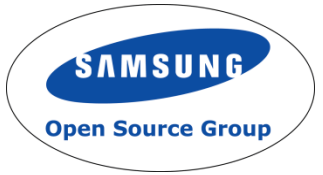


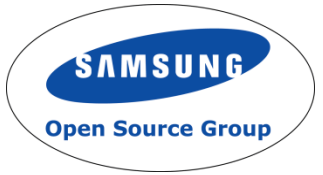
Rust for C++ programmers

Adenilson Cavalcanti
BSc MSc
WebKit & Blink committer
W3C CSS WG member

Summary

- Introduction
- The case for Rust
- Operators x traits
- Templates x generics
- Macros x attributes
- Threads x tasks
- Heap x boxes

Introduction

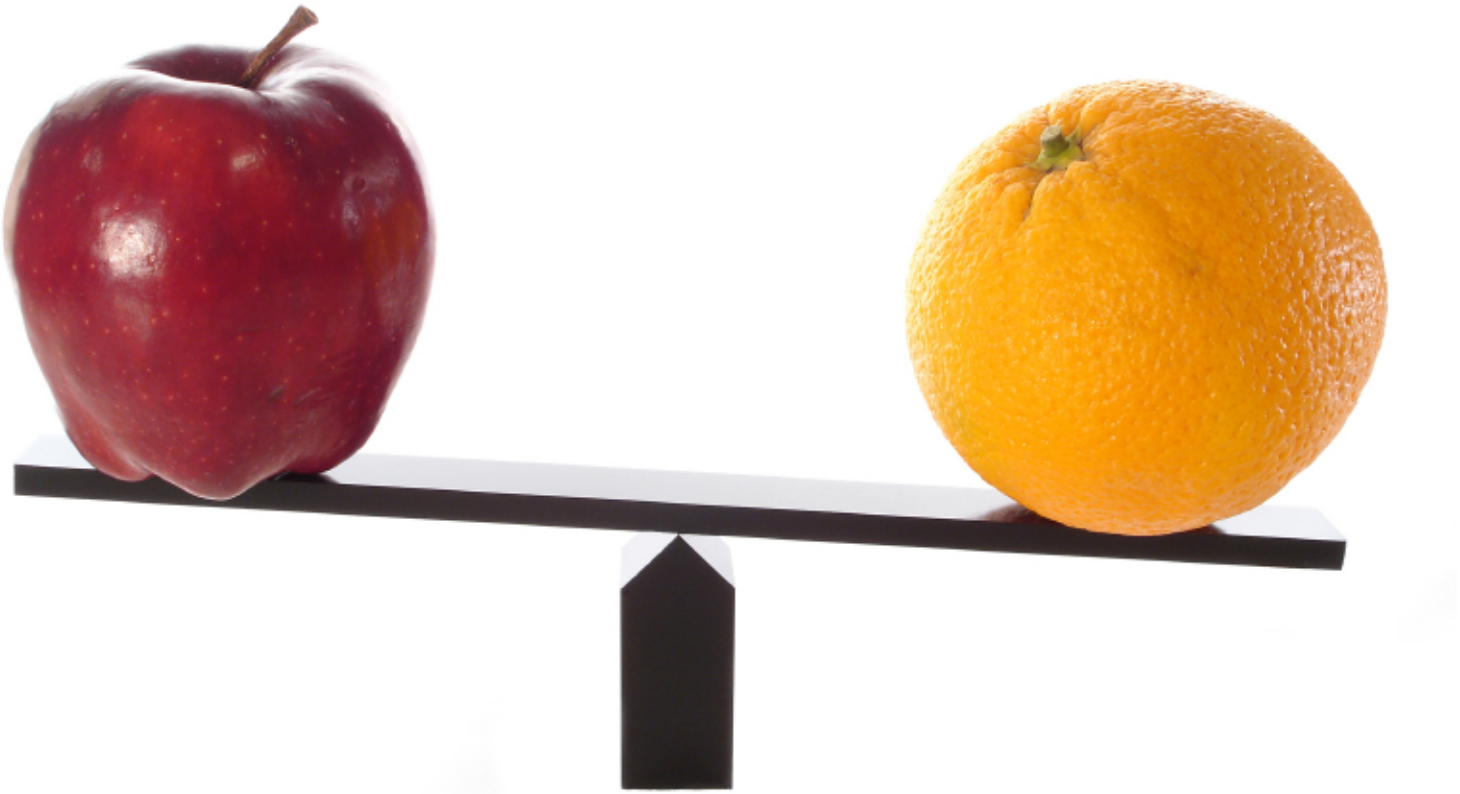


The problem:

