R Rust interface

RUST RHYMES WITH TRUST
— ANON.

R.1 Rust history R–1
R.2 Numbers in Rust R–2
R.3 Simple programs in Rust R–2
R.4 Variables in Rust R–5
R.5 Arrays in Rust R–5
R.6 Decoding floating-point numbers R–6
R.7 Calling C functions with floating-point arguments R–7
R.8 Calling C functions with pointer arguments R–9
R.9 Calling C functions with string arguments R–9
R.10 The Rust math library R–11
R.11 TO DO R–11

The Rust language was developed about the same time as Go, and after the `MathCW` library. The syntax of the two languages is quite different, but both languages have similar goals of strong type safety and efficient multithreading, both are compiled into native code, and both promise upward compatibility for long-term stability of software. Unlike Go, Rust does *not* supply run-time garbage collection, but Rust's handling of variables largely removes the need to free objects when they are no longer needed. We address that topic later in Section R.9 on page 10. Nevertheless, the experience of writing an interface from Go to the `MathCW` library was helpful in making a companion interface for Rust.

R.1 Rust history

Development of Rust began about 2006 as a side project of its chief architect, Graydon Hoare. Since 2009, it has been sponsored by Mozilla, and supported by a large team of volunteer programmers.

Two editions of the book *The Rust Programming Language*,[1] by Steve Klabnik and Carol Nichols, are a primary source of information about the language, and there are several other books, and easily findable online tutorials as well.[2]

At the time of writing this, it appears that there is only one major compiler for Rust,[3] but it runs on most common CPU types.[4] The Rust compiler, `rustc`, is available in the binary package systems for many distributions of GNU/LINUX, most variants of the BSD family, Apple MACOS, Oracle SOLARIS, and Microsoft WINDOWS. Another completely independent compiler, `mrustc`, is in development.[5]

[1] First edition: No Starch Press (2017), xxvii + 519 pages, ISBN-10 1-59327-828-4, ISBN-13 978-1-59327-828-1, LCCN QA76.73.R87 K53 2018. Second edition, No Starch Press (2019), xxix + 526 pages, ISBN-10 1-71850-044-0, ISBN-13 978-1-71850-044-0, LCCN QA76.73.R87.

[2] See, for example, `https://doc.rust-lang.org/book/`, `https://doc.rust-lang.org/reference/`, and `https://www.rust-lang.org/learn`.

[3] See `https://github.com/rust-lang/rust`.

[4] See *Rust Platform Support* at `https://forge.rust-lang.org/release/platform-support.html`.

[5] See `https://github.com/thepowersgang/mrustc/`.

R.2 Numbers in Rust

The signed and unsigned integer types are `i8`, `i16`, `i32`, `i64`, `i128`, `u8`, `u16`, `u32`, `u64`, and `u128`. Those names can be used as suffixes on integer constants when necessary. There are pointer-sized integer types called `isize` and `usize` that are used extensively in the language and its support libraries. A separate Boolean type, `bool`, occupies one byte and has values `true` and `false`. The integer types correspond to common hardware types on modern CPUs, are stored in CPU-dependent byte order, and are in two's-complement encoding.

Only two floating-point types, and number suffixes, are provided: `f32` and `f64`. Floating-point arithmetic conforms to the IEEE 754 standards, and the `rustc` compiler permits compile-time and run-time generation of Infinity, NaN, and signed zeros, and correctly supports a simple NaN test of the form `if x != x { error("x is a NaN"); }`.

Arithmetic operators in Rust are similar to those in C. Single underscores may separate digits of any numeric constant, or a trailing digit and a type suffix. However, there is regrettably no support in Rust for the hexadecimal floating-point constants introduced in C99.

R.3 Simple programs in Rust

In this appendix, user command lines are colored, highlighted code lines are discussed in the following text, and display lines are sometimes numbered for later reference.

