Systems 3

C Basics

Marcel Waldvogel

(Handout)

Department of Computer and Information Science
University of Konstanz

Winter 2019/2020

These slides are based on previous lectures held by Alexander Holupirek, Roman Byshko, and especially Stefan Klinger.

Chapter Goals

- What are the reasons, goals and organization of learning C in this course?
- What are the C compilation steps?
- What did Java borrow from C? How do they differ?
- How does character I/O work in C?
- How are functions defined and called? How are parameters passed?
- How are arrays laid out in memory? What does the compiler (not) know about arrays?

C popularity

- Requirements that make C mandatory:
  - embedded systems (close to hardware, scarce resources)
  - extreme performance (better usage of resources)
  - the world is built on C and C++ (with C++ being a superset of C)

    — Herb Sutter. C++ and Beyond.[1]
  - C is simple & powerful

    — Damien Katz (CouchDB). The Unreasonable Effectiveness of C.[2]
- Programming Languages Rankings
  - 1st/2nd place in TIOBE[3] (1989—2019)
  - 8th/9th place in RedMonk[4], with C++ ranking 5th—7th (2012—2019)

[1] https://www.youtube.com/watch?v=xcwxGzbTyms

[2] http://damienkatz.net/2013/01/the_unreasonable_effectiveness_of_c.html

[3] https://www.tiobe.com/tiobe-index/

[4] https://redmonk.com/kfitzpatrick/2019/07/31/june-2019-redmonk-programming-language-rankings-monkchat/

What is this course about?

System Programming

- With **system** we mean *operating system*.
- With **programming** we mean *using the interface* an operating system (OS) provides.
- With OS we mean UNIX-like OSs, *i.e.*, Linux.

Operating System

- Layer of software on top of bare hardware
- Shields programmers from the complexity of the hardware
- Presents an interface (of a virtual machine) that is easier to understand and program

Systems vs. Kernel programming

- Black Box Model is suitable for systems programming.
- However, knowledge about the system's internals is beneficial to use the system properly and to not work against it.
- Providing the system services is (mostly) kernel programming.