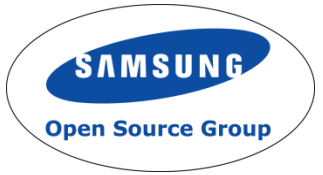
- **There are many C++ programmers**
- **But few Rust programmers**

The solution:
Why not explain Rust comparing it with modern C++11?

C++ x Rust



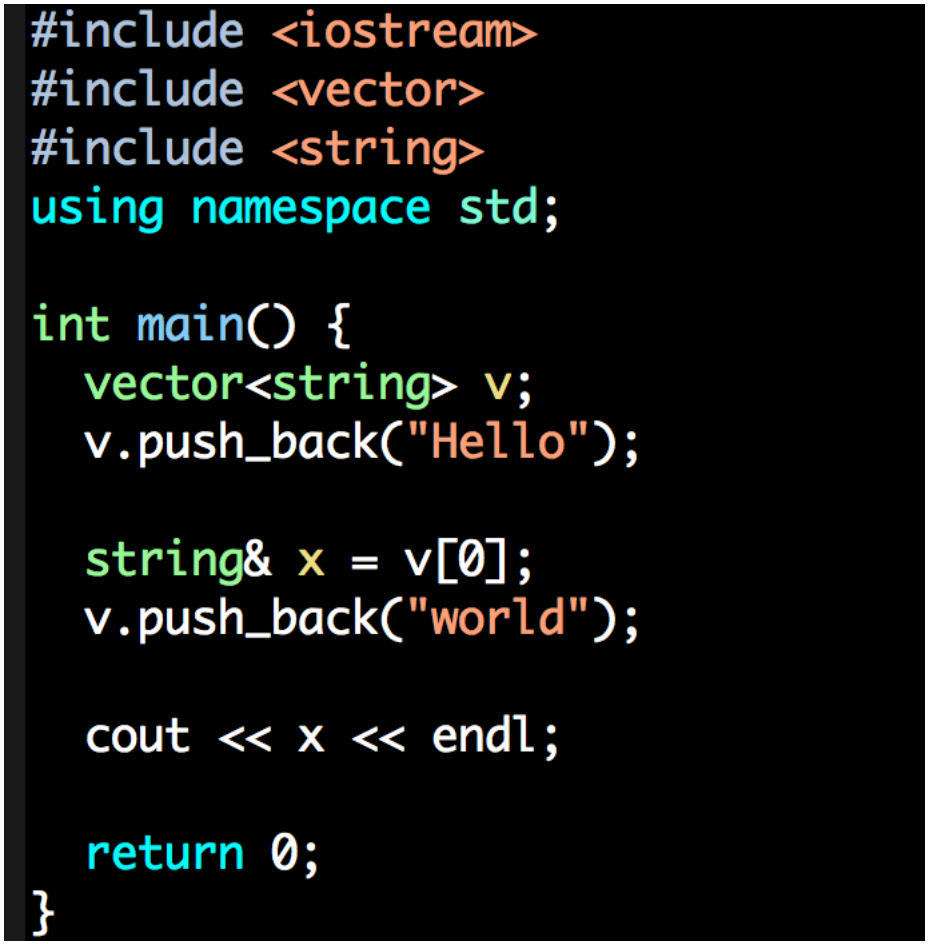
Why Rust?



The case for Rust

- It aims to solve some gruesome issues in C/C++.
- Be more expressive for hard tasks.
- Safety X performance: have both!

C++ example: anything wrong here?



