CMPS 12B Syllabus

Prof. Darrell Long
Department of Computer Engineering

Spring 2017

Course Objectives

This course presents an introduction to programming and data structures primarily using the **C** programming language, including basic data structures such as linked lists, queues, trees, hashing, sorting algorithms and basic computational complexity.

It assumes that you know the material covered in CMPS 12A, including the Java programming language. Java is an *object-oriented* language while **C** is not; Java is a managed language while **C** gives you access to (very nearly) the bare machine. Some faculty feel that the primary goal of this course is to teach you good object-oriented programming, I disagree, I believe it is to teach you *algorithms*, *abstraction* and *data structures*. For a thorough dose of object-oriented methodology you should take CMPS 109.

This course will be relatively fast-paced compared to CMPS 5J and CMPS 11, or CMPS 12A. Do not get behind in your reading, or your programming. Take notes: I use very few slides, and even if I did you *cannot pass* by just looking at the slides. You need to be active in learning, not just a passive consumer.

You will also learn important tools including UNIX (Linux), `git`, `make`, `cc`, `lldb`, `vi` or `emacs`, and of how to read and use `man` pages. These will be covered in detail in the laboratory portion (CMPS 12M).

Contacts

- Prof. Darrell Long, `darrell@ucsc.edu`
- Bharath Nagesh, `bnagesh@ucsc.edu`
- Juraj Juraska, `jjuraska@ucsc.edu`
- Nehal Bengre, `nbengre@ucsc.edu`
- Shubham Goel, `sgoel2@ucsc.edu`
- Sneha Das, `sndas@ucsc.edu`

Approximate Schedule

The pace of the course depends on a great extent to the degree that you as students engage in the class. We have basic material, as listed below, that we must cover. Beyond that there is a wealth of material that is in the domain of CMPS 12B that we can cover as well.

Week 1: Introduction to **C**, computing fundamentals

Week 2: Statements, functions, control flow, introduction to complexity

Week 3: Arrays and Bit Vectors

Week 4: Linked Lists

Week 5: Stacks and Queues

Week 6: Sorting

Week 7: Algorithm analysis

Week 8: Heaps and priority queues

Week 9: Binary trees

Week 10: Hashing

Grading

- 10% comes from weekly *notes* of the material covered in class. These must be turned in via `git` each Saturday by 0100 PDT.

  You probably have not encountered this before, but I have found that it really improves the learning (and grades) of students that do it.

- 20% comes from midterm examination. This will be closed-book, and you must be prepared to show identification.

- 20% comes from comprehensive final examination. This will be closed-book, and you must be prepared to show identification.

- 50% comes from weekly to bi-weekly programming assignments (the laboratory portion, CMPS 12M, is entirely programming). These must be turned in via `git` on the specified date by 0100 PDT.

Required Text

To do well you must *read the books* for this course, and so you should buy or rent copies of these books. I will assume that when I assign reading that you have done it; if you do not do the reading, then you bear the consequences.

- Janet Prichard (Author), Frank M. Carrano, *Data Abstraction and Problem Solving with Java: Walls and Mirrors*, Third Edition, ISBN 978-0132122306, 2010.

  I will be honest with you: I do not love this book. It is too long, but it is what has been used, and it presents the algorithms in Java, which should make it easier for you to follow. But it will require you to re-imagine the algorithms in **C**.

- Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, Second Edition, ISBN 978-0-131-10362-7, 1988.

  This is the original reference on the **C** programming language. You can use others (that are much longer) if you like, but this one is sufficient. I know that you can find a PDF on the Internet, but please, *do not steal*. Authors deserve to be paid for their work, just like you will want to be paid at your job.

