Advanced programming techniques

Andrew Koenig
ark@research.att.com
http://www.research.att.com/info/ark
Barbara E. Moo
bmoo@worldnet.att.net

Today’s topics

- Who are we?
- What is the purpose of this course?
- What will it cover?
- What will be expected of you?

Who are we?

- Andy:
  - Programming since 1967
  - Used lots of machines and languages
  - Coauthor of Columbia Computer Chess Program, 1970–71
  - Project editor of C++ standards committee

Who are we?

- Barbara:
  - 15+ years of software management
  - Managed 1st commercial C++ compiler
  - Directed AT&T's Worldnet® ISP development program
  - Managed various data processing applications

The purpose of the course

- Learn ideas in programming and system design that transcend any one language
- Learn that programming is not just coding
- Learn by doing

Underlying philosophy

- The software universe changes fast
  - It is easy to drown in details of one system or another
  - Nevertheless, there are things we can learn that can endure
- Programming is a human activity

The most important idea

- Abstraction
  - "the act or process of separating in thought, of considering a thing independently of its associations; or a substance independently of its attributes; or an attribute or quality independently of the substance to which it belongs" (OED)
  - "leaving out of a number of resembling ideas what is peculiar to each" (attributed to Locke by Priestly, 1782)
  - Selective ignorance (A.R. Koenig, 1990's)

Subsidiary themes

- Know, don't guess
  - what problem are you trying to solve
  - prove it's broken
  - speed matters—development, compilation, execution etc.
- Stuff happens
  - design to eliminate errors
  - underpromise & overdeliver
  - test early & often

Abstraction in practice

- Reversing an array

```
i = 0; j = n-1;
while (i < j) {
      swap(a[i], a[j]);
      ++i; --j;
}
```

- What assumptions appear in this fragment that we can ignore?

Generalizing the algorithm

- Reversing a sequence

```
p = a; q = a+n;
while (p != q) {
     --q;
     if (p != q) {
          swap(*p, *q);
          ++p;
     }
}
```

- What assumptions have we removed?

Other ways of reversing

- Copy from one sequence to another, reversing as you go.
- Attach a tag to every element, then sort the sequence.
- Push the elements on a stack, then pop them.

Other forms of abstraction

- Every programming language is an abstraction of a computer
- A file system abstracts a particular style of information storage and retrieval
- A relational database abstracts a different style

Abstractions create barriers

- You can't take advantage of what you're ignoring.
- An exam question from another course:
  - The purpose of an operating system is to keep users away from the computer. Discuss.
- "Good fences make good neighbors."

Why are barriers good?

- Information flow across barriers is controlled, and thereby reduced.
- When we design a large system, we can avoid having to learn about what is on the other side of a barrier.
- We can worry only about what is on our side, and what crosses the barrier.

Abstraction is rarely free

- Constrained information flow is usually less efficient.
- “Why can’t I just reach in and tweak that variable? It’s sitting right there…”

So what do we do?

- A key to successful programming is knowing how abstract to be and when.
  - Totally concrete programs take too long to write, and don't work.
  - Totally abstract programs take too long to run, and don't do enough.
- A sense of perspective is important.
  - $O(n \log n)$ is nearly $O(n)$, but $O(n^2)$ isn't.

The point of these examples

- Abstraction is useful
  - It is better to solve the reversing problem once and be done with it
  - More generally, we need a way to cope with problems that are too big to handle all at once
- Total abstraction is impossible
  - Ignoring everything leaves us with nothing

Learn by writing programs

- that do what was asked…
- …and do it clearly…
- …and with test data that proves it….
- …and which can gracefully handle the next change.

Modifiability

- Classroom exercises are always artificial
- Still, you can often pretend that you're writing a "real program"

What are “real programs?”

- The people who want them don’t always know what they want (even if they’re the authors)
- What they want changes over time
- Successful solutions suggest new problems
- Successful programs usually got that way a little at a time

General implications

- Solutions to problems are rarely final
- When writing a program, it is important to think about how it might change
- Well written programs will take plausible future changes into account
- Each aspect that might change should appear in as few places as possible

Homework implications

- Homework assignments are unrealistically small, when compared with commercial projects
- Therefore, you should be more aggressive about imagining future changes than you might be otherwise
- Homework programs should be better organized than their size suggests

An example

- Imagine an assignment to compute the prime numbers < 10000 and print them in columns
- What changes might we imagine for future versions?

