STL concepts

- Container: stores objects, supports iteration over the objects
  - Containers may be accessible in different orders
  - Containers may support adding/removing elements
  - e.g., vector, map, set, deque, list, multiset, multimap
- Iterator: interface between container and algorithm
  - Point to objects and move through a range of objects
  - Many kinds: input, forward, random access, bidirectional
  - Syntax is pointer like, analagous to (low-level) arrays
- Algorithms
  - find, count, copy, sort, shuffle, reverse, …

Iterator specifics

- An iterator is dereferenceable, like a pointer
  - `*it` is the object an iterator points to
- An iterator accesses half-open ranges, [first..last), it can have a value of last, but then not dereferenceable
  - Analagous to built-in arrays as we'll see, one past end is ok
- An iterator can be incremented to move through its range
  - Past-the-end iterators not incrementable

```
vector<int> v; for(int k=0; k < 23; k++) v.push_back(k);
vector<int>::iterator it = v.begin();
while (it != v.end()) { cout << *v << endl; v++;}
```


Design patterns

"... describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

Christopher Alexander, quoted in GOF

- Name
  - good name provides a handle for the pattern, builds vocabulary
- Problem
  - when pattern is applicable, context, criteria to be met, design goals
- Solution
  - design, collaborations, responsibilities, and relationships
- Forces and Consequences
  - trade-offs, problems, results from applying pattern: help in evaluating applicability

Iterator as Pattern

- (GOF) Provides access to elements of aggregate object sequentially without exposing aggregate's representation
  - Support multiple traversals
  - Supply uniform interface for different aggregates: this is *polymorphic iteration* (see C++ and Java)
- Solution: tightly coupled classes for storing and iterating
  - Aggregate sometimes creates iterator (Factory pattern)
  - Iterator knows about aggregate, maintains state
- Forces and consequences
  - Who controls iteration (internal iterator, external iterator)?
  - Who defines traversal method?
  - Robust in face of concurrent insertions and deletions?

STL overview

- STL implements generic programming in C++
  - Container classes, e.g., vector, stack, deque, set, map
  - Algorithms, e.g., search, sort, find, unique, match, …
  - Iterators: pointers to beginning and one past the end
  - Function objects: less, greater, comparators
- Algorithms and containers decoupled, connected by iterators
  - Why is decoupling good?
  - Extensible: create new algorithms, new containers, new iterators, etc.
  - Syntax of iterators reflects array/pointer origins, an array can be used as an iterator

STL examples: wordlines.cpp

- How does an iterator work?
  - Start at beginning, iterate until end: use [first..last) interval
  - Pointer syntax to access element and make progress

```
vector<int> v; // push elements
vector<int>::iterator first = v.begin();
vector<int>::iterator last  = v.end();
while (first < last) {
    cout << *first << endl;
    ++first;
}
```

  - Will the while loop work with an array/pointer?
- In practice, iterators aren't always explicitly defined, but passed as arguments to other STL functions

Review: what's a map, a set, a …

- Maps keys to values
  - Insert key/value pair
  - Extract value given a key, iterate over pairs
  - STL uses red-black tree, guaranteed O(log n) …
    - STL unofficially has a hash_map, see SGI website
  - Performance and other trade-offs?
- A set can be implemented by a map
  - Stores no duplicates, in STL guaranteed O(log n), why?
  - STL also has multimap

arrays and strings: what's a char *?

- Why not rely solely on string and vector classes?
  - how are string and vector implemented?
  - lower level access can be more efficient (but be leery of claims that C-style arrays/strings *required* for efficiency)
  - real understanding comes when more levels of abstraction are understood
- string and vector classes insulate programmers from inadvertent attempts to access memory that's not accessible
  - what is the value of a pointer?
  - what is a segmentation violation?

