Ownership and the Borrow Checker

SSP-RS — Safe System Programming (in Rust)

Samuel Tardieu Stefano Zacchiroli

2024-10-01

Memory and C pointers

The different memory area of a process

As seen in previous lessons, objects in memory can be stored in different memory areas:

- The `.rodata` section contains read-only data included at compile time.
- The `.data` and `.bss` sections contain global variables at fixed addresses determined at link time.
- The **heap** is general memory in which chunks of various sizes can be dynamically allocated.
- The **stack** area of a function exists during the function execution and should no longer be accessed once the function has returned.

All those areas correspond to memory addresses and can be indirectly accessed through pointers.

The different memory area of a process (cont.)

Below is an illustration of the various memory areas:

```
1  int value = 42;              // Global initialized variable: .data
2  int counter;                 // Global uninitialized variable: .bss
3
4  void f(size_t n) {
5    char arr[5] = "abcd";      // {'a','b','c','d','\0'} (5 bytes): .rodata
6                               // arr data (5 bytes reserved): stack
7    char *p1 = malloc(512);    // Area of 512 bytes: heap (/!\ should check result)
8    char *p2 = alloca(n);      // Area of n bytes: stack
9    … // Use arr, p1 and p2 here
10   free(p1);                  // Heap data must be freed
11 }                            // From there, the data stored on the stack must no longer be accessed
```

⚠ Those pointers need to be manipulated cautiously

- If the `p1` pointer is the only pointer to the allocated heap area and disappears without calling `free(p1)`, this area will become unreachable and will be unusable forever (*memory leak*).
- Content of pointer `p2` and address of array `arr` must not be kept anywhere after the function returns as the stack is no longer reserved and will be reused by other functions.

Some issues with pointers in C

Important warning

⚠ The example in the following slides has been constructed to illustrate many problems with pointers. Nobody would write code as bad as this, in any language.

Displaying an uppercase string

Objective

We want to write a function `print_upper(s)` which prints the following line when passed the string `"fOoBaR"` as an argument:

```
The uppercase version of `fOoBaR` is `FOOBAR`.
```

Here is a valid example of a `main()` function using our `print_upper()` function:

```
1 int main() {
2   print_upper("fOoBaR");
3 }
```

Displaying an uppercase string

Let's assume that we already have a `in_place_upper()` function which modifies a string in place and transforms every character to its uppercase counterpart. It also returns the string address for convenience:

```c
#include <ctype.h>   // For toupper()
#include <stdio.h>   // We'll need it later for printf()

// Uppercase the characters of s in place and return it
char *in_place_upper(char *s) {
  for (char *t = s; *t; t++)
    *t = toupper(*t);
  return s;
}
```

💡 *Try to remember what `in_place_upper()` does as we will use it later.*

Displaying an uppercase string: first attempt

```c
void print_upper(char *s) {
  printf("The uppercase version of `%s` is `%s`.\n", s, in_place_upper(s));
}

int main() {
  char *s = "fOoBaR";
  print_upper(s);
}
```

```
$ gcc -O3 -o t t.c -Wall -Wextra -Werror    # We allow no warning at all
$ ./t
[1]    1486161 segmentation fault (core dumped)  ./t
```

It appears that the string `"fOoBaR"` has been stored as read-only data with the program binary code. Assigning it to `s` does not make a copy, it only copies the pointer to the string data. It is then impossible to modify the string and the execution of `in_place_upper()` crashes.

Problem 1 The read-only or read-write status of the pointed data does not necessarily show in the pointer type (*e.g.*, `char *` is allowed to point to read-only data).

Displaying an uppercase string: taking ownership

We can declare `s` as an array on the stack so that we get proper ownership of its content:

```
1  void print_upper(char *s) {
2    printf("The uppercase version of `%s` is `%s`.\n", s, in_place_upper(s));
3  }
4
5  int main() {
6    char s[] = "fOoBaR";  // or: char s[7] = { 'f', 'O', 'o', 'B', 'a', 'R', '\0' };
7    print_upper(s);
8  }
```

```
The uppercase version of `FOOBAR` is `FOOBAR`.
```

Can you spot what happened?

- How and when are the arguments of `printf()` evaluated? When does `printf()` execute?
- What does `in_place_upper()` return?
- How many instances of a string are there in total?

Displaying an uppercase string: taking ownership

```c
void print_upper(char *s) {
  printf("The uppercase version of `%s` is `%s`.\n", s, in_place_upper(s));
}
```

Let's recap what the arguments of `printf()` are:

1. The first argument is the address of the format string.
2. The second argument is the address of `s`.
3. The third argument is the return value of `in_place_upper(s)` which is `s`.

At the time the code of `printf()` is called (after evaluating the arguments), its second and third arguments are the same (`s`) and point to the string which has been uppercased already.

What is problematic here is that we had already evaluated our second argument (as `s`) and before it was used in `printf()` the data pointed by `s` have been modified.

Problem 2 We can hold a pointer to some data while the data get modified behind our back.

