Lecture 17
Wrapup

CS211 – Fundamentals of Computer Programming II

Branden Ghena – Spring 2023

Slides adapted from:
Jesse Tov

Administrivia

- Projects spec feedback coming soon
  - ~20 groups have already gotten feedback
  - The rest should be today or tomorrow

- Submission on Gradescope is available
  - If it doesn’t pass tests there, it won’t compile when we go to run it

- Get working on your project code!

Today's Goals

- Review what you've learned and why it is useful
- Understand when to use or avoid C/C++ in future projects
- Brief overview of the Rust programming language
- Consider what's next after CS211

Outline

- **Course Goals**
- When should you use C and C++?
- Rust for C/C++ Programmers
- Review of Class Topics
- What's next?

So, why CS211?

- It’s going to make you a **much** better programmer
- It’s going to teach you a bunch of new skills
- It’s going to enable you to succeed in future classes

Formal goals

CS211:

- Teaches software design skills at a small-to-medium scale
  - Some smaller programs: Overlapped, Brickout
  - Some larger programs: Rank-choice Voting, Reversi

Formal goals

CS211:

- Teaches software design skills at a small-to-medium scale
  - Some smaller programs: Overlapped, Brickout
  - Some larger programs: Rank-choice Voting, Reversi

- Bridges students from *How to Design Programs* languages to industry-standard languages and tools
  - Unix shell: SSH, ls, cd,
  - C and C++ programming languages
  - CLion IDE
  - Make and CMake

Upsides to C and C++

- You are in charge of everything
  - You can do anything you want without constraints

- Capable of directly interacting with hardware (“systems language”)
  - Grab exactly as much memory as you need and manage it yourself
  - Makes it incredibly fast (~100x faster than Python)
  - Makes it incredibly efficient (no memory is wasted)

- These lead to the languages being very widely used
  - Top five programming languages for decades include C and C++

Downsides to C and C++

- You are in charge of everything
  - And nothing is taken care of for you

- Things you "can't" do are **UNDEFINED BEHAVIOR**
  - To enable portability, the languages just straight-up don't say what happens if you violate the rules
  - The computer could do *anything*

- Backwards compatibility means features are only ever added
  - You'll see this especially in C++, C just has less features total
  - C++ feels like a bunch of things stapled together
    - And there's an amazing programming language hiding in there

So why teach C and C++?

- You'll learn a lot more about programming
  - Syntax and ideas from C inspired a lot of other languages
  - Feels very different from Racket or Python

- You'll become a better programmer
  - You're going to run into a lot of errors and problems in this class
  - Hopefully they teach you to better design and plan your code

- Prepare you to dig deeper into computer systems
  - A "systems language" is needed to interact directly with hardware
  - Major options: Pascal, C, C++, Ada, Rust

Outline

- Course Goals
- **When should you use C and C++?**
- Rust for C/C++ Programmers
- Review of Class Topics
- What's next?

When should you use C?

- You probably shouldn’t

When should you use C?

- You probably shouldn’t

- Stronger: Don’t use C.

When should you use C?

- You probably shouldn't

- Stronger: Don't use C.

- Stronger still (and what I actually believe):

  Using C when you could use a safer language is engineering malpractice.

  C and **UNDEFINED BEHAVIOR** are the root of many security vulnerabilities

What is C good for?

- Very particular things

- Need for extreme efficiency and speed
  - Often efficient services for *other* programs
  - Systems Programming

- Low-level memory or hardware manipulation
  - Interact with raw memory
  - Computer Systems

Slowly we are replacing the need for C

- C is used for extreme efficiency and speed
  - Beware premature optimization
    - Often algorithm and library choice are more important than language
  - C++ (and others) are often good for this as well
- C is used for low-level memory or hardware manipulation
  - New languages like Rust are starting to meet the needs here

The value of learning C

- The impact it has on every other language you might learn
  - Java, Objective-C, C#, Go, Javascript, Swift, PHP, Perl, Python

  - You'll see lots of similar ideas
    - Structs
    - Curly braces and semicolons
    - if, while, for
    - Arrays and square bracket indexing

- You may use it for future systems courses: CS213, CS343, etc.
- Some experience helps you understand the danger

What about C++?

