The Rust Programming Language

Pramode C.E

March 2, 2017

A language that doesn't affect the way you think about programming, is not worth knowing.

Alan Perlis.

Why is Rust so exciting?

- Safe low-level systems programming
- Memory safety without garbage collection
- High-level abstractions (influenced by statically typed functional programming languages like ML) without run-time overhead
- Concurrency without data races

Why not just use C/C++?

The second big thing each student should learn is: How can I avoid being burned by C's numerous and severe shortcomings? This is a development environment where only the paranoid can survive.

`http://blog.regehr.org/archives/1393`

A bit of Rust history

- Started by Graydon Hoare as a personal project in 2006
- Mozilla foundation started sponsoring Rust in 2010
- Rust 1.0 released in May, 2015
- Regular six week release cycles

Core language features

- Memory safety without garbage collection
  - Ownership
  - Move Semantics
  - Borrowing and lifetimes
- Type Inference
- Algebraic Data Types (Sum and Product types)
- Exhaustive Pattern Matching
- Trait-based generics
- Iterators
- Zero Cost Abstractions
- Concurrency without data races
- Efficient C bindings, minimal runtime

Structure of this workshop

- Understanding the problems with C/C++
- Understanding Ownership, Borrow, Move semantics and Lifetimes
- Other features (depending on availability of time)

The Stack

```c
// a0.c

int main()
{
    int a = 1, b = 2;
    return 0;
}
```

The Stack

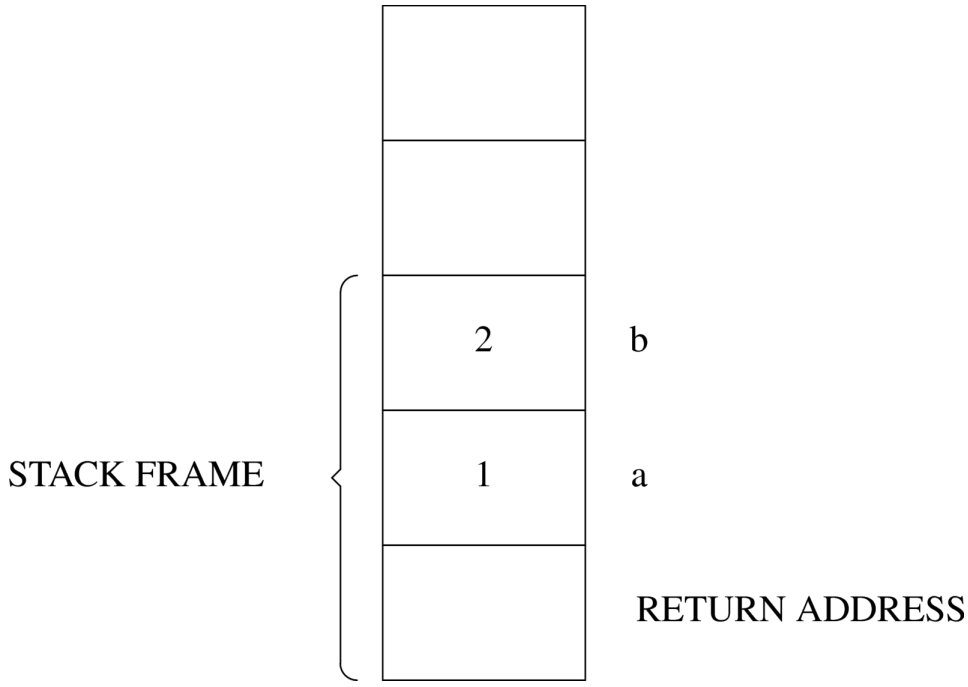

Figure 1: The C stack

Buffer overflow

```c
// a1.c
int main()
{
    int i=1;
    int a[4];
    int j=2;

    a[4] = 5; // bug
    a[5] = 6; // bug
    a[10000] = 7; // bug
}
```