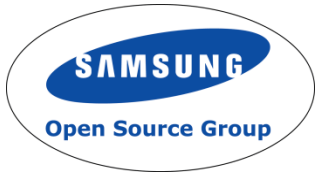
```cpp
#include <iostream>
#include <vector>
#include <string>
using namespace std;

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

  string& x = v[0];
  v.push_back("world");

  cout << x << endl;

  return 0;
}
```

http://doc.rust-lang.org/nightly/intro.html#ownership

Run on valgrind: warnings


```
a.cavalcanti ~/korea/rust4cpp $ valgrind ./crasher
==13007== Memcheck, a memory error detector
==13007== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==13007== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==13007== Command: ./crasher
==13007==
--13007-- ./crasher:
--13007-- dSYM directory is missing; consider using --dsymutil=yes
==13007== Invalid read of size 1
==13007==    at 0x100000FDD: std::__1::basic_ostream<char, std::__1::char_traits
<char> >& std::__1::operator<< <char, std::__1::char_traits<char>, std::__1::all
ocator<char> >(std::__1::basic_ostream<char, std::__1::char_traits<char> >&, sto
::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char>
 > const&) (in ./crasher)
==13007==    by 0x100000E7F: main (in ./crasher)
==13007==  Address 0x100015a10 is 0 bytes inside a block of size 24 free'd
==13007==    at 0x6B07: free (in /usr/local/Cellar/valgrind/3.10.0/lib/valgrind/
vgpreload_memcheck-amd64-darwin.so)
```

Using gdb to understand what is happening…

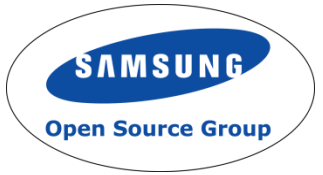



```
a.cavalcanti ~/korea/rust4cpp $ gdb ./crasher
GNU gdb 6.3.50-20050815 (Apple version gdb-1824) (Wed Feb  6 22:51:23 UTC 2013)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin"...Reading symbols for shared li
braries ... done

(gdb) b crash.cpp:10
Breakpoint 1 at 0x100000ae6: file crash.cpp, line 10.
(gdb) b crash.cpp:12
Breakpoint 2 at 0x100000cd3: file crash.cpp, line 13.
```

Address: 0x100103af0


```
(gdb) p v
$1 = {
  <std::__1::__vector_base<std::__1::basic_string<char>, std::__1::allocator<std
::__1::basic_string<char> > >> = {
    <std::__1::__vector_
    members of std::__1
llocator<std::__1::basic_
    __begin_ = 0x100103af0,
    __end_ = 0x100103b08,
    __end_cap_ = {
      <std::__1::__libcpp_compressed_pair_imp<std::__1::basic_string<char> *, st
d::__1::allocator<std::__1::basic_string<char> >, 2>> = {
        <std::__1::allocator<std::__1::basic_string<char> >> = {<No data fields>
},
        members of std::__1::__libcpp_compressed_pair_imp<std::__1::basic_string
<char> *, std::__1::allocator<std::__1::basic_string<char> >, 2>:
        __first_ = 0x100103b08
      }, <No data fields>}
  }, <No data fields>}
(gdb) c
Continuing.

Breakpoint 2, main () at crash.cpp:13
13          cout << x << endl;
```

Change! Now: 0x100103b10




```
Breakpoint 2, main () at crash.cpp:13
13          cout << x << endl;
(gdb) p v
$2 = {
  <std::__1::__vector_base<std::__1::basic_string<char>, std::__1::allocator<std
::__1::basic_string<char> > >> = {
    <std::__1::__vector_
    members of std::__1
llocator<std::__1::basic_
    __begin_ = 0x100103b10,
    __end_ = 0x100103b40,
    __end_cap_ = {
      <std::__1::__libcpp_compressed_pair_imp<std::__1::basic_string<char> *, st
d::__1::allocator<std::__1::basic_string<char> >, 2>> = {
        <std::__1::allocator<std::__1::basic_string<char> >> = {<No data fields>
},
        members of std::__1::__libcpp_compressed_pair_imp<std::__1::basic_string
<char> *, std::__1::allocator<std::__1::basic_string<char> >, 2>:
        __first_ = 0x100103b40
      }, <No data fields>}
  }, <No data fields>}
```

Address

Ref points to old address!

```
(gdb) p x
$3 = (string &) @0x100103af0: {
```

Compiler warnings won't catch it:

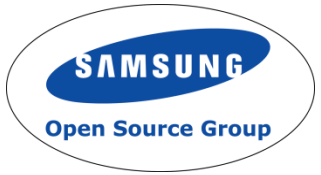
```
crash: crash.cpp
	g++ -pedantic -Wall -Wextra -Wcast-align -Wcast-qual -Wctor-dtor-privacy -Wdisabled-optimization -Wformat=2 -Winit-self -Wmissing-declarations -Wmissing-include-dirs -Wold-style-cast -Woverloaded-virtual -Wredundant-decls -Wshadow -Wsign-conversion -Wsign-promo -Wstrict-overflow=5 -Wswitch-default -Wundef -Werror -Wno-unused crash.cpp -o crasher
```

Rust to the rescue!

