Systems Programming and Computer Architecture

Janis Hutz
https://janishutz.com

January 6, 2026

TITLE PAGE COMING SOON

"*If you are using CMake to solve the exercises... First off, sorry that you like CMake*"

- Timothy Roscoe, 2025

HS2025, ETHZ

Summary of the Lectures and Lecture Slides

Quotes

"*An LLM is a lossy index over human statements*"

- Professor Buhmann, Date unknown

"*If you are using CMake to solve the exercises... First off, sorry that you like CMake*"

"*You can't have a refrigerator behave like multiple refrigerators*"

"*Why is C++ called C++ and not ++C? It's like you don't get any value and then it's incremented, which is true*"

- Timothy Roscoe, 2025

Contents

1 The C Programming Language **4**
1.1 Basics . . . 4
1.1.1 Control Flow . . . 5
1.1.2 Declarations . . . 6
1.1.3 Operators . . . 7
1.1.4 Arrays . . . 8
1.1.5 Strings . . . 8
1.1.6 Integers in C . . . 9
1.1.7 Pointers . . . 10
1.2 The C preprocessor . . . 12

2 x86 Assembly **13**

3 Hardware **14**

1 The C Programming Language

I can clearly C why you'd want to use C. Already sorry in advance for all the bad C jokes that are going to be part of this section

C is a compiled, low-level programming language, lacking many features modern high-level programming languages offer, like Object Oriented programming, true Functional Programming (like Haskell implements), Garbage Collection, complex abstract datatypes and vectors, just to name a few. (It is possible to replicate these using Preprocessor macros, more on this later).

On the other hand, it offers low-level hardware access, the ability to directly integrate assembly code into the `.c` files, as well as bit level data manipulation and extensive memory management options, again just to name a few.

This of course leads to C performing excellently and there are many programming languages whose compiler doesn't directly produce machine code or assembly, but instead optimized C code that is then compiled into machine code using a C compiler. This has a number of benefits, most notably that C compilers can produce very efficient assembly, as lots of effort is put into the C compilers by the hardware manufacturers.

There are many great C tutorials out there, a simple one (as for many other languages too) can be found here

1.1 Basics

C uses a very similar syntax as many other programming languages, like Java, JavaScript and many more... to be precise, it is *them* that use the C syntax, not the other way around. So:

File: 00_intro.c

```c
// This is a line comment
/* this is a block comment */
#include "01_func.h" // Relative import

int i = 0; // This allocates an integer on the stack

int main( int argc, char *argv[] ) {
    // This is the function body of a function (here the main function)
    // which serves as the entrypoint to the program in C and has arguments
    printf( "Argc: %d\n", argc );     // Number of arguments passed, always >= 1
                                      // (first argument is the executable name)
    for ( int i = 0; i < argc; i++ ) // For loop just like any other sane programming language
        printf( "Arg %d: %s\n", i, argv[ i ] ); // Outputs the i-th argument from CLI

    get_user_input_int( "Select a number" ); // Function calls as in any other language
    return 0;                                // Return a POSIX exit code
}
```

In C we are referring to the implementation of a function as a ***(function) definition*** (correspondingly, *variable definition*, if the variable is initialized) and to the definition of the function signature (or variables, without initializing them) as the ***(function) declaration*** (or, correspondingly, *variable declaration*).

C code is usuallt split into the source files, ending in `.c` (where the local functions and variables are declared, as well as all function definitions) and the header files, ending in `.h`, usually sharing the filename of the source file, where the external declarations are defined. By convention, no definition of functions are in the `.h` files, and neither variables, but there is nothing preventing you from putting them there.

File: 01_func.h

```c
#include <stdio.h> // Import from system path
                   // (like library imports in other languages)

int get_user_input_int( char prompt[] );
```

1.1.1 Control Flow

Many of the control-flow structures of C can be found in the below code snippet. A note of caution when using goto: It is almost never a good idea (can lead to unexpected behaviour, is hard to maintain, etc). Where it however is very handy is for error recovery (and cleanup functions) and early termination of multiple loops (jumping out of a loop). So, for example, if you have to run multiple functions to set something up and one of them fails, you can jump to a label and have all cleanup code execute that you have specified there. And because the labels are (as in Assembly) simply skipped over during execution, you can make very nice cleanup code. We can also use `continue` and `break` statements similarly to Java, they do not however accept labels. (Reminder: `continue` skips the loop body and goes to the next iteration)