Buffer overflow

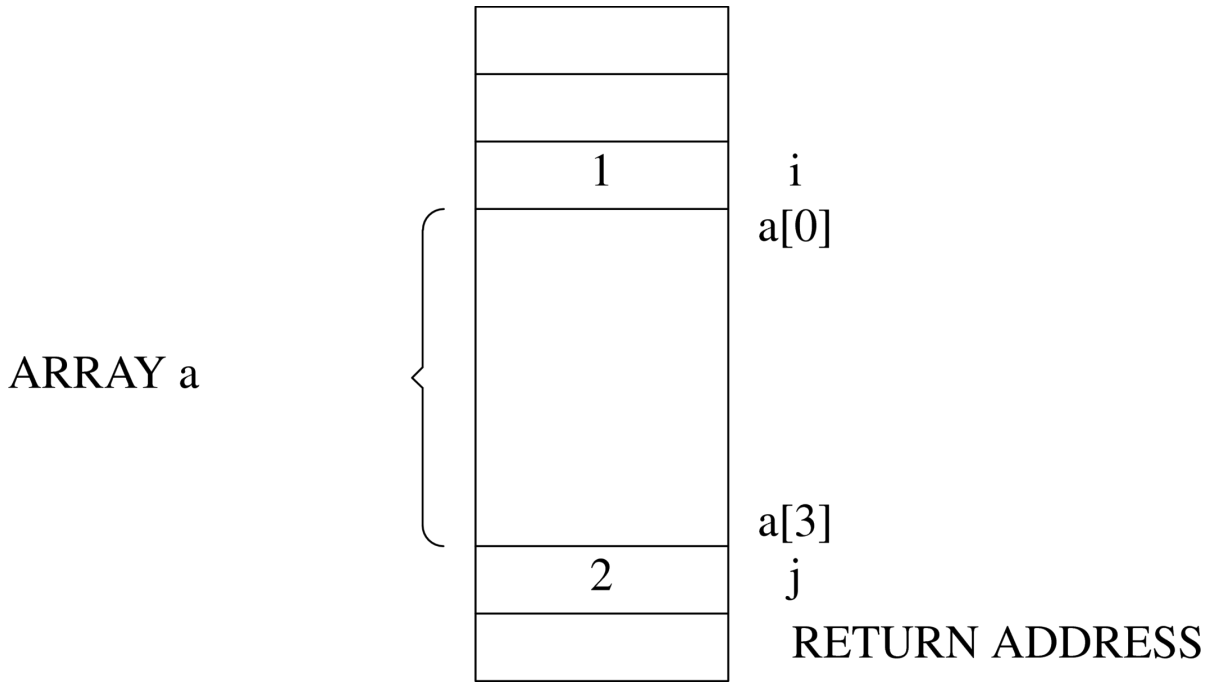

Figure 2: Buffer overflow diagram

Pointers in C

```c
// a2.c
int main()
{
    int a = 10;
    int *b;

    b = &a;
    *b = 20;
}
```

Pointers in C

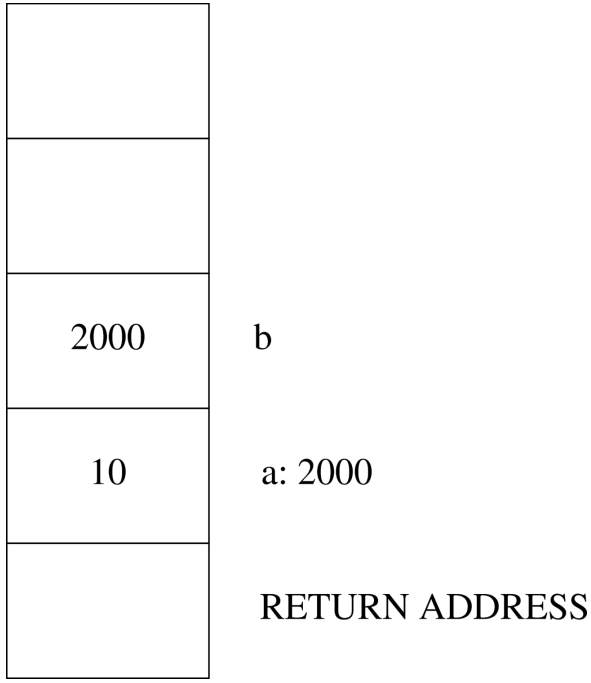

Figure 3: How pointer variables are stored in memory

Stack Frames - deallocations and allocations

```c
// a3.c
void fun2() { int e=5, f=6; }
void fun1() { int c=3, d=4; }

int main()
{
    int a=1, b=2;
    fun1(); fun2();
    return 0;
}
```