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

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

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

```cpp
#include <iostream>
#include <vector>
#include <string>
using namespace std;

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

  string& x = v[0];
  v.push_back("world");

  cout << x << endl;

  return 0;
}
```

Rust stops bugs from happening.



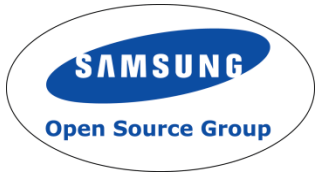

```
a.cavalcanti ~/korea/rust4cpp $ make rcrash
rustc rcrash.rs
rcrash.rs:6:5: 6:6 error: cannot borrow `v` as mutable because it is also borrow
ed as immutable
rcrash.rs:6     v.push("world");
                ^
rcrash.rs:5:14: 5:15 note: previous borrow of `v` occurs here; the immutable bor
row prevents subsequent moves or mutable borrows of `v` until the borrow ends
rcrash.rs:5     let x = &v[0];
                        ^
rcrash.rs:9:2: 9:2 note: previous borrow ends here
rcrash.rs:1 fn main() {
...
rcrash.rs:9 }
            ^
error: aborting due to previous error
make: *** [rcrash] Error 101
```

Introduction

How it is done

- A C++ example is given
- A similar Rust program is presented

Introduction



Objective

- Get used to Rust syntax
- Compare it with a mainstream language

C++ operators

SAMSUNG Open Source Group

Use

- Allows to overload operators (e.g. +, -, (), [], etc)
- Improves code readability and reuse

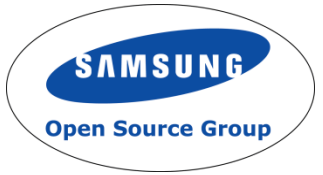
C++ operators

```
class Overloaded {
  int a, b;

public:
  Overloaded(int _a, int _b): a(_a), b(_b) {}

  Overloaded operator+(Overloaded &obj) {
    return Overloaded(a + obj.a, b + obj.b);
  }
};

int main(int argc, char* argv[])
{
  Overloaded a(10, 20), b(1, 2);
  Overloaded sumobj = a + b;
  return 0;
}
```

Rust traits

```
//http://doc.rust-lang.org/core/ops/
struct Overloaded {
    a: int,
    b: int,
}

impl Add<Overloaded, Overloaded> for Overloaded {
    fn add(&self, _rhs: &Overloaded) -> Overloaded {
        let mut res: Overloaded = Overloaded { a: 0, b: 0 };
        res.a = self.a + _rhs.a;
        res.b = self.b + _rhs.b;
        res
    }
}

fn main() {
    let a: Overloaded = Overloaded { a: 10, b: 20 };
    let b: Overloaded = Overloaded { a: 1, b: 2 };
    let sumobj: Overloaded;
    sumobj = a + b;
}
```

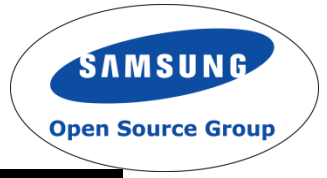
C++ templates



Use

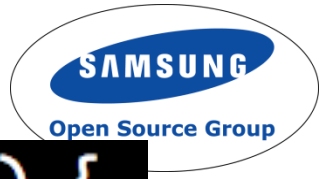
- Parametric types at compile type
- Improves performance and code reuse

C++ templates




```
template <typename T>
void swap(T &a, T &b) {
   T tmp = a;
   a = b;
   b = tmp;
}

int main(int argc, char* argv[])
{
   float f1(2.0), f2(1.0);
   swap(f1, f2);
}
```

Rust generics




```
fn swap<T: Clone>(x: &mut T, y: &mut T) {
    let t = x.clone();
    *x = y.clone();
    *y = t.clone();
}

fn main() {
    let mut a = 2i;
    let mut b = 1i;
    swap(& mut a, & mut b);
    println!("a: {}\t b: {}", a, b);
}
```

