The catalog description

CS 2303. SYSTEMS PROGRAMMING CONCEPTS. This course introduces students to a model of programming where the programming language exposes details of how the hardware stores and executes software. Building from the design concepts covered in CS 2102, this course covers manual memory management, pointers, the machine stack, and input/ output mechanisms. The course will involve large-scale programming exercises and will be designed to help students confront issues of safe programming with system-level constructs. The course will cover several tools that assist programmers in these tasks. Students will be expected to design, implement, and debug programs in C++ and C. The course presents the material from CS 2301 at a fast pace and also includes C++ and other advanced topics. Recommended background: CS 2102, CS 2103, or CS 2119 and/or substantial object-oriented programming experience.

Outline

- Let's look at the Canvas site.
- Syllabus organizes the activities according to the calendar.
- In Files, we find grading.xslx – you can enter your scores and estimate your grade
- We'll have in class quizzes on Tuesdays.
- We'll have labs on Wednesdays, which will start you on the homework.
- We'll discuss in lecture on Fridays one way to make progress on the homework.
- Homework is due on Monday. These are the large-scale programming exercises.
- How the hardware stores and executes software.
- Memory management
  - pointers
  - machine stack
  - input/output mechanisms
- Safe programming with system-level constructs
- Tools

Our Canvas site starts with the syllabus.

- Day by day, you can find
  - how to prepare
  - what to expect
  - what to do afterwards

Quizzes will test you on comprehension of concepts from reading and lecture.

- There is a quiz template.
- For the whole course at 100 points, each quiz is worth 2.5 points.
- There will be approximately 7 questions per quiz.
- Unless a dispensation is obtained in advance, quizzes are taken in class.

Sample quiz questions I

1. (0.357 points) Who or what creates function prototypes? Explain.

2. (0.357 points) What does a function prototype contain?

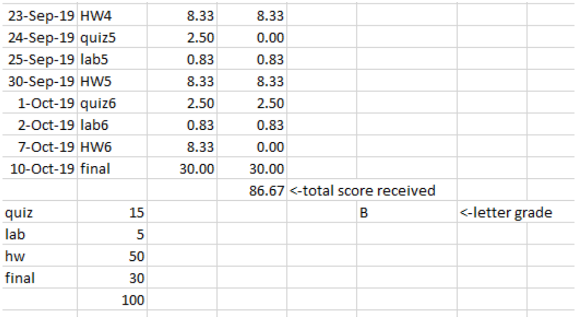
Labs are graded in the lab.

- Starter code will be posted on Canvas for you to download.
- Course staff will check that it runs for them, on the lab image, as furnished.
- It is your responsibility to get graded by a staff member.
- For your lab score, in lab, demonstrate to course staff that your instance runs.
- Be prepared to say what the code is doing.
- Each lab is worth 0.83 points.
- To earn at least 0.5 points, show that you can run the code.
- To earn full score, accomplish some progress towards the homework.

Homework is due on Monday morning before class.

- The code furnished by you for homework must run.
- If course staff cannot get your code to run, the burden is on you to demonstrate that it runs.
- All homework must use the test-driven development framework supplied to you in the starter code.
- You must add suitable test cases for each of your functions.
- Reading and lectures will discuss how to do this.
- Each homework is worth 8.3 points.
- Partial credit can be obtained for partial functionality.
- The rubric associated with the homework will suggest amounts of partial credit.

There is a file on Canvas that shows how to calculate your grade.



23-Sep-19	HW4	8.33	8.33			
24-Sep-19	quiz5	2.50	0.00			
25-Sep-19	lab5	0.83	0.83			
30-Sep-19	HW5	8.33	8.33			
1-Oct-19	quiz6	2.50	2.50			
2-Oct-19	lab6	0.83	0.83			
7-Oct-19	HW6	8.33	0.00			
10-Oct-19	final	30.00	30.00			
			86.67	<-total score received		
quiz	15				B	<-letter grade
lab	5					
hw	50					
final	30					
	100					

Figure: If you record your scores, the formula will calculate your grade.

Homework has previously taken students a lot of time, but does not need to.

- We are actively researching how it is that students use a lot of time completing their homework.
- A email will be or has been sent out inviting you to participate in this research.

We will study and practice manual memory management, which uses pointers.