File: 01_func.c

```c
#include "01_func.h"
#include <stdio.h>

int get_user_input_int( char prompt[] ) {
    int input_data;
    printf( "%s", prompt );           // Always wrap strings like this for printf
    scanf( "%d", &input_data );       // Get user input from CLI
    int input_data_copy = input_data; // Value copied

    // If statements just like any other language
    if ( input_data )
        printf( "Not 0" );
    else
        printf( "Input is zero" );

    // Switch statements just like in any other language
    switch ( input_data ) {
        case 5:
            printf( "You win!" );
            break; // Doesn't fall through
        case 6:
            printf( "You were close" ); // Falls through
        default:
            printf( "No win" ); // Case for any not covered input
    }

    while ( input_data > 1 ) {
        input_data -= 1;
        printf( "Hello World\n" );
    }

    // Inversed while loop (executes at least once)
    do {
        input_data -= 1;
        printf( "Bye World\n" );
        if ( input_data_copy == 0 )
            goto this_is_a_label;
    } while ( input_data_copy > 1 );

this_is_a_label:
    printf( "Jumped to label" );
    return 0;
}
```

1.1.2 Declarations

We have already seen a few examples for how C handles declarations. In concept they are similar (and scoping works the same) to most other C-like programming languages, including Java.

File: 02_declarations.c

```c
int my_int;              // Allocates memory on the stack.
                         // Variable is global (read / writable by entire program)
static int my_local_int; // only available locally (in this file)
extern const char *var;  // Defined in some other file
const int MY_CONST = 10; // constant (immutable), convention: SCREAM_CASE

enum { ONE, TWO } num; // Enum. ONE will get value 0, TWO has value 1

enum { O = 2, T = 1 } n; // Enum with values specified

// Structs are like classes, but contain no logic
struct MyStruct {
        int el1;
        int el2;
};

int fun( int j ) {
    static int i = 0;             // Persists across calls of fun
    short my_var = 1;             // Block scoped (deallocated when going out of scope)
    int my_var_dbl = (int) my_var; // Explicit casting (works between almost all types)
    return i;
}

int main( int argc, char *argv[] ) {
    if ( ( my_local_int = fun( 10 ) ) ) {
        // Every c statement is also an expression, i.e. you can do the above!
    }
    struct MyStruct test;            // Allocate memory on stack for struct
    struct MyStruct *test_p = &test; // Pointer to memory where test resides
    test.el1 = 1;                    // Direct element access
    test_p->el2 = 2;                 // Via pointer
    return 0;
}
```

A peculiarity of C is that the bit-count is not defined by the language, but rather the hardware it is compiled for.

C data type	typical 32-bit	ia32	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	4	8	8
long double	8	10/12	16

Table 1.1: Comparison of byte-sizes for each datatype on different architectures

By default, integers in C are signed, to declare an unsigned integer, use unsigned int. Since it is hard and annoying to remember the number of bytes that are in each data type, C99 has introduced the extended integer types, which can be imported from stdint.h and are of form int<bit count>_t and uint<bit count>_t, where we substitute the <bit count> with the number of bits (have to correspond to a valid type of course).

Another notable difference of C compared to other languages is that C doesn't natively have a boolean type, by convention a short is used to represent it, where any non-zero value means true and 0 means false. Since boolean types are quite handy, the ! syntax for negation turns any non-zero value of any integer type into zero and vice-versa. C99 has added support for a bool type via stdbool.h, which however is still an integer.

Notably, C doesn't have a very rigid type system and lower bit-count types are implicitly cast to higher bit-count data types, i.e. if you add a short and an int, the short is cast to short (bits 16-31 are set to 0) and the two are added. Explicit casting between almost all types is also supported. Some will force a change of bit representation, but most won't (notably, when casting to and from float-like types, minus to void)

Another important feature is that every C statement is also an expression, see above code block for example.

The void type has ***no*** value and is used for untyped pointers and declaring functions with no return value

It is also possible to define a custom type using typedef <type it represents> <name of the new type>

1.1.3 Operators

The list of operators in C is similar to the one of Java, etc. In Table 1.2, you can see an overview of the operators, sorted by precedence in descending order. You may notice that the & and * operators appear twice. The higher precedence occurrence is the address operator and dereference, respectively, and the lower precedence is bitwise and and multiplication, respectively.