Displaying an uppercase string: printing parts one after another

Let's serialize the printing so that we avoid this problem and add some self-congratulations in `main()`:

```
1  void print_upper(char *s) {
2    printf("The uppercase version of `%s`", s);
3    printf(" is `%s`.\n", in_place_upper(s));
4  }
5
6  int main() {
7    char s[] = "fOoBaR";
8    print_upper(s);
9    printf("Yeah, I was able to display `%s` in upper case at last.\n", s);
10 }
```

```
The uppercase version of `fOoBaR` is `FOOBAR`.
Yeah, I was able to display `FOOBAR` in upper case at last.
```

Oops. It looks like our string `s` has been modified as a side effect of `print_upper()`. This is unfortunate.

Displaying an uppercase string: duplicating the string

Let's make a copy of the string to display in uppercase and only transform the copy:

```c
#include <string.h>  // For strdup()

void print_upper(const char *s) {
  const char *upper = in_place_upper(strdup(s));
  printf("The uppercase version of `%s` is `%s`\n", s, upper);
}

int main() {
  print_upper("fOoBaR");
}
```

```
The uppercase version of `fOoBaR` is `FOOBAR`.
```

It looks perfect, right? We were even able to use a read-only string as an argument to `print_upper()`. But wait… Who is in charge of freeing the `upper` string? Isn't memory leaked there?

Displaying an uppercase string: checking for leaks

Let's use the `valgrind` tool to check for memory leaks:

```
$ valgrind -q --leak-check=full ./t
The uppercase version of `fOoBaR` is `FOOBAR`
==1877590== 7 bytes in 1 blocks are definitely lost in loss record 1 of 1
==1877590==    at 0x4841888: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==1877590==    by 0x493781E: strdup (strdup.c:42)
==1877590==    by 0x1091CB: print_upper (in ./t)
==1877590==    by 0x10906F: main (in ./t)
==1877590==
```

Indeed, memory allocated on the heap by `strdup()` (called from `print_upper()`) was not reclaimed. Once `print_upper()` returns, it is too late as the uppercased string pointer is no longer accessible.

Problem 3 The last pointer to a memory area can disappear without freeing the memory first, making it impossible to later free the data. This creates a memory leak.

Displaying an uppercase string: who is the owner?

In C, the same type can represent data that we own and that we must free when we are done, or data that we do not own and must **not** free when we are done.

For example:

- `char *strdup(const char *s)` returns a newly allocated string that must be freed by the caller using `free()`. Not doing so is a *memory leak*.
- `char *getlogin(void)` returns a statically allocated string that must **never** be passed to `free()`.

Problem 4 The pointer type gives no indication about who is supposed to free it and when.

Displaying an uppercase string: freeing memory

Ok, then let's free the memory returned by `strdup()` to avoid a memory leak:

```
1 #include <stdlib.h>  // For free()
2
3 void print_upper(const char *s) {
4   const char *upper = in_place_upper(strdup(s));
5   free(upper);
6   printf("The uppercase version of `%s` is `%s`\n", s, upper);
7 }
```

```
The uppercase version of `fOoBaR` is `?\\`.
```

Oops, it looks like we inserted the `free()` at the wrong place. We free the uppercased string at line 5 and try to display it at line 6. Sure, the mistake was ours, but why was there no warning?

Problem 5 A pointer can live longer than the data it points to.

causing the more general issue (which can also happen with uninitialized or `NULL` pointers):

Problem 6 A valid pointer can point to invalid data.

Displaying an uppercase string: a correct solution

```c
#include <ctype.h>   // For toupper()
#include <stdio.h>   // For printf()
#include <stdlib.h>  // For free()

// Uppercase the characters of s in place and return it
char *in_place_upper(char *s) {
  for (char *t = s; *t; t++)
    *t = toupper(*t);
  return s;
}

void print_upper(const char *s) {
  const char *upper = in_place_upper(strdup(s));
  printf("The uppercase version of `%s` is `%s`\n", s, upper);
  free(upper);
}

int main() {
  print_upper("fOoBaR");
}
```

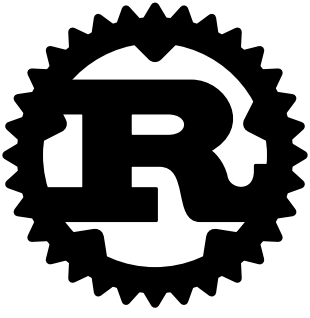
```
The uppercase version of `fOoBaR` is `FOOBAR`.
```

Displaying an uppercase string: summary

We have identified six deficiencies so far while trying to display our uppercased string.

- Problem 1 The read-only or read-write status of the pointed data does not necessarily show in the pointer type (*e.g.*, `char *` is allowed to point to read-only data).
- Problem 2 We can hold a pointer to some data while the data get modified behind our back.
- Problem 3 The last pointer to a memory area can disappear without freeing the memory first, making it impossible to later free the data. This creates a memory leak.
- Problem 4 The pointer type gives no indication about who is supposed to free it and when.
- Problem 5 A pointer can live longer than the data it points to.
- Problem 6 A valid pointer can point to invalid data.

