Few Prerequisites

Bitwise & Unsafe Rust

Few Prerequisites

Following are the prerequisites need to be covered before jumping into sea of embedded programming.

1. Binary Numbering System
2. Bitwise operators
3. Unsafe Rust
4. Raw Pointers
5. Mutable Static variables

Binary Number System

Binary Number System

Keep the points in mind in sequence.

1. Binary numbers are **made up of only 1's and 0's**. They don't contain any other number we usually see in decimals (i.e. 2,3,4,...,9). Here are few examples.

Decimal	0	1	2	4	13	21	27	45
Binary	0	1	10	100	1101	10101	11011	101101

Binary Number System

2. Binary numbers have **base of 2** in contrast our decimal numbers which have **base 10**. What base tells us? It tells how many combination can a number give us.

Decimal		Binary	
0000	Starts at 0	0000	Starts at 0
0009	Goes upto 9	0001	Goes upto 1
0010	Reset the digit and increment the next digit.	0010	Reset the current digit and increment the next
0019	Again goes upto 9	0011	What happens next?
0099	What happens with 99, now 2 digits are at optimum	0100	Initial 2 digits got reset and next digit will be incremented
0100	Both will got reset and next digit will be incremented.		

Binary Number System

Now, let's see how to read the binary numbers and also how to convert decimal numbers into binary numbers.

Significance of each bit.

Binary Base = 2

	Column 8	Column 7	Column 6	Column 5	Column 4	Column 3	Column 2	Column 1
Base^{exp}	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Weight	128	64	32	16	8	4	2	1

$2^0 = 1$

$2^1 = 2$

$2^2 = 2 * 2 = 4$

$2^3 = 2 * 2 * 2 = 8$

$2^4 = 2 * 2 * 2 * 2 = 16$

$2^5 = 2 * 2 * 2 * 2 * 2 = 32$

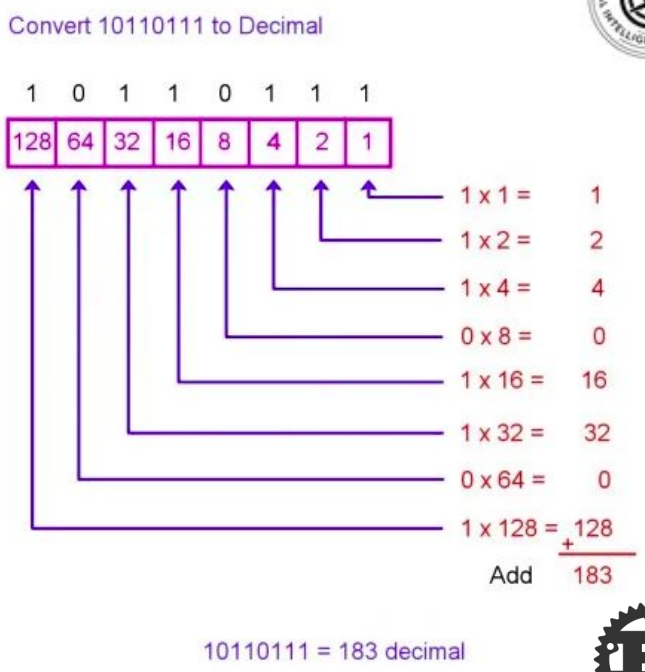
$2^6 = 2 * 2 * 2 * 2 * 2 * 2 = 64$

$2^7 = 2 * 2 * 2 * 2 * 2 * 2 * 2 = 128$

Binary Number System

Now we actually going to convert a decimal number into binary.

Keep the points we learnt so far while converting the number.



10110111 = 183 decimal

Bitwise Operators

Rust Bitwise Operator

Wondering! What these are?

Now you are familiar with binary numbers. Next we will see concept of Bitwise operators that based on the binary numbers.

There are variety of operators in programming languages (i.e. arithmetic, relational, logical and assignment operators). Bitwise operator is one of them.

