Systems 3

C Basics

Marcel Waldvogel

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.

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
  - 2nd place in TIOBE[3] (October 2015)
  - 9th place in RedMonk[4], with C++ ranking 5th (June 2015)

[1] https://www.youtube.com/watch?v=xcwxGzbTyms
[2] http://damienkatz.net/2013/01/the_unreasonable_effectiveness_of_c.html
[3] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
[4] http://redmonk.com/sogrady/2015/07/01/language-rankings-6-15/

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.

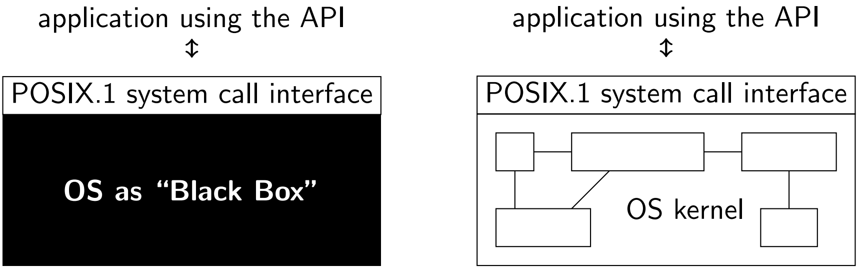

Figure: Black Box vs. White Box View of a UNIX System

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!”

```
#include <stdio.h>

int main()
{
   printf("Hello world!\n");
   return 0;
}
```

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
	`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.

```
if ()
else
switch ()

while ()
do  while ();
for (;;)

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.

```
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*

```c
/* print fahrenheit-celsius table for fahrenheit = 0, 20, ..., 300 */

#include <stdio.h>

#define LOWER 0    /* lower limit of table */
#define UPPER 300  /* upper limit */
#define STEP  20   /* step size */

int main(void)
{
  for (int fahr = LOWER; fahr <= UPPER; fahr += STEP)
    printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));

  return 0;
}
```

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

```c
#include <stdio.h>

/* copy input to output, v2 */
int main(void)
{
  int c;

  while ((c = getchar()) != EOF)
    putchar(c);

  return 0;
}
```

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:

```
type name( type parameter [, ...] )      /* or: name(void) */
{
declarations
statements
}
```

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

Function Terminology

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

```
int power(int base, int n)
{
  int i, p;

  p = 1;
  for (i = 0; i < n; ++i)
    p = p * base;
  return p;
}
```

  - 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*).

```
#include <stdio.h>

int func(int c)
{
  /* This is a conditional expression: */
  return c < 10 ? 10 : c;
}

int main(void)
{
  /* Bounds only known at runtime! */
  int a[func(getchar())];

  a[2] = 3;
  printf("%d\n", a[2]);
  return 0;
}
```

[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:

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

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

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

Question: How much memory does `arr` use?

More on Types

Structures, unions, enumerations
Defining types
Unscrambling C declarations

Type Conversions

Type Conversions

- Type ranking
  - `_Bool` → `char` → `short` → `int` → `long` → `long long` →
  - `float` → `double` → `long double`
- Typing constants
  - `L` (`long`), `LL` (`long long`)
  - `U` (`unsigned`), `UL` (`unsigned long`), `ULL` (`unsigned long long`)
  - `F` (`float`), `L` (`long double`)
- Automatic promotion to (`unsigned`) `int`
  - The signedness of the higher-ranked type takes precedence
  - For equal ranks, `unsigned` takes precedence

Structures

Declaring structures

- A structure allows a group of variables to be accessed via one name.
- To this end, a structure introduces a **new type**.

The definition has four parts:

```
struct tag {
    /* list of member declarations */
   type name...;
   type name...;
} variable...;
```

- the keyword `struct`
- an optional *structure tag*
- brace-enclosed list of declarations for the **members**
- list of variables of the new structure type (optional)

Declaration examples:

```
struct point {
   double x, y;
};
/* now "struct tag" serves as type name */
struct point p, q;
```

or equivalent

```
struct {
   double x, y;
} p, q;  /* directly name variables */
/* But you cannot reuse this struct! */
```

Using structures

- A list of constant member values in the right order initialises a structure. Or use individual members by name, in any order.

```
struct point p1 = { 320, 200 };
```

```
struct point p2 = { .x = 320 };
```

- Structures can be assigned as a unit, or be returned from a function.

```
struct point p = q;        /* copy all members */
struct point mkpoint();    /* declares function returning a point structure */
```

- Members can be accessed using name.member

```
struct point center;
printf("%f, %f\n", center.x, center.y);
```

- There is a shortcut for handling pointers to structs: ptr->name

```
struct point origin, *pp;
pp = &origin;                      /* so you can get the address of a structure */
printf("origin is (%f,%f)\n", (*pp).x, (*pp).y);
printf("origin is (%f,%f)\n", pp->x, pp->y);   /* this is equivalent */
```

- Structures can contain other structures

```
struct rect {
   struct point ul;
   struct point lr;
} square;

square.ul.x = 0; square.ul.y = 1;
square.lr.x = 2; square.lr.y = 0;
```

- Structures can be self-referential **via pointers**.

```
struct tnode {            /* the tree node: */
   int value;             /* node label */
   struct tnode *left;    /* left child */
   struct tnode *right;   /* right child */
};
```

- The **size** of a struct may be *larger* than the sum of its members!

```
struct demo {
   int i;
   char c;
};
```

```
/* prints 8 on my machine */
printf("%zu\n", sizeof(struct demo));
```

- Structures can be array elements

```
struct point {
   int x;
   int y;
} points[] = {
   { 0, 1 },
   { 2, 3 },
   { 3, 5 }
};
```

- Structures are passed to functions **by value**!

```
struct point add(struct point p1, struct point p2)
{
   p1.x += p2.x;
   p1.y += p2.y;

   return p1;
}
```

  - The **whole struct** is copied!
  - This also works for the `return` value!

