ECE 590.01
C++ Programming, Data structures, and Algorithms

Introduction
Admin

- Professor: Andrew Hilton
  - E-mail: adhilton@ee.duke.edu
  - Office: Hudson 211
  - Office Hours: M 11-12, Th 12:30-1:30
    - Or by appointment (e-mail me, we’ll setup a time)

- Tas:
  - Andre Van Rynbach
  - Perry Chaubal
  - Parker Kuivila
  - Patrick Pensabene
A bit about me

• Teaching is my primary job
  • Don’t be afraid to come to my office hours!
  • Don’t be afraid to ask me to setup some other office hours time!

• Please, feel free to call me “Drew”
  • If you are uncomfortable with that, Dr. Hilton or Prof. Hilton are ok
A bit about you

• Before we get too much further, I’d like you all to introduce yourselves:
  • Some of you: I had last semester
  • Everyone else: Will try to learn your names quickly
  • (Think about homework groups as we do this!)
• Go around the room:
  • Name (even if I had you before)
    • Your chance to change what I call you
  • Quick summary of your prior programming experience
    • Include what language(s)
    • E.g.,
      • “One intro class in Java 5 years ago”.
      • “No experience at all”
    • etc...
Overview

• We are going to learn **programming**
  • How to program: applicable to any language
  • Meta-cognition: thinking about how you think
  • Solve a problem -> describe algorithm -> code

• In **C++**
  • Start with straight up C: learn pointers etc
  • Then add C++: classes, templates, etc.

• With a focus on **data structures** and **algorithms**
  • Abstract data types: interface vs implementation
  • Right tools for the right job
  • Efficiency/high performance
Environment: Linux/UNIX

- We are going to do this in a Linux/UNIX environment
  - gcc/g++
  - gdb
  - make
  - Some shell scripting (bash)
  - subversion (svn)
  - emacs or vim
    - Your choice: pick one and get good at it

- Why?
  - Because we are going to make real programmers out of you!
Assignments

- 4 Homeworks: write code to do some task I describe
  - Done in groups (2 or 3)
  - Same groups for entire semester

- 1 Project: write code to do something you choose
  - Also in groups (may be different from homework group)
  - Proposal: Written document (pdf) due before spring break
    - Clearly articulate requirements/goal for project
    - Describe rough plan for how to go about it
    - Discuss any areas where your expertise is lacking that you expect to need particular help with

- 2 Exams: 1 Midterm + 1 Final
- Best 7 of 9 Quizzes (cultural reference—anyone?)
Homework/project groups

• Choose your groups carefully!

• Homework groups:
  • Can you help each other learn?
  • Do you have similar goals/ambitions?
  • What sort of skill levels would you like?
    • Comparable to yours?
    • Someone who can help you out?
    • Someone you can teach/help out?

• Project groups:
  • Should share common interest/vision
  • Ambition?
  • Skill level?
Grade Breakdown

- Homeworks: 32% (4 @ 8% each)
  - 70% of each grade: implementation
  - 30% group demos: do you understand your groups code, did you contribute, can you explain what’s going on here?

- Project: 19%
  - 5%: Proposal write-up
  - 10%: Ambition
  - 60%: Implementation
  - 25%: Demo

- Quizzes: 7%
  - 50% attendance / 50% correctness

- Midterm: 19%

- Final: 23%
Class Attendance/Quizzes

- I expect you to attend class..
  - But understand that sometimes things come up and you need to miss
- We will have 9 “quizzes” throughout the semester in lecture
  - You get 50% for being there, 50% for correct answers
  - Drop 2 lowest quiz grades (count 7 of 9)
  - Questions mostly check if paying attention/understanding discussion
    - Recommend asking questions if you are unclear on things
    - Reading in advance of lecture
    - May try out a few different quiz formats
- Drop quizzes: account for needing to miss lecture
  - Long term circumstances, please talk to me
Academic Integrity

• Academic Integrity Expectations
  • I take academic integrity **VERY** seriously, and you should too
  • Basic principles for Duke in general:
    • I will not lie, cheat, or steal in my academic endeavors, nor will I accept the actions of those who do.
    • I will conduct myself responsibly and honorably in all my activities as a Duke student.
  • If I suspect academic misconduct in my class...
    • Reported to the appropriate Associate Dean
    • Due process to determine if you did commit academic misconduct
    • If found responsible,
      • I will give you a 0 on the assignment
      • Appropriate Associate Dean may apply additional penalties
Academic Integrity: Homeworks

- Expectations for homeworks + project
  - You will work in groups of 2 or 3, so collaborate freely within group
  - Should not discuss specifics of homework solutions with other students outside your group
    - May freely discuss general class material, study for exams etc
    - Do not exchange (or look at/show) code with other groups
    - Do not get someone else to help you write it or debug it
    - Exceptions: TAs and myself
  - Outside resources for general information:
    - Can reference webpages/texts/etc for general information
      - Examples: general C syntax, Textbook, etc...
    - If you do so, cite the resource in your homework (comment in code)
    - Note that you may not download code and re-use it, even if you cite it
Academic Integrity: Exams

- Exams in this class are individual effort
  - No outside resources/help except 1 page of notes
    - No textbook
    - No talking to friends/group members
    - No text messages/cell phones/laptops/calculators/smart phones
  - Before I return your exams, I will photocopy and keep a random subset
    - If you request a re-grade, I will compare your solutions to my photocopy. If they have been changed, I will report the incident directly.