Very low precedence belongs to boolean operators && and ||, as well as the ternary operator and assignment operators

<table>
<tr><th>Operator</th><th>Associativity</th></tr>
<tr><td>() [] -> .</td><td>Left-to-right</td></tr>
<tr><td>! ~ ++ -- + - * & (type) sizeof</td><td>Right-to-left</td></tr>
<tr><td>* / %</td><td>Left-to-right</td></tr>
<tr><td>+ -</td><td>Left-to-right</td></tr>
<tr><td><< >></td><td>Left-to-right</td></tr>
<tr><td>< <= >= ></td><td>Left-to-right</td></tr>
<tr><td>== !=</td><td>Left-to-right</td></tr>
<tr><td>& (logical and)</td><td>Left-to-right</td></tr>
<tr><td>^ (logical xor)</td><td>Left-to-right</td></tr>
<tr><td>| (logical or)</td><td>Left-to-right</td></tr>
<tr><td>&& (boolean and)</td><td>Left-to-right</td></tr>
<tr><td>|| (boolean or)</td><td>Left-to-right</td></tr>
<tr><td>? : (ternary)</td><td>Right-to-left</td></tr>
<tr><td>= += -= *= /= %= &= ^=—= <<= >>=</td><td>Right-to-left</td></tr>
<tr><td>,</td><td>Left-to-right</td></tr>
</table>

Table 1.2: C operators ordered in descending order by precedence

Associativity

- Left-to-right: $A + B + C \mapsto (A + B) + C$
- Right-to-left: A += B += C $\mapsto$ (A += B) += C

As it should be, boolean and, as well as boolean or support early termination.

The ternary operator works as in other programming languages result = expr ? res_true : res_false;

As previously touched on, every statement is also an expression, i.e. the following works

```
printf("%s", x = foo(y)); // prints output of foo(y) and x has that value
```

Pre-increment (++i, new value returned) and post-increment (i++, old value returned) are also supported by C.

C has an assert statement, but do not use it for error handling. The basic syntax is assert(expr);

1.1.4 Arrays

C compiler does not do any array bound checks! Thus, always check array bounds. Unlike some other programming languages, arrays are ***not*** dynamic length.

The below snippet includes already some pointer arithmetic tricks. The variable `data` is a pointer to the first element of the array.

File: `03_arrays.c`

```
#include <stdint.h>
#include <stdio.h>

int main( int argc, char *argv[] ) {
    int data[ 10 ];                     // Initialize array of 10 integers
    data[ 5 ] = 5;                      // element 5 is now 5
    *data = 10;                         // element 0 is now 5
    printf( "%d\n", data[ 0 ] );       // print element 0 (prints 10)
    printf( "%d\n", *data );           // equivalent as above
    printf( "%d\n", data[ 5 ] );       // print element 5 (prints 5)
    printf( "%d\n", *( data + 5 ) ); // equivalent as above
    int multidim[ 5 ][ 5 ];             // 2-dimensional array
                                        // We can iterate over it using two for-loops
    int init_array[ 2 ][ 2 ] = {
        {1, 2},
        {3, 4}
    };                          // We can initialize an array like this
    int empty_arr[ 4 ] = {}; // Initialized to 0
    return 0;
}
```

1.1.5 Strings

C doesn't have a `string` data type, but rather, strings are represented (when using `ASCII`) as `char` arrays, with length of the array $n + 1$ (where n is the number of characters of the string). The extra element is the termination character, called the `null character`, denoted `\0`. To determine the actual length of the string (as it may be padded), we can use `strnlen(str, maxlen)` from `string.h`

File: `04_strings.c`

```
#include <stdio.h>
#include <string.h>

int main( int argc, char *argv[] ) {
    char hello[ 6 ] = "hello";                           // Using double quotes
    char world[ 6 ] = { 'w', 'o', 'r', 'l', 'd', '\0' }; // As array

    char src[ 12 ], dest[ 12 ];
    strncpy( src, "ETHZ", 12 );      // Copy strings (extra elements will be set to \0)
    strncpy( dest, src, 12 );        // Copy strings (last arg is first n chars to copy)
    if ( strncmp( src, dest, 12 ) ) // Compare two strings. Returns 1 if src > dest
        printf( "Hello World" );
    strncat( dest, " is in ZH", 12 ); // Concatenate strings
    return 0;
}
```