Stack Frames - allocations and deallocations

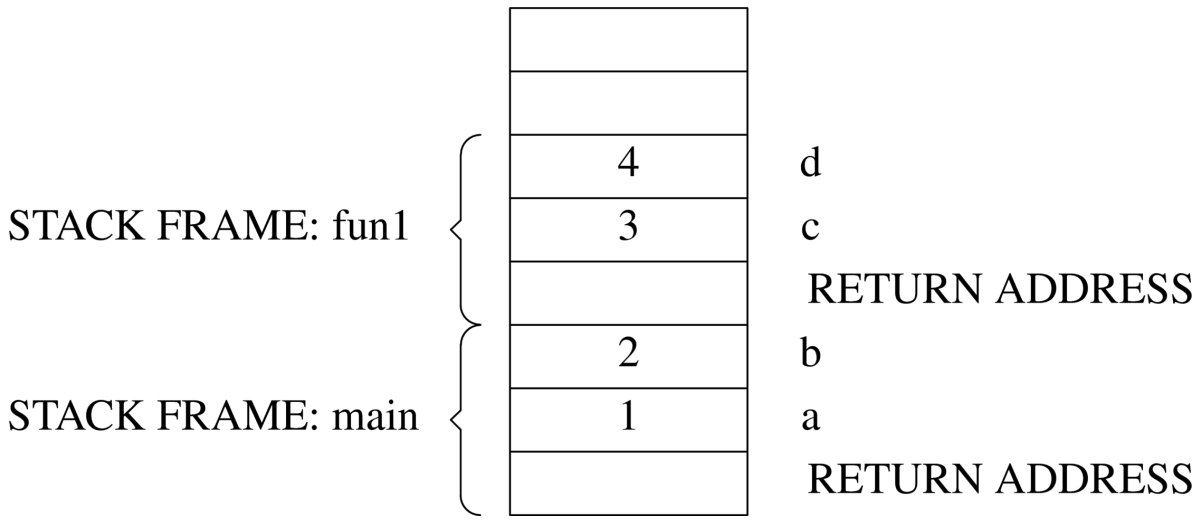

Figure 4: Multiple stack frames

Stack Frames - allocations and deallocations

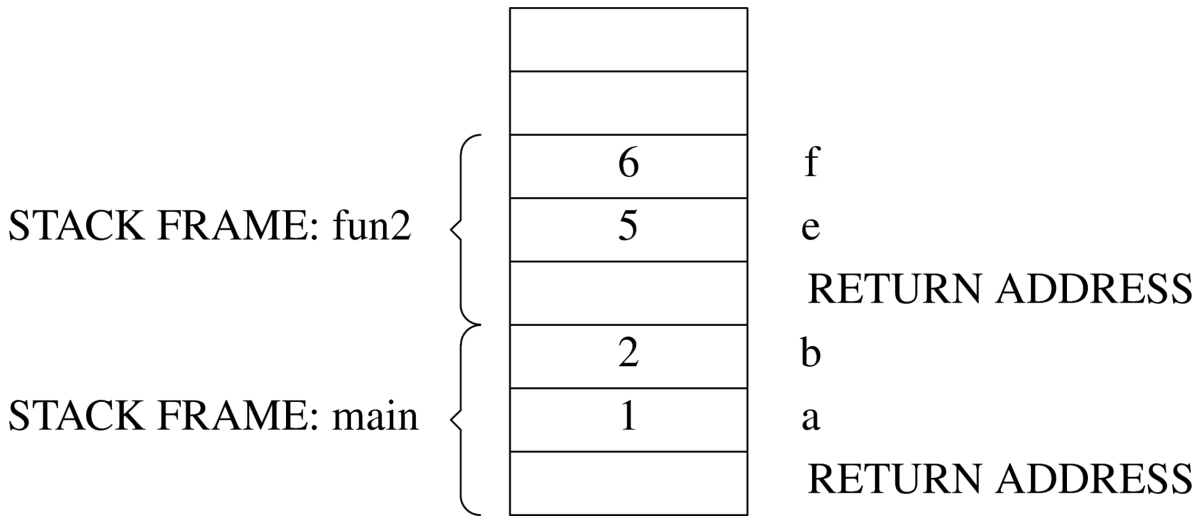

Figure 5: Multiple stack frames

Dangling Pointers

```c
// a4.c
void fun2() { int m = 1; int n = 2; }
int* fun1() {
    int *p; int q = 0;
    p = &q; return p; // bug
}

int main() {
    int *a, b; a = fun1();
    *a = 10; fun2();
    b = *a;
}
```

