Software Fundamentals

AY 2025/2026

Andreea Costea

1st September 2025

TU Delft

Staff

Eric Jerman

Glenn Weeland (Head TA)

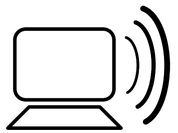
Koen Langendoen

Charlie Ciaś

Max Guichard

Utkarsh Verma

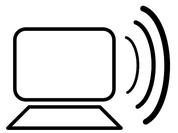
Goal

Specification  Translate →  Code

> “Function `foo` takes two numbers as input and returns their sum.”

Specification?

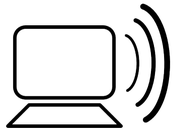
Goal

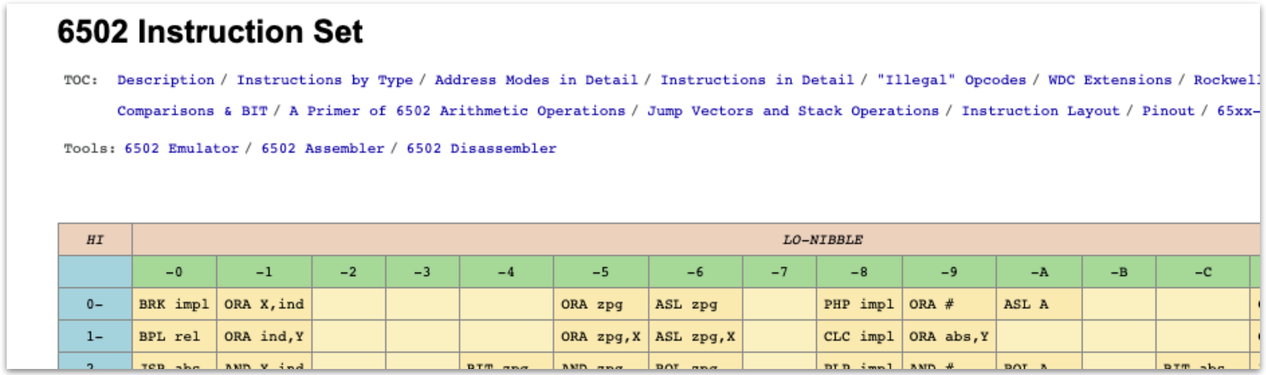
Specification  Translate →  Code

"Function `foo` takes two numbers as input and returns their sum."

Specification?

Goal

Specification  Translate →  Code



6502 Instruction Set

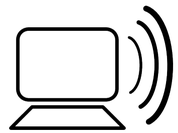
TOC: Description / Instructions by Type / Address Modes in Detail / Instructions in Detail / "Illegal" Opcodes / WDC Extensions / Rockwell
Comparisons & BIT / A Primer of 6502 Arithmetic Operations / Jump Vectors and Stack Operations / Instruction Layout / Pinout / 65xx-

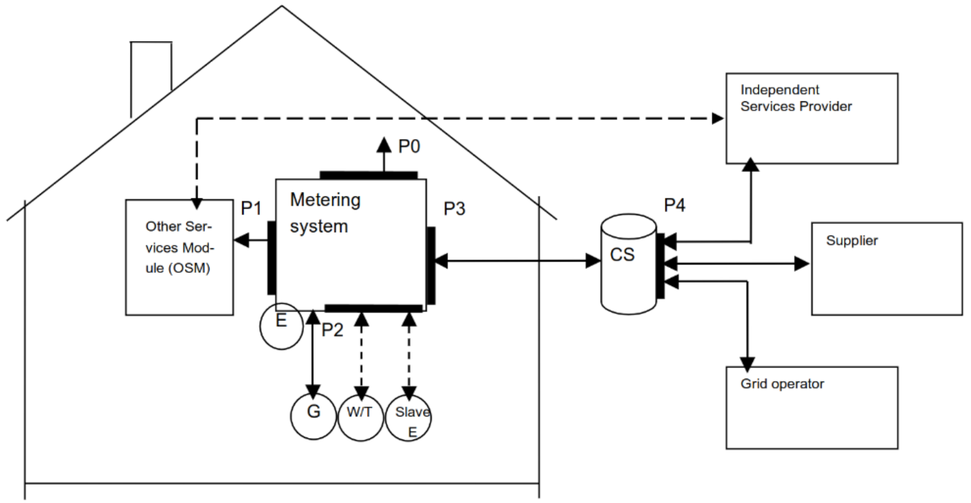
Tools: 6502 Emulator / 6502 Assembler / 6502 Disassembler