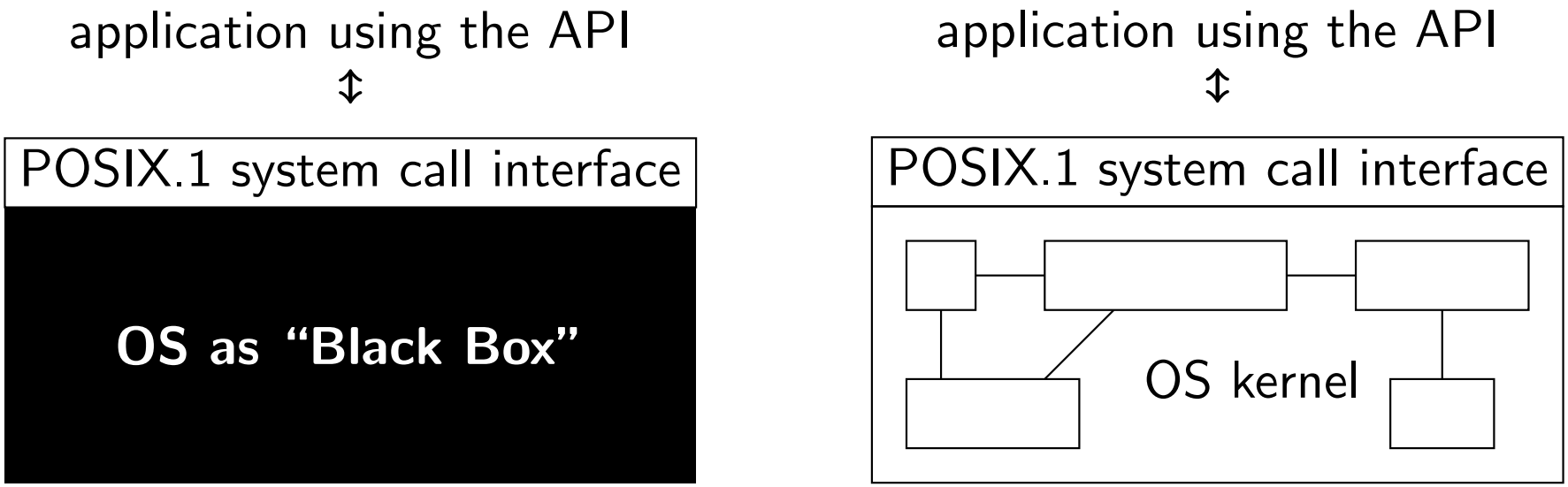

Figure: Black Box vs. White Box View of a UNIX System

Gentle introduction to C

C standardization

- ISO/IEC 9899:1990 Programming Language C, (C89 or C90)
- ISO/IEC 9899:1999 Programming Language C, (C99)
- ISO/IEC 9899:2011 Programming Language C, (C11)

Note We will focus on C99, *i.e.*, use `-std=c99` as compiler flag.

First C Program

Print the sentence: “Hello world!”

```
1 #include <stdio.h>
2
3 int main()
4 {
5    printf("Hello world!\n");
6    return 0;
7 }
```

Compiler

- Before executing a program, we have to translate it to machine code. The most popular compiler is gcc.
- We want to get all compiler errors and warnings:
  - Compile (we will) your code with

```
$ gcc -std=c99 -g -Wall -Wextra -Wpedantic -Wbad-function-cast \
> -Wconversion -Wwrite-strings -Wstrict-prototypes source.c
```

  - This will provide you helpful information from the compiler
  - You will gain **no points at all** for a programming exercise if the compiler stops with an **error**.
  - We will subtract **3 points** for every **compiler warning**.
- The tutors will show you on Thursday how to use the compiler.

Compilation on a UNIX-like OS

```
$ gcc hello.c
$ ls
a.out   hello.c
$ ./a.out
Hello world!
```

engine	filename	description
editor	`hello.c`	source code
preprocessor	`hello.i`	source w/ preproc. directives expanded
compiler	`hello.s`	assembler code
assembler	`hello.o`	object code ready to be linked
linker	`a.out`	executable

(Use `-save-temps` to preserve these files)

Basic instructions

There are many instructions which you already know from Java.

```
1 if ()
2 else
3 switch ()
4
5 while ()
6 do  while ();
7 for (;;)
8
9 i++; ++i; i += 1; ...
```

C vs. Java

C	Java
∼1970, procedural, low(er)-level	1995, object-oriented, high-level
compiled to machine code	compiled to byte code
suitable for systems programming	—
explicit `free()`	garbage collection
explicit pointers (+arithmetic)	implicit pointers in object variables
—	native threading
type casting	type checking
preprocessor	method overloading
default public	default private
global variables	—
`goto` statement	—
`struct`, `union`, bitfields	object
varargs	—

Basic data types

`char` a single byte. By definition, this is the unit of measurement for memory size.

`int` an integer, typically reflecting the natural size of integers on the host machine

`float` single-precision floating point

`double` double-precision floating point

`short` and `long` are *qualifiers* that can be applied to integers:

```
short int i;
long int f;
unsigned long d;
```

The qualifiers `signed` and `unsigned` can be applied to `char` and any integer.

printf revisited

```
#include <stdio.h>
int printf(const char *format, ...);
```

printf(3) is a general-purpose output formatting function.[5]

- 1st argument is the string of characters to be printed.
  - Each **%** indicates **where** one of the other arguments
  - and **in what form** it is to be printed.
- Each % in the 1st arg is paired with the 2nd, 3rd arg etc.

```
17 printf("%d\t%d\n", fahr, celsius);
```

- %d, for instance, specifies an integer argument, so fahr and celsius are printed with a tab (\t) between them.