What if a programming language had been designed to prevent those problems from happening in the first place? Let's give Ferris, the Rust mascot, a warm welcome.



Rust and ownership

Rust ownership is based on some simple but effective principles.

Every allocated object has exactly one owner

Every object present in memory has exactly one **owner**. For example, an object allocated on the stack can belong to a local variable. Or, an object allocated on the heap can belong to a field of a structure.

The disappearance of the owner implies the destruction of the object

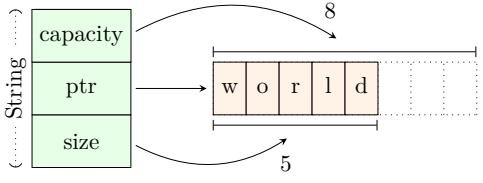
If the owner of an object disappears, the destructor of the object is called. For example, if a local variable owns an object allocated on the heap and the variable goes out of scope (it is **dropped** in Rust terminology), the object will be destroyed and its memory reclaimed.

```
1  fn main() {
2    let s = String::from("world");   // s owns the newly created String (on the heap)
3    println!("Hello, {s}!");
4  }  // s goes out of scope, the memory allocated for the String data is freed
```

Rust: anatomy of a String

In Rust, a `String` is made of a structure containing three fields[1]. The character data is stored in the heap and dynamically allocated.

- A `capacity`, which represents the number of bytes of the allocated memory area.
- A `ptr` which points to the allocated memory area.
- A `size` which represents the number of bytes used in the allocated memory area.



The owner of a `String` can push new characters at the end of the string. If the new size exceeds the capacity, a larger memory area will be allocated, the content will be copied, and the former memory area will be freed.

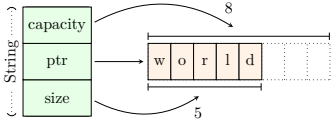
When the destructor of a `String` runs, it returns the allocated memory to the system, preventing any memory leak.

[1] The real structure is a bit more convoluted but ends up with those three fields.

Rust: transferring ownership

When a variable is assigned to another, the new one takes the ownership of the object. Here, the `String` structure with three fields is **moved** from `s` to `t`. The heap memory is left untouched.

```
1  fn main() {
2    let s = String::from("world");
3    let t = s;   // t is now the owner of the string created above
4    println!("Hello, {t}!");
5  }   // t is dropped, the string memory is freed
```



After the `String` is moved from `s` to `t`, it belongs to `t` and `s` does not contain an object anymore. The compiler knows this and inserts no code (for destruction) when `s` goes out of scope. However, when `t` goes out of scope, the compiler will call its destructor.

Rust: use of moved value

Does `s` contain really nothing after its value has been moved to `t`? What if we use it anyway?

```
1 fn main() {
2   let s = String::from("world");
3   let t = s;
4   println!("Hello, {s}!");
5 }
```

In this case, we get a compilation error:

```
error[E0382]: borrow of moved value: `s`
 --> t.rs:4:21
  |
2 |   let s = String::from("world");
  |       - move occurs because `s` has type `String`, which does not implement the `Copy` trait
3 |   let t = s;
  |           - value moved here
4 |   println!("Hello, {s}!");
  |                     ^ value borrowed here after move
```

We'll come back to this error shortly, but let's talk about integers first.

Rust: Copy types

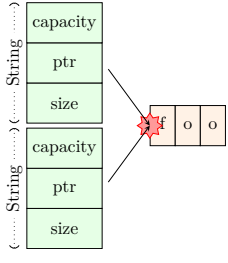
```
fn main() {
  let s = 42;
  let t = s;
  println!("The value of s is {s}");
}
```

This code compiles and runs as expected. Why is that?

Some types in Rust implement **the `Copy` trait**: they can be copied (using a bitwise copy) and destroyed without any side effect. For these types, the value is always copied instead of moved. This applies to all builtin integer and floating point types.

Could `String` implement the `Copy` trait? Absolutely not!

- The two `ptr` would point to the same memory area after the copy, but pushing characters in one string could lead to a reallocation, the other string would then point to a freed memory area.
- It could not have a destructor (since a destructor would constitute a side effect) and would leak memory.



For now, think of a trait as a property that applies to all values of a given type.

Back to the error message

This message is obscure:

```
error[E0382]: borrow of moved value: `s`
 --> t.rs:4:21
```

“borrow”? What does that mean here?

```
1  fn main() {
2    let s = String::from("world");
3    let t = s;
4    println!("Hello, {s}!");
5  }
```

At least we now understand this part:

```
2 |   let s = String::from("world");
  |       - move occurs because `s` has type `String`, which does not implement the `Copy` trait
```

And then “borrowed” again:

```
3 |   let t = s;
  |           - value moved here
4 |   println!("Hello, {s}!");
  |                    ^ value borrowed here after move
```

Borrowing must be an important concept if the compiler cannot stop talking about it. It is indeed.