Here is the traditional greeting program in Rust, and its execution:

```
1  % cat hello.rs
2  fn main()
3  {
4      println!("Hello, world ... this is the Rust language");
5  }
6
7  % rustc hello.rs && ./hello
8  Hello, world ... this is the Rust language
```

That short program could have been collapsed to a single line, but we prefer a more standard layout. Notice that no header files or package commands are needed in this example: I/O facilities are already known to the compiler. The exclamation mark after the name in line 4 indicates that it is not really a function, but rather, a compile-time macro.[6]

Here is a second program that prints a small numerical table:

```
% cat numbers.rs
fn main()
{
    println!(" k  k-as-f32  1/(k-as-f32)");

    for k in 0 .. 11
    {
        println!(" {:2} {:6.1} {:14.8}", k, k as f32, 1.0f32 / (k as f32));
    }
}

% rustc numbers.rs && ./numbers
 k  k-as-f32  1/(k-as-f32)
  0    0.0            inf
  1    1.0     1.00000000
  2    2.0     0.50000000
```

[6] See `https://doc.rust-lang.org/0.10/guide-macros.html`.

```
 3    3.0     0.33333334
 4    4.0     0.25000000
 5    5.0     0.20000000
 6    6.0     0.16666667
 7    7.0     0.14285715
 8    8.0     0.12500000
 9    9.0     0.11111111
10   10.0     0.10000000
```

The Rust `for` loop here is an iterator: the variable `k` is automatically typed as `usize` and is available only inside the loop. It takes values from the beginning of the range up to, but *excluding*, the end of the range. An alternate operator `..=` is available to *include* the range end, but is less commonly used. You can step through the range in reverse with `for k in (0 .. 11).rev()`.

Our example introduces the unusual method of type conversion in Rust: it is neither a C-like parenthesized type cast, nor a built-in function, but instead uses the keyword `as` followed by a type name.

The Rust `format!()` function that underlies the `println!()` function provides a powerful set of braced formatting directives; our example uses only decimal output, but binary, octal, and hexadecimal variants are supported, along with left, right, and center justification of output values.[7]

Here is a third example showing extraction and printing of character values from a Unicode string in UTF-8 encoding:

```
% cat chars.rs
fn main()
{
    println!(" c  c-as-i32");

    for c in "La saison d'été".chars()
    {
        println!(" {}   {:^8}", c, c as i32);
    }
}

% rustc chars.rs && ./chars
 c  c-as-i32
 L      76
 a      97
         32
 s     115
 a      97
 i     105
 s     115
 o     111
 n     110
         32
 d     100
 '      39
 é     233
 t     116
 é     233
```

Character strings in Rust are immutable, and unlike C, cannot be indexed directly. A character string is stored as a three-element structure containing a pointer to an array of UTF-8 glyph values, a current length, and a capacity.

[7] See `https://doc.rust-lang.org/std/fmt/`.

Each glyph value requires one to four bytes, and is guaranteed to be a valid UTF-8 sequence. The `chars()` method suffix on the string extracts the array part as glyph numbers, *not* bytes, and the iterator steps over them in turn.

