

## Modern C

Jens Gustedt

INRIA, FRANCE

ICUBE, STRASBOURG, FRANCE

*E-mail address:* jens.gustedt@inria.fr

*URL:* [http://icube-icps.unistra.fr/index.php/Jens\\_Gustedt](http://icube-icps.unistra.fr/index.php/Jens_Gustedt)

*This is a preliminary version of this book compiled on February 11, 2017, but should already contain all material of a final version.*

*You might find a more up to date version at*

*<http://icube-icps.unistra.fr/index.php/File:ModernC.pdf> (inline)*

*[http://icube-icps.unistra.fr/img\\_auth.php/d/db/ModernC.pdf](http://icube-icps.unistra.fr/img_auth.php/d/db/ModernC.pdf) (download)*

*You may well share this by pointing others to my home page or one of the links above.*

*Since this is not completely finished, please don't distribute the file itself, yet.*

*I may have found a publisher that is willing to publish this book under a Creative Commons license, so please stay put.*

*All rights reserved, Jens Gustedt, 2017*

*Special thanks go to the people that encouraged the writing of this book by providing me with constructive feedback, in particular Cédric Bastoul, Lucas Nussbaum, Vincent Loechner, Kliment Yanev, Szabolcs Nagy, Marcin Kowalczyk, Ali Asad Lotia, Richard Palme ... and probably some that I missed to mention.*



PRELIMINARIES. The C programming language has been around for a long time — the canonical reference for it is the book written by its creators, Kernighan and Ritchie [1978]. Since then, C has been used in an incredible number of applications. Programs and systems written in C are all around us: in personal computers, phones, cameras, set-top boxes, refrigerators, cars, mainframes, satellites, basically in any modern device that has a programmable interface.

In contrast to the ubiquitous presence of C programs and systems, good knowledge of and about C is much more scarce. Even experienced C programmers often appear to be stuck in some degree of self-inflicted ignorance about the modern evolution of the C language. A likely reason for this is that C is seen as an "easy to learn" language, allowing a programmer with little experience to quickly write or copy snippets of code that at least appear to do what it's supposed to. In a way, C fails to motivate its users to climb to higher levels of knowledge.

This book is intended to change that general attitude. It is organized in chapters called "Levels" that summarize levels of familiarity with the C language and programming in general. Some features of the language are presented in parts on earlier levels, and elaborated in later ones. Most notably, pointers are introduced at Level 1 but only explained in detail at Level 2. This leads to many forward references for impatient readers to follow.

As the title of this book suggests, today's C is not the same language as the one originally designed by its creators Kernighan and Ritchie (usually referred to as K&R C). In particular, it has undergone an important standardization and extension process now driven by ISO, the International Standards Organization. This led to three major publications of C standards in the years 1989, 1999 and 2011, commonly referred to as C89, C99 and C11. The C standards committee puts a lot of effort into guaranteeing backwards compatibility such that code written for earlier versions of the language, say C89, should compile to a semantically equivalent executable with a compiler that implements a newer version. Unfortunately, this backwards compatibility has had the unwanted side effect of not motivating projects that could benefit greatly from the new features to update their code base.

In this book we will mainly refer to C11, as defined in JTC1/SC22/WG14 [2011], but at the time of this writing many compilers don't implement this standard completely. If you want to compile the examples of this book, you will need at least a compiler that implements most of C99. For the changes that C11 adds to C99, using an emulation layer such as my macro package P99 might suffice. The package is available at <http://p99.gforge.inria.fr/>.

Programming has become a very important cultural and economic activity and C remains an important element in the programming world. As in all human activities, progress in C is driven by many factors, corporate or individual interest, politics, beauty, logic, luck, ignorance, selfishness, ego, sectarianism, ... (add your primary motive here). Thus the development of C has not been and cannot be ideal. It has flaws and artifacts that can only be understood with their historic and societal context.

An important part of the context in which C developed was the early appearance of its sister language C++. One common misconception is that C++ evolved from C by adding its particular features. Whereas this is historically correct (C++ evolved from a very early C) it is not particularly relevant today. In fact, C and C++ separated from a common ancestor more than 30 years ago, and have evolved separately ever since. But this evolution of the two languages has not taken place in isolation, they have exchanged and adopted each other's concepts over the years. Some new features, such as the recent addition of atomics and threads have been designed in a close collaboration between the C and C++ standard committees.