HI	*LO-NIBBLE*													
	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	
0-	BRK impl	ORA X,ind				ORA zpg	ASL zpg		PHP impl	ORA #	ASL A			
1-	BPL rel	ORA ind,Y				ORA zpg,X	ASL zpg,X		CLC impl	ORA abs,Y				
2-	JSR abs	AND X,ind			BIT zpg	AND zpg	ROL zpg		PLP impl	AND #	ROL A		BIT abs	

Specification?

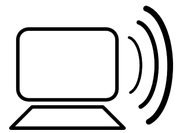
Goal

Specification  Translate  Code



Specification?

Goal

Specification  Translate →  Code

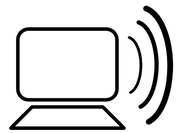
```rust
#[test]
fn test_add() {
    assert_eq!(add(3, 4), 7);
}

#[test]
fn test_subtract() {
    assert_eq!(subtract(-10, 7), -17);
}

#[test]
fn test_multiply() {
    assert_eq!(multiply(-10, 7), -70);
```

Specification?

Goal

Specification  Translate →  Code

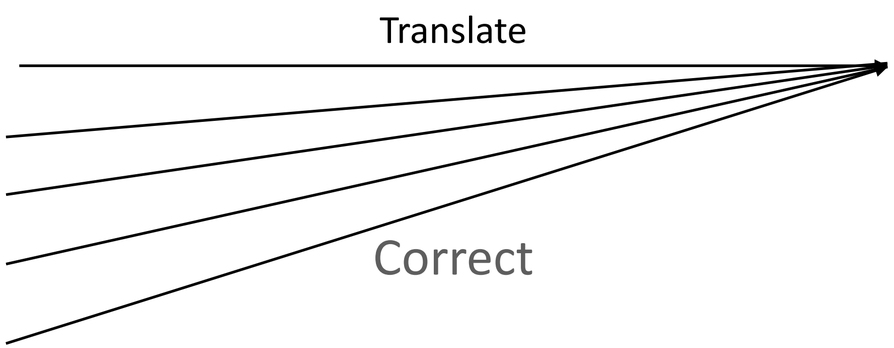
```
pub fn is_equal_v2(a: &i64, b: i64) -> bool {
```

Specification?

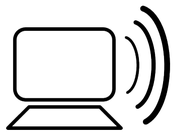
Goal

Specification

Translate



Correct



Code

Informal / Formal

Ambiguos

Un-/Semi-/structured

Graphical

Mathematical / Logical

Executable

Precise

Maintainable

Safe

"Function `foo` takes two numbers as input and returns their sum."

Study Goals

After this course, you will be able to:

1. Explain the programming language concepts followed in **Rust**.
2. **Design, implement** and **debug** a small software system in Rust following the language standard (including proper coding style).
3. **Set up a project** and build environment, using the Rust ecosystem.
4. Use **Git** to version and share source code contributions for collaborative development.
5. Evaluate and integrate code contributions of other **team** members.

Software Fundamentals

Programming

Choices in programming languages

Making safe, reliable and correct programs

Developing software together

Hardware Fundamentals

Digital Computer Systems

Discrete Signals and Systems

Design of Control Systems

Software Fundamentals

Part 1

Lectures (twice a week)

Individual Assignment (start this week!)

