CPS 108, Spring 2003

- **Software Design and Implementation**
  - **Object oriented programming and design**
    - good design helps do away with late night Teer-fests, but some late nights are inevitable
    - your toolkit must include mastery of language/programming and design

- **What’s in the course?**
  - **C++ and Java, team projects, mastery exams**
    - team projects can be more and less than the sum of their parts
  - **high-level abstractions, low-level details**
    - patterns, heuristics, and idioms
Program Design and Implementation

• **Language independent principles of design and programming**
  - **design heuristics**
    - coupling, cohesion, small functions, small interfaces ...
  - **design patterns**
    - factories, adapter, MVC aka observer/observable, ...

• **Language specific:**
  - **Idioms**
    - smart pointers, vectors/arrays, overloaded operators ...
  - **idiosyncracies, idiocies**
    - must define virtual destructor, stream zoo in Java, ...
Administrivia

- check website and news regularly
  - duke.cs.cps108
- Grading (see web pages)
  - group projects: small, medium, large
  - mastery programs (solo or semi-solo endeavors)
  - readings and summaries
  - tests
- Evaluating team projects, role of TA, UTA, consultants
  - face-to-face evaluation, early feedback
- Compiling, tools, environments, Linux, Windows
  - g++ 2.95, Java 2 aka 1.4, JRE, ...
Classes: Review/Overview

- A class encapsulates state and behavior
  - Behavior first when designing a class
  - Information hiding: who knows state/behavior?

- State is private/protected; some behavior is public
  - Private/protected helper functions
  - A class is called an object factory, creates lots of instances

- Classes communicate and collaborate
  - Parameters: send and receive
  - Containment: has a reference to
  - Inheritance: is-a
C++ (and Java) class construction

- **C++ uses** .h and .cpp, **Java uses** .java
  - Documentation different (javadoc vs. ???)

- **Default, overloaded, copy constructor**
  - tvector, string, Date
  - Default constructor needed in C++, where?
  - Copy constructor needed to avoid shallow copy
  - In C++ destructors needed to free resources/self, Java?
  - Clone makes copy in Java (rare), share is default

- **Private, protected, public, (package)**
  - Private default in C++, package default in Java
  - Per method declaration in Java, class sections in C++
Design Criteria

Good design comes from experience, experience comes from bad design

Fred Brooks (or Henry Petroski)

- Design with goals:
  - ease of use
  - portability
  - ease of re-use
  - efficiency
  - first to market
  - ????
How to code

- **Coding/Implementation goals:**
  - Make it run
  - Make it right
  - Make it fast
  - Make it small

- **spiral design (or RAD or !waterfall or ...)**
  - what’s the design methodology?
XP and Refactoring

*(See books by Kent Beck (XP) and Martin Fowler (refactoring))*

• **eXtreme Programming (XP) is a lightweight design process**
  - Communication: unit tests, pair programming, estimation
  - Simplicity: what is the simplest approach that works?
  - Feedback: system and clients; programs and stories
  - Courage: throw code away, dare to be great/different

• **Refactoring**
  - Change internal structure without changing observable behavior
  - Don’t worry (too much) about upfront design
  - Simplicity over flexibility (see XP)
Modules, design, coding, refactor, XP

- Make it run, make it right, make it fast, make it small
- Do the simplest thing that can possibly work (XP)
  - Design so that refactoring is possible
  - Don’t lose sight of where you’re going, keep change in mind, but not as the driving force [it will evolve]

- Refactor: functionality doesn’t change, code does
  - Should mean that new tests aren’t written, just re-run
  - Depends on modularity of code, testing in pieces

- What’s a module in C++
  - Could be a class, a file, a directory, a library, a namespace
  - We should, at least, use classes, files, directories
Design Heuristics: class/program/function

(see text by Arthur Riel)

- **Coupling**
  - classes/modules are independent of each other
  - goal: minimal, loose coupling
  - do classes collaborate and/or communicate?

- **Cohesion**
  - classes/modules capture one abstraction/model
  - keep things as simple as possible, but no simpler
  - goal: strong cohesion (avoid kitchen sink)

- **The open/closed principle**
  - classes/programs: open to extensibility, closed to modification
Tapestry classes -> STL

- **What’s the difference between tvector and vector**
  - Safety and the kitchen sink
    - What happens with t[21] on a 21-element vector?
    - Part of STL means crufty code (whose viewpoint?)
  - What about Java analog?

- **Differences in wordlines.cpp and tapwordlines.cpp**
  - Map compared to tmap, what other kinds of maps?
  - Sets and vectors, which is easier to use?

- **Anything not clear in either program? Are these programs object-oriented?**
C++ idioms/general concepts

- **Genericity**
  - Templates, STL, containers, algorithms

- **Copy/Assignment/Memory**
  - Deep copy model, memory management “required”

- **Low-level structures**
  - C-style arrays and strings compared to STL, Tapestry

- **const**
  - Good for clients, bad for designers/coders?

- **From C to C++ to Java**
  - function pointers, function objects, inheritance
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
  - \( *\textit{it} \) is the object an iterator points to

- An iterator accesses half-open ranges, \([\text{first}..\text{last})\), it can have a value of \textit{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?
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

```cpp
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, STL/Tapestry

- Maps keys to values
  - Insert key/value pair
  - Extract value given a key
  - STL uses red-black tree, Tapestry uses bst or hashtable
    - STL unofficially has a hash_map, see SGI website
  - Performance and other trade-offs?

- In Tapestry, there’s an inheritance hierarchy of tmap, BSTMap, HMap
  - The hash-table requires map of string->value
    - Makes programming simpler, too restrictive in practice
  - See tapwordlines.cpp
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];
```

```
0   1   32  33  98  99
```

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

- malloc must be cast, is NOT typesafe (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(...)?

- 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?

```c
char * 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 style strings/string functions

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

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

- Are these less cryptic?

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

- 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;
  }
  ```

- 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`?

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

```c
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

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

- 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?**


```c
int atoi(const char * sp);
char * itoa(int num);
string itoa(int num);
string uitoa(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’s hard about itoa (or atoi)?

- What’s the naïve way of coding itoa?
  - Performance implications?
  - Alternatives?
  - What does the standard do?

- What’s the naïve way of coding atoi?
  - Where can problems happen?
  - What are choices for dealing with them?
  - What does the standard do?

- What will you do?
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;
```