Rust and borrowing

Let's assume that you are the owner of an object `x` of type `T` and that a function needs to use `x` temporarily. You can choose one of the following options:

1. Give the object `x` to the function (this will transfer the ownership from you to the function). The function can then return the object back to you once it is done with it.
2. Or lend the object `x` to the function which then **borrows** it. A borrowed object is represented by a **reference** `&x` of type `&T`. When the reference goes out of scope, nothings happens as the owner still owns the object:

```
1  fn display_twice(r: &String) {   // r is a borrowed String
2    println!("{r}, and then {r} again");
3  } // r (which is a reference to a String) is dropped, nothing happens
4
5  fn main() {
6    let s = String::from("I talk");
7    display_twice(&s);              // Lend s to display_twice
8    println!("We still own the string `{s}` here");
9  } // s is dropped, the String is freed
```

So a reference is like a pointer in C, right?

Wrong! A reference in Rust is much more than a pointer in C[2]:

- A reference is like a pointer with a specified **lifetime**. The reference cannot live longer than the object it points to. This is verified at compile time by the **borrow checker**.
- As a consequence, you cannot store a reference in an object that would live longer than the reference lifetime. For example, you cannot store a reference you were given to into a global variable unless you have borrowed an object which lives forever.
- You cannot destroy an object on which you only have a reference, unless you replace it by another valid object[3]. Remember, you have only borrowed the object, you are not the owner, so you are expected to return it in good shape.
- A reference always points to an existing object and can never be null.

[2] We will see later than Rust has pointers too, used for low-level programming and interfacing, *e.g.,* with C.

[3] provided you have the right to modify it, more on this in a few slides

The borrow checker in action

```
1  fn main() {
2    let r: &String = {
3      let s = String::from("Hello, world!");
4      &s   // In Rust, blocks evaluate to their last expression
5    };
6    println!("r references the string {r}");
7  }
```

```
error[E0597]: `s` does not live long enough
 --> t.rs:4:5
  |
2 |   let r: &String = {
  |       - borrow later stored here
3 |     let s = String::from("Hello, world!");
4 |     &s
  |     ^^ borrowed value does not live long enough
5 |   };
  |   - `s` dropped here while still borrowed
```

The string `s` lives from its creation (line 3) to the end of its scope (line 5). The borrow checker does not let us store a reference to `s` in `r`, which lives up to line 7.

How does it help us so far?

How do references protect us from the problems we had identified with C pointers?

- Problem 1 The read-only or read-write status of the pointed data does not necessarily show in the pointer type (*e.g.*, `char *` is allowed to point to read-only data).
- Problem 2 We can hold a pointer to some data while the data get modified behind our back.
- Problem 3 ~~The last pointer to a memory area can disappear without freeing the memory first, making it impossible to later free the data. This creates a memory leak.~~ *(at the time a reference to an object dies, the object owner is still alive)*
- Problem 4 ~~The pointer type gives no indication about who is supposed to free it and when.~~ *(the object is freed when its owner dies)*
- Problem 5 ~~A pointer can live longer than the data it points to.~~ *(nope, not in Rust)*
- Problem 6 ~~A valid pointer can point to invalid data.~~ *(nope, not in Rust)*

But do Rust references suffer from problems 1 and 2? No they don't, let's see why.

Rust and immutability

By default, Rust objects are manipulated through read-only (immutable) variables.

```
1 fn main() {
2   let s = String::from("Hello, world");
3   s.push('!');
4   println!("{s}");
5 }
```

```
 --> t.rs:3:3
  |
2 |   let s = String::from("Hello, world");
  |       - help: consider changing this to be mutable: `mut s`
3 |   s.push('!');
  |   ^^^^^^^^^^ cannot borrow as mutable
```

Note that the `push` method on `s` wanted to borrow the string, not take ownership of it. Of course, we want the variable `s` to still be the owner of the string after we have added the `!` character. We will see later how methods can declare what they want to do with the object (receive it, borrow it, etc.).

Let's follow the compiler suggestion and add `mut` there.

Rust and mutability

Let's make the variable `s` mutable by adding `mut` so that we can modify the object it owns:

```
1 fn main() {
2   let mut s = String::from("Hello, world");
3   s.push('!');
4   println!("{s}");   // Displays Hello, world!
5 }
```

We can also lend a mutable reference (on a mutable object) to a function using `&mut`:

```
1 fn add_punctuation(r: &mut String) {
2   r.push('!');       // We can call the method directly on the mutable reference, neat heh?
3 }
4
5 fn main() {
6   let mut s = String::from("Hello, world");
7   add_punctuation(&mut s);
8   println!("{s}");
9 }
```

Note how `add_punctuation()` re-lends the mutable reference to the `push()` method.

We are getting further

Problem 1 ~~The read-only or read-write status of the pointed data does not necessarily show in the pointer type (*e.g.*, `char *` is allowed to point to read-only data).~~ *(a mutable reference is necessary to modify the data)*

Problem 2 We can hold a pointer to some data while the data get modified behind our back.

