Hacking in C

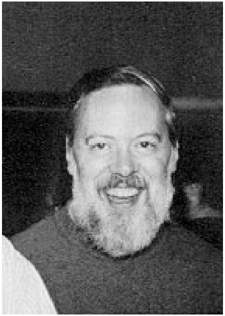
The C programming language

Radboud University, Nijmegen, The Netherlands

Spring 2018

The C programming language

- Invented by Dennis Ritchie in the early 70s
- First “Hello World” program written in C
- UNIX (and Linux) is written in C
- Still one of the top-5 most used programming languages
- Compilers for almost all platforms
- Many “interesting” security issues



Source: Wikipedia

C standards and “standards”

- First definition in Kernighan&Ritchie: *“The C Programming Language”*
- Also known as K&R C, book appared in 1978
- Standardized by ANSI in 1989 (**C89**) and ISO (**C90**)
- Second edition of K&R book used “ANSI C”, i.e., C89
- In 1995, ANSI published an amendment to the C standard (“C95”)
- In 1999, ISO standardized updated C, ANSI adopted (**C99**)
- Current standard is **C11**, standardized (ANSI and ISO) in 2011
- Standard draft online: `https://port70.net/~nsz/c/c11/n1570.html`
- Compilers like `gcc` or `clang` also support GNU extensions
- Default for `gcc`: C11 plus GNU extensions (aka gnu11)
- You can switch `gcc` to other C standards using, e.g., `-std=c89`
- Use `-pedantic` flag to issue warnings if your code doesn't conform to the standard

C vs. C++

- C is the basis for C++, Objective-C, and many other languages
- C is **not a subset of C++**, e.g.,

  ```
  int *x = malloc(sizeof(int) * 10);
  ```

  is valid (and perfectly reasonable) C, but not valid C++!
- You can "mix" C and C++ code, but you have to be very careful
- In C++, declare C functions as `extern "C"`, for example:

  ```
  extern "C" int mycfunction(int);
  ```

- Now you can call `mycfunction` from your C++ code
- Use compiler by the same vendor to compile
- Lets you use, e.g., highly optimized C libraries
- Common scenario:
  - Write high-speed code in C (and assembly)
  - Write so-called wrappers around this for easy access in C++

A “portable assembler”

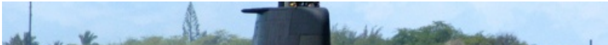
> *C has been characterized (both admiringly and invidiously) as a portable assembly language*
>
> —Dennis Ritchie

- Idea of assembly:
  - Programmer has full control over the program
  - Choice of instructions, register allocation etc. left to programmer
  - Programmer has “raw access” to memory
  - Need to rewrite programs for each architecture
  - Need to re-optimize for each microarchitecture
- Idea of C:
  - Take away some bits of control from the programmer
  - Stay as close as possible to assembly, but stay portable
  - In particular: give programmer raw access to memory
  - Use compiler to generate code for different architectures
  - Use compiler to optimize for different microarchitectures

“If programming languages were. . .”

- . . . vehicles
  http://crashworks.org/if_programming_languages_were_vehicles/
- . . . countries
  https://www.quora.com/If-programming-languages-were-countries-which-country-would-each-l
- . . . GoT characters
  https://techbeacon.com/if-programming-languages-were-game-thrones-characters
- . . . beer
  https://www.topcoder.com/blog/if-programming-languages-were-beer/
- . . . boats
  http://compsci.ca/blog/if-a-programming-language-was-a-boat/

“C is a nuclear submarine. The instructions are probably in a foreign language, but all of the hardware itself is optimized for performance.



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!

Unspecified behavior

- Unspecified behavior is “implementation-specific”
- Semantics not defined by the standard, but have to specified by the compiler
- Reason: allow better optimization
- Examples:
  - Shifting negative values to the right (e.g., `int a = (-42) >> 3`)
  - Order of subexpression evaluation (e.g., `f(g(), h())`)
  - Sizes of of various types (more later)
  - Representation of data types (more later)
  - Number of bits in one byte
- Fairly hard to write fully specified C programs
- For this course: if not otherwise stated assume `gcc` (version 6.x or 7.x) compiling for AMD64.

Undefined behavior

- Different from unspecified behavior: **undefined behavior**
- Program reaches a state in which it may do **anything**
  - It may crash with arbitrary error code
  - It may silently corrupt data
  - It may give the right result
  - The behavior may be "randomly" different in independent runs
- **Undefined behavior means that the whole program has no meaning anymore!**
- This is essentially always a bug, often security critical
- Examples:
  - Access an array outside the bounds
  - More generally: access memory at "illegal" position
  - Overflowing a signed integer (`(INT_MAX+1)`)
  - Left-shifting a signed integer (`(-42) << 3`)
- It is totally acceptable for a program to delete all your data when running into undefined behavior
- Sometimes we can *make* a program do this (or something similar)
- Most attacks in the course: exploit undefined behavior

C compilation

- Four steps involved in compilation, can stop at any of those
- First step: Run the preprocessor (`gcc -E`)
  - Include code from `#include` directives
  - Expand macros from `#define` directives
  - Expand compile-time (static) conditionals `#if`
  - The C preprocessor is almost Turing complete
  - See `https://github.com/orangeduck/CPP_COMPLETE` for a Brainfuck interpreter written in the C preprocessor
- Second step: Run compilation proper (`gcc -S`)
  - Go from C to assembly level
  - This is where you get syntax errors
- Third step: Generate machine code (`gcc -c`)
  - Generates so-called **object files**
- Fourth step: Linking (simply run `gcc`, this is default)
  - Put object files together to a binary
  - Linker errors include missing functions or function duplicates
  - Also include external libraries here (e.g., `-lm`)
  - Caution: order of arguments can matter!

