**Standard Libraries**

- In C++ there is the *Standard Library*, formerly known as the *Standard Template Library* or STL
  - Emphasizes generic programming (using templates)
  - Write a sorting routine, the implementation depends on
    - Elements being comparable
    - Elements being assignable

  We should be able to write a routine not specific to int, string or any other type, but to a generic type that supports being comparable/assignable

- In C++ a templated function/class is a code-factory, generates code specific to a type at compile time
  - Arguably hard to use and unsafe

---

**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, analogous 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
  - Analogous 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

  ```cpp
  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, apply in MultiSet)?
  - Who defines traversal method?
  - Robust in face of insertions and deletions?

**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

**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

---

Software Design
Address calculations, what is sizeof(...)?

- 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:
  
\[
(d + 3) - d = \_
\]

```c
int * a = new int[100];
a[33] is the same as *(a+33)
if a is 0x00a0, then a+1 is 0x00a1, 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)
```

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?

```c
int a = new int[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
char * s = new char[6];
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?
- what’s “wrong” with this code?
  ```c
  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;
  }
  ```
- Are these less cryptic?
  ```c
  while (s[count]) count++;
  // OR, is this right?
  char * t = s;
  while (*t++);
  return t-s;
  ```
- how many chars examined for 10 character string?
- solution?
More string functions (from <string.h>)

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

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

- 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

  ```c
  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
  ```c
  int * ap = new int[10];
  ```

- both a and ap represent ‘arrays’, but ap is an lvalue

- arrays converted to pointers for function calls:
  ```c
  char s[] = "hello";
  ```

- multidimensional arrays and arrays of arrays
  ```c
  int a[20][5];
  int * b[10]; for(k=0; k < 10; k++) b[k] = new int[30];
  ```

Microsoft question, 108 question

- Write atoi, write itoa, which is harder?

- Questions? Issues? Problems?

  ```c
  int atoi(const char * sp);
  char * itoa(int num);
  string itoa(unsigned int num); // what's the difference?
  ```

- Difference between const char * p and char * const p
  - one is a pointer to a constant character
  - one is a constant pointer to a character

What about ints and unsigned ints?

- What are the largest and smallest integer values?
  - Where defined? What is standard? What is common?
  - Typically we have fabs(INT_MIN) > fabs(INT_MAX)
  - Typically –INT_MIN is negative

- What does this depend on? Do we need to be aware of this?
  - Two's complement is nearly universal
  - Unsigned values are your friends

  ```c
  int x = INT_MIN;
  x = -x;
  unsigned int y = -x;
  ```