Problem 3 ~~The last pointer to a memory area can disappear without freeing the memory first, making it impossible to later free the data. This creates a memory leak.~~ *(at the time a reference to an object dies, the object owner is still alive)*

Problem 4 ~~The pointer type gives no indication about who is supposed to free it and when.~~ *(the object is freed when its owner dies)*

Problem 5 ~~A pointer can live longer than the data it points to.~~ *(nope, not in Rust)*

Problem 6 ~~A valid pointer can point to invalid data.~~ *(nope, not in Rust)*

How does Rust help with problem 2? Let's kill the suspense right there.

Mutability and numerability

Rust enforces, at compile time, additional rules on mutable objects and references:

1. An object cannot be modified as long as any reference (mutable or not) to it is alive:

```
fn main() {
  let mut s = String::from("Hello, world");
  let t = &s;
  s.push('!');       // Error: t (reference on s) is alive while attempting this operation
  println!("{t}");
}
```

2. A mutable reference cannot coexist (be alive at the same time) with any other reference (mutable or not) to an object.

```
fn main() {
  let mut s = String::from("Hello, world");
  let t = &s;
  let r = &mut s;    // Error: t (reference on s) is alive while attempting this
  r.push('!');
  println!("{t}");
}
```

Job done

Problem 1 ~~The read-only or read-write status of the pointed data does not necessarily show in the pointer type (*e.g.*, `char *` is allowed to point to read-only data).~~ *(a mutable reference is necessary to modify the data)*

Problem 2 ~~We can hold a pointer to some data while the data get modified behind our back.~~ *(Rust prevents this at compile time)*

Problem 3 ~~The last pointer to a memory area can disappear without freeing the memory first, making it impossible to later free the data. This creates a memory leak.~~ *(at the time a reference to an object dies, the object owner is still alive)*

Problem 4 ~~The pointer type gives no indication about who is supposed to free it and when.~~ *(the object is freed when its owner dies)*

Problem 5 ~~A pointer can live longer than the data it points to.~~ *(nope, not in Rust)*

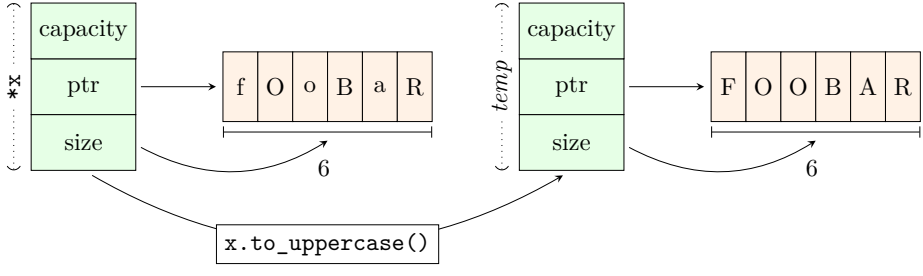
Problem 6 ~~A valid pointer can point to invalid data.~~ *(nope, not in Rust)*

Now that the superiority of Rust references over C pointers has been established, let's see how to use them in practice.

`in_place_upper()` in Rust

```
1 fn in_place_upper(x: &mut String) {
2   *x = x.to_uppercase();
3 }
```

The code on line 2 first creates a temporary `String` object with an uppercase version of the string referenced by `x`. A new memory area will be allocated to hold “FOOBAR”.



The content of `x` is then replaced by the temporary `String` object: all fields are replaced. Just before doing so, the previous memory area (containing `"fOoBaR"`) is freed.

`in_place_upper()` in Rust: modifying an object

```rust
fn in_place_upper(x: &mut String) {
  *x = x.to_uppercase();
}
```

You might be surprised that `in_place_upper()` was able to drop the object that was borrowed through `x` as the function was not the owner of the object.

This is not a problem here: another object has replaced the borrowed one, the caller will still be the rightful owner of an object of type `String` although it is a different one. This is equivalent to having replaced all three fields of the borrowed `String` one by one (if we had access to those fields).

Note

Some might argue that `in_place_upper()` is not equivalent to its C counterpart. In this function, only the `x` variable is modified in-place, while the `String` content it points to is replaced with another string. This is indeed true, but the next slide will explain why we chose not to modify the string character by character.

`in_place_upper()` in Rust: why not character by character?

You might have noticed that our `in_place_upper()` function is different in C and in Rust:

- In C we replace each `char` in place (one byte each) by its uppercase version.
- In Rust we build a new `String` with the uppercase version of the whole string.

```c
char *in_place_upper(char *s) {
  for (char *t = s; *t; t++)
    *t = toupper(*t);
  return s;
}
```

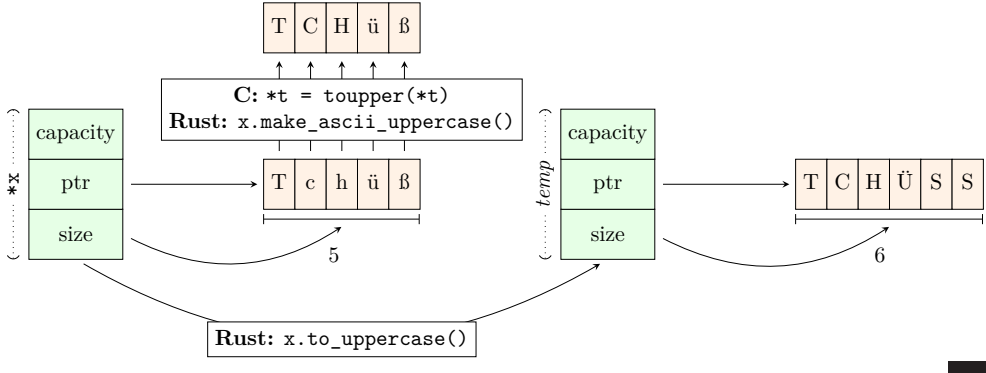
```rust
fn in_place_upper(x: &mut String) {
  *x = x.to_uppercase();
}
```