- Related exam policies:
  - Questions? Raise hand, TA or I will come to you (don’t get up)
  - Need restroom? Raise hand, we will let you go one at a time
  - No calculators/smart phones: too easy to use to chat
Academic Integrity: Mini-Quizzes

- You may use your book, notes, and/or slide print outs
  - I.e., any hard-copy written material
- You may not discuss with other class mates,
  - Talking/text messages/note passing etc.
- You may use e-readers/laptops but ONLY for course notes
  - No internet,
  - No chat
  - Don’t ruin it for everyone else
Academic Integrity: General

- Some general guidelines
  - If you don’t know if something is OK, please ask me.
  - If you think “I don’t want to ask, you will probably say no” that is a good sign it’s NOT acceptable.
  - If you do something wrong, and regret it, please come forward—I recognize the value and learning benefit of admitting your mistakes. (Note: this does NOT mean there will be no consequences if you come forward).
  - If you are aware of someone else’s misconduct, you should report it to me or another appropriate authority.
    - Within your homework group, this becomes even stronger: if you are aware that one of your group members has committed misconduct on a homework submission for your group, you are complicit in it if you do not report it.
Course Problems

• Can’t make midterms / final, other conflicts
  • Tell us early and we will schedule alternate time

• Irresolvable group problems
  • Come see me. Will allow group changes in extreme circumstances
  • Prefer you try to resolve issues first.

• Other problems:
  • Feel free to talk to me, I’m generally understanding and will try to work with you
  • Some problems may extend well beyond my course
    • Talk to your Director of Master Studies (me)/ Director of Graduate Studies (Dr. Cummer)
    • Talk to the Associate Dean of your Program
Resources

- Piazza
  - Discussions, questions, etc
  - Announcements I make: required reading
  - Other discussions: strongly recommended reading
- TBD: Assignment submission
  - Hopefully by subversion
- Course Web Page
  http://people.ee.duke.edu/~adh39/courses/spring_2013/ece590.01/
Textbook, etc.

- **Text:** *Data Structures and Algorithm Analysis in C++* (Weiss)
  - Readings to supplement lecture
  - Good to see material multiple ways
  - Readings: mostly in order
    - Section 12.2 with Chapter 4
    - Chapters 8, 11, and rest of 12 are optional
  - Dates on syllabus: approximate

- **Now:**
  - Start reading Chapter 1
Recitation

• Recitation: Fridays
  • Andre (same TA as 590.03 last fall) will lead it
  • This week: Linux + UNIX including subversion (svn)

• Strongly recommended, but not mandatory
  • Responsible for material in it
  • But no attendance taken/no quizzes

• Exception: midterm exam will be during recitation time
  • Friday March 1st, 2013
First task: Learn HOW to program

- First step of this course: learn **how** to program
  - Even if you have prior programming experience, pay attention!
  - This is how you should always approach problems at the edge of your abilities
    - Which is where you want to be working:

---

**Spectrum of Programming Problems**
Aside: Why you want to be there

• You want to work at challenging problems, why?

• Too easy:
  • Mentally un-stimulating (boring)
  • Generally could get paid more to work on harder things
  • Don’t improve at all
Aside: Why you want to be there

• You want to work at challenging problems, why?

• Too hard:
  • Frustrating, un-productive
  • Generally won’t meet goals etc.
  • Also won’t improve
Aside (cont’d): Improvement

With time and practice, this:
Aside (cont’d): Improvement

With time and practice, this:

Turns into this:
Aside (cont’d): Improvement

With time and practice, this:

Turns into this:

And eventually this:

And now your ideal spot is what was previously impossible
Variety of backgrounds

- You all have a wide variety of backgrounds
  
  Some have very little experience
  
  Some have a moderate amount
  
  Some a fair bit
Too much background

• Really strong programmer?
  • Should not be in this course!

• Aside:
  • Some of you are registered for compilers...
  • Need to be a pretty strong programmer for that
  • Probably should not be in both classes!
How we will deal with this

• For the less experienced students
  • “Ramp up” problems: Extra credit with a limit
  • Easier problems, bring you up to speed
  • Worth extra credit if your final letter grades is below a threshold
    • E.g., only counts if below a ‘B’ can bring you from C+ to B-, or B- to B, but not B to B+.
  • Individual effort (not done in groups!)

