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

Admin
- Errors in original schedule (Spring break off by 1 month)
  - Revised homework/reading/etc data on corrected syllabus on web
  - Midterm exam: 3/1/2013 (Friday)
    - Not the day before spring break!
  - Project Proposals: 2/15/2013
- Piazza up—discussion board
- Homework 1 out soon
- Speaking of homework...

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
    - 0/50 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)

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

```plaintext
==28511== HEAP SUMMARY:
==28511==     in use at exit: 0 bytes in 0 blocks
==28511==      total heap usage: 38 allocs, 38 frees, 2,349,844 bytes
==28511==  all heap blocks were freed -- no leaks are possible
==28511== For counts of detected and suppressed errors, rerun with: -v
==28511== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

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

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 part 2?

- 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
    - Write most of my code in emacs, scp’d 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? Sooo 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

Picking up where we left off

- Who can remind us what we were talking about last time?

Picking up where we left off

- Who can remind us what we were talking about last time?

  - 5 steps to write a program
    - Useful anytime you writing a "hard but doable" program
    - Good to learn even if you have some experience
      - May not have thought of this explicitly before
    - Step 1?
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

   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…

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 blueprint 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 if the problem is in your "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
    check_and_process_options(argc, argv);
    read_config_file(config_filename);
  • ... Repeat for all the functions you assume you have.

Bottom-up design

• Other alternative: bottom-up design
  • Build all the smaller pieces you think you need
  • Then have them put together
  • How do you know what you need?
    • Still have to think about the big picture first somewhat
    • Just don’t fill in the higher level algorithms yet

Very basic example: isPrime

• Let’s see these steps in action on a relatively simple problem:
  • Given a number \( N \), is \( N \) prime?

• Step 1?

• Step 1?
  • Work (at least) one instance of the problem ourselves by hand
  • Is 7 prime?
  • Note: "yes, I just know it is" is not the way to go
  • If you can only come up with "I just know it," try larger: is 94723 prime?
  • Probably won’t work it all out, but might get the gist of what to do for 7.
Is 7 prime?

- We'll let's see:
  - \(7/2 = 3 \text{ R } 1\)
  - \(7/3 = 2 \text{ R } 1\)
  - \(7/4 = 1 \text{ R } 3\)
  - \(7/5 = 1 \text{ R } 2\)
  - \(7/6 = 1 \text{ R } 1\)

- Yes, 7 is prime
- Step 2?

Is 7 prime?

- We'll let's see:
  - \(7/2 = 3 \text{ R } 1\)
  - \(7/3 = 2 \text{ R } 1\)
  - \(7/4 = 1 \text{ R } 3\)
  - \(7/5 = 1 \text{ R } 2\)
  - \(7/6 = 1 \text{ R } 1\)

- Yes, 7 is prime
- Step 2?
  - Write down exactly what we did
  - May require thinking about things you did intuitively

Step: Write down what we did

- Steps to see if 7 is prime:
  - Check if 7 is divisible by 2 (its not)
  - Check if 7 is divisible by 3 (its not)
  - Check if 7 is divisible by 4 (its not)
  - Check if 7 is divisible by 5 (its not)
  - Check if 7 is divisible by 6 (its not)

- Answer "yes"

- Note that these steps work for 7 (and only for 7)
  - Really boring as an algorithm: no parameters, kind of useless
  - "Certificate" that 7 is prime: don't believe me? Find a step you disagree with.
  - Not our focus, but nice aside

Step 3: Generalize

- Steps to see if \(N\) is prime:
  - Check if \(N\) is divisible by 2 (its not)
  - Check if \(N\) is divisible by 3 (its not)
  - Check if \(N\) is divisible by 4 (its not)
  - Check if \(N\) is divisible by 5 (its not)
  - Check if \(N\) is divisible by 6 (its not)

- Answer "yes"

- What to have steps to see if \(N\) is prime
  - Look for patterns, repetition
  - Figure out why each number is what it is

- These are all ??
  - Are they always ? for any value of \(N\)?