Labs and Tutorials (twice a week)

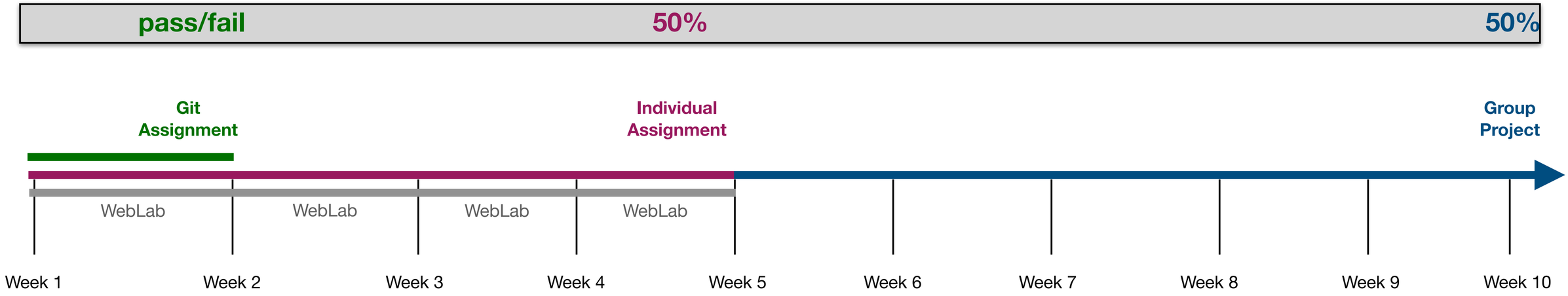
Part 2

Group Project

No lectures!

Mandatory attendance of at least **one lab a week**!

Evaluation



TODO this week

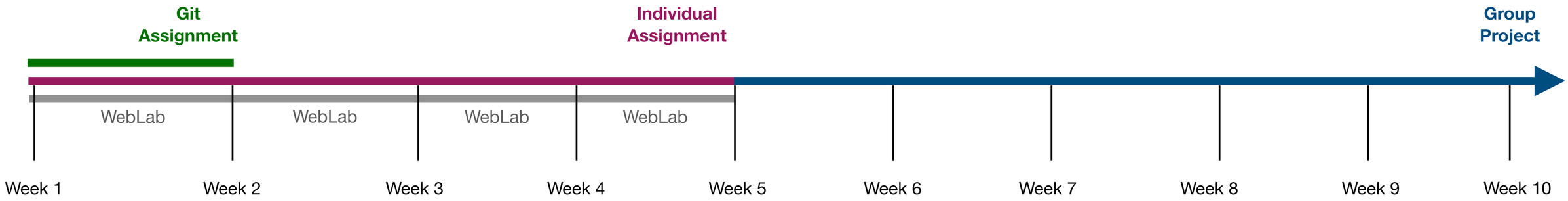
Enjoy exploring and hacking with Rust!

Register on **Brightspace** so we can share the Git repository and your individual assignment with you.

Attend the lab session to complete the **Git assignment** (deadline: Sunday).

Work through the exercises on **WebLab**.

Start the **Individual Assignment.**



Resources

Software Fundamentals website.

Books:

- The Rust Programming Language (Available Online)
  by Steve Klabnik; Carol Nichols; The Rust Community,

- Rust for Rustaceans by Jon Gjengset

Software:

- Linux (recommend Fedora)

- Install Rust through `rustup`, avoid Ubuntu/Debian's repository

Software Fundamentals website

An Introduction to Rust

Andreea Costea

Delft University of Technology

2025-09-01

(slides adapted from Jana Dönszelmann)

TU Delft

- Why choosing a programming language matters?
- Why did we choose Rust?
- Some basics of Rust

Tell me about you

Question:

What programming languages have you used in the past? And what for?

- work, hobby, in teams, alone?

Drones!

Question:

What properties do we care about for the software of this drone?

Programming Languages for Embedded Systems

- We’re teaching about Rust

Question:

What other options are there?

Programming Languages for Embedded Systems

- We’re teaching about Rust
- C
- C++

From the https://osdev.org wiki: people have written kernels in:

Forth, Lisp, C#, Modula-2, Ada, Bliss, Smalltalk, PL/1, Assembly, Zig, D

Programming Languages for Embedded Systems

- We're teaching about Rust
- C
- C++

From the https://osdev.org wiki: people have written kernels in:

Forth, Lisp, C#, Modula-2, Ada, Bliss, Smalltalk, PL/1, Assembly, Zig, D

Question:

Can you use python on embedded systems?

