Welcome!

Program Design and Analysis II
CPS 100
LSRC B101
M, W, F 10:30-11:20

Professor: Jeffrey Forbes
Course Topics

- **Design concepts**
  - A bit more C++
  - Complexity

- **Data Structures**
  - Sets
  - Trees
  - Maps

- **Algorithms**
  - Sorting
  - Game playing
  - Inheritance
  - Linked lists
  - Tries
  - Graphs
  - Searching
Administrivia

- Read web page *regularly*
  http://www.cs.duke.edu/education/courses/fall01/cps100
- Read newsgroup *regularly*
  news:duke.cs.cps100

- Overview handout
Frequently Asked Questions

• What is the prerequisite? (choose one)
  ∙ CPS 6
  ∙ 4 or 5 on AP Computer Science AB exam
  ∙ Instructor’s permission

• How does this course fit into the curricula?
  ∙ Required for majors & minors
  ∙ Solid grounding in programming, data structures, and algorithms
On the subject of questions…

- Did you ask any good questions today?
  - *Ideas and Information* by Nobel prize winning physicist Arno Penzias
  - Questions which illuminate help nourish ideas
  - Children are born curious
  - Fear of public displays of ignorance prevents learning
- Participate in class
- Go to office hours
- Make study groups with your classmates
Efficient Programming

- Designing and building efficient programs efficiently requires knowledge and practice
  - Hopefully the programming language helps, it’s not intended to get in the way
  - Object-oriented concepts, and more general programming concepts help in developing programs
  - Knowledge of data structures and algorithms helps

- Tools of the engineer/scientist programmer
  - A library or toolkit is essential, don’t reinvent the wheel
  - Someone must build the tools
  - Programming is not just art, not just science, not just engineering
This reads words, how can we count different/unique words?

```cpp
vector<string> list;
string filename, word;
cin >> filename;
ifstream input(filename.c_str());
CTimer timer;
timer.Start();
while (input >> word) {
    list.push_back(word);
}
timer.Stop();
cout << "read " << list.size() << " words in ";
cout << timer.ElapsedTime() << " seconds" << endl;
```
Tracking different/unique words

- **We want to know how many times ‘the’ occurs**
  - Do search engines do this? Does the number of occurrences of “basketball” on a page raise the priority of a web page in some search engines?
    - Downside of this approach for search engines?

- **Constraints on solving this problem**
  - We must read every word in the file (or web page)
  - We must search to see if the word has been read before
  - We must process the word (bump a count, store the word)

  - Are there fundamental limits on any of these operations? *Where should we look for data structure and algorithmic improvements?*
Search: measuring performance

- **How fast is fast enough?**

  ```cpp
  bool search(const tvector<string> & a, 
               const string & key)
  // pre:  a contains a.size() entries
  // post: return true if and only if key found in a
  {
    int k; int len = a.size();
    for(k=0; k < len; k++)
      if (a[k] == key) return true;
    return false;
  }
  ```

- **C++ details: parameters? Return values? Vectors?**
- **How do we measure performance of code? Of algorithm?**
  - Does processor make a difference? P4, G4, ???
- **Is this the way you would find entries in a phonebook?**
Genericity through templates

- **What kind of object can we put in a vector?**
  - What kind of object can we sort?
  - What kind of object can we print: `cout << t << endl;`

- **What is a vector? How is it different from the class Date?**
  - Container class, what does it contain? Why use it?

- **Genericity is a good thing, program to a more abstract idea rather than something more concrete**
  - Sorting function for sorting int, string, double, ...
  - In C++ genericity done with templates and sometimes with inheritance; useful in different situations
Selection Sort: The Code (selectsort2.cpp)

```cpp
void SelectSort(tvector<int> & a)
// pre: a contains a.size() elements
// post: elements of a are sorted in non-decreasing order
{
    int j, k, temp, minIndex, numElts = a.size();
    // invariant: a[0]..a[k-1] in final position
    for(k=0; k < numElts - 1; k++)
    {
        minIndex = k;              // minimal element index
        for(j=k+1; j < numElts; j++)
        {
            if (a[j] < a[minIndex])
            {
                minIndex = j;      // new min, store index
            }
        }
        temp = a[k];      // swap min and k-th elements
        a[k] = a[minIndex];
        a[minIndex] = temp;
    }
}
```
What changes if we sort strings?

- The parameter changes, the definition of `temp` changes
  - Nothing else changes, code independent of type
  - We can use features of language to capture independence

- We can have different versions of function for different array types, with same name but different parameter lists
  - Overloaded function: parameters different so compiler can determine which function to call
  - Still problems, duplicated code, new algorithm means …?