Here is another example that shows extraction of substrings:

```
% cat strslice.rs
fn main() {
    let a = "Hello, world ... this is the Rust language";
    let u = "Hello Olá Gruß";

    println!("              0000000000111111111122222222223333333333444");
    println!("              0123456789012345678901234567890123456789012");
    println!("a          =  {}",  a);
    println!("&a         =  {}",  &a);
    println!("a          = {:?}", a);
    println!("&a         = {:?}", &a);
    println!("&a[ 0.. 5] = {}",   &a[ 0.. 5]);
    println!("&a[ 7..12] = {}",   &a[ 7..12]);
    println!("&a[34..42] = {}",   &a[34..42]);

    println!("");

    println!("                         000000000011111");
    println!("                         012345678901234");
    println!("u                     =  {}",  u);
    println!("&u                    =  {}",  &u);
    println!("u                     = {:?}", u);
    println!("&u                    = {:?}", &u);
    println!("&u[0..5]              =  {}",  &u[0..5]);
    println!("&u[6..16]             =  {}",  &u[6..16]);
    println!("u.chars()             = {:?}", &u.chars());
    println!("u.bytes()             = {:?}", &u.bytes());
    println!("u.to_string()         = {:?}", u.to_string());
    println!("u.to_string()         =  {}",  u.to_string());
    println!("&u.to_string()[6..16] = {:?}", &u.to_string()[6..16]);
}

% rustc strslice.rs && ./strslice
              0000000000111111111122222222223333333333444
              0123456789012345678901234567890123456789012
a          =  Hello, world ... this is the Rust language
&a         =  Hello, world ... this is the Rust language
a          = "Hello, world ... this is the Rust language"
&a         = "Hello, world ... this is the Rust language"
&a[ 0.. 5] = Hello
&a[ 7..12] = world
&a[34..42] = language

                         000000000011111
                         012345678901234
u                     =  Hello Olá Gruß
&u                    =  Hello Olá Gruß
u                     = "Hello Olá Gruß"
&u                    = "Hello Olá Gruß"
```

```
&u[0..5]              =  Hello
&u[6..16]             =  Olá Gruß
u.chars()             = Chars(['H', 'e', 'l', 'l', 'o', ' ', 'O', 'l', 'á',
u.bytes()             = Bytes(Cloned { it: Iter([72, 101, 108, 108, 111, 32,
u.to_string()         = "Hello Olá Gruß"
u.to_string()         =  Hello Olá Gruß
&u.to_string()[6..16] = "Olá Gruß"
```

The first string contains only ASCII characters, so each takes only one byte, and the range selectors easily extract substrings. However, the second string has two non-ASCII characters, á and ß, each requiring two bytes. The range selectors are in *bytes*, not glyphs, and have to be chosen carefully to avoid a fatal error from indexing into the middle of a multibyte glyph. The braced format directives illustrate variations in output formatting.

R.4 Variables in Rust

One of the more unusual features of Rust is its treatment of variables in functions. The previous example showed two variables beginning with the keyword `let` and ending with assignments of initial values. What they did not reveal is that such a declaration creates a name for a value, but that value cannot be changed later. To declare a mutable variable requires two keywords: `let mut`. The compiler warns if such a variable is assigned only once. It is an error in Rust to use a variable before it is assigned a value, and the compiler warns of multiple assignments without intervening uses of the variable. Here are some examples of declarations with, and without, initializers:

```
let v     : f32 = 0.0;
let mut w : f32 = -1.0;
let x     : f32;
let mut y : f32;
const Z   : f32 = 1.25;
```

The compiler warns if named constants are not spelled entirely in uppercase letters.

R.5 Arrays in Rust

One-dimensional arrays are declared with a bracketed type and length, separated by a semicolon, as in this example:

```
% cat vecinit.rs
fn main()
{
    const N   : usize = 5;
    let mut x : [f64 ; N] = [1.0 ; N];
    let mut y : [f64 ; N] = [0.0 ; N];
    let z     : [f64 ; N] = [0.0, 1.0, 2.0, 3.0, 4.0];

    for k in 0 .. N
    {
        x[k] = -x[k];
        y[k] = x[k] - 3.0;
    }

    println!("N = {}",    N);
    println!("x = {:?}",  x);
    println!("y = {:?}",  y);
```

```
    println!("z = {:+?}", z);
}
```

```
% rustc vecinit.rs && ./vecinit
N = 5
x = [-1.0, -1.0, -1.0, -1.0, -1.0]
y = [-4.0, -4.0, -4.0, -4.0, -4.0]
z = [+0.0, +1.0, +2.0, +3.0, +4.0]
```

They should generally be initialized at the point of declaration to avoid compiler warnings about unused variables. The bracketed initializers of `x` and `y` repeat the initial value `N` times. For `z`, we used instead an explicit comma-separated list of `N` elements. The format directives `{:?}` and `{:+?}` result in the printing of entire arrays.

