Hacking in C 2020

The C programming language

Thom Wiggers

iCIS | Digital Security
Radboud University

Table of Contents

Introduction

Undefined behaviour

Abstracting away from bytes in memory

Integer representations

Table of Contents

Introduction

Undefined behaviour

Abstracting away from bytes in memory

Integer representations

The C programming language

- Invented by Dennis Ritchie in 1972–1973
- Not one of the first programming languages: ALGOL for example is older.
  - Another predecessor is `B`.
- Closely tied to the development of the Unix operating system
- Unix and Linux are mostly written in C
- Compilers are widely available for many, many, many platforms
- Still in development: latest release of standard is `C18`. Popular versions are `C99` and `C11`.
- Many compilers implement extensions, leading to versions such as `gnu18`, `gnu11`.
- Default version in GCC `gnu11`

Programming for hardware

- Initially C was co-developed with Unix
- Unix is an operating system: main job is managing hardware
- C was used to replace the assembly code and implement new sofware for Unix
- Writing code in assembly:
  - Almost no abstraction
  - Full control
  - No types, no bounds checking: everything is just bytes to the CPU
  - Direct access to CPU and memory
  - Choice of instructions, register allocation left to programmer
  - Need to do everything from scratch for different CPUs
  - If a microarchitecture is released with new features, may need to re-implement parts of the code

Comparing C to assembly code

- C takes away some control from the programmer
- C is portable (*in theory*)
  - In practice, different compilers may not be fully compatible (Microsoft vs GCC)
  - You need to stay away from hardware-specific features and hardware-specific assumptions
  - You need to stay away from implementation-defined behaviour (turn on at least `-pedantic` on GCC)
- Compiler can translate C code to the target CPU
- Compiler can optimize code for you, for the target microarchitecture
- C still gives raw access to memory
- Gives you types to detect some errors, but lets you convert between any of them, often even implicitly.

Comparing C to C++

- C++ was originally developed from C
- C++ is not a strict superset of C.
  `int *x = malloc(sizeof(int) *10);` is valid C, but not C++!
  In C++ you will need to cast the `void*` pointer (but you should use `new` in C++).
- It is easy to write some code in C and then call it from C++ code, however.
  - Commonly used when high-performance code is written in C and a nice-to-use wrapper is written in C++.
  - Not restricted to C++, many languages have such a *foreign function interface* to link to libraries compiled from C.
  - For example: Numpy (Python) implements many core maths operations in C for performance reasons.

Table of Contents

Introduction

Undefined behaviour

Abstracting away from bytes in memory

Integer representations

Syntax and semantics

Syntax of a programming language

- Spelling and grammar rules
- Defines the language of valid programs
- Syntax errors are caught by the compiler
- Classical example: forget a ; at the end of a line

Semantics of a programming language

- Defines the meaning of a valid program
- In many languages semantics are fully specified
- Runtime errors (exceptions) are part of the semantics
- C is not fully specified!

Implementation-defined behaviour

- Some behaviour is unspecified by the standard and was left to be specified by the compiler.
- Reason: simplify compiler implementation and allow compilers to optimize things better.
- Often such behaviour is also specific to the hardware that you're running the software on
- Examples:
  - Order of subexpression evaluation: `f(g(), h())`.
  - Sizes of types (more later)
  - Signedness of `char`
  - Number of bits in a byte
- For most of this course, we assume GCC 7+ on a 64-bit AMD64 cpu.

Undefined behaviour

- Certain specific actions are defined as undefined behaviour (UB)
- When a program reaches UB, one or more of the following may happen
  - Crash every time
  - Crash 0.01% of the time
  - Crash not when you test it, but only you use it as a library
  - Delete everything on your hard drive
  - Murder some puppies
  - Light your house on fire
  - All of the above, and still give the right result
- The existence of UB anywhere in your program makes the entire thing meaningless!
  - Reason: compilers make assumptions based on it *not* existing, which may change the meaning of your program
- Often UB leads to exploitable security problems.

Examples of undefined behaviour

- Accessing memory out of bounds
- Reading uninitialized memory
- Division by zero
- Dereferencing a null pointer
- Signed integer overflow (`INT_MAX +1`)
- Left-shifting a signed integer (`(-42) << 3`)
- Shifting by more than the size of the type (`char x = 1; 1 << 100;`)
- Returning nothing from a non-void function (`int f() {}`)

Compilers can find some of these problems, but for weird reasons, those warnings are often switched off by default! Make sure you enable `-Wall -Wextra` when you compile code.

Table of Contents

Introduction

Undefined behaviour

Abstracting away from bytes in memory

Integer representations

Values

- A program typically applies operations to values (add, sub, mul)
- In assembly, you need to carefully manage where you store values
  - Limited number of registers, often necessary to spill to stack
- The compiler takes care of that for you in higher-level languages
- When calling a function `void f(int x) { x += 10;}` as `f(y)` you pass it the value of `y`.
  - this is called call-by-value
  - The compiler copies `x` if necessary
  - Modifying the passed value in `f` won't change it outside the function: `y=10; f(y); printf("y = %d\n", y);` will still print `10`.