This is due to the fact that Rust strings are Unicode strings encoded in UTF-8:

- Unicode strings contain characters for all known written languages. The uppercase version of “ß” (1 character) in German is “SS” (2 characters). We cannot uppercase it in place.
- UTF-8 uses variable-length encoding: some characters are represented by several bytes (up to 4). We cannot treat a string as independent bytes representing characters.

`in_place_upper()`: result of uppercasing “Tchüß”

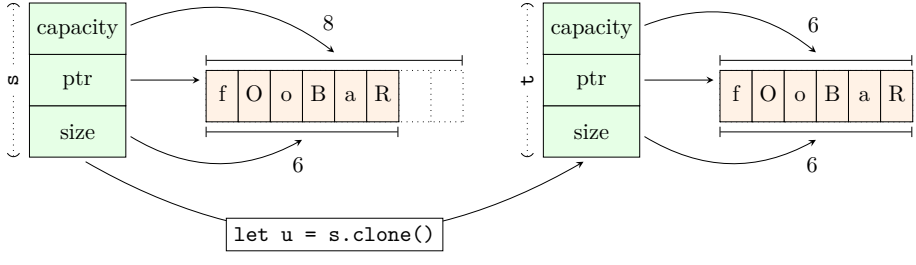
- “TCHÜSS” in Rust: use proper capitalization rules and create a new string with `x.to_uppercase()`.
- “TCHüß” in C: no rule for “ü” or “ß” in the limited ASCII character set.
- Note: Rust also has a C-like `make_ascii_uppercase()` method on strings.



`print_upper()` in Rust

```rust
fn print_upper(s: &String) {
  let mut u = s.clone();
  in_place_upper(&mut u);
  println!("The uppercase version of `{s}` is `{u}`");
}
```

`s.clone()` makes a clone of the string `s` that we have borrowed and stores it in the mutable variable `u` as a new `String` object (because `String` implements the `Clone` trait).



`u` is then mutably borrowed by `in_place_upper()` which will uppercase it.

The whole Rust program

The annotated program shows when memory is allocated or freed. No memory is leaked despite the fact that we have zero explicit deallocations.

```
1  fn in_place_upper(x: &mut String) {     // x is mutably borrowed and can be modified
2    // Replace content of x by uppercase version and drop previous content
3    *x = x.to_uppercase();
4  }
5
6  fn print_upper(s: &String) {            // s is immutably borrowed and cannot be modified
7    let mut u = s.clone();                // Make u a clone of s (allocate memory and copy)
8    in_place_upper(&mut u);               // Make u uppercase
9    println!("The uppercase version of `{s}` is `{u}`");
10 }  // u goes out of scope here and its memory is reclaimed
11
12 fn main() {
13   print_upper(&String::from("fOoBaR")); // The "fOoBaR" string is dropped
14                                         // at the end of this statement.
15 }
```

Note that this is **not idiomatic Rust**. We will learn how to write better and clearer code later.

Does Rust use a garbage collector?

No! Rust does not need a garbage collector:

- A conservative garbage collector (for example in Java) scans the heap from time to time and tries to find allocated memory which is not transitively referenced by an object on the stack or a global variable. Memory is freed at unpredictable times and can grow excessively.
- A counting garbage collector (for example in Python) counts the number of time each object is referenced. When the count drops to 0, the object is destroyed. However, having cross-references (or self-references) will maintain the counter to a higher value and memory can be leaked.
- The Rust compiler knows exactly, at compile time, when the owner of an object goes out of scope or is overwritten. Instruction for dropping the object and calling its destructor are inserted at this point in the machine code. The memory is freed as soon as possible.

References and lifetimes

What is a lifetime?

In Rust, a lifetime represents a span of code:

- starting at an object creation
- and ending at the object destruction.

Example

```
1  fn main() {
2    let a = String::from("aaaa");
3    let b = String::from("bbbb");
4    {
5      let c = String::from("cccc");
6    }
7    let d = String::from("dddd");
8  }
```

- a lifetime: line 2 (creation) to line 8 (destruction)
- b lifetime: line 3 (creation) to line 8 (destruction)
- c lifetime: line 5 (creation) to line 6 (destruction)
- d lifetime: line 7 (creation) to line 8 (destruction)

Object destruction

An object stored in a variable is destroyed:

- when the variable (its owner) goes out of scope
- or just before a new object is assigned to the variable.