  - No: they are just 7 because we picked \(N=7\)
  - In general, want to check divisibility of \(N\)
### Step 3: Generalize

**Steps to see if \( N \) is prime:**
- Check if \( N \) is divisible by 2 (its not)
- Check if \( N \) is divisible by 3 (its not)
- Check if \( N \) is divisible by 4 (its not)
- Check if \( N \) is divisible by 5 (its not)
- Check if \( N \) is divisible by 6 (its not)
- Answer "yes"

**These are all 7?**
- Are they always 7 for any value of \( N \)?
- No: they are just 7 because we picked \( N=7 \)
- In general, want to check divisibility of \( N \)

**What about 2..6?**
- Do we always check exactly these?
- No...
  - Where do we start?
  - Where do we end?

---

### Step 3: Generalize

**Steps to see if \( N \) is prime:**
- Count from 2 to \( N-1 \) (inclusive), for each number \( X \) that you count
  - Check if \( N \) is divisible by \( X \) (its not)
  - Answer "yes"

**What about 2..6?**
- Do we always check exactly these?
  - No...
  - Where do we start? Always 2
  - Where do we end? \( N-1 \)

---

### Step 3: Generalize

**Steps to see if \( N \) is prime:**
- Count from 2 to \( N-1 \) (inclusive), for each number \( X \) that you count
  - Check if \( N \) is divisible by \( X \) (its not)
  - Answer "yes"

**What about 2..6?**
- Do we always check exactly these?
  - No...
  - Where do we start? Always 2
  - Where do we end? \( N-1 \)

---

### Step 3: Generalize

**Steps to see if \( N \) is prime:**
- Count from 2 to \( N-1 \) (inclusive), for each number \( X \) that you count
  - Check if \( N \) is divisible by \( X \) (its not)
  - Answer "yes"

**What about 2..6?**
- Do we always check exactly these?
  - No...
  - Where do we start? Always 2
  - Where do we end? \( N-1 \)

---

### Step 3: Generalize

**Steps to see if \( N \) is prime:**
- Count from 2 to \( N-1 \) (inclusive), for each number \( X \) that you count
  - Check if \( N \) is divisible by \( X \) (its not)
  - Answer "yes"

**What about 2..6?**
- Do we always check exactly these?
  - No...
  - Where do we start? Always 2
  - Where do we end? \( N-1 \)

**These steps look reasonable:**
- Clear/simple/straightforward: no ambiguity, very step-by-step
- Are they right?
  - Difficult: prove it correct (we’re not doing this)
  - Become more confident, but never certain: test it
  - Testing can find the presence of errors, not the absence.
Step 4: Test your generalized steps

- Try out your steps on other values
- Make sure you can get some "yes" answers and some "no" answers
- Are there any corner cases?
  - Where your algorithm has to do something special for a special value?
  - Test those explicitly
- Get at least "statement coverage"
  - Should test every step at least once
- For this, what should we test with?

Step 4: Test with 0

- Steps to see if \( N \) is prime:
  - Count from 2 to \( N-1 \) (inclusive), for each number \( X \) that you count
    - Check if \( N \) is divisible by \( X \)
      - If so: stop and answer "no"
    - If not: (nothing special, keep going)
    - Answer "yes"
  - Let's see: 0 is NOT prime (infinite divisors)
    - Count from 2 to -1?
      - We actually implicitly assumed we were counting up...
      - At least, I did...
      - So nothing in this range
    - Get an answer of "yes"—oops.

Step 4a: Fix our generalized steps

- Steps to see if \( N \) is prime:
  - If \( N \) is less than or equal to 1, stop and answer "no"
  - Count from 2 to \( N-1 \) (inclusive), for each number \( X \) that you count
    - Check if \( N \) is divisible by \( X \)
      - If so: stop and answer "no"
      - If not: (nothing special, keep going)
    - Answer "yes"
  - Fix algorithm:
    - All primes are > 1
    - So we can answer "no" immediately for \( <= 1 \)
    - What about \( N=2.76 \) or \( N="hello world" \) or \( N=false \)?
    - Wrong type of data: we're only concerned with strongly typed languages where this is not permissible

Step 5: translate to code

- Steps to see if \( N \) is prime:
  - If \( N \) is less than or equal to 1, stop and answer "no"
  - Count from 2 to \( N-1 \) (inclusive), for each number \( X \) that you count
    - Check if \( N \) is divisible by \( X \)
      - If so: stop and answer "no"
      - If not: (nothing special, keep going)
    - Answer "yes"
  - Language dependent:
    - Let's do Java first