Alternative versions

- Compute something other than primes
- Compute more primes (too many to fit in memory)
- A different output format
- Do something with the primes other than print them

Modularity

- If you intertwine computation and printing, it becomes harder to change either one
- It is better to keep them separate and define a clean (i.e. as simple as possible) interface between them

Modularity example

- Right:

```
if (p is prime)
    print(p);
```

- Wrong:

```
if (p is prime) {
    buffer[n] = p;
    if (++n == buffer_size)
        flush_buffer();
}
```

What is expected of you?

- Come to class. Ask questions.
  Be critical. Think for yourself.
- Form project teams (3–6 people).
  Propose a project; get approval; do it.
- Watch the calendar.

Schedule

- Class: M, W 1:30–2:50
- Project deadlines:
  - Proposals due Friday, February 26
  - Presentations in class March 1 and 3
  - Revised proposals due Monday, March 22
  - Projects due Monday May 17 (during finals)

Course grades

- Project counts 40%; all team members get the same project grade
- Homework counts 40%; exams 20%
- Grades will usually be based on medians, not means
- Project must be complete to receive a grade at all!

Calibration

- Last year's grades:
  - A (including ±): 15
  - B (including ±): 19
  - C (± not allowed): 6
  - D (± not allowed): 1
  - F: none

Program grades

- Does it work?
  - Does it do what it is supposed to do?
  - For the project: Does it do what the proposal said it would do? Does it do more? How ambitious is it?
- How easy is it to tell that it works?
- How well is it written?

Mechanics of programming

- We will probably use C++ for most examples, explaining as we go.
- You can do homework in any language.
- You know more than we do about the local computing facilities.
- You may wish to consider language preferences when choosing project partners.

Project teams

- Form your own teams (3–6 people)
- The team picks the project (try to choose something fun and useful)
- Get started early; ask if you need help

Project proposals

- Pretend that we run a venture capital company
- The proposal is what you will use to convince us to fund your startup
- You *must* do what you proposed!
  - Don't be too ambitious
  - Make it work correctly; then add to it

Project essentials

- Estimate task duration and report actual
- Design—even (especially!) if it changes during development
- Test plan
  - You can develop the test facilities while you're developing the system
  - One person should probably work exclusively on testing
- Documentation (external and internal)
- Organization (who is doing what?)

Homework, part 1
(due Monday)

- Write a program to generate a permuted index.
  - An index in which each phrase is indexed by every word in the phrase:

The quick brown fox

```
        the quick   brown fox
The quick brown     fox
            The     quick brown fox
                    The quick brown fox
```

Suggested Strategy
(thanks to the AWK book)

- Read a line and generate rotations
  - each rotation puts a different word first and rotates previous first word to end
- Sort the rotations
- Unrotate and print the index
  - find beginning of original phrase in the rotation and put phrase together printed with appropriate formatting

Illustration

- After rotations, we should have:
  - The quick brown fox
  - quick brown fox The
  - brown fox The quick
  - fox The quick brown
- After sorting, we should have:
  - brown fox The quick
  - fox The quick brown
  - quick brown fox The
  - The quick brown fox

Illustration, continued

- When we print the output, we must remember how much each line was rotated:

  brown fox | The quick
  fox | The quick brown
  quick brown fox | The
  The quick brown fox |

- All that's left is to swap the two parts of the line when we print it.

Homework, part 2

- Assume you're using a programming language that supports strings, in which evaluating `s+t` takes *O(len(s)+len(t))* time. How long does this loop take?

```
s = "";
while (--n >= 0)
   s = s + x;
```

- Prove it.

Suggested reading (part 1)

(All published by Addison-Wesley)

- Bentley: *Programming Pearls* and *More Programming Pearls*
- Brooks: *The Mythical Man-Month*
- Gamma, Helm, Johnson, and Vlissides: *Design Patterns*
- Koenig: *C Traps and Pitfalls*
- Koenig and Moo: *Ruminations on C++*
- Lippman and Lajoie: *C++ Primer*, 3rd edition

Suggested reading (part 2)

- Reade: *Elements of Functional Programming*
- Sethi: *Programming Languages—Concepts and Constructs*
- Stroustrup: *The Design and Evolution of C++*
- Stroustrup: *The C++ Programming Language*