Multidimensional array declarations, however, have a surprise in both the syntax and the order of the dimensions:

```
% cat multidim.rs
const K: usize = 2;
const M: usize = 3;
const N: usize = 4;

fn main()
{
    let m2 = [ [2.0; N]; M];     // NB: array of size M x N!
    let m3 = [[[3.0; N]; M]; K]; // NB: array of size K x M x N!

    println!("m2 = {:?}", m2);
    println!("m3 = {:?}", m3);
}
```

```
% rustc multidim.rs && ./multidim
m2 = [[2.0, 2.0, 2.0, 2.0], [2.0, 2.0, 2.0, 2.0], [2.0, 2.0, 2.0, 2.0]]
m3 = [[[3.0, 3.0, 3.0, 3.0], [3.0, 3.0, 3.0, 3.0], [3.0, 3.0, 3.0, 3.0]],
      [[3.0, 3.0, 3.0, 3.0], [3.0, 3.0, 3.0, 3.0], [3.0, 3.0, 3.0, 3.0]]]
```

The types of `m2` and `m3` are inferred from that of the initializers, here `f64`, that being a common practice in some Rust software. Explicit typing is done like this:

```
let mut m2 = [ [2.0 as f64; N]; M];
let mut m3 = [[[3.0 as f64; N]; M]; K];
```

R.6 Decoding floating-point numbers

Rust provides two ways to access the bits of a floating-point number: either via a C-style union, or via the `to_bits()` method. Here is an example of both approaches:

```
% cat getbits.rs
fn main()
{
    #[repr(C)]                          // for union syntax support
    union un32 { x : f32, i : u32 };
    let u = un32 { x : 1.0 };

    unsafe
    {
```

```
10              println!("u.x = {}",      u.x);
11              println!("u.i = {:#08x}", u.i);
12          }
13
14          let e : u32; let f : u32; let h : u32;
15          let j : u32; let s : u32; let y : f32;
16
17          y = -1.0;
18          j = y.tobits() as u32;
19
20          println!("y   = {}",      y);
21          println!("j   = {:#08x}", j);
22
23          s = j >> 31;                        // sign bit
24          e = (j >> 23) & 0xff;               // biased 8-bit exponent
25          f = j & 0x003fffff;                 // stored 23-bit fraction
26          h = if e == 0 { 0 } else { 1 };     // hidden bit for significand
27
28          println!("sign            = {}",        s);
29          println!("biased exponent = {:#04.2x}", e);
30          println!("stored fraction = {:#08.6x}", f);
31          println!("significand     = {:#08.6x}", (h << 23) | f);
32      }
33
34      % rustc getbits.rs && ./getbits
35      u.x = 1
36      u.i = 0x3f800000
37      y   = -1
38      j   = 0xbf800000
39      sign            = 1
40      biased exponent = 0x7f
41      stored fraction = 0x000000
42      significand     = 0x800000
```

That example introduces in line 4 a Rust attribute, `#[repr(C)]`, that applies to the following statement, and makes the `union` declaration known to the compiler. The following `let` declares and initializes the union variable `u`. However, use of its members requires a wrapper `unsafe {...}`. The `to_bits()` alternative in line 18 is easier, although it must be correctly cast with the trailing `as u32`.

Once an integer is available to hold the exact bit pattern of a floating-point number, it is straightforward to use shifting and masking to extract its fields, as shown in lines 23–26.

In Rust, expressions return values, but statements do not. However, the `if ... else if ... else ...` construct is defined to be an *expression*, rather than a *statement*, so we use it that way in the assignment to `h` in line 26.

