CSCI 331:
Introduction to Computer Security

Lecture 2: C Review

Instructor: Dan Barowy

Williams

1

SureVeyor User Study

Come to experiment with our domain-specific programming language designed for social sciences! All backgrounds are welcomed!

For details, contact us at dwb1@williams.edu or ys5@williams.edu

Date: Monday, Sep 11

Time: 4pm–6pm

Location: Ward Lab

There will be pizzas!

Scan the QR code =>

2

Topics

Drop/add deadline: Friday, 15th of September

Anonymous feedback

C review

3

Discussion

4

Discuss Schneier reading.

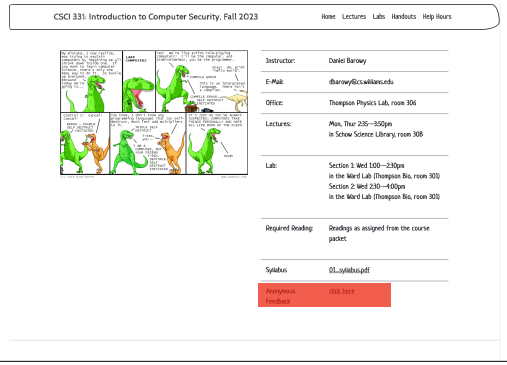
Feedback

Anonymously or eponymously

5

You are always welcome to send me feedback, either via email or anonymously, via the course website.

Anonymous Feedback



6

You can find it here.

Your to-dos

7

1. First lab (Lab 0) **tomorrow**.
   Do you know what section you are in?
2. Reading (pseudoterminals) **due Thu**.
   i. Please use crib notes form
   ii. Reading discussions will be on **Thursdays** from this point forward
3. Second lab (Lab 1) is posted, **due 9/24**.

Readings for Lab 0

8

1. Lab 0 writeup.
   Not a bad idea to skim labs ahead of time.

Lab 0

9

If you have a **laptop** that you plan to use for the semester, please **bring it** to our **first lab meeting**.

If you prefer to use a **lab machine**, you **don't need to bring anything**.

The C Programming Language

10

Let's refresh our memory on the C language. You should have had some prior exposure via 237. We are going to go deeper, so prepare yourself to learn more! These two people, Brian Kernighan on the left, and Dennis Ritchie on the right, invented the C programming language in the early 1970s. C's invention was in a large part motivated by the desire for a "portable" programming language. In other words, Kernighan and Ritchie wanted a programming language where programs could be written once and reused on different computers. Believe it or not, in the early 1970s, portable programs were NOT common!

Activity: What do you know about C?

11

Take a moment and look at the following list of terms. If there are terms whose meanings you don't know, write them down. We will spend some time next class discussing these.

Let's start with the easy stuff

```
$ gcc helloworld.c
```

Like Java, C programs need to be **compiled** before you can run them.

12

The C compiler ignores many problems

```
$ gcc -Wall helloworld.c
```

So you should always ask it to report **warnings**.

13

If you don't like a.out

```
$ gcc -Wall helloworld.c -o helloworld
```

Tell the compiler what you want the output **named**.

14

C Background

1. Despite its quirks, it has many of the **features that you know and love** in Java/Python, etc. (it looks sort of like Java!)
2. Often used in **low-level** or "systems" programming.
3. Nearly as **fast** as expert assembly code; usually faster than non-expert assembly.
4. **No safety net**. Very easy to write programs with subtle bugs.
   1. **No garbage collector: no memory safety.**
   2. **No bounds checker: off-by-one is subtle!**
   3. **No objects: roll your own!!**
   4. **No strings: null-terminated char arrays!!!**
   5. **This list is not exhaustive!!!!**

15

16

The problem with C is **not** its **complexity**.
The problem is its **simplicity**.

Remember the following **rules** and you'll be **OK**!

17

Rule 0:

Pointers are for **pointing at** other values in **memory**.

```c
#include <stdio.h>

int main() {
  int num = 4;
  int *num_ptr = &num;
  printf("num = %d, and it is stored at %p.\n", num, num_ptr);
  return 0;
}
```

The value stored in num_ptr is the address of the location of the num variable. Both num and num_ptr are "automatic variables" and are stored on the call stack.

18

Rule 1:

Whenever you **store data**, you must **always** ask C to **reserve memory** for some **duration**.

```c
#include <stdio.h>

int main() {
  int num = 331;
  printf("%d rocks!\n", num);
  return 0;
}
```

short (automatic)

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

int main() {
  int *num_ptr = malloc(sizeof(int));
  if (!num_ptr) {
    printf("Unable to allocate.\n");
    exit(1);
  }
  *num_ptr = 331;
  printf("%d rocks!\n", *num_ptr);
  return 0;
}
```

long (allocated)

If you don't ask for anything special, values are given automatic duration, meaning that they are invalid at the end of their scope (e.g., when a function returns, or outside of a loop, etc). If you use malloc, a value has "allocated duration" and must be MANUALLY freed later. Note that in the program on the right, num_ptr is itself a variable whose value has automatic duration; the thing IT POINTS TO has allocated duration. We will discuss this more next class.

Recap & Next Class

Today:

- Schneier discussion
- Feedback
- Some C

Next class:

- More C