Programming Languages for Embedded Systems

- We're teaching about Rust
- C
- C++

From the https://osdev.org wiki: people have written kernels in:

Forth, Lisp, C#, Modula-2, Ada, Bliss, Smalltalk, PL/1, Assembly, Zig, D

Question:

Why shouldn't you use python on an embedded system?

Programming Languages for Embedded Systems

- We’re teaching about Rust
- C
- C++

From the https://osdev.org wiki: people have written kernels in:

Forth, Lisp, C#, Modula-2, Ada, Bliss, Smalltalk, PL/1, Assembly, Zig, D

So clearly, the features of a programming language matters.

Programming Languages for Embedded Systems

Question:

What properties do we care about when we want to use a programming language for embedded systems?

Programming Languages for Embedded Systems

Question:

What properties do we care about when we want to use a programming language for embedded systems?

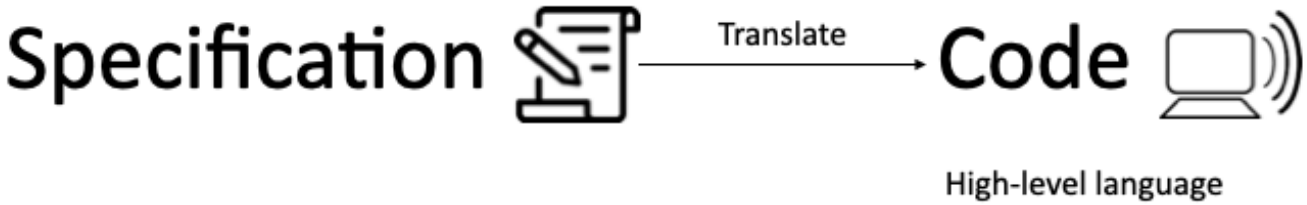
- Compiled
- Low-level access to locations in memory
- Precise control over all program resources
- Guarantees about correctness

Programming Languages for Embedded Systems

Question:

What properties do we care about when we want to use a programming language for embedded systems?

- Compiled
- Low-level access to locations in memory
- Precise control over all program resources
- Guarantees about correctness



Programming Languages for Embedded Systems

Question:

What properties do we care about when we want to use a programming language for embedded systems?

- Compiled
- Low-level access to locations in memory
- Precise control over all program resources
- Guarantees about correctness

Programming Languages for Embedded Systems

Question:

What properties do we care about when we want to use a programming language for embedded systems?

- Compiled
- Low-level access to locations in memory
- Precise control over all program resources
- Guarantees about correctness

Question:

Is there a conflict in these requirements?

Problems with low level control and safety

Access a peripheral (great!):

```c
int main() {
  (int *)(address_of_peripheral) = 10;
}
```

Problems with low level control and safety

Access a peripheral (great!):

```c
int main() {
  (int *)(address_of_peripheral) = 10;
}
```

this works for any random address too (**not ok!**):

```c
int main() {
  (int *)(0x12345678) = 10;
}
```

Problems with low level control and safety

```c
char *alloc_str(char *src) {
  size_t len = strlen(src);
  char *dst = malloc(len);
  memcpy(dst, src, len);
  return dst;
}

int main() {
  char *something = alloc_str("something");
  printf("%s\n", something);
  free(something);
}
```

https://godbolt.org/z/aP5cj16cT

How far can we go?

```c
int main() {
  char *arr = malloc(10);
  for (int i = 0; i < 1500; i++) {
    arr[i] = 5;
  }
}
```

https://godbolt.org/z/15qqq74oe

Undefined Behavior

```C++
1   int main () {
2     while (1) {}
3   }
4
5 int unreachable() {
6     std::cout << "hello, world" << std::endl;;
7 }
```

https://godbolt.org/z/qKMeE9xfb

Undefined Behavior

```C++
int main () {
  while (1) {}
}

int unreachable() {
    std::cout << "hello, world" << std::endl;;
}
```

https://godbolt.org/z/qKMeE9xfb

- In some compilers it's common to **not define** certain behavior.
- The compiler is allowed to assume those cases never happen
- The programmer should simply make sure those cases never happen!