• For the stronger students
  • Last 10 points of each assignment are harder/require learning something new on your own/making assignment polished/etc.
Algorithms vs Code

• Two distinct things to keep separate:
  • Writing an algorithm
    • Clear set of step-by-step instructions
    • Solves any problem in a certain class of problems
    • Parameterized to identify which particular problem
  • Implementing that algorithm in a programming language
    • Translating the steps into the syntax of a particular language
    • As well as testing, debugging, etc...

• Note that once you have an algorithm, its generally easy to translate into any language you want
“Solves any problem in a certain class of problems”
  - Why not just “solve a problem?”
  - What do I mean here?

We generally don’t want to write an algorithm that solves just one problem:
  - E.g., “Is 7 prime?”

Instead, we want to have an algorithm for a general class of problems:
  - E.g., “Given a number N, is N prime?”
  - Note that this algorithm is parameterized over N.
5 steps to writing a program

1. Work an instance of the problem yourself
   - Maybe a few to get the feel of it if it's hard
   - Can't do this? Either need domain knowledge or clarification

- Domain knowledge:
  - Knowledge specific to problem domain: chemistry, physics, biology, micro-architecture, ...
  - If you don't have the relevant domain knowledge, you can't hope to write a program about it—and should find this out in step 1.
  - If you are stuck here, you should read up or seek out a domain expert...
5 steps to writing a program

1. Work an instance of the problem yourself
   - Maybe a few to get the feel of it if it's hard
   - Can't do this? Either need **domain knowledge** or clarification

2. Write down exactly what you did to solve that instance
   - In a level of sophistication that a small child who is good at math (and precisely obedient) could perform
5 steps to writing a program

1. Work an instance of the problem yourself
   - Maybe a few to get the feel of it if it's hard
   - Can't do this? Either need **domain knowledge** or clarification

2. Write down exactly what you did to solve that instance
   - In a level of sophistication that a small child who is good at math (and precisely obedient) could perform

3. Generalize your steps
   - Find repetitions
   - Determine what numbers depend on parameters
5 steps to writing a program

1. Work an instance of the problem yourself
   - Maybe a few to get the feel of it if it's hard
   - Can't do this? Either need **domain knowledge** or clarification

2. Write down exactly what you did to solve that instance
   - In a level of sophistication that a small child who is good at math (and precisely obedient) could perform

3. Generalize your steps
   - Find repetitions
   - Determine what numbers depend on parameters

4. Test your generalized steps on another instance
   - Generalized wrong? Find it now!
5 steps to writing a program

1. Work an instance of the problem yourself
   • Maybe a few to get the feel of it if its hard
   • Can’t do this? Either need **domain knowledge** or clarification

2. Write down exactly what you did to solve that instance
   • In a level of sophistication that a small child who is good at math
     (and precisely obedient) could perform

3. Generalize your steps
   • Find repetitions
   • Determine what numbers depend on parameters

4. Test your generalized steps on another instance
   • Generalized wrong? Find it now!

5. Translate steps to code
   • The only part that depends on the programming language!
5 steps to writing a program

1. Work an instance of the problem yourself
   • Maybe a few to get the feel of it if it's hard
   • Can't do this? Either need **domain knowledge** or clarification

2. Write down exactly what you did to solve that instance
   • In a level of sophistication that a small child who is good at math (and precisely obedient) could perform

3. Generalize your steps
   • Find repetitions
   • Determine what numbers depend on parameters

4. Test your generalized steps on another instance
   • Generalized wrong? Find it now!

5. Translate steps to code
   • The only part that depends on the programming language!

---

Steps 1—4: develop algorithm
No really, plan before you code

• How do you build a skyscraper? (Or even a house?)
  • Option 1: Start building, figure out where things go as you build
  • Option 2: Have an architect carefully plan everything out, developing a blue print that specifies every detail of the construction. Get it approved by the city etc... THEN break ground and start building.

• Sadly, most programmers choose the analog of option 1
  • Note that is OK iff you are in the “green zone” and can do steps 1—4 trivially in your head.
Big systems: Abstraction

• Abstraction: separate interface from implementation
  • Key to any large system (sound familiar?)

• In software, divide things up into modules/functions
  • Plan your algorithm assuming other modules obey their interface

• Step in algorithm is complex?
  • Abstract it out into a function
  • Have clear idea of what it does (interface)
  • Then come back and figure out its implementation
  • Just make sure its behavior is well defined and implementable
Top-down design

- Top-down design:
  - Start with algorithm for main:
    - Check and process the options
    - Read the configuration file
    - As long as there is input
      - Process the input
      - Print the result
    - Free resources

Most of these steps turn into their own functions

```c
check_and_process_options(argc, argv);
read_config_file(config_filename);
```

... 