Dangling pointers

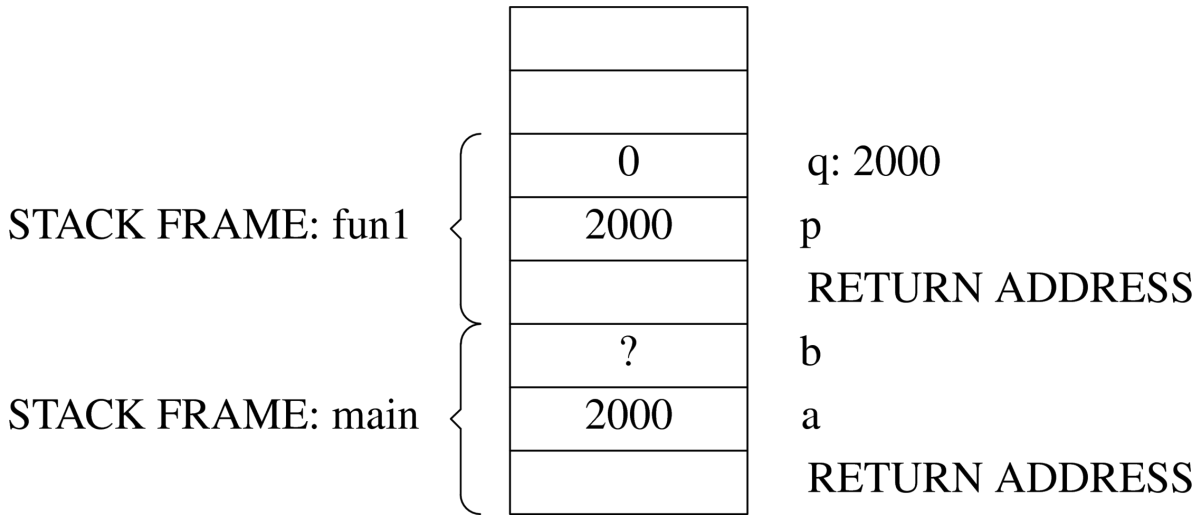

Figure 6: Dangling pointer

Null pointer dereferencing

```c
// a6.c
#include <strings.h>
#include <stdio.h>
int main()
{
    char s[100] = "hello";
    char *p;
    p = index(s, 'f');
    *p = 'a'; // bug!
    return 0;
}
```

Heap allocation - malloc and free

```c
// a7.c
#include <stdlib.h>
void fun()
{
    char *c;
    c = malloc(10*sizeof(char));
    /* do some stuff here */
    free(c);
}
int main()
{
    fun();
}
```

Heap allocation - malloc and free

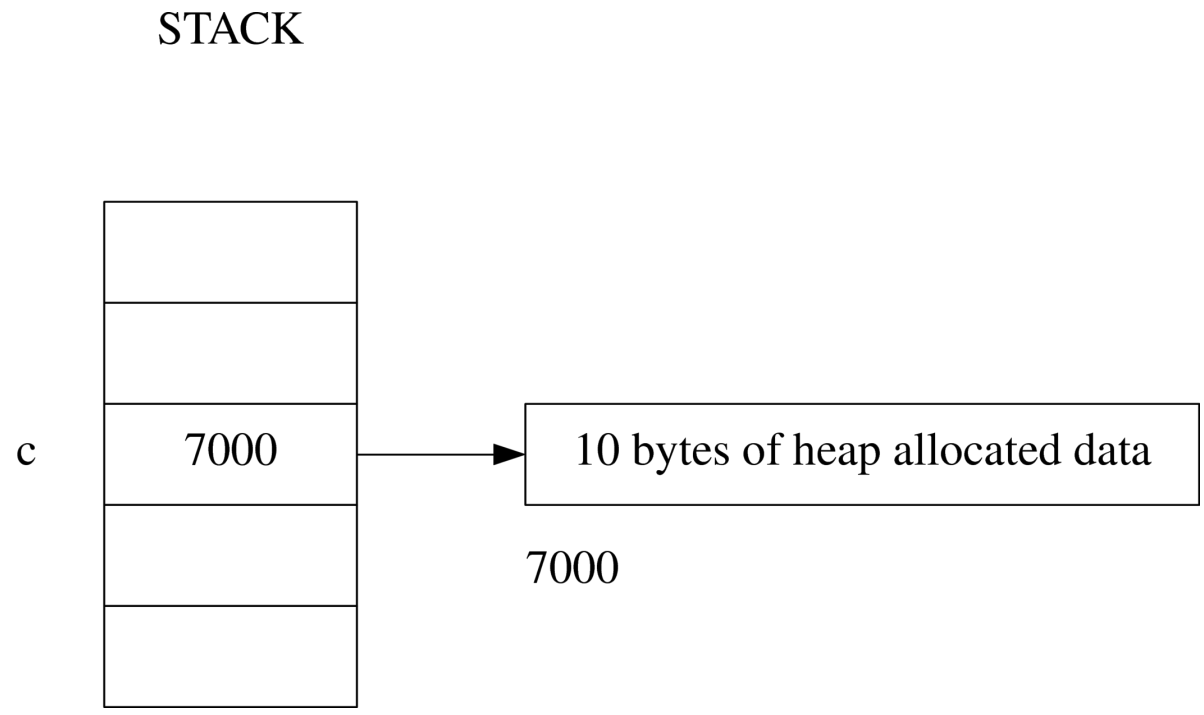

Figure 7: A stack location pointing to a heap location

Memory leaks

```c
// a8.c
#include <stdlib.h>
void fun()
{
    char *c;
    c = malloc(10*sizeof(char));
    /* do some stuff here */
}
int main()
{
    fun(); // bug! memory leak.
}
```