R.7 Calling C functions with floating-point arguments

To access a function in the C math library, `-lm`, requires definition of a suitable function prototype, followed by use of the function inside `unsafe {...}`:

```
% cat sqrt-in-c.rs
const EPSILON32 : f32 = 1.0f32 / 8_388_608.0f32;

extern "C"
```

```
{
    fn sqrtf(x: f32) -> f32;
}

fn fma32(a : f32, b : f32, c : f32) -> f32
{
    ((a as f64) * (b as f64) + (c as f64)) as f32
}

fn main()
{
    let mut x : f32;
    let mut r : f32;
    let mut s : f32;

    for k in 1 ..= 10
    {
        x = k as f32;
        unsafe { s = sqrtf(x); }
        r = fma32(s, s, -x) / x;        // fma32() for accurate computation of s*s - x
        println!("k = {:2}  x = {:2}  sqrtf(x) = {:1.8e}  relerr = {:-8.2e} = {:+5.3} ulps",
                 k, x, s, r, r / EPSILON32);
    }
}
```

```
% rustc sqrt-in-c.rs && ./sqrt-in-c
k =  1  x =  1  sqrtf(x) = 1.00000000e0  relerr =   0.00e0 = +0.000 ulps
k =  2  x =  2  sqrtf(x) = 1.41421354e0  relerr = -3.42e-8 = -0.287 ulps
k =  3  x =  3  sqrtf(x) = 1.73205078e0  relerr = -3.59e-8 = -0.301 ulps
k =  4  x =  4  sqrtf(x) = 2.00000000e0  relerr =   0.00e0 = +0.000 ulps
k =  5  x =  5  sqrtf(x) = 2.23606801e0  relerr =  2.94e-8 = +0.246 ulps
k =  6  x =  6  sqrtf(x) = 2.44948983e0  relerr =  7.28e-8 = +0.611 ulps
k =  7  x =  7  sqrtf(x) = 2.64575124e0  relerr = -5.53e-8 = -0.464 ulps
k =  8  x =  8  sqrtf(x) = 2.82842708e0  relerr = -3.42e-8 = -0.287 ulps
k =  9  x =  9  sqrtf(x) = 3.00000000e0  relerr =   0.00e0 = +0.000 ulps
k = 10  x = 10  sqrtf(x) = 3.16227770e0  relerr =  2.43e-8 = +0.203 ulps
```

The `extern "C"` block supplies a prototype for the C function, but with Rust-style argument and return types. Apart from the wrapper, use of the C function looks exactly that of a native function. The compiler knows how to find the C math library, so nothing further needs to be done to help it. Changing the function implementation to that in the `MathCW` library requires only an extra option:

```
% rustc sqrt-in-c.rs -lmcw && ./sqrt-in-c
```

The coding in our `fma32()` function deserves a remark. Many programming languages require function values to be supplied by a `return` statement with an expression, and Rust supports that. However, the recommended practice in Rust code is to provide the return value as the *last expression* in the function body, *without a trailing semicolon*. The compiler reports an error if you include that semicolon. The type promotions from `f32` to `f64` ensure that the computation of the fused multiply-add is accurate, and subject to only *one* rounding. The wider range of `f64` in IEEE 754 arithmetic guarantees that neither underflow nor overflow is possible in the product part, and the wider precision guarantees that the product is always exact.

R.8 Calling C functions with pointer arguments

For some of the functions in the `MathCW` library, we need to pass pointers to scalars and arrays. Here is an example that uses the accurate vector summation function, `vsum()`:

```
% cat vsum.rs
extern "C"
{
   fn vsum(perr: * mut f64, n: i32, v: * const f64) -> f64;
}

const N : usize = 101;

const TWO_TO_MINUS_46 : f64 = 1.0 / 70_368_744_177_664.0;

fn main()
{
    let mut err : f64 = 0.0;
    let mut v   : [f64 ; N] = [0.0 ; N];
    let sum     : f64;

    for k in 0 .. N
    {
        v[k] = k as f64;
        v[k] += TWO_TO_MINUS_46;
    }

    unsafe { sum = vsum(& mut err, N as i32, & v[0]); }

    println!("N = {}  sum = {:2}  err = {:.2e}", N, sum, err);
}

% rustc vsum.rs -lmcw && ./vsum
N = 101  sum = 5050.000000000001  err = 5.26e-13
```

The first argument to `vsum()` is a pointer to a scalar. The second argument is a C `int`, which on all modern platforms is a signed 32-bit value. The last argument is a pointer to a vector.