- With function templates we replace duplicated code maintained by programmer with compiler generated code
Creating a function template

```cpp
template <class Type>
void SelectSort(tvector<Type> & a)
// pre: a contains a.size() elements
// post: elements of a are sorted in non-decreasing order
{
    int j,k,minIndex,numElts = a.size();
    Type temp;
    // invariant: a[0]..a[k-1] in final position
    for(k=0; k < numElts - 1; k++)
    {
        minIndex = k;              // minimal element index
        for(j=k+1; j < numElts; j++)
        {
            if (a[j] < a[minIndex])
            {
                minIndex = j;      // new min, store index
            }
        }
        temp = a[k];      // swap min and k-th elements
        a[k] = a[minIndex];
        a[minIndex] = temp;
    }
}

● When the user calls this code, different versions are compiled
```
Some template details

- Function templates permit us to write once, use several times for several different types of vector
  - Template function “stamps out” real function
  - Maintenance is saved, code still large (why?)

- What properties must hold for vector elements?
  - Comparable using < operator
  - Elements can be assigned to each other

- Template functions capture property requirements in code
  - Part of generic programming
  - Some languages support this better than others
**Templates and function objects**

- In a templated sort function vector elements must have certain properties (as noted previously)
  - Comparable using `operator <`
  - Assignable using `operator =`
  - Ok for int, string, what about Date? ClockTime?
- What if we want to sort by a different criteria
  - Sort strings by length instead of lexicographically
  - Sort students by age, grade, name, ...
  - Sort stocks by price, shares traded, profit, ...
- We can’t change how `operator <` works
  - Alternative: write sort function that does NOT use `<`
  - Alternative: encapsulate comparison in parameter, pass it
Function object concept

- To encapsulate comparison (like operator <) in a parameter
  - Need convention for parameter: name and behavior
  - Other issues needed in the sort function, concentrate on being clients of the sort function rather than implementors

- Name convention: class/object has a method named `compare`
  - Two parameters, the vector elements being compared (might not be just vector elements, any two parameters)

- Behavior convention: `compare` returns an int
  - zero if elements equal
  - +1 (positive) if first > second
  - -1 (negative) if first < second
Function object example

```cpp
class StrLenComp
{
    public:
        int compare(const string& a, const string& b) const
            // post: return -1/+1/0 as a.length() < b.length()
            {
                if (a.length() < b.length()) return -1;
                if (a.length() > b.length()) return 1;
                return 0;
            }
};

// to use this:
StrLenComp scomp;
if (scomp.compare(“hello”, “goodbye”) < 0) …
```

- We can use this to sort, see strlensort.cpp
  - Call of sort: `InsertSort(vec, vec.size(), scomp);`
Using function object for search

```cpp
class WcountComp
{
    public:
        int compare(const Wcount& a, const Wcount& b) const
        // post: return -1/+1/0 as a.length() < b.length()
        { 
            if (a.myWord < b.myWord) return -1;
            if (a.myWord > b.myWord) return 1;
            return 0;
        }
};

// to use this:
WcountComp comp;
// search using comparer
int index = bsearch(list, wc, comp);

- We can use this to sort, see strlensort.cpp
  ➤ Call of sort: InsertSort(vec, vec.size(), scomp);
```
Structuring data: `sortreadwords.cpp`

- **Search for a word using binary search**
  - Differences from sequential/linear search?

- What’s a precondition for binary search to work?

- **How can we store new words so that binary search will work?**
  - Add to end of vector and sort the vector
  - Add to end of vector and shift (down) until location found
  - Advantages of one method over another?

- What about the C++ details in using a struct/class to store data, how are comparisons made?
Review/Preview: Anagrams/Jumbles

- Brute-force approach to finding anagrams/solving Jumbles
  - Brute-force often thought of as “lack of thought”
  - What if the better way requires too much thought?
  - What if there’s nothing better?
- nelir, nelri, neirl, neirl, neril, nleir, nleri, nlier, nlrei, nlrie, nielr, nierl, niler, nilre, nirel, ... lenir, lenri, leinr, leirn, lerni, lerin, liner
  - What’s the problem here?
  - Is there a better method?
Brute force? permama.cpp

// find anagram of word in wordSource
// list is a vector [0, 1, 2, ..., n]
Permuter p(list);
int count = 0;
string copy(word);// makes copy the right length

{
    p.Current(list);
    for(k=0; k < list.size(); k++)
    {
        copy[k] = word[list[k]];
    }
    if (wordSource.contains(copy))
    {
        cout << "anagram of " << copy << endl;
        break;  // find first anagram only
    }
}
Quantifying brute force for anagrams

- On one machine make/test a word takes $10^{-5}$ seconds/word
  - $9!$ is 362,880, how long does this take?
  - What about a ten-letter word?