1.1.6 Integers in C

As a reminder, integers are encoded as follows in big endian notation, with x_i being the i-th bit and w being the number of bits used to represent the number:

- ***Unsigned***: $\sum_{i=0}^{w-1} x_i \cdot 2^i$
- ***Signed***: $-x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-1} x_i \cdot 2^i$ (two's complement notation, with x_{w-1} being the sign-bit)

The minimum number representable is 0 and -2^{w-1}, respectively, whereas the maximum number representable is $2^w - 1$ and $2^{w-1} - 1$. `limits.h` defines constants for the minimum and maximum values of different types, e.g. `ULONG_MAX` or `LONG_MAX` and `LONG_MIN`

We can use the shift operators to multiply and divide by two. Shift operations are usually *much* cheaper than multiplication and division. Left shift (`u << k` in `C`) always fills with zeros and throws away the extra bits on the left (equivalent to multiplication by 2^k), whereas right shift (`u >> k` in `C`) is implementation-defined, either arithmetic (fill with most significant bit, division by 2^k. This however rounds incorrectly, see below) or logical shift (fill with zeros, unsigned division by 2^k).

Signed division using arithmetic right shifts has the issue of incorrect rounding when number is < 0. Instead, we represent $s/2^k = s + (2^k - 1)$ `>>` k for $s < 0$ and $s/2^k = s >> k$ for $s > 0$

In expressions, signed values are implicitly cast to unsigned

This can lead to all sorts of nasty exploits (e.g. provide -1 as the argument to `memcpy` and watch it burn, this was an actual exploit in FreeBSD)

Addition & Subtraction

A nice property of the two's complement notation is that addition and subtraction works exactly the same as in normal notation, due to over- and underflow. This also obviously means that it implements modular arithmetic, i.e.

$$\texttt{Add}_w(u, v) = u + v \text{ mod } 2^w \quad \text{and} \quad \texttt{Sub}_w(u, v) = u - v \text{ mod } 2^w$$

Multiplication & Division

Unsigned multiplication with addition forms a commutative ring. Again, it is doing modular arithmetic and

$$\texttt{UMult}_w(u, v) = u \cdot v \text{ mod } 2^w$$

1.1.7 Pointers

On loading of a program, the OS creates the virtual address space for the process, inspects the executable and loads the data to the right places in the address space, before other preparations like final linking and relocation are done.

Stack-based languages (supporting recursion) allocate stack in frames that contain local variables, return information and temporary space. When a procedure is entered, a stack frame is allocated and executes any necessary setup code (like moving the stack pointer, see later). When a procedure returns, the stack frame is deallocated and any necessary cleanup code is executed, before execution of the previous frame continues.

In `C` a pointer is a variable whose value is the memory address of another variable

Of note is that if you simply declare a pointer using `type * p;` you will get different memory addresses every time. The (Linux)-Kernel randomizes the address space to prevent some common exploits.

File: 05_pointers.c

```c
#include "01_func.h" // See a few pages up for declarations
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

void a_function( int ( *func )( char * ), char prompt[] ) {
    ( *func )( prompt ); // Call function with arguments
}

int main( int argc, char *argv[] ) {
    int x = 0;
    int *p = &x;          // Get x's memory address
    printf( "%p\n", p );  // Print the address of x
    printf( "%d\n", *p ); // Dereference pointer (get contents of memory location)
    *p = 10;              // Dereference assign
    int **dbl_p = &p;     // Double pointer (pointer to pointer to value)
    int *null_p = NULL;   // Create NULL pointer
    *null_p = 1;          // Segmentation fault due to null pointer dereference

    // pointer arithmetic
    int arr[ 3 ] = { 2, 3, 4 };
    char c_arr[ 3 ] = { 'A', 'B', 'C' };
    int *arr_p = &arr[ 1 ];
    char *c_arr_p = &c_arr[ 1 ];
    c_arr_p += 1; // Now points to c_arr[2]
    arr_p -= 1;   // Now points to arr[0]

    char *arr_p_c = (char *) arr_p; // Cast to char pointer (points to first byte of arr[0])
    printf( "%d", *( arr_p - 5 ) ); // No boundary checks (can access any memory)
    assert( arr == &( arr[ 0 ] ) ); // Evaluates to true
    int new_arr[ 3 ] = arr;         // Compile time error (cannot use other array as initializer)
    int *new_arr_p = &arr[ 0 ];     // This works

    a_function( &get_user_input_int, c_arr );

    return EXIT_SUCCESS;
}
```

Some pointer arithmetic has already appeared in section 1.1.4, but same kind of content with better explanation can be found here

Note that when doing pointer arithmetic, adding 1 will move the pointer by `sizeof(type)` bits.