- The benefit of manual memory management is to obtain more computational complexity in execution, from a given machine size.
- The technique is to use the same memory for different information, over time.
- The cost is, thoughtfulness on the part of the developer.
- To obtain memory, we call a language function malloc, and tell it how many bytes we want.
- When that much memory is available, the execution system will return a memory location,
  a pointer to a location in memory, where our reservation of memory is to be found.
- When we are finished with the memory we obtained, we must free it, else the benefit is lost.
- People have had difficulty with this.
- We will practice it.

We will practice using pointers.

- Because we tell malloc how many bytes of storage we need, we need a way to find out; it is easy. We use the sizeof() operator.
- We will get into the habit of checking whether our request for memory was successful.
- We will learn to initialize pointers, rather than allow random values to appear in them.
- We will contemplate the consequence of using uninitialized pointers.

We will consider the machine stack.

- The memory provided by the hardware is used by the compiler.
- The compiler reserves different regions of memory for different purposes.
- "Reserved", "stack", "heap" and special purpose addresses are nominally different regions.
- "Nominally", because stack overflow is use of one region by another.

There used to be a device used by cafeterias, for stacking trays.

- The computer stack is named after this device.
- New clean trays from dishwashing are placed on the top of the stack.
- Customers remove trays from the stack, at the top.
- People familiar with tax accounting procedures will recognize the similarity to the Last-In, First-Out queue.

Developers can be cognizant of the computer's stack

- When/just before a program function is invoked, it frequently can be that its arguments are placed on the stack.
- Imagine each argument on a single tray.
- When/just before the function returns, its return value is placed on the stack (likewise, a single tray).
- At the assembly language level, the function has the responsibility to remove those trays carrying input values.
- In C and C++, the compiler does this for us.

A recursive call may place too many trays onto the stack.

- Because a recursive routine invokes itself a number of times related to the size of its input,
- and because it can use space on the stack each time,
- and because it might not clean up after itself until later,
- the finite amount of space allocated to the stack can be insufficient.
- If the memory location pointing to the stack's top is adjusted beyond the region allocated for the stack,
- then the stack overflows.

There are other ways to place too many trays onto the stack.

- The ability to ask too much of the stack is not restricted to recursion.
- Using malloc, we can obtain a specified amount of space on the stack.
- Working with C strings, we must be conscientious.
- If we have a pointer to memory we can use, but we continue to write, incrementing the pointer without checking the extent, we can overflow the stack.

We will consider input/ output mechanisms.

- User input from the keyboard, and
- console output
- are our first examples.
- We will also discuss input from and output to text files.
- Binary files are also of interest.
- Input from sensors at fixed addresses in the machine, and
- output to effectuators, likewise at fixed addresses, will also be mentioned.

Aspects of safety in programming.

- Many things can be considered safety, such as avoiding physical damage.
- Here we intend protecting against buffer overflow.

Tools that assist programmers.

- Tools for documentation and debugging are traditionally taught in this course.
- Test driven development replaces both of these.

Consider playing chess, or a similar strategic game.

- We can learn the moves.
- For winning, especially against skilled opponents, we need to do more.
- In this course we shall first learn syntax and semantics.
- These are like the moves in chess.
- A game (in our case, program) will be legal, meaningful, not necessarily efficient or beautiful.
- We shall study approaches:
  - How to get started
  - How to know when we are finished
  - How to obtain assurance our implementation is correct.
  - What refactoring is.

The test-driven development approach considers test cases before implementation.

- People have been known to say, “How can you test something that isn’t there yet?”
- The answer is, that’s easy, we invoke a function with some values for its arguments,
- we generate the correct answer for that case.
- We call the function with those values,
- we compare the returned answer with the known correct answer.
- If a function does not exist at all, an error, perhaps during compilation, will occur.