Nevertheless, many differences remain and generally all that is said in this book is about C and not C++. Many code examples that are given will not even compile with a C++ compiler.

**Rule A** *C and C++ are different, don't mix them and don't mix them up.*

ORGANIZATION. This book is organized in *levels*. The starting level, *encounter*, will introduce you to the very basics of programming with C. By the end of it, even if you don't have much experience in programming, you should be able to understand the structure of simple programs and start writing your own.

The *acquaintance* level details most principal concepts and features such as control structures, data types, operators and functions. It should give you a deeper understanding of the things that are going on when you run your programs. This knowledge should be sufficient for an introductory course in algorithms and other work at that level, with the notable caveat that pointers aren't fully introduced yet at this level.

The *cognition* level goes to the heart of the C language. It fully explains pointers, familiarizes you with C's memory model, and allows you to understand most of C's library interface. Completing this level should enable you to write C code professionally, it therefore begins with an essential discussion about the writing and organization of C programs. I personally would expect anybody who graduated from an engineering school with a major related to computer science or programming in C to master this level. Don't be satisfied with less.

The *experience* level then goes into detail in specific topics, such as performance, reentrancy, atomicity, threads and type generic programming. These are probably best discovered as you go, that is when you encounter them in the real world. Nevertheless, as a whole they are necessary to round off the picture and to provide you with full expertise in C. Anybody with some years of professional programming in C or who heads a software project that uses C as its main programming language should master this level.

Last but not least comes *ambition*. It discusses my personal ideas for a future development of C. C as it is today has some rough edges and particularities that only have historical justification. I propose possible paths to improve on the lack of general constants, to simplify the memory model, and more generally to improve the modularity of the language. This level is clearly much more specialized than the others, most C programmers can probably live without it, but the curious ones among you could perhaps take up some of the ideas.



## Contents

Level 0. Encounter	1
1. Getting started	1
1.1. Imperative programming	1
1.2. Compiling and running	3
2. The principal structure of a program	6
2.1. Grammar	6
2.2. Declarations	7
2.3. Definitions	9
2.4. Statements	10
Level 1. Acquaintance	13
Warning to experienced C programmers	13
3. Everything is about control	14
3.1. Conditional execution	15
3.2. Iterations	17
3.3. Multiple selection	20
4. Expressing computations	22
4.1. Arithmetic	22
4.2. Operators that modify objects	25
4.3. Boolean context	26
4.4. The ternary or conditional operator	28
4.5. Evaluation order	28
5. Basic values and data	30
5.1. Basic types	32
5.2. Specifying values	34
5.3. Initializers	37
5.4. Named constants	38
5.5. Binary representations	42
6. Aggregate data types	50
6.1. Arrays	50
6.2. Pointers as opaque types	55
6.3. Structures	56
6.4. New names for types: <b>typedef</b>	60
7. Functions	62
7.1. Simple functions	62
7.2. <b>main</b> is special	63
7.3. Recursion	65
8. C Library functions	70
8.1. Mathematics	74
8.2. Input, output and file manipulation	74
8.3. String processing and conversion	83
8.4. Time	87
8.5. Runtime environment settings	89

8.6. Program termination and assertions	91
Level 2. Cognition	95
9. Style	95
9.1. Formatting	95
9.2. Naming	96
10. Organization and documentation	99
10.1. Interface documentation	100
10.2. Implementation	103
10.3. Macros	103
10.4. Pure functions	105
11. Pointers	108
11.1. Address-of and object-of operators	109
11.2. Pointer arithmetic	109
11.3. Pointers and <b>struct</b> s	112
11.4. Opaque structures	114
11.5. Array and pointer access are the same	115
11.6. Array and pointer parameters are the same	115
11.7. Null pointers	116
11.8. Function pointers	117
12. The C memory model	121
12.1. A uniform memory model	121
12.2. Unions	122
12.3. Memory and state	123
12.4. Pointers to unspecific objects	125
12.5. Implicit and explicit conversions	125
12.6. Effective Type	127
12.7. Alignment	127
13. Allocation, initialization and destruction	129
13.1. <b>malloc</b> and friends	130
13.2. Storage duration, lifetime and visibility	138
13.3. Initialization	142
13.4. Digression: a machine model	144
14. More involved use of the C library	148
14.1. Text processing	148
14.2. Formatted input	153
14.3. Extended character sets	155
14.4. Binary files	162
15. Error checking and cleanup	163
Level 3. Experience	167
16. Performance	167
Safety first	167
Optimizers are good enough	168
Help the compiler	168
16.1. Inline functions	169
16.2. Avoid aliasing: <b>restrict</b> qualifiers	172
16.3. Measurement and inspection	173
17. Functionlike macros	179
17.1. how does it work	180
17.2. Argument checking	182
17.3. Accessing the calling context	185
17.4. Variable length argument lists	186