Example

```
1  fn main() {
2    let mut a = String::from("aaaa");
3    let b = String::from("bbbb");
4    a = String::from("AAAA");
5  }
```

- a lifetime (string “aaaa”): line 2 (creation) to line 4 (destruction)
- b lifetime: line 3 (creation) to line 5 (destruction)
- a lifetime (string “AAAA”): line 4 (creation) to line 5 (destruction)

Inheriting lifetimes

Many lifetimes are not named explicitly. In the following example, the lifetime of the returned reference is the same as the lifetime of the function parameter:

```
1  struct Person {
2    first_name: String,
3    last_name: String,
4  }
5
6  fn first_name(person: &Person) -> &String {  // The output lifetime is the input lifetime
7    &person.first_name
8  }
```

The compiler will ensure at compile time that the reference will not survive the `Person`:

```
1  fn main() {
2    let s: &String = {
3      let person = Person { first_name: String::from("John"), last_name: String::from("Doe") };
4      first_name(&person)  // ERROR: `person` (borrowed value) does not live long enough
5    };
6    println!("The person first name is {s}");  // Error
7  }
```

Lifetime limitations

The following function attempts to return the shortest string using references:

```
1 fn shortest(s1: &String, s2: &String) -> &String {
2   if s1.len() <= s2.len() { s1 } else { s2 }
3 }
```

This code will not compile:

```
1 | fn shortest(s1: &String, s2: &String) -> &String {
  |                 -------      -------     ^ expected named lifetime
  |                                            parameter
  |
  = help: this function's return type contains a borrowed value, but the
  = signature does not say whether it is borrowed from `s1` or `s2`
help: consider introducing a named lifetime parameter
```

The compiler tells us that we have to indicate which input lifetime (if any) should be used for the returned reference. Let's do this.

Naming lifetimes

In our case, we want the lifetime of the returned reference (let's call it `'lr`) to be no wider than any of the lifetimes of the input references (let's call them `'l1` and `'l2`):

```
1  // 'lr, 'l1 and 'l2 are three generic lifetime parameters to this function
2  fn shortest<'lr, 'l1: 'lr, 'l2: 'lr>(s1: &'l1 String, s2: &'l2 String) -> &'lr String {
3      if s1.len() <= s2.len() { s1 } else { s2 }
4  }
5
6  fn main() {
7      let s1 = String::from("John");
8      let s2 = String::from("Doe");
9      let s: &String = shortest(&s1, &s2);
10     println!("The shortest string is {s}");
11 }
```

By adding the constraints that `'l1` and `'l2` must live at least as long as `'lr`, using `'l1: 'lr` and `'l2: 'lr`, the compiler ensures at compile time that the returned reference will never outlive its content.

Interlude: some Rust syntax

Some Rust syntaxic elements we have encountered already

```rust
struct Person { first_name: String, last_name: String }

fn first_name(person: &Person) -> &String {
  &person.first_name
}

fn shortest<'lr, 'l1: 'lr, 'l2: 'lr>(s1: &'l1 String, s2: &'l2 String) -> &'lr String {
    if s1.len() <= s2.len() { s1 } else { s2 }
}
```

- A function is created using `fn`, function names are usually lower cased. The main function is named `main`.
- A structure type is created using `struct`, composite type names are usually CamelCased and field names lower cased.
- Every expression returns a value (no need to use `return`) unless it ends with `;`, in which case it is a statement and returns `()` (the equivalent of `void` in C).
- Lifetimes are prefixed by `'`, generic parameters are enclosed in `<>` and constraints are given using `:`.

Some Rust syntaxic elements we have encountered already (cnt'd)

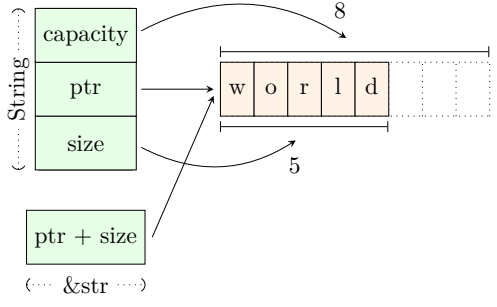
```rust
fn main() {
  let mut s = String::from("Hello");
  s.push_all(", world!");
  println!("s is {s} and its length is {}", s.len());
}
```

- Functions (such as `from`) can be associated with a type (such as `String`) and called as `type::function(…)` (such as `String::from("Hello")`).
- Variables are declared with `let` (immutable) or `let mut` (mutable).
- Methods are called with a `.` as in `s.push_all(", world!")`.
- Macros are suffixed with `!`. `println!()` takes a format string and the list of substitutions.

Back to references and lifetimes

`String` and `str`

While `String` represents an owned string, which can be modified if mutable, `&str` is a reference containing a *fat pointer* (or *wide pointer*) to a memory area representing a string. `&str` contains the address in memory as well as the size of the string in bytes, and as any reference is associated with a lifetime.



For example, a string literal in the source code such as `"Hello, world!"` is a `&str`. Its lifetime is the special lifetime `'static`, which means “until the program ends”.