R.9 Calling C functions with string arguments

Because of the more complex representation of strings in Rust, calling a C function with a string argument requires run-time translation to a temporary string whose storage must somehow be freed after its use. Here is an example for setting the payload of a quiet NaN:

```
% cat nan.rs
use std::ffi::CString;
use std::os::raw::c_char;

extern "C"
{
   fn nan(tag: * const c_char) -> f64;
}

```

```
10      fn main()
11      {
12          let payload = "0xeffacefacade";
13          let qnan    = qnan_with_payload(payload);
14
15          println!("nan({:?}) -> {} = {:#018.16x}", payload, qnan, qnan.to_bits() as u64);
16      }
17
18      fn qnan_with_payload(payload : & str) -> f64
19      {
20          let c_payload = CString::new(payload.to_string()).expect("CString::new failed");
21
22          unsafe { nan(c_payload.as_ptr()) }
23      }
24
25      % rustc nan.rs -lmcw && ./nan
26      nan("0xeffacefacade") -> NaN = 0x7ff8effacefacade
```

Lines 2–3 reference two Rust packages for string conversion and storage management. The prototype for `nan()` on line 7 declares the argument as `* const c_char`; its type in C would be `const char *`. Lines 12–13 declare two immutable variables whose types are inferred from their initializers.

Foreign functions are regarded by Rust as unsafe, so we wrapped the call to the C function in a Rust function. Line 20 shows how the reference to a Rust string is converted at run time to an object from which a C string can be obtained and used in the next line.

What happens to the dynamically-allocated memory for `c_payload` in line 20 when `qnan_with_payload()` returns? In many programming languages, code like that would produce a *memory leak* that could lead to performance loss, or even run-time failure, if the function were called many times. Fixing that bug would require inserting a call to a storage deallocator just before the function return.

There is no bug, and no leak, here because of the unusual handling of variables in Rust functions. When objects are created in a function, they are automatically added to a *drop list*, and on return, all objects in that list are explicitly freed by a call to `drop()`. That design decision means that Rust programmers rarely need to think about deallocation: their programs run as if there were run-time garbage collection, even though there is none.

In rare cases, such as creation of large objects inside loops, it might be desirable to free storage early. In our case, we could do that with this minor revision:

```
fn qnan_with_payload(payload : & str) -> f64
{
    let c_tag = CString::new(payload.to_string()).expect("CString::new failed");
    let qnan : f64 = unsafe { nan(c_tag.as_ptr()) };
    drop(c_tag);        // release storage early
    qnan
}
```

All Rust objects have exactly one *owner*, and assigning one object to another transfers ownership. Similarly, passing an object to a function transfers ownership, and results in its destruction just before the function returns. That is why function arguments that are complicated objects are commonly passed by *reference*, as we did here with the `payload` argument, so they survive function calls.[8]

[8] For more on the Rust language concept of *ownership*, and whether variables are stored on the *stack* where they are available only during their scope, or on the *heap*, where explicit allocation and deallocation is needed, see `https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html`.

R.10 The Rust math library

Unlike in C, mathematical functions in Rust are part of the standard library, so no compiler library option `-lm` is needed. However, the functions are actually *methods*, which means that they can be used in two different styles, as shown in the highlighted lines in this example:

```
% cat sqrts.rs
fn main()
{
    let x : f32 = 5.0;
    let y : f32 = 7.0;
    println!("x = {}  x.sqrt()     = {}", x, x.sqrt());
    println!("y = {}  f32::sqrt(y) = {}", y, f32::sqrt(y));
}

% rustc sqrts.rs && ./sqrts
x = 5  x.sqrt()     = 2.236068
y = 7  f32::sqrt(y) = 2.6457512
```

The Rust mathematical function repertoire is extensive,[9] and similar to that in the C library, `-lm`. Notable omissions are equivalents of the useful C functions, `ldexp()`, `lgamma()`, `logb()`, `nextafter()`, `scalb()`, `scalbn()`, `scalbln()`, and `tgamma()`, as well as the several functions that provide access to IEEE 754 flags, rounding and precision control, and exception handling.

R.11 TO DO

- Implement `vsum()` in C and compare with output from `vsum2`: returned error seems inaccurate.
- Mention `cargo` build and documentation manager.
- Building a library of `MathCW` prototypes via awk.
- Using the `MathCW` prototype library.

[9] See `https://doc.rust-lang.org/std/primitive.f32.html` and `https://doc.rust-lang.org/std/primitive.f64.html`.