//http://doc.rust-lang.org/src/core/home/rustbuild/src/rust-buildbot/slave/nightly-linux/build/src/libcore/mem.rs.html#270-284
http://en.cppreference.com/w/cpp/utility/move

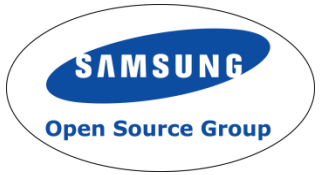
C++ macros

SAMSUNG Open Source Group

Use

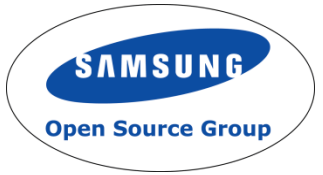
- Conditional compilation, performance code, etc
- When possible, use inline functions instead.

C++ macros




```
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {

#ifdef __APPLE__
  cout << "Hello, I'm running in OSX!" << endl;
#else __linux__
  cout << "Hello, I'm running in Linux!" << endl;
#endif

  return 0;
}
```

Rust attributes




```
#[cfg(target_os = "macos")]
fn where_am_i() {
    println!("Hello, I'm running in OSX!")
}

#[cfg(target_os = "linux")]
fn where_am_i() {
    println!("Hello, I'm running in Linux!")
}

fn main() {
    where_am_i();
}
```

C++ threads

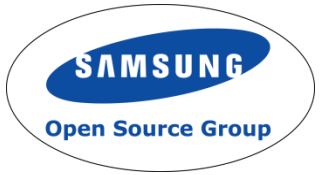
SAMSUNG Open Source Group

Use

- Parallelism, performance.
- Can get messy pretty fast.

```cpp
#include <thread>
#include <vector>
#include <iostream>

using namespace std;

static const int ntasks = 10;

void thread_code() {
  cout << "this is thread: " << std::this_thread::get_id() << endl;
}

int main(int argc, char *argv[])
{
  vector<thread*> jobs;

  for (int i = 0; i < ntasks; ++i) {
    jobs.push_back(new thread(thread_code));
  }

  for (auto* task: jobs) {
    task->join();
    // Try to comment this and run inside valgrind! :-)
    delete task;
  }
}
```

Rust tasks




```
static NTASKS: int = 10;

fn main() {
    for i in range(0, NTASKS) {

        spawn(proc() {
                println!("this is task number {}", i)
            });
    }
}
```

C++ heap

SAMSUNG
Open Source Group

Use

- Memory allocation of objects (i.e. stack has a fixed size).
- Be careful with leaks (smart pointers are your friend).

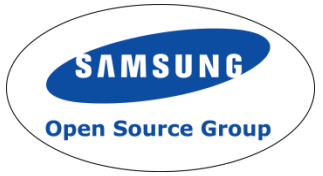
C++ heap use

```
#include <iostream>
#include <memory>
using namespace std;

struct Point {
  long x, y;
  Point(long _x, long _y): x(_x), y(_y) {}
};

int main(int argc, char *argv[])
{
  Point stack(0, 0);
  auto_ptr<Point> heap(new Point(1, 1));
  cout << "stack size: " << sizeof(stack) << "\theap: "
       << sizeof(heap) << endl;

  return 0;
}
```

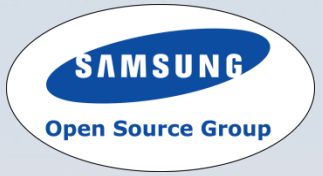
Rust boxes


```
use std::mem;

struct Point {
    x: int,
    y: int,
}

fn main() {
    let stack = Point { x: 0, y: 0 };
    let heap: Box<Point> = box Point { x: 1, y: 1 };

    println!("size stack: {}\theap: {}",
             mem::size_of_val(&stack),
             mem::size_of_val(&heap));
}
```

Conclusions



- Rust has equivalent features to modern C++;
- The same programming style won't work to both languages: you have to code in the right style;
- For smaller tasks, C++ may have more compact code;
- For more complex tasks (i.e. multitask), Rust seems to be more expressive;
- Most importantly: Rust avoids common traps in C++.



Questions?

Thank you.