

- 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 (yeas, 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?

VECTORS

- A vector is a relocating, expandable, polymorphic array.
 - They are 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.
- Before you begin:

```
#include <vector>
```
- Declaring a vector:

```
vector<int> a(3); // a vector holding ints, of (initial) size 3
vector<char> b; // a vector holding chars, of default initial size 0
```
- Accessing values in a vector:

```
a[1] = a[1] + 5;
```

 - `operator[]` does **not** check for array bounds.

- Offers only first-order data types.
 - Also offers generic, handy algorithms.
- Includes, between other convenient types, well-behaved replacements for arrays (`vector`).
 - In C++ proper, strings are no longer a subtype of arrays. In particular the class `string` is not even in the STL.
- For a reference of STL types, see for instance
http://www.cppreference.com/cpp_stl.html

VECTORS (CONT'D)

- Other goodies:
 - You can obtain the size of a vector by using the member function `size()`.
 - You can **resize** a vector using the member function `resize(int)`.
 - * Expensive operation!
 - You cannot however initialize a vector using a literal array or for that matter any array.
 - * You have to use a loop to initialize the values in a vector, or be happy with the default. The empty constructors of the elements **are** called.

```
void get_ints (vector<int> array) {
    int read_so_far = 0, input;
    while ( cin >> input ) {
        if ( read_so_far == array.size() )
            array.resize(array.size() * 2 + 1);
        array[read_so_far++] = input;
    }
    array.resize(read_so_far);
}
```

VECTORS (CONT'D)

- Other goodies:
 - You can obtain the size of a vector by using the member function `size()`.
 - You can **resize** a vector using the member function `resize(int)`.
 - * Expensive operation!
 - You cannot however initialize a vector using a literal array or for that matter any array.
 - * You have to use a loop to initialize the values in a vector, or be happy with the default. The empty constructors of the elements **are** called.

```
void get_ints (vector<int>& array) {
    int read_so_far = 0, input;
    while ( cin >> input ) {
        if ( read_so_far == array.size() )
            array.resize(array.size() * 2 + 1);
        array[read_so_far++] = input;
    }
    array.resize(read_so_far);
}
```

SIZE, CAPACITY, PUSHING

- There are two “sizes”: how many elements are stored in the vector (`size()`) and how many elements can be held (`capacity()`).
 - **But** don't get excited, when you declare `vector<int> a(3)` the **size** is set to 3, even if you did not put anything in there explicitly.
 - In most of the cases, you should forget the existence of `capacity()`.
- Another version of `get_ints`:

```
void get_ints (vector<int>& array) {
    array.resize(0);
    while ( cin >> input ) {
        array.push_back(input);
    }
}
```

- The member function `push_back` increases the size by 1, and adds the argument as the last element in the vector.
 - * Capacity is also increased if needed.

STRINGS AS FIRST-ORDER OBJECTS

- **Not** really in the STL (strings are not polymorphic), but related to the idea of first-order objects.
- Before you begin: `#include <string>`
- Declaring strings:

```
string s; // an (initially empty) string
string s1("hello"); // a string initialized by means of a string literal
```

Operation on string <code>s</code>	Result
<code>s.length()</code>	returns the length of <code>s</code>
<code>s[2]</code>	accesses the third character in <code>s</code>
<code>s = "hi";</code>	assignment operator
<code>s == "hi"</code>	true! ; special functions no longer needed
<code>s >= "hello"</code>	true!
<code>s = s + " there"</code>	<code>s</code> becomes "hi there"
<code>s += " there"</code>	same as above
<code>s.c_str()</code>	returns a pointer to the C string held by <code>s</code> (<code>const char*</code> , null-terminated)

TO USE OR NOT TO USE

- Arrays or vectors? C strings or C++ strings? The choice is yours.
- Probably C++ constructs are better:
 - Cleaner programs (because they are first-order objects), therefore
 - Less prone to programming errors, and
 - Improved understandability of the resulting code.
- An experienced programmer would probably choose between efficiency (C types are better in this respect) and understandability.
 - But then an experienced programmer will choose understandability in all but the most special cases.
- There are however problems with the STL types (inherited from the problems with templates).
 - Most importantly, compiler errors are reported in somewhere else than the place of occurrence.
 - So program with caution.

- **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.

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();
}
```

- 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)`;

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 do not hold any data (instead, they **operate** on some provided data).
 - So they are not classes, they are functions; or rather “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>`.

- 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 $-x$

Comparison:

`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);
```