- We’re willing to do some pre-processing to make the time to find anagrams quicker
  - Often find that some initialization/up-front time or cost saves in the long run
  - We need a better method than trying all possible permutations
  - What properties do words share that are anagrams?
Preliminaries: C++ in permana.cpp

- **What is a dictionary?**
  - What is a class, what are the methods, why use it?
  - What properties of the class do methods depend on, class invariants?

- **What is a tvector and why is it used instead of an array?**
  - How are elements added to the vector?
  - Differences between tvector and vector (STL class)?

- **What is a Permuter and how does it work?**
  - Where is information about this class found?
  - What patterns of use does a permuter exhibit?
Toward a faster anagram finder

- Words that are anagrams have the same letters; use a letter fingerprint or signature to help find anagrams
  - Count how many times each letter occurs:
    - “teacher” 1 0 1 0 2 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0
    - “cheater” 1 0 1 0 2 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0

- Store words, but use fingerprint for comparison when searching for an anagram
  - How to compare fingerprints using operator ==
  - How to compare fingerprints using operator <

- How do we make client programmers unaware of fingerprints? Should we do this?
OO and C++ features we’ll use

- We’ll use an *adapter* or *wrapper* class called Anaword instead of a string
  - Clients can treat Anaword objects like strings, but the objects are better suited for finding anagrams than strings
  - The Anaword for “bear” prints as “bear” but compares to other Anaword objects as 11001000000000000100000000

- C++ allows us to overload operators to help, not necessary but good cosmetically
  - Relational operators == and <
    - What about other operators: >, <=, >=, and !=
  - Stream operator <<

- How should we implement overloaded operators?
Pointers

- Pointers are essential in many programming applications
  - Indirect references are often useful
    - Publish your email as foo@hotmail.com, but forward it to wherever you “really” are as you change jobs, for example
  - Allow data to be shared rather than duplicated
    - Sort a list of people/grades by name and by grade, we can maintain one list of people, and two lists of indexes, one sorted by name, one sorted by grade
  - Facilitate inheritance
    - Essential for OO
  - Implement data structures
    - Lists, trees, …
Pointer basics

- Memory is allocated *dynamically* at runtime from the heap

  - Contrast to *statically* allocated at compile time
    - Static variables take up space on the runtime stack, program executable may be large as a result
      ```
      void foo(const Date& d)
      {
          string s;
          int y;
          tvector<int> scores(20);
      }
      ```
    - Scores isn’t 20x bigger than y, why?

- Pointers reference memory, a pointer is different from the object it points to. There is a *pointer* and a *pointee*.
Syntax and semantics of pointers

```c
void foo()
{
    string s("hello");
    string * sp = new string("world");
    string * sp2;  // never do this!!!

    int slen = s.length();
    int splen = sp->length();
    // splen = (*sp).length();
    int splen2 = sp2->length();
}
```

- Memory allocated dynamically using `new`
  - What happens to `s` when `foo` terminates?
- Dereference a pointer using `*`, get at the object pointed to
  - `a->` is shorthand for `(*a)`.
- Pointers that don’t point at something are BIG TROUBLE
Pointer/Pointee confusion?

- **Pass-by-value, can we change the parameter?**

```cpp
void doStuff(Date * d) {          void doStuff2(Date * d) {
    d = new Date();             *d += 1;
}]
```

```cpp
Date * flagDay = new Date(6,14,2001);
doStuff(flagDay);
cout << *flagDay << endl;
```

- **In case things aren’t confusing enough**
  - `const Date * d;  // pointee is constant`
  - `Date * const d;  // pointer is constant`
Vectors of pointers

void readWords(istream& input, tvector<string>& list) 
   // post: all words in input stored in list 
   { 
      string word;  
      while (input >> word)  
      {    list.push_back(word); 
      }  
      cout << "read " << list.size() << " words" << endl; 
   }  

• What changes if we use tvector<string * > instead?

• What happens if we use code below which uses the address-of operator for vector of pointers (don’t do this at home)

      list.push_back(&word);
Sidebar: implementing swap in C

- Unlike C++, there are no reference parameters in C
  - Simulate pass-by-reference using pointers, what happens to actual parameters x and y in code below?

```c
void swap(string& a, string& b)
{
    string temp = a;
    a = b;
    b = a;
}

int main()
{
    string x("first"), y("second");
    swap(x, y);
}
```
Implementing swap in C

- In C we must pass pointers, and use address-of operator to simulate reference parameters, *is the picture different?*
  ‣ In C++ the pointers are hidden, harder to make mistakes?

```c
void swap(string * a, string * b) {
    string temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    string x("first"), y("second");
    swap(&x, &y);
}
```