Contiguous chunks of memory

- In C++ allocate using array form of new

```
int * a = new int[100];
double * b = new double[300];
```

- new [] returns a pointer to a block of memory
  - how big? where?
- size of chunk can be set at runtime, not the case with

```
int a[100];
cin >> howBig;
int a[howBig];
```

- delete [] a; // storage returned

```
int * a = new int[100];
```


```
a is a pointer
*a is an int
a[0] is an int (same as *a)
a[1] is an int
a+1 is a pointer
a+32 is a pointer
*(a+1) is an int (same as a[1])
*(a+99) is an int
*(a+100) is trouble
a+100 is valid for comparison
```

of pointer values

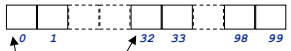
C-style contiguous chunks of memory

- In C, malloc is used to allocate memory

```
int * a = (int *)
   malloc(100 * sizeof(int));
double * d = (double *)
   malloc(200 * sizeof(double));
```

- malloc must be cast, is NOT type-safe (returns void *)
  - void * is 'generic' type, can be cast to any pointer type
- free(d); // return storage
- We WILL NOT USE malloc/free

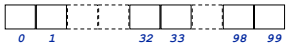
```
int * a = (int *)
     malloc(100*sizeof(int));
```




```
a is a pointer
*a is an int
a[0] is an int (same as *a)
a[1] is an int
a+1 is a pointer
a+32 is a pointer
*(a+1) is an int (same as a[1])
*(a+99) is an int
*(a+100) is trouble
a+100 is valid for comparison
```


Address calculations, what is sizeof(…)?

```
int * a = new int[100];
```




```
a[33] is the same as *(a+33)
if a is 0x00a0, then a+1 is
0x00a4, a+2 is 0x00a8
(think 160, 164, 168)
```

```
double * d = new double[200];
```


```
*(d+33) is the same as d[33]
if d is 0x00b0, then d+1 is
0x00b8, d+2 is 0x00c0
(think 176, 184, 192)
```

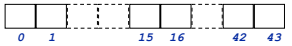
- x is a pointer, what is x+33?
  - a pointer, but where?
  - what does calculation depend on?
- result of adding an int to a pointer depends on size of object pointed to
- result of subtracting two pointers is an int:

```
(d + 3) - d == _____
```


More pointer arithmetic

- address one past the end of an array is ok for *pointer comparison only*
- what about *(begin+44)?
- what does begin++ mean?
- how are pointers compared using < and using == ?
- what is value of end - begin?

```
char * a = new char[44];
char * begin = a;
char * end = a + 44;
```




```
while (begin < end)
{
   *begin = 'z';
   begin++;  // *begin++ = 'z'
}
```

What is a C-style string?

- array of char terminated by sentinel '\0' char
  - sentinel char facilitates string functions
  - '\0' is nul char, unfortunate terminology
  - how big an array is needed for string "hello"?
- a string is a pointer to the first character just as an array is a pointer to the first element
  - `char * s = new char[6];`
  - what is the value of s? of s[0]?
- char * string functions in <string.h>

C style strings/string functions

- strlen is the # of characters in a string
  - same as # elements in char array?

```
int strlen(char * s)
// pre: '\0' terminated
// post: returns # chars
{
    int count=0;
    while (*s++) count++;
    return count;
}
```

- Are these less cryptic?

```
while (s[count]) count++;
// OR, is this right?
char * t = s;
while (*t++);
return t-s;
```

- what's "wrong" with this code?

```
int countQs(char * s)
// pre: '\0' terminated
// post: returns # q's
{
  int count=0;
  for(k=0;k < strlen(s);k++)
    if (s[k]=='q') count++;
  return count;
}
```

- how many chars examined for 10 character string?
- solution?

<string.h> aka <cstring> functions

- strcpy copies strings
  - who supplies storage?
  - what's wrong with s = t?

```
char s[5];
char t[6];
char * h = "hello";
strcpy(s,h); // trouble!
strcpy(t,h); // ok

char * strcpy(char* t,char* s)
//pre: t, target, has space
//post: copies s to t,returns t
{
   int k=0;
   while (t[k] = s[k]) k++;
   return t;
}
```

- strncpy copies n chars (safer?)
- what about relational operators <, ==, etc.?
- can't overload operators for pointers, no overloaded operators in C
- strcmp (also strncmp)
  - return 0 if equal
  - return neg if lhs < rhs
  - return pos if lhs > rhs

```
if (strcmp(s,t)==0) // equal
if (strcmp(s,t) < 0)// less
if (strcmp(s,t) > 0)// ????
```


Arrays and pointers

- These definitions are related, but not the same

```
int a[100];
int * ap = new int[10];
```

- both a and ap represent 'arrays', but ap is an lvalue
- arrays converted to pointers for function calls:

```
char s[] = "hello";
// prototype: int strlen(char * sp);
cout << strlen(s) << endl;
```

- multidimensional arrays and arrays of arrays

```
int a[20][5];
int * b[10]; for(k=0; k < 10; k++) b[k] = new int[30];
```