Repeat for all the functions you assume you have.
To be continued

- We will talk more about devising algorithms next time
- For now, a few more admin things...
Ground Rules

- Your code must compile
  - With `gcc --std=gnu99 --pedantic -Wall -Werror` on Linux
    - (or `g++ --std=gnu99 --pedantic --Wall -Werror` )
  - Must have a Makefile (more on this later) which builds everything
  - Don’t compile? Max score: 5/60 on implementation for “main” part of homework
    - 0/60 for last 10 points

- Your program must valgrind cleanly
  - No memory leaks
  - No use of un-initialized values
  - Valgrind issues? 5 to 15 points off (2—7 on L10P)

- You should code defensively
  - Handle un-expected problems gracefully: give errors, act sanely
  - Segfault? 5 to 20 points depending on conditions (1—9 on L10P)
Breaking down the rules

- **gcc (or g++):** compilation program
  - Turns C source code into an executable binary (more later)
- **--std=gnu99**
  - We’ll use C99 with Gnu extensions
  - Allows for (int i = 0 ... and //comments
- **-pedantic –Wall**
  - Apply the rules pedantically (conform to the rules strictly)
  - Warn about everything possible (Wall = W[arn] all)
- **-Werror**
  - Treat warnings as errors: don’t let you get away with them
Valgrind

- Valgrind
  - Checks for memory leaks (more later)
  - And other runtime errors (invalid memory access, use of uninitialized values)
  - Should get a report that looks like this:

```
==28511== HEAP SUMMARY:
==28511==     in use at exit: 0 bytes in 0 blocks
==28511== total heap usage: 38 allocs, 38 frees, 2,369,944 bytes allocated
==28511== All heap blocks were freed -- no leaks are possible
==28511==
==28511== For counts of detected and suppressed errors, rerun with: -v
==28511== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```
Why these rules?

- Non-compiling code isn’t “almost there”...
  - “I almost got it to compile” = “I almost started testing and debugging it”

- “I almost built you a house: the permit office says I only need to change these 10 things about the plans before I can start building”

- Getting your code to compile should NOT be hard
  - If you are struggling with this, your code almost certainly is nowhere near working

- Careless mistakes at the end (accidently changed something)?
  - “Sorry Mr. Customer, we really had this working a week ago, its only what we delivered that is broken...”
Why these rules?

• Non-compiling code isn’t “almost there”...
  • “I almost got it to compile” = “I almost started testing and debugging it”
  • Getting your code to compile should NOT be hard
    • If you are struggling with this, your code almost certainly is nowhere near working

• Bad software kills people
  • Seriously: Therac 25, Patriot Missile Failure, Panama NIO radiation overdoes,...
  • Also, non-fatal, but really bad consequences: $440 M trading error for Knight Capital Group, Ariane 5 explosion, phone system failure, Mariner 1, east coast blackout of 2003, 1998 Mars orbiter,...
  • Therefore, I want you all to learn to write good software:
    • No sloppiness
    • Code defensively
Semi-rules

- A few “semi-rules”
  - Can’t/won’t really enforce these, but you really really should

- Debug in gdb
  - Learn it
  - Love it
  - Become an expert
  - Debugging skill distinguishes OK programmers from great programmers

- Learn and use a programmer’s editor: emacs or vim
  - I use emacs
  - Andre uses vim
  - Feel free to try both, pick one and become an expert at it
Programmer’s editors: Emacs or vim

• “But I like <eclipse, visual studio, ...>”
  • That is only because you do not know the power of a real editor
  • Also, every job I’ve ever had except one I’ve used emacs
    • And everyone has either used emacs or vim

• IDEs edit one or a few languages decently
  • Vim/emacs edit everything well:
    • C, C++, Java
    • SML
    • Scheme/LISP
    • Assembly
    • LaTeX
    • ...

Programmer’s Editors: continued

- Emacs and vim are also ubiquitous
  - Almost every Linux/UNIX system has both of them
  - My experience: Had to work on 1 AIX system with only vim
    - Wrote most of my code in emacs, scped it, and made minor edits
- They are also useable across remote connections
  - Graphical IDE over X-11 forwarding (or VNC) from halfway around the world? Soooo painful
  - Note: working on remote systems incredibly common in the real world
- So: my advice to you
  - Learn emacs and/or vim
  - Become an expert in ONE (you can’t be a true expert in both):
    muscle memory
Wrap-up there for today

• We’ll end here for today

• Friday (recitation):
  • Linux/Unix

• Monday (lecture):
  • Algorithms
  • How to approach ANY programming problem you ever have

• Soon:
  • Homework 1