Addresses

- You can get the address of a variable using the `&` operator:
  `int a; &a`
- You then obtain a pointer to `a`
- A pointer to a type is denoted as `type*`, e.g. `int*`, `char*`.

We will return to pointers later

Types

- The hardware only understands memory as a bunch of bytes that it can perform certain operations on
- Bytes are sets of 8 bits
- For writing software, other types are helpful to help determine semantics
  - it's helpful that a compiler gives an error when you call `strlen(3)`.
- You can program without really understanding how these types map to bytes.
- But we can have more fun if we do know how it works

char

- The most elementary data type
- Almost anywhere exactly 1 byte (required by POSIX)
- Can be used to store characters: `char a = '2';`
- But char is an 8-bit integer type
- We can just assign any 8-bit integer value to `char` types.

```
char a = '2';
char b = 2;
char c = 50;
```

- In fact, `a == c` because ASCII character '2' is 50.
- Writing `'A' + 3` is perfectly valid and will result in `'D'`.

Tricky `char`

How many times will the following line be printed?

```
for (char i = 42; i >= 0; i--) {
  printf("Crypto stands for cryptography");
}
```

- Trick question! It is compiler-defined if `char` is signed (-128–127) or unsigned (0–255).
- On `amd64`, `char` is signed, so it will terminate.
- On `Aarch64` (64-bit `ARMv8`), `char` is unsigned, so it will loop forever.
- Always write `signed char` or `unsigned char` in portable software.

Integral types

- Other types that are important:
  - `short`: at least two bytes
  - `int`: *typically* 4 bytes (but sometimes only two bytes!)
  - `long`: either 4 bytes or 8 bytes (different between Linux and Windows!)
  - `long long`: 8 bytes
- Each of these are in `signed` (default) and `unsigned` variants
- Find the size of a type: `printf("%zu\n", sizeof(int));`
- We can also do this via variable: `int x; sizeof(x);`
- We can write integer literals as:
  - Decimal: `255`
  - Octal: `0377` (prefix `0`)
  - Hexadecimal: `0xFF` (prefix `0x`)

Other integer types

- There is a special integer type to indicate sizes: `size_t`
- For example returned by `sizeof`, expected as argument by `malloc`
- Pointers also have a specific size, 8 bytes on amd64

Better integer types

- All those varying byte sizes of `int` et al. make it hard to write efficient portable code
- Solution: use fixed-size integer types defined by `stdint.h`
  - `uint8_t` is an 8-bit unsigned integer
  - `int8_t` is an 8-bit signed integer
  - `uint16_t` is a 16-bit unsigned integer
  - ...
  - `int64_t` is a 64-bit signed integer

Floating-point and complex values

- C also defines 3 "real" types:
  - `float`: usually 32-bit IEEE 754 "single-precision" floats
  - `double`: usually 64-bit IEEE 754 "double-precision" floats
  - `long double`:: usually 80-bit "extended precision" floats
- Corresponding "complex" types (need to include `complex.h`)
- This course: not much float hacking
- However, this is fun, see *"What every computer scientist should know about floating point arithmetic"*
  www.itu.dk/~sestoft/bachelor/IEEE754_article.pdf
- Small example:

```
double a; /* assume IEEE 754 standard */
// snip
a += 675539941055744;
a -= 675539941055744;
```

- What does this code do to a?
- Answer: it rounds a according to the currently set rounding mode

Excursion: `printf`

`printf` is a function that *print*s something according to a *f*ormat string.

```c
#include <stdio.h>
printf("%d", a); /* prints signed integers in decimal */
printf("%u", b); /* prints unsigned integers in decimal */
printf("%x", c); /* prints integers in hexadecimal */
printf("%o", c); /* prints integers in octal */
printf("%lu", d); /* prints long unsigned integer in decimal */
printf("%llu", d); /* prints long long unsigned integer in decimal
printf("%p", &d); /* prints pointers (in hexadecimal) */
printf("%f", e); /* prints single-precision floats */
printf("%lf", e); /* prints double-precision floats */
printf("%llf", e); /* prints extended-precision floats */
printf("%zu", f); /* prints a size_t as unsigned decimal*/
printf("%" PRIu8, g); /* prints a uint8_t */
printf("%" PRIu64, h); /* prints a uint64_t */
printf("%" PRId64, i); /* prints a int64_t */
printf("%" PRIx64, i); /* prints a (u)int64_t as hex */
```


Implicit type conversion

- Sometimes we want to evaluate expressions involving different types
- Example:

```
float pi, r, circ;
pi = 3.14159265;
circ = 2*pi*r;
```