Memory abstraction 1: where data is stored

- Programmers typically don't know where data is stored
- For example, a variable can sit in
  - a register of the CPU
  - in any of the caches of the CPU
  - in RAM
  - on the hard drive (in so-called swap space)
- Compiler makes decisions about register allocation
- Compiler has some bit of influence on caching
- Other decisions are made by the OS (and the CPU)
- Sometimes important: always read the variable from memory
- C has keyword `volatile` to enforce this
- Disables certain optimization

Where is data allocated?

- C has the & operator that returns the address of a variable
- Example:
  - Let's say we have a variable `int x = 12`
  - Now `&x` is the address where `x` is stored, aka a **pointer to `x`**
- Much more on pointers later, for the moment let's print them:

```
char x; int i; short s; char y;
printf("The address of x is %p\n", &x);
printf("The address of i is %p\n", &i);
printf("The address of s is %p\n", &s);
printf("The address of y is %p\n", &y);
```

- Note the %p format specifier for pointers
- The "inverse" of & is `*`, i.e., `*(&x)` gives the value of `x`

register

- Important task for the compiler: **register allocation**
- Map live variables (whose values are still needed) to registers
- Typical goal: minimize amount of “register spills”
- C lets programmers “help” the compiler with keyword `register`
- Quote from Erik’s slides:

  “you should never ever use this! Compilers are much better than you are at figuring out which data is best stored in CPU registers.”
- I agree that I never (?) use `register`
- Reason: I am (often) better than the compiler at figuring out which data is best stored in CPU registers…
- … and then I write in assembly and avoid the compiler alltogether
- Problem with `register`: no guarantee that the value isn’t spilled
- Requesting the address of a `register` variable is invalid!

Memory abstraction 2: how data is stored

- You can think of memory as an array of bytes
- For this course: a byte consists of 8 bits
- Computer programs work with different data types
- Important step of compilation: map other types to bytes
- Idea of C: you can program without needing to understand this mapping
- Idea of this course: you can have more fun with C if you do!
- The CPU likes to see the memory as an array of words
- Words typically consist of several bytes (e.g., 4 or 8 bytes)
- (Most) registers have the size of machine words
- Often loads and stores are more efficient when **aligned** to a word boundary
- von Neumann architecture: also programs are just bytes in memory
- Only difference between data and program: what you do with it

char

- Most basic data type: `char`
- From the C11 standard:

  "An object declared as type char is large enough to store any member of the basic execution character set."
- More useful definition: a `char` is a byte, i.e., the smallest addressable unit of memory
- In all relevant scenarios: a `char` is an 8-bit integer
- Traditionally a `char` is used to represent ASCII characters, yields two common ways to initialize a `char`:

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

- Which of those values are equal?
- It's `a` and `c`, because `'2'` has ASCII value 50.

Another quick question...

- What does the following code do?:

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

- Answer: it depends (and it really does!)
- C standard does not define whether `char` is signed or unsigned
- Make explicit by using `signed char` or `unsigned char`

Other integral types

- C11 provides 4 more integral types (each signed and unsigned):
  - `short`: at least 2 bytes
  - `int`: typically 4 (but sometimes 2) bytes
  - `long`: typically 4 or 8 bytes
  - `long long`: at least 8 bytes (in practice: exactly 8 bytes)
- GNU extension: `__int128` for architectures that support it
- Common misconception: `long` is as long as a machine word
- Think about how this would work on an 8-bit microcontroller...
- Find size of any type in bytes using `sizeof`, e.g.:

```
int a;
printf("%zd", sizeof(a));
printf("%zd", sizeof(long));
```

- Integral constants can be written in
  - Decimal, e.g., 255
  - Hexadecimal, using `0x`, e.g., `0xff`
  - Octal, using `0`, e.g., `0377`

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 lecture: 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 */
...
a += 6755399441055744;
a -= 6755399441055744;
```

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

Printing values

Have already seen various examples of **format strings**, let's summarize:

```
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 */
```

There's quite a few more, but these get you fairly far.

stdint.h

- Often we need to know how large an integer is
- Example: crypto primitives are optimized to work on, e.g., 32-bit words
- Solution: Fixed-size integer types defined in `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
- Problem: how do we print them in a portable way?
- `printf("%llu\n", a);` for `uint64_t a` may produce warnings
- Solution: `printf("%" PRIu64 "\n", a)`
- For signed values, e.g., `PRId64`
- Printing in hexadecimal: `PRIx64`

Implicit type conversion

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

```
float pi, r, circ;
a = 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:

```
unsigned int a,b;
...
unsigned long long r = a*b;
```

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

```
unsigned long long r = (unsigned long long)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 “unspecified behavior” – it’s C after all)

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: $\{-2^{k-1}, \dots, 2^{k-1} - 1\}$
- Example: signed (8-bit) byte: $\{-128, \dots, 127\}$
- 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
```

- A quick poll: 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:
  - PowerPC
  - UltraSPARC
- ARM can switch endianess (is "bi-endian")
- The problem with little-endian intuition is just that we write left-to-right (but use Arabic numbers)