- More ambiguous than C

- Definitely don't use *old* C++
  - We learned modern C++14
    - Includes many more standard libraries
    - Includes safer memory management (smart pointers)
  - C++ Core Guidelines is a good place to start

- There are other languages with many of the benefits without the confusing parts
  - But really big, important software often eventually ends up in C++

Use the right programming language for the job

- Remember: there is no *best* programming language
  - Every tool is situational

- C and C++ are *not* good for simple programs and demonstrations
  - So use something simpler, like Python

- But if we wrote all of our video game engines in Python, games would be very limited in what they could do
  - So use something more complex, like C++

Break + example Go code

- I'm guessing that few of you have used Go
  - But do you understand it?
- Where does code start?
- What is the type of d?

main.go blog

```
1   package main
2
3   import "fmt"
4
5   type day string
6
7   func (d day) getDayInfo() (string, string) {
8       if d == "Monday" {
9           return "Great Day", "Sunny"
10      }
11      return "Unknown day", "Sunny anyways"
12  }
13
14  func main() {
15      var d day = "Monday"
16      fmt.Println(d.getDayInfo()) // Output is Great Day Sunny
17  }
18
19  func (d day) printDay() {
20      fmt.Println(d)
21  }
```

Break + example Go code

- I'm guessing that few of you have used Go
  - But do you understand it?
- Where does code start?
  - **main()**
- What is the type of d?
  - **day which is a string**

main.go blog

```
1   package main
2
3   import "fmt"
4
5   type day string
6
7   func (d day) getDayInfo() (string, string) {
8       if d == "Monday" {
9           return "Great Day", "Sunny"
10      }
11      return "Unknown day", "Sunny anyways"
12  }
13
14  func main() {
15      var d day = "Monday"
16      fmt.Println(d.getDayInfo()) // Output is Great Day Sunny
17  }
18
19  func (d day) printDay() {
20      fmt.Println(d)
21  }
```

Outline

- Course Goals
- When should you use C and C++?
- **Rust for C/C++ Programmers**
- Review of Class Topics
- What's next?

Background on Rust

- Relatively new programming language (1.0 release in 2015)

- Supports low-level "systems programming"
  - Like C, C++, Ada, Go, Pascal, and few others

- Selling points of Rust
  - Modern language features
    - Zero-cost abstractions, foreign-function interfaces
    - Package management, built-in support for testing
  - Compile-time memory safety
    - No uninitialized variables, no use-after-free or double-free
  - Lack of undefined runtime behavior
    - Array access is bounds-checked

“Hello World” in Rust

```
fn main() {
  println!("Hello 🌍!");
}
```

“Hello World” in Rust

```
fn main() {
    println!("Hello 🌍!");
}
```

- `main()` is a function, and it’s the starting point for Rust programs
  - Takes no arguments, returns no values
  - Separate ways to get input arguments or return error codes

“Hello World” in Rust

```
fn main() {
  println!("Hello 🌍!");
}
```

- `main()` is a function, and it’s the starting point for Rust programs
  - Takes no arguments, returns no values
  - Separate ways to get input arguments or return error codes
- Strings in Rust are Unicode

“Hello World” in Rust

```
fn main() {
    println!("Hello 🌍!");
}
```

- `main()` is a function, and it’s the starting point for Rust programs
  - Takes no arguments, returns no values
  - Separate ways to get input arguments or return error codes
- Strings in Rust are Unicode
- A little weird: `println!()` is a macro function
  - Handles most argument stuff at compile time to generate better errors
  - Macro is code that generates code at compile-time

Types in Rust

- Types in Rust are explicit about their size
  - Listed in number of bits, so `i8` ≈ `char` and `u32` ≈ `unsigned int`

	Types	Literals
Signed integers	`i8`, `i16`, `i32`, `i64`, `i128`, `isize`	`-10`, `0`, `1_000`, `123i64`
Unsigned integers	`u8`, `u16`, `u32`, `u64`, `u128`, `usize`	`0`, `123`, `10u16`
Floating point numbers	`f32`, `f64`	`3.14`, `-10.0e20`, `2f32`
Strings	`&str`	`"foo"`, `"two\nlines"`
Unicode scalar values	`char`	`'a'`, `'α'`, `'∞'`
Booleans	`bool`	`true`, `false`

Working with variables

```
fn main() {
   let x: i32 = 10;
   println!("x: {}", x);
}
```

Working with variables

```
fn main() {
    let x: i32 = 10;
    println!("x: {}", x);
}
```