17.5. Type generic programming	192
18. Variations in Control Flow	199
18.1. Sequencing	200
18.2. Short jumps	202
18.3. Functions	203
18.4. Long jumps	205
18.5. Signal handlers	209
19. Threads	217
19.1. Simple inter-thread control	220
19.2. Thread local data	220
19.3. Critical data and critical sections	221
19.4. Communicating through condition variables	223
19.5. More sophisticated thread management	226
20. Atomic access and memory consistency	228
20.1. The “happend before” relation	229
20.2. Synchronizing C library calls	230
20.3. Sequential consistency	231
20.4. Other consistency models	232
 Level 4. Ambition	 235
21. The <b>register</b> overhaul	236
Overview	237
21.1. Introduce <code>register</code> storage class in file scope	237
21.2. Typed constants with <code>register</code> storage class and <code>const</code> qualification	238
21.3. Extend ICE to register constants	241
21.4. Functions	243
21.5. Unify designators	244
22. Improve type generic expression programming	248
22.1. Storage class for compound literals	249
22.2. Inferred types for variables and functions	250
22.3. Anonymous functions	252
23. Improve the C library	255
23.1. Make the presence of all C library headers mandatory	255
23.2. Add requirements for sequence points	260
23.3. Provide type generic interfaces for string functions	262
24. Modules	265
24.1. C needs a specific approach	265
24.2. All is about naming	265
24.3. Modular C features	266
25. Simplify the object and value models	267
25.1. Remove objects of temporary lifetime	267
25.2. Introduce comparison operator for object types	268
25.3. Make <b>memcpy</b> and <b>memcmp</b> consistent	268
25.4. Enforce representation consistency for <b>_Atomic</b> objects	268
25.5. Make string literals <b>char const []</b>	268
25.6. Default initialize padding to 0	268
25.7. Make <code>restrict</code> qualification part of the function interface	268
25.8. References	269
26. Contexts	269
26.1. Introduce evaluation contexts in the standard	269
26.2. Convert object pointers to <b>void*</b> in unspecific context	270
26.3. Introduce <code>nullptr</code> as a generic null pointer constant and deprecate <b>NULL</b>	270

List of Rules	271
Listings	282
Bibliography	285
Index	289





LEVEL 0

## Encounter

This first level of the book may be your first encounter with the programming language C. It provides you with a rough knowledge about C programs, about their purpose, their structure and how to use them. It is not meant to give you a complete overview, it can't and it doesn't even try. On the contrary, it is supposed to give you a general idea of what this is all about and open up questions, promote ideas and concepts. These then will be explained in detail on the higher levels.

### 1. Getting started

In this section I will try to introduce you to one simple program that has been chosen because it contains many of the constructs of the C language. If you already have experience in programming you may find parts of it feel like needless repetition. If you lack such experience, you might feel overwhelmed by the stream of new terms and concepts.

In either case, be patient. For those of you with programming experience, it's very possible that there are subtle details you're not aware of, or assumptions you have made about the language that are not valid, even if you have programmed C before. For the ones approaching programming for the first time, be assured that after approximately ten pages from now your understanding will have increased a lot, and you should have a much clearer idea of what programming might represent.

An important bit of wisdom for programming in general, and for this book in particular, is summarized in the following citation from the *Hitchhiker's guide to the Galaxy*:

**Rule B** *Don't panic.*

It's not worth it. There are many cross references, links, side information present in the text. There is an Index on page 288. Follow those if you have a question. Or just take a break.

**1.1. Imperative programming.** To get started and see what we are talking about consider our first program in Listing 1:

You probably see that this is a sort of language, containing some weird words like “**main**”, “**include**”, “**for**”, etc. laid out and colored in a peculiar way and mixed with a lot of weird characters, numbers, and text “*Doing some work*” that looks like an ordinary English phrase. It is designed to provide a link between us, the human programmers, and a machine, the computer, to tell it what to do — give it “orders”.