Unions

- A *union* is a **variable** that may hold (at different times) objects of **different types** and sizes.
- Unions provide a way to manipulate different kinds of data in a **single area of storage**.

The syntax is similar to structures:

```
union tag {
    /* list of member declarations */
   type name...;
   type name...;
} variable...;
```

- the keyword `union`
- an optional *union tag*
- brace-enclosed list of declarations for the **members**
- list of variables of the new union type (optional)

- Union variables will be large enough to hold the **largest** of the member types. (the specific size is implementation-dependent)
- It is the programmer's responsibility to keep track of which member currently holds a value. **Only one** can be used at any time.

```c
union demo {
   int i;
   double d;
   char c;
};

union demo u;

printf("size: %zu\n", sizeof(u));

u.i = 23;   /* now u.d and u.c contain garbage! */
printf("u.i: %-16d   u.d: %-16e   u.c: '%c'\n", u.i, u.d, u.c);

u.d = 4.2;   /* now u.i and u.c contain garbage! */
printf("u.i: %-16d   u.d: %-16e   u.c: '%c'\n", u.i, u.d, u.c);

u.c = 'X';   /* now u.i and u.d contain garbage! */
printf("u.i: %-16d   u.d: %-16e   u.c: '%c'\n", u.i, u.d, u.c);
```

```
$ ./a.out
size: 8
u.i: 23                u.d: 6.952931e-310      u.c: ''
u.i: -858993459        u.d: 4.200000e+00       u.c: '□'
u.i: -858993576        u.d: 4.200000e+00       u.c: 'X'
```

Use case 1: Saving space

- Usually occur as a part of a larger struct that also has implicit or explicit information about the data.
- Used to save space.

Example Zoological information on certain species. First attempt:

```
struct creature {
   char has_backbone;
   char has_fur;
   short num_of_legs_in_excess_of_4;
};
```

However...

- All creatures are either vertebrate or invertebrate.
- Only vertebrates have fur and only invertebrates have more than four legs.
- Nothing has more than four legs and fur.

That is why...

```
union secondary_characteristics {
   char has_fur;
   short num_of_legs_in_excess_of_4;
};
struct creature {
   char has_backbone;  /* indicates valid union field! */
   union secondary_characteristics form;
};

struct creature naked_mole_rat = {
   .has_backbone = 'y',
   .form.has_fur = 'n'      /* Note the .form prefix */
};
```

Use case 2: Data interpretation

```
union bits32_tag {
    int whole;  /* one 32-bit value */
    char byte[4];  /* four 8-bit bytes */
} value;
```

- Take the whole with `value.whole`
- Take 3rd byte with `value.byte[2]`

Notes

- You need to check your compiler's documentation to make proper use of this!
- Generally, structs are about one hundred times more common than unions.

Enumerations

- Enumerations provide a convenient way to associate **constant integer** values with **names**.
- An alternative to `#define` with the advantage that the values can be generated automatically.
- A compiler can warn about missing `case`s in `switch` statements over an enumeration.
- A **debugger** may also be able to print values of enumeration variables in symbolic form.

Definition syntax:

```
enum tag {
  name,
  name = val,
  ...
} variable...;
```

- the keyword `enum`
- an optional *enumeration tag*
- brace-enclosed list of *members*, sep. by **comma**, with optional assignment
- optional list of variables of the new type

- Declaring an enumeration is similar to `enum` and `struct`.

```c
enum answer { no, yes }; /* definition */
enum answer x; /* declaration */
```

```c
/* shorthand */
enum { no, yes } x;
```

- An enumeration is a list of **constant integer values**.

```c
enum answer { no, yes };

enum answer x;
int i;

x = no;
x = 42;      /* x is just an int */
i = yes;
no = 23;     /* invalid — not an lvalue! */
```

- If not assigned explicitly, the names are assigned consecutive integer constants, starting from 0.
- Enumeration continues from an explicit assignment.

```c
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
JUL, AUG, SEP, OCT, NOV, DEC };
/* FEB is 2, MAR is 3 ... */
```

Enumerations

- A **function** itself *is not* a variable.
- But it is possible to define **pointers** to functions.
- Can be **assigned**, placed in **arrays**, **passed** to/**returned** from functions.

Syntax step-by-step examples of declarations:

- `int fun(char c, double x);` nothing new!
  - The expression `fun('Q', 3.14)` is of type `int`.
- `int *fun(char c, double x);` nothing new!
  - The expression `*fun('Q', 3.14)` is of type `int`.
  - Dereferencing `fun('Q', 3.14)` is of type `int`.
- `int (*fun)(char c, double x);`
  - The expression `(*fun)('Q', 3.14);` is of type `int`.
  - Dereferencing `fun`, and applying the result to `'Q',3.14`, is of type `int`.

⇒ We have just dereferenced a function!

Example

```c
size_t strlen(const char *s);    /* available with #include <string.h> */

size_t (*fp)(const char *);
fp = &strlen;
printf("result = %zu\n", (*fp)("hello world"));
```

- This can be abbreviated:

```c
printf("result = %zu\n", fp("hello world"));
```

- Nicer with `typedef`

```c
typedef size_t (*func)(const char *);
func fp = &strlen;
```

Note Function pointers are heavily used in the real world! *E.g.*,

- pass a comparing function to a queue datastructure;
- installing signal handlers (*cf.* OS lecture, and later in this course); and
- abstractions (syscall interface, subclasses, VFS, ...).

Question Can you read `int (*(*f)(void))[2]` ?