Bitwise operators: & **,** | **,** ^ **,** ! **,** << **,** >>

Assume values: **A** = 2 (b10) , **B** = 3 (b11)

Bitwise AND (&)

It performs a Boolean AND operation on each bit of its integer arguments.

Example:

A & B => 2, How come this result evaluated ?

1 0 & 1 1
1 0

main.rs saving...

```
1  fn main() {
2      let a:i32 = 2;
3      let b:i32 = 3;
4
5      let mut result:i32;
6
7      result = a & b;
8      println!("(a & b) => {}",result);
9  }
```

Result :

```
(a & b) => 2
```

Bitwise OR (|)

It performs a Boolean OR operation on each bit of its integer arguments.

Example:

A | B => 3, How come this result evaluated ?

main.rs saving...

```
1  fn main() {
2      let a:i32 = 2;
3      let b:i32 = 3;
4
5      let mut result:i32;
6
7      result = a | b;
8      println!("(a | b) => {}",result);
9  }
```

1 0 | 1 1
1 1

Result :

```
(a | b) => 3
```

Bitwise XOR (^)

It performs a Boolean exclusive OR operation on each bit of its integer arguments.

Example:

A ^ B => 1, How come this result evaluated ?

main.rs saving...

```
1  fn main() {
2    let a:i32 = 2;
3    let b:i32 = 3;
4
5    let mut result:i32;
6
7    result = a ^ b;
8    println!("(a ^ b) => {}",result);
9  }
```

1 0 ^ 1 1
0 1

Result :

```
(a ^ b) => 1
```

Bitwise NOT (!)

It is a unary operator and operates by reversing all the bits in the operand.

Example:

!B => -14, How come this result evaluated ?

main.rs saving...

```
1  fn main() {
2      let b : i8 = 13;
3
4      let mut result : i8 ;
5      result = !b;
6      println!("(!b) => {} , {:b}",result,result);
7
8  }
```

! 0 0 0 0 1 1 0 1
1 1 1 1 0 0 1 0 (- 1 4)

Result :

```
(!b) => -14 , 11110010
```

Bitwise left shift (<<)

It moves all the bits in its first operand to the left by the number of places specified in the second operand.

Example:

A << B = 16, How come this result evaluated ?

(2) 0 0 0 0 0 0 1 0 << 3
(16) 0 0 0 1 0 0 0 0

main.rs saving...

```
1  fn main() {
2      let a : i8 = 2;
3      let b : i8 = 3;
4
5      let mut result : i8 ;
6      result = a << b;
7      println!("(a << b) => {}",result);
8
9  }
```

Result :

```
(a << b) => 16
```

Bitwise right shift (>>)

Binary Right Shift Operator. The left operand's value is moved right by the number of bits specified by the right operand.

Example:

A >> B = 0, How come this result evaluated ?

main.rs saving...

```
1  fn main() {
2   let a : i8 = 2;
3   let b : i8 = 3;
4
5   let mut result : i8;
6   result = a >> b;
7   println!("(a >> b) => {}",result);
8  }
```

(2) 00000010 >> 3
(0) 00000000

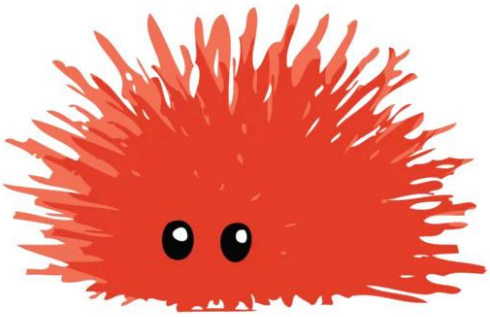
Result :

```
(a << b) => 16
```


PRESIDENTIAL
INITIATIVE
ARTIFICIAL INTELLIGENCE AND COMPUTING

Unsafe Rust

Unsafe Rust



What is that?