Recommended Texts

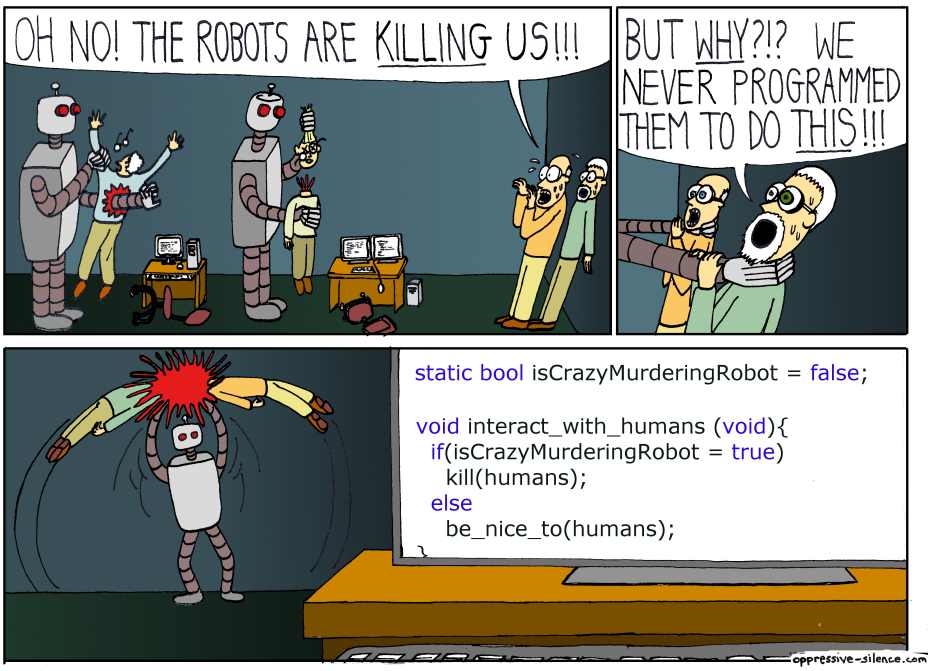
A large part of CMPS 12M (the laboratory component) will be learning how to use common programming tools. To be a Computer Scientist or Computer Engineer you must be competent with these tools.

- Arnold Robbins, *vi and Vim Editors* (Pocket Reference), O'Reilly, Second Edition, ISBN 978-1-449-39217-8, 2011.
- Richard E. Silverman, *Git Pocket Guide*, O'Reilly, ISBN 978-1-449-32586-2, 2013
- Arnold Robbins, Elbert Hannah and Linda Lamb, *Learning the vi and Vim Editors*, O'Reilly, Seventh Edition, ISBN 978-0-596-52983-3, 2008.
- Cameron Newham, *Learning the bash Shell:* UNIX *Shell Programming*, O'Reilly, ISBN 978-0-596-00965-6, 2005.
- Jon Loeliger and Matthew McCullough, *Version Control with Git: Powerful tools and techniques for collaborative software development*, O'Reilly, Second Edition, ISBN 978-1-449-31638-9, 2012.

Writing Good Code

One of the primary goals of this course is to begin to teach you to think clearly and logically. As you will hear me say, "*Simplicity is the queen of the virtues, and Clarity her sister*," which means that unless there is a compelling reason not to do so your code should be as simple, clear and elegant as possible. Others should be able to read it and immediately understand what you are doing.

We are working in **C**, and it is a dangerous programming language. That means that, unlike Java, it will do little to protect you when are not careful with the code that you write. It will check basic types, but it will not check that you have not run past the end of an array, whether a pointer is valid, or whether you used the correct operator when you wrote `if (x = 0)` when you really meant to write `if (x == 0)`. If what you write is valid **C**, then the compiler assumes that you know what you are doing and will emit code to do it.



By using `git` you can track the changes that you make, and back up to a working version if you break (or accidently delete) your program. Revision control programs like `git` are essential tools (some IDEs even integrate them).

Do not be afraid of recursion. Your goal is clarity and simplicity. If the most natural or clear expression of an algorithm is recursive, then use it. Do not worry too much about efficiency in most cases since the compiler can usually transform most recursion (which is tail recursion) into iteration. The cost of recursion versus an explicit stack is negligible.

Consider for example,

```
// Compute n! = n(n - 1)!
uint64_t f(uint64_t n)
{
        if (n == 0) { return 1; }
        else        { return n * f(n - 1); }
}
```

versus:

```
// Compute n! iteratively.
uint64_t f(uint64_t n)
{
    uint64_t t = 1;
        for (int i = n; i > 1; i = i - 1)
```

```
        {
                t = t * i;
        }
    return t;
}
```

Although this is a trivial example, ask yourself: Which is easier to understand? Which conforms most directly to the definition of factorial?

Now consider Ackerman's function:

$$A(m,n) = \begin{cases} n+1 & m = 0 \\ A(m-1,1) & m > 0 \wedge n = 0 \\ A(m-1,A(m,n-1)) & \text{otherwise.} \end{cases} \tag{1}$$

It is easily written recursively, but writing it iteratively is a challenge to say the least!

```
uint64_t A(uint64_t m, uint64_t n)
{
        if (m == 0)                { return n + 1; }
        else if (m > 0 && n == 0) { return A(m - 1, 1); }
        else                       { return A(m - 1, A(m , n - 1)
    ); }
}
```

Academic Honesty

Academic honesty is very important in computer science, and college in general. The goal of this course is for you to learn the material, not simply for you to get a mark on your transcript saying you passed the class. Because we've had issues with academic honesty in the past (not unique to this course or even UCSC—CS faculty at other universities have had similar issues), we felt it was necessary to be specific about specific scenarios.

All students in the class must sign and turn in an acknowledgment that they understand the cheating policy for the class. We will not accept assignments from a student unless she or he has turned in a signed agreement. Of course, we encourage you to ask for clarifications in the academic policy if you have any questions.