- C uses complex rules to implicitly convert types
- Often these rules are perfectly intuitive:
  - Convert "less precise" type to more precise type, preserve values
  - Compute modulo 2^{16}, when casting from `uint32_t` to `uint16_t`
- However, these rules can be rather counterintuitive:

```
unsigned int a = 1;
int b = -1;
if(b < a) printf("all good\n");
else printf("WTF?\n");
```

Explicit casts

- Sometimes we need to convert explicitly
- Example: multiply two (32-bit) integers:

```
uint32_t a,b;
...
uint32_t r = a*b;
```

- By "default", result of `a*b` has 32-bits; upper 32 bits are "lost"
- Fix by casting one (or both) factors:

```
uint64_t r = (uint64_t)a*b;
```

- Can also use this to, e.g., truncate floats:

```
float a = 3.14159265;
float c = (int) a;
printf("%f\n", trunc(a));
printf("%f\n", c);
```

- Careful, this does not generally work (undefined behavior ahead)!

A small quiz

What do you think this program will print?

```
unsigned char x = 128;
signed char y = x;
printf("The value of y is %d\n", y);
```

(Obviously, the answer is "undefined behavior" – it's C after all)

Table of Contents

Introduction

Undefined behaviour

Abstracting away from bytes in memory

Integer representations

Two's complement

- Can represent a signed integer as "sign + absolute value"
- Disadvantage: zero has two representations (`0` and `-0`)
- Other idea: flip all bits in `a` to obtain `-a`.
- This is known as "ones complement"
- Still: zero has two representations
- Much more common: two's complement
  - flip all bits in `a`
  - add 1
- Sanity test: `a == -(-a)`
- Range of k-bit signed integer: $\{-2^{k-1}, \ldots, 2^{k-1} - 1\}$
- Example: signed (8-bit) byte: $\{-128, \ldots, 127\}$
- Can use the same hardware for signed and unsigned addition

Endianess

- Let's consider the 32-bit integer 287454020 =0x11223344
- How would you put it into memory. . . ,like this?:

| 11 | 22 | 33 | 44 |

0x0...0 0x0...1 0x0...2 0x0...3

- How about like this?

| 44 | 33 | 22 | 11 |

0x0...0 0x0...1 0x0...2 0x0...3

- What do you find more intuitive?

Endianess, let's try again

- Take 4-byte integer $a = \sum_{i=0}^{3} a_i 2^{8i}$
- The a_i are the bytes of a
- How would you put it into memory. . . ,like this?:

```
|    a0    |    a1    |    a2    |    a3    |

0x0...0    0x0...1   0x0...2   0x0...3
```

- Or would you rather have this?

```
|    a3    |    a2    |    a1    |    a0    |

0x0...0    0x0...1   0x0...2   0x0...3
```

- Again a quick poll: What do you find more intuitive?

Endianess, the conclusion

- Least significant bytes at low addresses: little endian
- Most significant bytes at low addresses: big endian
- This is short for "little/big endian byte first"
- Most CPUs today use little endian
- Examples for big-endian CPUs:
  - Classic PowerPC
  - UltraSPARC
- ARM and POWER8 can switch endianess (is "bi-endian"); usually used little-endian
- The problem with little-endian intuition is just that we write left-to-right (but use Arabic numbers)
- Endianness wil become important again when we need to write memory addresses later

Memory addresses

- On 32-bit x86 processors, addresses were 4 bytes.
- Current AMD64 processors support up to 2^{48} bytes of memory (256TiB)
  - This means you need 6 bytes to represent 2^{48} addresses
  - 8 Bytes are used for addresses though.
    - Upper 3 bytes are either in `0x000000...-0x00007f...`, or `0xffff80...-0xffffff...`
    - On Linux, the first is userspace and the second is kernelspace
    - `0x000080...-0xffff7f...` are not used

Back to pointers

We can print the address of a variable:

```
int a = 4; /* https://xkcd.com/221/ */
int* a_ptr = &a;
printf("The value of the variable     a = %d\n",  a);
printf("The address of the variable   a = %p\n", &a);
printf("The value of the variable a_ptr = %p\n",  a_ptr);
printf("The value pointed to by   a_ptr = %d\n", *a_ptr);
```

Output:

```
The value of the variable     a = 4
The address of the variable   a = 0x7ffd1be9fb8c
The value of the variable a_ptr = 0x7ffd1be9fb8c
The value pointed to by   a_ptr = 4
```

Variable a is stored very high in the user-space memory, because `int a` defines a stack variable.

Heap addresses

We can print the address of a variable:

```
int* a_ptr = malloc(sizeof(int));
*a_ptr = 4;  /* https://xkcd.com/221/ */
printf("The value stored at a_ptr = %d\n", *a_ptr);
printf("The value of        a_ptr = %p\n",  a_ptr);
free(a_ptr);  /* need to manually manage heap */
```

Output:

```
The value  a = 4
The addr  &a = 0x55b899d552a0
```

a_ptr is somewhere halfway user-space memory, as it is on the heap. Note that we have been writing *a_ptr to dereference the pointer, to get the value stored at the address!