```java
boolean isPrime(int N) {
    // ... (same as previous steps)
}
```

- We mostly just go line by line
Step 5: translate to code

boolean isPrime(int N) {
    if (n <= 1) { return false; }
    for (int X = 2; X <= N-1; X++) {
        if (isDivisibleBy(N,X)) {
            return false;
        } else {
            (nothing special, keep going)
        }
    }
    return true;
}

Step 5: translate to code

boolean isPrime(int N) {
    if (n <= 1) { return false; }
    for (int X = 2; X <= N-1; X++) {
        if (isDivisibleBy(N,X)) {
            return false;
        } else {
            (nothing special, keep going)
        }
    }
    return true;
}

Step 5: translate to code

boolean isPrime(int N) {
    if (n <= 1) { return false; }
    for (int X = 2; X <= N-1; X++) {
        if (isDivisibleBy(N,X)) {
            return false;
        } else {
            (nothing special, keep going)
        }
    }
    return true;
}

Step 5: translate to code

boolean isPrime(int N) {
    if (n <= 1) { return false; }
    for (int X = 2; X <= N-1; X++) {
        if (isDivisibleBy(N,X)) {
            return false;
        } else {
            (nothing special, keep going)
        }
    }
    return true;
}
Might clean this up a bit

```java
boolean isPrime(int n) {
    if (n <= 1) { return false; }
    for (int x = 2; x <= n-1; x++) {
        if (isDivisibleBy(n, x)) {
            return false;
        }
    }
    return true;
}
```

Other languages?

- C/ C++: almost identical

**isPrime: Java**

```java
boolean isPrime(int n) {
    if (n <= 1) { return false; }
    for (int x = 2; x <= n-1; x++) {
        if (isDivisibleBy(n, x)) {
            return false;
        }
    }
    return true;
}
```

**isPrime: C++**

```java
bool isPrime(int n) {
    if (n <= 1) { return false; }
    for (int x = 2; x <= n-1; x++) {
        if (isDivisibleBy(n, x)) {
            return false;
        }
    }
    return true;
}
```

**isPrime: C**

```c
int isPrime(int n) {
    if (n <= 1) { return 0; }
    for (int x = 2; x <= n-1; x++) {
        if (isDivisibleBy(n, x)) {
            return 0;
        }
    }
    return 1;
}
```

**isPrime: SML**

```sml
fun isPrime n =
    if n <= 1
    then false
    else let fun
        lprimex =
            if x <= n-1
                then if isDivisibleBy(n, x) then false
                    else lprimex (x+1)
                else true
            in
                lprimex 2
            end
    in
        lprimex n
    end
```
Code: what next?

- Step 5: write the code
  - Step 5a: test and debug the code
    - Fairly confident our algorithm is right...
    - But could be wrong for cases we didn't test
    - Implementation of the algorithm may also be wrong
    - Start with same test cases as in step 4
      - They don't work? Implementation problem
      - Try other test cases
        - Easier to large # of tests on code: can automate it
        - Problems: Algorithm or implementation?
        - Can re-work algorithm by hand to see
        - More on debugging soon

Testing

- Even if the problem is easy, test your code
  - Test is really well
  - Mistakes are easy to make
  - Cannot be certain of correctness from testing, but more confidence that it is correct is important

- Useful technique: random testing
  - Want to test cases you didn't think of...
    - But you won't think of writing test cases for those!
    - Write a random test generator: it may come up with cases you didn't
      - Have used this with great success in The Real World (ITRW).
      - Bigger picture-wise: writing extra tools can be helpful

Algorithms: common patterns

- Many algorithms are quite similar
  - Recognizing common patterns can help you design algorithms in the future

- Figure out if something is true of all things in a range/set/list/etc...
  - Check them all: find one that violates the condition—return false
  - After checking them all, return true

- Figure out if something is true of any things in a range/set/list/etc...
  - Check them all: find one that meets the condition—return true
  - After checking them all, return false

Rest of the semester overview

- We'll see more patterns over the semester
  - Discuss as they come up

- For now, wrap up with a quick overview of the rest of the semester
  - Next two weeks: learn C
    - Have a language to write these in
    - Will build into C++ later

- Data structures
  - Places to hold data for more complex/interesting manipulation
  - Abstraction: interface vs. implementation
Semester overview (cont’d)

- Datastructures (cont’d)
  - Start simple (linked lists)
  - And get more complex (BSTs, Hashables, Heaps)
  - Also, switch to C++: templates, inheritance, etc...
  - And Big-O so we can talk about efficiency on large data sets
- Brief discussion of sorting
- Some more data structures
  - Graphs: In depth
  - Touch briefly on others
- Advanced topics in inheritance
  - Object layout, virtual inheritance, mixins
- Concurrency
  - Multi-threading, locks, lock-free data structures

Next time: C

- Next time: we start on C
- Readings: Chapters 1 and 2