FIRST-ORDER OBJECTS

- Ideally, all the data types in a programming language should be first-order objects.
  - I.e., all the data types should be manipulated in the “usual ways.”
  - They should be comparable using the normal operators, passed by value (unless explicitly stated otherwise) to functions, etc. etc.
- C++ has gone a step further than Java in this respect.
  - Indeed, even the “primitive” types can be considered classes; there is only one class of objects in the C++ discourse.
- But then take (yes, please take) arrays (and thus strings).
  - They cannot be manipulated in the usual way.
  - Indeed, they are in fact pointers to the actual content, so they cannot be meaningfully compared using usual operators, are always passed by reference to functions, etc.
  - Tired of that `strcmp` yet?

THE C++ STANDARD TEMPLATE LIBRARY (STL)

- Ofers only first-order data types.
  - Also offers generic, handy algorithms.
- Includes, between other convenient types, well-behaved replacements for arrays (vector).
  - Polymorphic in the usual sense, not Java or Lisp sense.
  - I.e., you can declare vectors that hold any data type, but a given vector instance can hold data of a single type.
  - Quick random access but slow copying and expansion.
- In C++ proper, strings are no longer a subtype of arrays. In particular the class string is not even in the STL (strings are not polymorphic).
- For a reference of STL types, see for instance

  `http://www.cppreference.com/cpp_stl.html`

OTHER STL TYPES

- Lists: the opposite of vectors, fast insertions and deletions, slower random access.
  - Header: `<list>`
  - Sample declaration: `list<int> l;`
  - Some interesting member functions: `push_front`, `push_back`, `size`, `front`, `pop_front`, `reverse`, `merge` (on sorted lists), `sort`.
- `list` and `vector` are sequence containers.
- There are also associative containers, such as sets.

ITERATORS

- Iterators are objects which move through a collection or container of other objects, selecting them one at a time.
- Iterators are not pointers, but they are useful for the same jobs.
  - A pointer is actually a special case of iterator.
- Operations on an iterator `itr`:
  - `itr++` advances the iterator to the next location.
  - `*itr` returns a reference to the object stored at location pointed at by `itr`.
  - `itr1==itr2` (`itr1!=itr2`) return true if `itr1` and `itr2` refer (do not refer) to the same location.
- Containers define several iterators. They also define iterator types.
  - For instance, there are two iterators defined for the class `string`: `begin()` and `end()`
  - the type `string::iterator` is also defined. In other words, the type of the `begin()` is `string::iterator begin(void);`

USING ITERATORS

```
#include <string>
#include <string.h>
#include <iostream>
using namespace std;

char* end_str (char* str)
{ char* p = str;
  while (*p != ’\0’) p++;
  return p;  }

int my_strcmp(char* s1, char* s2) {
  char* p1 = s1;
  char* p2 = s2;
  while( p1 != end_str(s1) &&
         p2 != end_str(s2) ) {
    cout << "Compare " << *p1 <<
         " with " << *p2 << "\n";
    if ( *p1 != *p2 )
      return (*p1 < *p2) ? -1 : 1;
    p1++;
    p2++;
  }
  return strlen(s2) - strlen(s1);
}
```

```
int main () {
  char* cs1 = "hello world";
  char* cs2 = "hello";
  string ss1(cs1);
  string ss2(cs2);

  cout << my_strcmp(cs1,cs2) << ", "
       << my_strcmp(ss1,ss2) << "\n";
}

int my_strcmp(string& s1, string& s2) {
  string::iterator p1 = s1.begin();
  string::iterator p2 = s2.begin();
  while( p1 != s1.end() &&
         p2 != s2.end() ) {
    cout << "Compare " << *p1 <<
         " with " << *p2 << "\n";
    if ( *p1 != *p2 )
      return (*p1 < *p2) ? -1 : 1;
    p1++;
    p2++;
  }
  return s2.size() - s1.size();
}
```

OTHER ITERATORS

- The iterators presented above are in fact forward iterators.
- Other types of iterators:
  - Bidirectional: same as forward iterator, plus
    - `itr--` sets the iterator to the previous location. We can traverse the container forward as well as backward.
  - Random access: same as bidirectional iterator, plus assignment:
    - `itr=itr1` sets the iterator `itr` to point to the same location as `itr1`.
    - Actually, `string::iterator` is a type for random access iterator. So we can do:

```
string::iterator p1;     // compare with:
p1 = s1.begin();         // string::iterator p1 = s1.begin();
```

ALGORITHMS

- Algorithms do not hold any data (instead, they operate on some provided data).
  - So they are not classes, they are functions; or rathrer “recipes for functions.”
    - Remember, now all our objects are first-class, so we can write functions that can be applied on a wide collection of data types.
    - In other words, we can write generic functions.
    - In other words, we can write things we can really call algorithms (as opposed to algorithm implementations).
  - A first simple algorithm: receives a function f and a value x, and applies f on x.