`String` and `str` (cnt'd)

`String::from()` takes a `&str`, makes a copy of it in the heap and returns a `String` object:

```
1 fn main() {
2   let s = "Hello";                // s type is &'static str
3   let mut t = String::from(s);    // String::from() takes a &str and makes a copy
4   t.push_all(", world!");         // Since we own t and it is mutable, we can modify it
5   println!("s = {s}, t = {t}");  // s = Hello, t = Hello, world!
6 } // t is freed here. s is a reference so it has no destructor to run.
```

Note that `s` is a reference to a string literal `"Hello"` which is probably stored in a read-only memory of the program. When `s` goes out of scope, nothing happens:

- Dropping a reference doesn't drop its content (the reference does not own it).
- There is nothing to free as no heap memory has been allocated for the `&str`.

`String` and `str` (cnt'd)

It is possible to use the `as_ref()` method on a `String` to get a `&str` with the same lifetime:

```
fn main() {
  let mut s: String = String::from("Hello");
  let t: &str = s.as_ref();   // t lifetime cannot be longer than s lifetime
  s.push_all(", world!");     // ERROR: cannot modify s since we have a reference (t) on it
  println!("t = {t}");
}
```

Due to some compiler magic that will be explained later when we study *traits*, you can pass a reference to a `String` everytime a `&str` is expected:

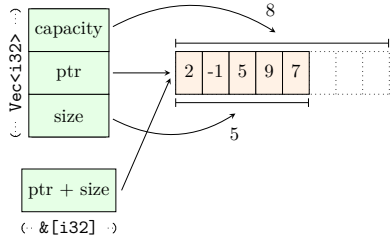
```
fn display_twice(s: &str) {
  println!("{s}/{s}");
}

fn main() {
  let s: String = String::from("Hello");
  display_twice(s.as_ref());   // Will display Hello/Hello
  display_twice(&s);           // Will display Hello/Hello
  display_twice("foo");        // Will display foo/foo
}
```

`Vec` and slices

Similarly to `String`, Rust has a generic type `Vec` that represents a (possibly mutable) vector of consecutive elements in memory:

- `Vec<i32>` represents a vector of `i32` values
- `Vec<String>` represents a vector of `String` objects

The `capacity` and `size` are expressed in terms of number of elements.



As with `&str`, a reference to a slice, denoted as `&[i32]` or `&[String]` in our examples, represents a memory area with a beginning and a number of elements. A reference to a `Vec<T>` can be used everytime a `&[T]` is expected.

`Vec`, arrays and slices

Contrary to vectors, arrays have a fixed size. They can also be transformed into a reference to a slice:

```
1  fn display_len(name: &str, t: &[i32]) {
2    println!("The len of {name} is {}", t.len());
3  }
4
5  fn main() {
6    let arr: [i32; 5] = [10, 20, 30, 40, 50];
7    display_len("arr", &arr);              // The len of arr is 5
8    let mut vec: Vec<i32> = vec![10, 20, 30, 40, 50];
9    vec.push(60);
10   display_len("vec", &vec);              // The len of vec is 6
11 }
```

References on slices (`&[T]`) and `&str` are much more powerful than C raw pointers:

- They are references, so they have a lifetime and cannot survive the object they are pointing to.
- They embed the size of the data they point to and prevents overflowing their buffer.

Mutable slices

Slices can be mutable as well if the underlying data is:

```
1  /// Double all integers present in the slice
2  fn double(s: &mut [i32]) {
3    for i in s {   // i is of type &mut i32 and will iterate over the slice elements
4      *i *= 2;
5    }
6  }
7
8  fn main() {
9    let mut arr: [i32; 5] = [10, 20, 30, 40, 50];
10   double(&mut arr);
11   println!("arr = {arr:?}"); // Prints: arr = [20, 40, 60, 80, 100]
12   let mut vec: Vec<i32> = vec![1, 2, 3, 4, 5];
13   vec.push(6);
14   double(&mut vec);
15   println!("vec = {vec:?}"); // Prints: vec = Vec<2, 4, 6, 8, 10, 12>
16 }
```

The `{:?}` modifier displays values whose type implement the `Debug` trait.

To summarize

- `Vec<T>` represents a vector of objects of type `T`. Those objects are stored consecutively on the heap, and the memory is reallocated as needed when the capacity is no longer sufficient.
- `[T; N]` represents an array of `N` objects of type `T`. The size is fixed, the array can be allocated in a read-only area, on the stack or on the heap (more on that in a later course).
- `&[T]` represents reference on a slice of consecutive objects of type `T`. This reference contains a *fat pointer* which includes the size of the underlying data as well as the data lifetime.

Also:

- `Vec<T>` and `String` objects have a fixed size (they contain `capacity`, `ptr` and `size`), even though they contain a pointer to data of arbitrary length stored on the heap. They can be stored in other structures or in variables, or passed to functions.
- Arrays such as `[T; N]` also have a fixed size, generally `N` times the size of an object of type `T` (possibly more due to alignment constraints).
- `str` and `[T]` have arbitrary sizes. They can be passed around only as references, which include the size of the data automatically.