[5] Not part of the C language, but defined in ANSI X3.159-1989 ("ANSI C")

Printing with `printf`

specifier	print as ...
`%d`	decimal integer
`%6d`	decimal, at least 6 characters wide
`%f`	floating point
`%6f`	floating point, at least 6 characters wide
`%.2f`	floating point, 2 characters after decimal point
`%6.2f`	floating point, at least 6 wide and 2 after decimal point

- Further `printf(3)` recognizes `%o` for octal, `%x` for hexadecimal, `%c` for character, `%s` for string, `%p` for address (pointer), ...
- ISO C: 7.19.6 : Formatted input/output functions

Symbolic constants

- Bad practice to bury "magic numbers" in a program
- Convey little information, hard to change in a systematic way
- A `#define` line defines a *symbolic name*

```
1  /* print fahrenheit-celsius table for fahrenheit = 0, 20, ..., 300 */
2
3  #include <stdio.h>
4
5  #define LOWER 0    /* lower limit of table */
6  #define UPPER 300  /* upper limit */
7  #define STEP  20   /* step size */
8
9  int main(void)
10 {
11   for (int fahr = LOWER; fahr <= UPPER; fahr += STEP)
12     printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
13
14   return 0;
15 }
```

Character input and output

- Standard library provides e.g. `getchar(3)` and `putchar(3)`.

```
#include <stdio.h>

int getchar(void);
int putchar(int c);
```

- `putchar(3)` prints a character to *stdout* each time it is called.
- `getchar(3)` reads the next input byte from *stdin* stream

Why does `getchar` *return* an `int` instead of `char`?

- Handle errors (returning distinctive value `EOF`) (end of file; a symbolic name, defined in `<stdio.h>`), which cannot be confused with data.
- The return type must hold `EOF` in addition to any possible `char`.

Why does `putchar` *accept* an `int` instead of `char`?

- Backward compatibility (smallest parameter used to be `int`).

File Copying

Given `getchar` and `putchar` we can write a surprising amount of useful code without knowing anything more about input and output.

Algo Copying input to output one character at a time

read a character
while character is not end-of-file indicator **do**
  output the character just read
  read a character
end while

File Copying, v1

```c
#include <stdio.h>

/* copy input to output, v1 */
int main(void)
{
  int c = getchar();

  while (c != EOF) {
    putchar(c);
    c = getchar();
  }
  return 0;
}
```

File Copying, v2

- An assignment, such as `c = getchar()` is an expression and has a value (value of the left hand side after the assignment)
- An assignment can appear as part of a larger expression

```
1  #include <stdio.h>
2
3  /* copy input to output, v2 */
4  int main(void)
5  {
6    int c;
7
8    while ((c = getchar()) != EOF)
9      putchar(c);
10
11   return 0;
12 }
```

Functions

power(m,n)

- So far only `printf(3)`, `getchar(3)`, and `putchar(3)`
- Implement `power(m,n)` to raise an integer *m* to the power[6] of *n*.

A function definition has the form:

```
1 type name( type parameter [, ...] )      /* or: name(void) */
2 {
3 declarations
4 statements
5 }
```

[6] Only handles positive powers of small integers, in real life take `pow(3)`

Function Terminology

- A **function definition** gives signature and implementation:

```
4  int power(int base, int n)
5  {
6    int i, p;
7
8    p = 1;
9    for (i = 0; i < n; ++i)
10     p = p * base;
11   return p;
12 }
```

  - A **parameter** is a variable named in the argument list, *e.g.*, `base`, `n`.
  - An **argument** is a value used in a call of the function.

- A **function declaration** omits the implementation:

```
int power(int base, int n);  /* no body! */
```

  - A function must be declared *before* it can be used!
  - A definition also declares a function.
  - We will not need to write declarations for some time...

Call by value, call by reference

In C, all function arguments are passed **by value**

- The called function is given the values of its arguments in **temporary variables** (lifetime of function's execution) rather than the originals.
- The callee **cannot directly alter** a variable in the calling function.