You may use pointer arithmetic on whatever pointer you'd like (as long as it's not a null pointer). This means, you *can* make an array wherever in memory you'd like. The issue is just that you are likely to overwrite something, and that something might be something critical (like a stack pointer), thus you will get ***undefined*** behaviour! (This is by the way a common concept in `C`, if something isn't easy to make more flexible (example for `malloc`, if you pass a pointer to memory that is not the start of the `malloc`'d section, you get undefined behaviour), in the docs mention that one gets undefined behaviour if you do not do as it says so... RTFM!)

As already seen in the section arrays (section 1.1.4), we can use pointer arithmetic for accessing array elements. The array name is treated as a pointer to the first element of the array, except when:

- it is operand of `sizeof` (return value is $n \cdot$ `sizeof(type)` with n the number of elements)
- its address is taken (then `&a == a`)
- it is a string literal initializer. If we modify a pointer `char *b = "String";` to string literal in code, the `"String"` is stored in the code segment and if we modify the pointer, we get undefined behaviour

Fun fact : `A[i]` is always rewritten `*(A + i)` by compiler.

Another important aspect is passing by value or by reference. You can pass every data type by reference, you can not however pass an array by value.

Another interesting concept that `C` has to offer is body-less loops:

```c
int x = 0;
while ( x++ < 10 ); // This is (of course) not a useful snippet, but shows the concept
```

`C` also has an option to pass functions as arguments to functions, called function pointers. A function is passed using the typical address syntax with the `&` symbol is annotated as argument using `type (* name)(type arg1, ...)` and is called using `(*func)(arg1, ...)`.

1.2 The C preprocessor

To have gcc stop compiliation after running through cpp, the C preprocessor, use gcc -E <file name>.

Imports in C are handled by the preprocessor, that for each #include <file1.h>, the preprocessor simply copies the contents of the file recursively into one file.

Depending on if we use #include <file1.h> or #include "file1.h" the preprocessor will search for the file either in the system headers or in the project directory. Be wary of including files twice, as the preprocessor will recursively include all files (i.e. it will include files from the files we included)

The C preprocessor gives us what are called preprocessor macros, which have the format #define NAME SUBSTITUTION.

File: 00_macros.c

```c
#include <stdio.h>
#include <stdlib.h>
#define FOO       BAZ
#define BAR( x ) ( x + 3 )
#define SKIP_SPACES( p )                                    \
    do {                                                    \
        while ( p > 0 ) { p--; }                            \
    } while ( 0 )
#define COMMAND( c ) { #c, c##_command } // Produces { "<val(c)>", "<val(c)>_command" }

#ifdef FOO // If macro is defined, ifndef for if not defined
    #define COURSE "SPCA"
#else
    #define COURSE "Systems Programming and Computer Architecture"
#endif

#if 1
    #define OUT HELLO // if statement
#endif

int main( int argc, char *argv[] ) {
    int i = 10;
    SKIP_SPACES( i );

    printf( "%s", COURSE );

    return EXIT_SUCCESS;
}
```

To avoid issues with semicolons at the end of preprocessor macros that wrap statements that cannot end in semicolons, we can use a concept called semicolon swallowing. For that, we wrap the statements in a do ... while(0) loop, which is removed by the compiler on compile, also taking with it the semicolon.

There are also a number of predefined macros:

- __FILE__: Filename of processed file
- __LINE__: Line number of this usage of macro
- __DATE__: Date of processing
- __TIME__: Time of processing
- __STDC__: Set if ANSI Standard C compiler is used
- __STDC_VERSION__: The version of Standard C being compiled
- ...many more

In headers, we typically use #ifndef __FILENAME_H_ followed by a #define __FILENAME_H_ or the like to check if the header was already included before

2 x86 Assembly

3 Hardware

Remember: Rust and the like have an `unsafe` block... `C`'s equivalent to this is

```c
int main( int argc, char *argv[] ) {
    // Unsafe code goes here
}
```

i.e. ***YOU are the one that makes `C` code safe!***