Use-after-free

```c
// a9.c
#include <stdlib.h>
void fun(char *t) {
    /* do some stuff here */
    free(t);
}
int main() {
    char *c;
    c =  malloc(10 * sizeof(char));
    fun(c);
    c[0] = 'A'; //bug! user-after-free
}
```

Double free

```c
// a10.c
#include <stdlib.h>
void fun(char *t) {
    /* do some stuff here */
    free(t);
}
int main() {
    char *c;
    c =  malloc(10 * sizeof(char));
    fun(c);
    free(c); //bug! double free
}
```

Undefined behaviours and optimization

```c
// a11.c
#include <limits.h>
#include <stdio.h>
// compile the code with optimization (-O3) and without
int main() {
    int c = INT_MAX;
    if (c+1 < c)
        printf("hello\n");
    printf("%d\n", c+1);
}
```

Undefined behaviours

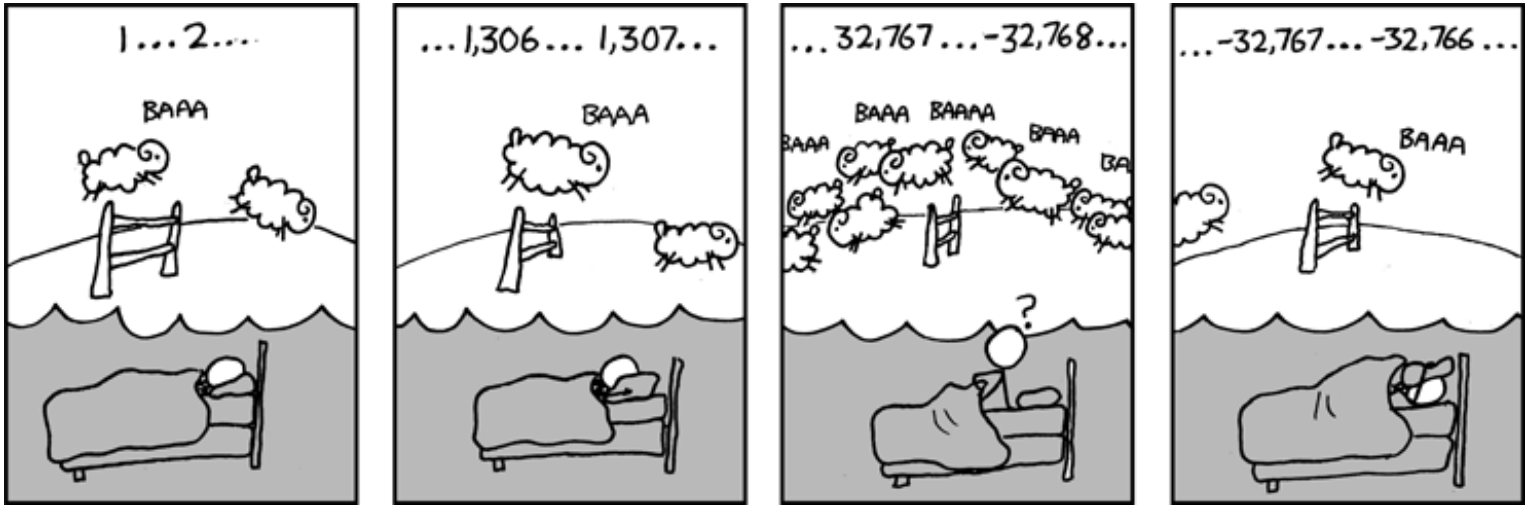

Figure 8: https://www.explainxkcd.com/wiki/images/c/cc/cant_sleep.png

Undefined behaviours

> *When tools like the bounds checking GCC, Purify, Valgrind, etc. first showed up, it was interesting to run a random UNIX utility under them. The output of the checker showed that these utility programs, despite working perfectly well, executed a ton of memory safety errors such as use of uninitialized data, accesses beyond the ends of arrays, etc. Just running grep or whatever would cause tens or hundreds of these errors to happen.*

From: http://blog.regehr.org/archives/226

Undefined behaviours

> *More and more, I'm starting to wonder how safety-critical code can continue being written in C.*

A comment on: http://blog.regehr.org/archives/232

Conclusion

Figure 9: rustpoem