Call **by reference** is possible

- by passing the **address** of a variable (*aka.* a pointer).
- The callee can access the variable *indirectly* by **dereferencing** the address.
- The pointer itself is passed by value.
- We will discuss pointers in more detail at a later point.

One Dimensional Arrays

Syntax: `memberType arrayName[ numberOfMembers ];`

- Most simple:

```
int a[2];   /* at this point, the contents are undefined! */
a[0] = 23;  /* store 23 in 1st cell. */
a[1] = 42;
```

- Shortcut:

```
int a[2] = {23, 42};   /* initialize right away */
```

- Even shorter:

```
int a[] = {23, 42};    /* Compiler figures out size of array. */
```

- If not all items are given, the rest is initialised to 0.

```
int a[8] = {23, 42};   /* is the same as */
int a[] = {23, 42, 0, 0, 0, 0, 0, 0};
```

- Use `for` loop to initialize bigger arrays, or `memset`(3) (*cf.* later).

Multidimensional arrays

- Most simple:

```
int a[2][3];        /* at this point, the contents are undefined */
a[1][2] = 52;       /* assign to 3rd cell in 2nd array */
```

- Classic:

```
int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

- Shortcut:

```
int a[][3] = {{1, 2, 3}, {4, 5, 6}};
```

You may omit *only* the most significant (first, *i.e.*, outer) dimension!

- Stored in memory linearly, *i.e.*:

1	2	3	4	5	6

- Use `for` loop to initialize bigger arrays, or memset(3) (*cf.* later)
- If not all items are given, the rest is initialised to 0.

```
int a[3][4] = { {1,2}, {3} };   /* is the same as */
int a[][4] = { {1, 2, 0, 0}, {3, 0, 0, 0}, {0, 0, 0, 0} };
```

Fixed- and variable-length arrays

- C90 allowes only **constant**[7] **expressions** as array dimensions.

```
#define SIZE 1024
int a[42 * SIZE];
```

- In C99, **variable-length arrays** (VLAs) have been introduced.
  - They **cannot be initialised** in their declaration.
  - **Caution:** VLAs are rather tricky, and have a bunch of interesting consequences. You will not need them for your exercises.
  - You **cannot change** the size of a VLA once it is declared (*i.e.*, they are not *dynamic*).

```
1  #include <stdio.h>
2
3  int func(int c)
4  {
5    /* This is a conditional expression: */
6    return c < 10 ? 10 : c;
7  }
8
9  int main(void)
10 {
11   /* Bounds only known at runtime! */
12   int a[func(getchar())];
13
14   a[2] = 3;
15   printf("%d\n", a[2]);
16   return 0;
17 }
```

[7] *i.e.*, can be computed at compile-time by the compiler

Character arrays

Definition A **string** is an array of characters terminated with a `'\0'` character (nul; numerical value is zero). Yes, that is *nul*, with only one ℓ

- So is `"hello\n"` is stored as

`h`	`e`	`l`	`l`	`o`	`\n`	`\0`

- A string containing n characters requires $n + 1$ memory!
- A string does not know its own length.

Note You may have an **array of characters**, with none of them being nul.

- Perfectly valid, but **not a string!**
- String manipulating functions probably **will fail** on that data!

Initialization of character arrays

- Character by character:

```
char str[3];
str[0] = 'o';
str[1] = 'k';
str[2] = '\0';
```

- Shorter:

```
char str[] = {'o', 'k', '\0'};
```

- Initialising from a string literal:

```
char str[20] = "ok";     /* str[2] and onwards are automatically assigned '\0' */
```

- Without giving the dimension:

```
char str[] = "ok";                                  /* The dimension will be... What?  */
```

Arrays of character arrays

- Initialised from string literals:

```
1 char arr[3][12]= { "University",
2   "of",
3   "Konstanz" };
```

- You are only allowed to omit the **outermost** dimension:

```
1 char arr[][12]= { "University",
2   "of",
3   "Konstanz" };
```

Question: How much memory does `arr` use?