- `x` is a variable of type `i32` with initial value `10`

Working with variables

```
fn main() {
    let x: i32 = 10;
    println!("x: {}", x);
}
```

- `x` is a variable of type `i32` with initial value `10`
- Curly brackets denote an expression you want to print

Rust playground - Variables

- Test out Rust code online in a browser
  - https://play.rust-lang.org/

- Let's try out that function and play around with some things
  - Variable types and initialization
  - Modify the variable value

- https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=cd7cc8a19bbf719cdbc6eb4b06edbb29

Type inference in Rust

```
fn takes_u32(x: u32) {
  println!("u32: {x}");
}

fn takes_i8(y: i8) {
    println!("i8: {y}");
}
```

```
fn main() {
    let x = 10;
    let y = 20;
    takes_u32(x);
    takes_i8(y);
    // below would fail
    // takes_u32(y);
}
```

- Rust figures out what type you meant if you leave it out
  - But everything *does* still have a type

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=aae8903ef418552d0a10b731e6a11390

Rust has structs which can have methods

```
#[derive(Debug)]
struct Person {
 name: String,
 age: u8,
}

impl Person {
 fn say_hello(&self) {
   println!("Hello, my name is {}", self.name);
 }
}

fn main() {
  let peter = Person {
    name: String::from("Peter"),
    age: 27,
  };
  peter.say_hello();
}
```

Rust has structs which can have methods

```
#[derive(Debug)]
struct Person {
 name: String,
 age: u8,
}

impl Person {
 fn say_hello(&self) {
   println!("Hello, my name is {}", self.name);
 }
}

fn main() {
  let peter = Person {
    name: String::from("Peter"),
    age: 27,
  };
  peter.say_hello();
}
```

Struct with two fields

Rust has structs which can have methods

```
#[derive(Debug)]
struct Person {
 name: String,
 age: u8,
}

impl Person {
 fn say_hello(&self) {
   println!("Hello, my name is {}", self.name);
 }
}

fn main() {
  let peter = Person {
    name: String::from("Peter"),
    age: 27,
  };
  peter.say_hello();
}
```

Tells compiler to create default code that will print the struct for debugging

Rust has structs which can have methods

```
#[derive(Debug)]
struct Person {
 name: String,
 age: u8,
}
```

All member functions of the struct

```
impl Person {
 fn say_hello(&self) {
   println!("Hello, my name is {}", self.name);
 }
}
```

```
fn main() {
  let peter = Person {
    name: String::from("Peter"),
    age: 27,
  };
  peter.say_hello();
}
```

Rust has structs which can have methods

```
#[derive(Debug)]
struct Person {
 name: String,
 age: u8,
}

impl Person {
 fn say_hello(&self) {
   println!("Hello, my name is {}", self.name);
 }
}

fn main() {
  let peter = Person {
    name: String::from("Peter"),
    age: 27,
  };
  peter.say_hello();
}
```

Creating a struct

Rust has structs which can have methods

```
#[derive(Debug)]
struct Person {
 name: String,
 age: u8,
}

impl Person {
 fn say_hello(&self) {
   println!("Hello, my name is {}", self.name);
 }
}

fn main() {
  let peter = Person {
    name: String::from("Peter"),
    age: 27,
  };
  peter.say_hello();
}
```

Calling a method

Rust Playground - Methods

- https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=2925da360685d3be97f2c5726499344a

- Things to try
  - Print out the entire struct
  - Create a constructor with “new”
  - Look at some error messages
    - Try using a string literal as a message

Rust solves memory ownership issues

- C++ example of an ownership issue
  - Reference points to a value that no longer exists

```
int main() {
 std::vector<std::string> v;
 v.push_back("Hello");

 string& x = v[0]; // gets reference to item
 v.push_back("world"); // may reallocate memory

 std::cout << x << "\n"; // UNDEFINED BEHAVIOR
}
```

Rust prevents ownership issues at compile-time

- The following code errors
  - https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=e7bee1ca785182ce1c7e0c1ea3d21748

```
fn main() {
    let mut v = vec![];
    v.push("Hello");

    let x = &v[0];
    v.push("world");

    println!("{}", x);
}
```

Way more to Rust

- Don’t have time for an extensive review of the language

- Course in Rust
  - Most of these slides are borrowed directly from here
  - https://google.github.io/comprehensive-rust/hello-world.html