Let's look at some starter code. I

- Use this file every time, especially in homework.
- We can change the HW1 to HW2, etc.
- You can use this file later in your programming life, too.

```
#include <stdio.h>
#include <stdlib.h>
#include "tests.h"
#include "production.h"
#include "TMSName.h"

int main(int argc, char* argv[]) {
puts("!!!Let's do HW1!!!");
if(tests())
{
production(argc, argv);
}
```

Let's look at some starter code. II

```
else
{
puts("Tests did not pass.");
}
return EXIT_SUCCESS;
}
```

Let's look at some starter code, a homework step for your name. I

- This is just for homework.
- Use your own name.

```
/*
* TMSName.h
*  Created on: Jul 10, 2019
*      Author: Therese
*/

#ifndef TMSNAME_H_
#define TMSNAME_H_

#endif /* TMSNAME_H_ */
```

Separate out the test handling in one or more files. I

```
#include "tests.h"
#include "production.h"

bool tests()
{
bool answer = false;
return answer;
}
```

Each function needs at least one test. I

```
#include "tests.h"
#include "production.h"

bool tests()
{
bool answer = false;
bool ans1 = testFunction1();
bool ans2 = testFunction2();
bool ans3 = testFunction3();
...
answer = ans1 && ans2 && ans3;
return answer;
}
```

Each test needs at least two test cases. I

- Or else it could pass the test by always returning that one right answer.

```
bool testMake2DArray()
{
bool ok = false;
bool ok1 = false;
bool ok2 = false;
//here is a test case, needs rows and cols
int rows = 3;
int cols = 4;
int* the2DArray_p = make2DArray(rows,cols);
//if the array works, it can be written to,
//and subsequently read from
//and will have remembered the correct answer
int testPointRow = 2;
```

Each test needs at least two test cases. II

```
int testPointCol = 1;
int testPointValue = 17;
*(the2DArray_p+testPointRow*cols+ testPointCol)
  = testPointValue;
if(*(the2DArray_p+testPointRow*cols+ testPointCol)
     == testPointValue)
{
//testcase passed
ok1 = true;
}
free(the2DArray_p);
//here is another test case
rows = 5;
cols = 7;
the2DArray_p = make2DArray(rows,cols);
testPointRow = rows-1;
testPointCol = cols-1;
```

Each test needs at least two test cases. III

```
testPointValue = 13;
*(the2DArray_p+testPointRow*cols+ testPointCol)
 = testPointValue;
if(*(the2DArray_p+testPointRow*cols+ testPointCol)
     == testPointValue)
{
//testcase passed
ok2 = true;
}
free(the2DArray_p);
ok = ok1 && ok2;
return ok;
}
```

We have seen two kinds of include files mentioned. I

```
#include "tests.h"
#include "production.h"
```

- Those above are include files we create with the editor.
- Those below are provided with the implementation of C we are using.

```
 #include <stdio.h>
#include <stdlib.h>
```

- We use include files for function prototypes, among other kinds of statements.

Abstraction

- Computer science education research reveals that abstraction is good for you, and difficult at first.
- We see abstraction in object-oriented languages: We can build an object, such as car, that has parts, such as wheels, tires, doors.
- At the higher level of abstraction, it is a car.
- At a lower level of abstraction, it is the assemblage of the parts.
- At a yet lower level of abstraction, it is a lot of lines of code.
- We have also seen abstraction in mathematics.
- We can add 3 things to 4 things, and get 7 things, without much concern for what the things are.
- We can add x things to y things, and get $x + y$ things, without much concern for what the things are.
- We can learn to view a function as a unit, then implement it at a lower level of abstraction, as a bunch of lines of code, and after sufficient testing, view the function's implementation confidently as a unit, at the higher level of abstraction.

Top-down, middle-out and bottom-up design approaches exist. I

- One example of top-down thinking is, at the top level, we have the test suite and the production code.
- We run the test suite, and if it passes, we try out the production code.
- Then, moving down a level of abstraction, we ask, which tests we have.
- At a next lower level of abstraction, we ask, which test cases do we have.
- One example of a middle-out approach is, we have a bunch of objects interacting,
  we have several platforms housing the various parts of the implementation
  we can design the inter-object communication system, before knowing which object we are interconnecting,
  or knowing how they are implemented.

Top-down, middle-out and bottom-up design approaches exist. II

- This approach can help with multiple vendors, when planning to modify prior implementations.
- An example of bottom-up is, we design a fast processor, we design a programming model, so that the programmer does not have to understand hardware much at all, we design a compiler, so that the programmer really just works with the compiler, we might have standard libraries, so the programmer can invoke system functions, and so on, through patterns, such as model-view-controller.

We can go through the prior code samples, and discuss the statements.