The Good Programmer Myth

- A good programmer knows to avoid undefined behavior
- If someone causes a memory safety bug, they can’t have been a very good programmer
  - Look in the manual! It clearly states that this is undefined behavior!

The Good Programmer Myth

- A good programmer knows to avoid undefined behavior
- If someone causes a memory safety bug, they can’t have been a very good programmer
  - Look in the manual! It clearly states that this is undefined behavior!

- Realy?! Heartbleed: million of users affected by stable code written by professional programmers

The Good Programmer Myth

- A good programmer knows to avoid undefined behavior
- If someone causes a memory safety bug, they can't have been a very good programmer
  - Look in the manual! It clearly states that this is undefined behavior!

- Realy?! Heartbleed: million of users affected by stable code written by professional programmers
- Chrome: "Around 70% of our high severity security bugs are memory unsafety problems"

The Good Programmer Myth

- A good programmer knows to avoid undefined behavior
- If someone causes a memory safety bug, they can't have been a very good programmer
  - Look in the manual! It clearly states that this is undefined behavior!

- Realy?! Heartbleed: million of users affected by stable code written by professional programmers
- Chrome: "Around 70% of our high severity security bugs are memory unsafety problems"

- Bugs aren't always local
- Code review misses bugs (Khoshnoud, Fatemeh, et al.)
- https://steveklabnik.com/writing/memory-safety-is-a-red-herring

We're teaching you Rust

- By default, Rust does not contain any undefined behavior
- If you do want control, you can ask for it:

```Rust
unsafe {
  *(0x1234_5678usize as *const u8) = 10;
}
```

- But don't, you don't usually need it!

Fewer bugs in android: https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html

- Why choosing a programming language matters?
- Why did we choose Rust?
- **Some Basics of Rust**

Anatomy of a Program

```Rust
const A: usize = 3;                 // CONSTANTS AND STATICS
static B: i32 = 5;

struct Point {                      // TYPES
  x: f32,                           // with fields
  y: f32
}

fn example () {}                    // FUNCTIONS

fn main() { }                       // ONE MAIN FUNCTION--special function, program entry point
```

Anatomy of a Program

```Rust
const A: usize = 3;                 // CONSTANTS AND STATICS
static B: i32 = 5;

struct Point {                      // TYPES
  x: f32,                           // with fields
  y: f32
}

mod foo {                           // MODULES
  fn example () {}                  // FUNCTIONS
}

fn main() { }                       // ONE MAIN FUNCTION--special function, program entry point
```

Anatomy of a Program

```Rust
const A: usize = 3;                 // CONSTANTS AND STATICS
static B: i32 = 5;

struct Point {                      // TYPES
  x: f32,                           // with fields
  y: f32
}

mod foo {                           // MODULES
  pub fn example () {}              // FUNCTIONS
}

use foo::example;                   // IMPORTS

fn main() { }                       // ONE MAIN FUNCTION--special function, program entry point
```

Most of Rust code is writing expressions

```Rust
fn example () -> i32 {                    //BLOCK
}

fn main () {                              //BLOCK
}
```

Most of Rust code is writing expressions

```Rust
fn example () -> i32 {                  //BLOCK
  let a: i32 = 40;                      // VARIABLE BINDINGS
  let b: i32 = 2;                       // STATEMENT
  a + b                                 // IMPLICIT RETURN
}

fn main () {                            //BLOCK
  let x: i32 = example();
  println!("Hello, world: {}", x);      // MACROS
}
```

https://godbolt.org/z/ffaf15sdd

Most of Rust code is writing expressions

Rust

```
fn example () -> i32 {                  //BLOCK
  let a: i32 = 40;                      // VARIABLE BINDINGS
  let b: i32 = 2;                       // STAMEMENT
  a + b                                 // IMPLICIT RETURN
}

fn main () {                            //BLOCK
  let x: i32 = { example() + 6 };
  println!("Hello, world: {}", x);      // MACROS
}
```

