
<tr>
<td>not_equal_to</td>
<td>BinaryPredicate</td>
<td>arg1 != arg2</td>
</tr>
<tr>
<td>greater</td>
<td>BinaryPredicate</td>
<td>arg1 &gt; arg2</td>
</tr>
<tr>
<td>less</td>
<td>BinaryPredicate</td>
<td>arg1 &lt; arg2</td>
</tr>
<tr>
<td>greater_equal</td>
<td>BinaryPredicate</td>
<td>arg1 &gt;= arg2</td>
</tr>
<tr>
<td>less_equal</td>
<td>BinaryPredicate</td>
<td>arg1 &lt;= arg2</td>
</tr>
<tr>
<td>logical_and</td>
<td>BinaryPredicate</td>
<td>arg1 &amp;&amp; arg2</td>
</tr>
<tr>
<td>logical_or</td>
<td>BinaryPredicate</td>
<td>arg1</td>
</tr>
<tr>
<td>logical_not</td>
<td>UnaryPredicate</td>
<td>!arg1</td>
</tr>
<tr>
<td>unary_negate</td>
<td>Unary Logical</td>
<td>!(UnaryPredicate(arg1))</td>
</tr>
<tr>
<td>binary_negate</td>
<td>Binary Logical</td>
<td>!(BinaryPredicate(arg1, arg2))</td>
</tr>
</tbody>
</table>