Basics

These rules apply to everyone who isn't explicitly in your project group, and only apply to graded assignments. Programming projects, notes and examinations, however, are graded, so these rules apply.

1. You may not work on your assignment with anyone.
2. You may not show your code or design to anyone.

3. You may not have anyone "walk you through" an assignment, describe a solution in detail, or sit with you as you work on it.

4. You may not provide such assistance to anyone, either. This includes friends, family members, tutors, current and former students, paid consultants, and random people on the Internet.

5. You may not post code or questions from your project on-line to ask others for help. This means anywhere on-line, including Piazza (ask us in person), independent message boards (*e.g.* StackExchange) and file sharing sites.

There are cases where you're allowed to get help from someone else that might be useful for a project, as the scenarios below show. That's why you are required to document any help you get for each assignment you turn in. This doesn't apply to help from the course staff, assigned project partners, course textbooks, course web site, and code and documentation (*e.g.* manual pages) supplied with Linux, all of which you may freely use without explicit acknowledgment.

If you get solutions from somewhere you shouldn't and you documented your source when you turned in the assignment, we won't consider it cheating. We may give you less credit (or no credit) for that part of the assignment—after all, the assignment was for you to do it—but we won't report you for cheating, either. Of course, this only applies if you document your source when you turn the assignment in; doing so when you get caught doesn't count.

Bottom line: document your sources.

Scenarios

These scenarios are *shamelessly stolen* from CMPS 111. For all these scenarios, assume that Alice and Bob are not in the same project group. Since they're not, it doesn't matter whether one of them isn't a student.

Getting `git` to work

Alice can't figure out how `git` works, even after reading the on-line guide to `git`. She asks Bob for help in getting her repository set up on her installation of FreeBSD.

Is this OK? Yes, it's OK for Bob to help Alice to get `git` to work because `git` isn't related to a particular assignment, but rather is more general.

Understanding existing code

The second assignment requires that you modify the FreeBSD scheduler to use lottery scheduling. Alice and Bob sit down together to figure out how the existing scheduler works—what files are involved, and what the existing routines do.

Is this OK? Yes, it's OK for Bob and Alice to figure out how the existing code works. They could do this without seeing the current assignment—they might want to better understand a part of FreeBSD—and they don't need to know the details of the assignment to do so.

However, Alice and Bob must be careful not to discuss details of how they might change the scheduler; they should stick to discussing how the existing one works.

Planning an approach

Alice and Bob now decide that, since they understand how the existing code works, they should figure out how they're going to approach the problem. They write up their designs separately. However, Bob isn't confident about his design, so he asks Alice to look it over and give him some feedback.

Is this OK? No, this is not OK, and would be considered a violation of the academic honesty policy. It's a violation of Rules 2 and 3.

Debugging

Bob is having trouble with the `execvp()` system call, so he copies the line of code out of his shell program (Assignment 1) into a Piazza post and asks for help understanding why it's not working.

Is this OK? No, this is not OK, and would be considered a violation of the academic honesty policy. It violates Rule 4.

Understanding system calls

Bob is still having trouble with the `execvp()` system call, so he asks his friend Carlos to explain how it works.

Is this OK? As long as Carlos hasn't seen or heard any details about the assignment itself, this is OK. Bob is just asking for background information that happens to be relevant for his assignment, but Carlos can't do the assignment for him.

However, if Carlos knows the details of the assignment or Bob provides his code, it's likely this is a violation of the academic honesty policy because of Rule 3. The only exception might be if Carlos has seen the assignment but carefully avoids referencing it. Course staff are good at this, but it's unlikely that others will be less careful, causing an academic honesty issue. This is why you document all sources of help!

Asking targeted questions

While doing her shell program (Assignment 1), Alice wants to know how to break an input string into tokens (individual words). She asks her friend Carlos, who graduated last year and now works at Google, to show her how to do this. Carlos helps her out by showing her sample code that accomplishes the goal. Alice looks over the code and then goes home and writes her own code based on what she saw.

Is this OK? At a minimum, Alice needs to document the help she got from Carlos, regardless of what else she does. If she didn't keep a copy of Carlos's code and waited for at least an hour before starting on her own code, this is OK—she's doing the work herself. But if she printed the code and used it as a template, or works off highly detailed notes she took during the help session, this violates Rule 3.

Previous assignments

Bob has a friend who took the class last year. He sees that this year's assignment is somewhat different from last year's, so he asks his friend Carlos to give him a copy of a solution to last year's assignment.

Is this OK? No, this is not OK—it violates Rule 3, and is a violation of the academic honesty code. It doesn't matter that the assignment is different from last year's, since there are very likely significant overlaps between the assignments.