Most of Rust code is writing expressions

```Rust
fn example () -> i32 {                  //BLOCK
  let a: i32 = 40;                      // VARIABLE BINDINGS
  let b: i32 = 2;                       // STAMEMENT
  a + b                                 // IMPLICIT RETURN
}

fn main () {                            //BLOCK
  let x: i32 = { example() + 6 };
  println!("Hello, world: {}", x);      // MACROS
}
```

- Each of those expressions ending with a semicolon is known in Rust as a **statement**
- A block is made up of 0 or more statements, followed by at most one expression

Mutability

```Rust
fn example () -> i32 {
  let a: i32 = 40;                    // IMMUTABLE
  let b: i32 = 2;                     // IMMUTABLE
  a + b
}

fn main () {
  let x: i32 = { example() + 6 };     // IMMUTABLE
  println!("Hello, world: {}", x);
}
```

Unless otherwise specified, variables are immutable.

https://godbolt.org/z/ffaf15sdd

Mutability

```rust
fn example () -> i32 {
  let a: i32 = 40;                    // IMMUTABLE
  let b: i32 = 2;                     // IMMUTABLE
  a + b
}

fn main () {
  let mut x: i32 = { example() + 6 }; // MUTABLE
  x += 5;
  println!("Hello, world: {}", x);
}
```

- Unless otherwise specified, variables are immutable.
- Use `mut` to make them mutable.

References

```rust
fn example () -> i32 {
  let a: i32 = 40;                    // IMMUTABLE
  let b: i32 = 2;                     // IMMUTABLE
  a + b
}

fn main () {
  let mut x: i32 = { example() + 6 }; // MUTABLE
  let y: &i32 = &x;                   // REFERENCE
  *y += 5;                            // ERROR: cannot assign behind a `&` reference
  println!("Hello, world: {}", y);
}
```

- Create a reference to a value with `&`

https://godbolt.org/z/xGcYbMqfa

References

```rust
fn example () -> i32 {
  let a: i32 = 40;                    // IMMUTABLE
  let b: i32 = 2;                     // IMMUTABLE
  a + b
}

fn main () {
  let mut x: i32 = { example() + 6 }; // MUTABLE
  //let y: &i32 = &x;                 // IMMUTABLE REFERENCE
  let y: &mut i32 = &mut x;           // MUTABLE REFERENCE
  *y += 5;
  println!("Hello, world: {}", y);
}
```

- Create a reference to a value with `&`
- References carry mutability permissions: `&T` vs `&mut T`

(mostly) Automatic Memory Management

```rust
// a string always contains a length
fn alloc_str(inp: &str) -> String {
    String::from(inp)
}

fn main() {
    let x = alloc_str("something");
    println!("{x}");

    // no free needed!
}
```

- No garbage colletor required

Loops

```rust
fn main() {
  let mut c: usize = 0;
  while c < 10 {
    println!("the counter is {c}");
    c += 1;
  }
}
```

Loops

```Rust
fn main() {
  for c in 0..10 {
    println!("the counter is {c}");
  }
}
```

Conditionals

```Rust
fn main() {
  for c in 0..10 {
    if c != 3 {
      println!("the counter is {c}");
    }
  }
}
```

Basics of Rust: Recap

- Expression oriented language

Basics of Rust: Recap

- Expression oriented language
- Variables are immutable by default, use `mut` to make them mutable

Basics of Rust: Recap

- Expression oriented language
- Variables are immutable by default, use `mut` to make them mutable
- References with `&` and `&mut`

Basics of Rust: Recap

- Expression oriented language
- Variables are immutable by default, use `mut` to make them mutable
- References with `&` and `&mut`
- Automatic memory management (no garbage collector)

Basics of Rust: Recap

- Expression oriented language
- Variables are immutable by default, use `mut` to make them mutable
- References with `&` and `&mut`
- Automatic memory management (no garbage collector)
- (for now) Control flow with `if`, `while`, and `for`

Assignment:

- Form pairs
- Go to https://projecteuler.net/archives
- Try one of 1, 5, or 14, or a slightly harder one: 18

Then:

- Go to https://play.rust-lang.org and program it :)
- See how far you get, I'll walk around.
- If you get stuck somewhere? Also look at: https://doc.rust-lang.org/book/