- Rust for Systems Programmers
  - Targets people who know C++ and care about “systems” topics
    - Some of this is likely not understandable yet for CS211 students
  - https://github.com/nrc/r4cppp

Break + Question

- What does this code print?

```
fn main() {
 let v = vec![10, 20, 30];

 for x in v {
   println!("x: {x}");
 }

 for i in (0..10).step_by(2) {
   println!("i: {i}");
 }
}
```

Break + Question

- What does this code print?

```
fn main() {
 let v = vec![10, 20, 30];

 for x in v {
   println!("x: {x}");
 }

 for i in (0..10).step_by(2) {
   println!("i: {i}");
 }
}
```

Output:

```
x: 10
x: 20
x: 30
i: 0
i: 2
i: 4
i: 6
i: 8
```

Outline

- Course Goals
- When should you use C and C++?
- Rust for C/C++ Programmers
- **Review of Class Topics**
- What's next?

What did we learn in CS211?

- In reverse order:
  - Game Design
  - C++ Programming
  - C Programming
  - Unix Shell

Game Design

- Model, View, Controller concept
  - **Model** handles the program state
  - **View** displays information based on the state
  - **Controller** modifies the state based on user input

- Breaking a system up into these three parts enables more robust, testable code
  - Applicable to any interactive program, not just games

C++ Programming

- Object Oriented Programming
  - Using objects and methods
  - Creating our own Classes

- Encapsulation
  - Internal state should be private
  - Only expose operations that maintain validity of our internal state

- Resource Acquisition Is Initialization (RAII)
  - Wrap resources in an object
  - Allocate when constructed and deallocate when automatically destructed

C Programming

- C syntax and structure
  - If, while, for
  - Functions and return values
  - Headers and Source files

- Types and Variables
  - Name, Object, Value
  - Type determines the kind of value and size of object

- Memory management
  - Stack, Data, and Heap segments
  - When to `malloc()` and `free()` and possible errors

Unix Shell (a.k.a. Linux terminal)

- SSH access to remote machines
  - This will be a recurring need in future classes

- Interacting with files and programs
  - cd, ls
  - Relative and absolute paths
  - Providing flags to programs and looking up documentation

More background on CS tools

- We don’t have a good class on this
  - CS150 and CS211 try to give you some basics
- One good source of material: MIT course
  - “The Missing Semester of Your CS Education”
  - https://missing.csail.mit.edu/
- Another approach: use terminal
  - The more you use it, the more you google how to do things, and the better you’ll get at it

Recommendation: don’t forget about Unix

- Keep playing around with Unix shell
  - Incredibly useful tool for software development and productivity
  - Several options
    - Native MacOS
    - Windows Subsystem for Linux (WSL)
    - Linux installed in a virtual machine (Virtualbox is a good choice)
  - Installing Linux on a virtual machine yourself is a good experience
    - Free and only takes an hour
    - And then you can wreck it, with no consequences

Outline

- Course Goals
- When should you use C and C++?
- Rust for C/C++ Programmers
- Review of Class Topics
- **What's next?**

More CS classes!

- CS211 is a pre-requisite for CS213
  - Obvious next step while you're still fresh with C programming

- CS111, CS150, and CS211 are the "programming classes"
  - Teach you how to program
  - Teach you programming languages

- Future classes in CS are "computer science classes"
  - Teach you how to understand computation and computers
  - How do we use computers to understand and effect our world
    - You'll write programs along the way

New languages

- "Wait, but I only know like four programming languages?!!"
  - Learning others will be up to you

- The same ideas you've already learned will apply
  - Types and Imperative Programming
  - Functional Programming
  - Debugging and Testing

- Lots of great guides online for popular languages

Full-Stack Programming

- A benefit to being a "computer scientist" versus "knowing a programming language"
  - Our curriculum teaches you multiple different parts of the software stack
- You can understand front-end (user-facing) software
  - Probably something like Python or Javascript
- You can understand back-end (software-facing) software
  - Probably something like C++

Plenty More Testing and Debugging

- If you’re going to do a lot of programming, debugging is the most useful skill
  - You get better with lots of practice

- Learning to test your code will help you be more successful
  - Especially on big projects

Outline

- Course Goals
- When should you use C and C++?
- Rust for C/C++ Programmers
- Review of Class Topics
- What's next?