- Hidden language inside Rust
- Gives us extra superpowers

Why is it?

- It let us do the operations which compiler restricts us from
- Without unsafe Rust we can’t do operations on hardware level

Unsafe Superpowers:

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- Implement an unsafe trait

Unsafe Rust

While using unsafe Rust keep few things in mind:

- Unsafe rust doesn't disable borrow checker or any other safety feature.
- It gives you only access to these four features and compiler don't check for memory safety in them.
- Inside unsafe block the code doesn't need to be necessarily dangerous.
- While writing code keep the unsafe block smaller.
- For the isolation of unsafe code, it's better to enclose unsafe code with in a safe abstraction. Wrapper!
- When using unsafe rust you as programmer have to ensure that you are accessing memory in a valid way.

Unsafe Rust

A simple unsafe code block:

main.rs saved

```rust
fn main() {

  let mut num = 5;

  let r1 = &num as *const i32;
  let r2 = &mut num as *mut i32;

  unsafe{
    println!("r1 is : {}", *r1);
    println!("r2 is : {}", *r2);
  }

}
```

Raw Pointers

Raw pointer

Unsafe Rust has two new types called **raw pointers** that are similar to references. They can be immutable or mutable and can be written as:

- *const T (Immutable)
- *mut T (Mutable)

Note: Asterisk " ***** " is not a dereferencing operator, it's part of type name.

Let's see how to create an immutable and mutable raw pointer from **reference**.

```
3   let mut num = 5;
4
5   let r1 = &num as *const i32;
6   let r2 = &mut num as *mut i32;
```

Is something wrong! We didn't include **unsafe** keyword here? Well, its okay to create raw pointers outside unsafe but it's must to include while dereferencing them.

Raw pointer

Now, let's create a raw pointer whose validity can't be guaranteed.

main.rs saved

```
1  fn main() {
2
3      let address = 0x012345usize;
4      let r = address *const i32;
5
6  }
```

Above approach to access memory is not recommended, however, you might see or write this kind of code.

Dereference a raw pointer

At the end we will access these raw pointers or dereference them.

Case 1:
Creating a raw pointer and dereference it from a valid reference is completely okay.

main.rs saving...

```rust
fn main() {

  //case 1
  let mut num = 5;

  let r1 = &num as *const i32;
  let r2 = &mut num as *mut i32;

  unsafe{
    println!("r1 is : {}", *r1);
    println!("r2 is : {}", *r2);
  }
}
```

Upon running this code we'll get the result:

```
r1 is : 5
r2 is : 5
```

What if we print values of r1 and r2.

```
r1 is: 0x7ffd359774a4
r2 is: 0x7ffd359774a4
```

Dereference a raw pointer

Case 2:

But when creating a reference from arbitrary location of memory and trying dereference it. It could be problematic. There might be data at that location or might be not.

We'll get the following result upon executing this code.

main.rs saving...

```rust
fn main() {

  //case 2
  let address = 0x012345usize;
  let r = address as *const i32;

  unsafe {
    println!("r is : {:?}", *r);
  }
}
```

```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 0.49s
     Running `target/debug/playground`
timeout: the monitored command dumped core
/root/entrypoint.sh: line 8:     8 Segmentation fault      timeout --signal=KILL ${timeout} "$@"
```

Raw pointer

How **raw pointers** different from **references**!

- Are allowed to ignore the borrowing rules by having both immutable and mutable pointers or multiple mutable pointers to the same location
- Aren’t guaranteed to point to valid memory
- Are allowed to be null
- Don’t implement any automatic cleanup

Static Variables

Static variables

- Global variables in Rust are called static variables
- Static variables are similar to **Constants**
- Naming style of static variables is in SCREAMING_SNAKE_CASE
- Type annotation for static variables is must (like constants).
- Rust allows access read-only static variables (immutable static variables), however doesn't allow access to immutable static variables.
- Static variables always have fixed address in memory.

Summary