    ```
    template<class UnaryFunc, class T>
    void call_func(T& x, UnaryFunc f) {
      f(x);
    }
    ```

  - You don’t always have to roll your own algorithms. Handy functions are provided in STL. They are grouped in the header `<algorithm>`.

ALGORITHMS (CONT'D)

- So algorithms are functions.
- But then functions (and thus algorithms) are also types, so we must be able to define functions as classes.
  - How?

ALGORITHMS (CONT'D)

- So algorithms are functions.
- But then functions (and thus algorithms) are also types, so we must be able to define functions as classes.
  - How?
  - By defining the function application operator, i.e., `operator()`
  - Example: binary comparison objects.

```
template <class T> struct tmax {
  bool operator() (const T& a, const T& b) { return (a > b) ? a : b; }
};

int main () {
    tmax<int> max;  // max is now a function (and also an object)
    cout << max(1, 2) << endl;
}
```

Ugly, much like macro definition for generic functions!

Binary comparison, revisited

- We can however move the template inside the class:

```
struct tmax {
    template <class T> T operator()(T a, T b) {
        return (a > b) ? a : b;
    }
};

int main () {
    tmax max;

    cout << max(1, 2) << endl;                             // on int
    cout << max(string("alpha"), string("beta")) << endl; // on string
    cout << max(1.5, 6.3) << endl;                         // on float
    // cout << max (1.5, 6) << endl;                       // not going to work
                                                           // (why?)
}
```

FUNCTION OBJECTS IN STL

- Most operators have equivalent functions in STL
- Header that needs to be included: `<functional>`

```
#include <functional> // for greater<> and less<>
#include <algorithm> //for sort()
#include <vector>
using namespace std;

int main()
{
    vector <int> vi;
    //..fill vector
    sort(vi.begin(), vi.end(), greater<int>() );//descending
    sort(vi.begin(), vi.end(), less<int>() ); //ascending
}
```

Function Objects in STL (cont'd)

Arithmetic:

`plus`	→	addition `x + y`
`minus`	→	subtraction `x - y`
`multiplies`	→	multiplication `x * y`
`divides`	→	division `x / y`
`modulus`	→	remainder `x % y`
`negate`	→	negation `- x`

Commparison:

`equal_to`	→	`x == y`
`not_equal_to`	→	`x != y`
`greater`	→	`x > y`
`less`	→	`x < y`
`greater_equal`	→	`x >= y`
`less_equal`	→	`x <= y`

Logical:

`logical_and`	→	`x && y`
`logical_or`	→	`x || y`
`logical_not`	→	`! x`

- Compute the by-element addition of two lists of integer values, placing the result back into the first list:

```
transform(listOne.begin(), listOne.end(),
          listTwo.begin(), listTwo.begin(), plus<int>() );
```

ACCESS STATE INFORMATION IN FUNCTIONS

- Functions declared as objects can also access state information (much like static local variables, only simpler to control)

```
class iotaGen
{
public:
  iotaGen (int start = 0) : current(start) { }
  int operator() () { return current++; }
private:
  int current;
};

int main {
    vector<int> aVec(20);
    generate(aVec.begin(), aVec.end(), iotaGen(1));
}
```

STL ALGORITHMS

- Algorithms already defined in the STL (implemented as function templates):

```
template <class _Tp>
const _Tp& min(const _Tp& __a, const _Tp& __b) {
    return __b < __a ? __b : __a;
}

template <class _Tp>
const _Tp& max(const _Tp& __a,  const _Tp& __b) {
    return  __a < __b ? __b : __a;
}
```

MORE INTERESTING STL ALGORITHMS

- Search:

```
template <class Iter, class Predicate>
Iter find_if (Iter begin, Iter end, Predicate pred);
```

- Binary search:

```
template <class Iter, class Val>
Iter find (Iter begin, Iter end, Val what);
```

- Counting:

```
template <class Iter, class Val>
Iter count (Iter begin, Iter end, Val what);
```

- Sorting:

```
template <class RandomIter>
RandomIter sort (RandomIter begin, RandomIter end);
```

- Merging two sorted lists:

```
template <class Iter>
Iter merge (Iter begin1, Iter end1, Iter begin2, Iter end2, Iter dest);
```