**Rule 0.1.1.1** *C is an imperative programming language.*

In this book, we will not only encounter the C programming language, but also some vocabulary from an English dialect, *C jargon*, the language that helps us *to talk about C*. It will not be possible to immediately explain each term the first time it occurs. But I will explain each one, in time, and all of them are indexed such that you can easily cheat and *jump*<sup>C</sup> to more explanatory text, at your own risk.

As you can probably guess from this first example, such a C program has different components that form some intermixed layers. Let's try to understand it from the inside out.

LISTING 1. A first example of a C program

```

1  /* This may look like nonsense, but really is -*- mode: C -*- */
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  /* The main thing that this program does. */
6  int main(void) {
7      // Declarations
8      double A[5] = {
9          [0] = 9.0,
10         [1] = 2.9,
11         [4] = 3.E+25,
12         [3] = .00007,
13     };
14
15     // Doing some work
16     for (size_t i = 0; i < 5; ++i) {
17         printf("element_%zu_is_%g, \ tits_square_is_%g\n",
18             i,
19             A[i],
20             A[i]*A[i]);
21     }
22
23     return EXIT_SUCCESS;
24 }

```

1.1.1. *Giving orders.* The visible result of running this program is to output 5 lines of text on the command terminal of your computer. On my computer using this program looks something like

	Terminal
0	> ./getting-started
1	element 0 is 9,                its square is 81
2	element 1 is 2.9,            its square is 8.41
3	element 2 is 0,              its square is 0
4	element 3 is 7e-05,         its square is 4.9e-09
5	element 4 is 3e+25,         its square is 9e+50

We can easily identify parts of the text that this program outputs (*prints*<sup>C</sup> in the C jargon) inside our program, namely the blue part of Line 17. The real action (*statement*<sup>C</sup> in C) happens between that line and Line 20. The statement is a *call*<sup>C</sup> to a *function*<sup>C</sup> named **printf**.

	getting-started.c
17	<b>printf</b> ("element_%zu_is_%g, \ tits_square_is_%g\n",
18	i,
19	A[i],
20	A[i]*A[i]);

Here, the **printf** *function*<sup>C</sup> receives four *arguments*<sup>C</sup>, enclosed in a pair of *parenthesis*<sup>C</sup>, “( ... )”:

- The funny-looking text (the blue part) is a so-called *string literal*<sup>C</sup> that serves as a *format*<sup>C</sup> for the output. Within the text are three markers (*format specifiers*<sup>C</sup>), that mark the positions in the output where numbers are to be inserted. These markers start with a "%" character. This format also contains some special *escape characters*<sup>C</sup> that start with a backslash, namely "\t" and "\n".
- After a comma character we find the word "i". The thing "i" stands for will be printed in place of the first format specifier, "%zu".
- Another comma separates the next argument "A[i]". The thing this stands for will be printed in place of the second format specifier, the first "%g".
- Last, again separated by comma, appears "A[i]\*A[i]", corresponding to the last "%g".

We will later explain what all of these arguments mean. Let's just remember that we identified the main purpose of that program, namely to print some lines on the terminal, and that it "orders" function `printf` to fulfill that purpose. The rest is some *sugar*<sup>C</sup> to specify which numbers will be printed and how many of them.

**1.2. Compiling and running.** As it is shown above, the program text that we have listed can not be understood by your computer.

There is a special program, called a *compiler*, that translates the C text into something that your machine can understand, the so-called *binary code*<sup>C</sup> or *executable*<sup>C</sup>. What that translated program looks like and how this translation is done is much too complicated to explain at this stage.<sup>1</sup> However, for the moment we don't need to understand more deeply, as we have that tool that does all the work for us.

**Rule 0.1.2.1** *C is a compiled programming language.*

The name of the compiler and its command line arguments depend a lot on the *platform*<sup>C</sup> on which you will be running your program. There is a simple reason for this: the target binary code is *platform dependent*<sup>C</sup>, that is its form and details depend on the computer on which you want to run it; a PC has different needs than a phone, your fridge doesn't speak the same language as your set-top box. In fact, that's one of the reasons for C to exist.

**Rule 0.1.2.2** *A C program is portable between different platforms.*

It is the job of the compiler to ensure that our little program above, once translated for the appropriate platform, will run correctly on your PC, your phone, your set-top box and maybe even your fridge.

That said, there is a good chance that a program named `c99` might be present on your PC and that this is in fact a C compiler. You could try to compile the example program using the following command:

Terminal

```
0 > c99 -Wall -o getting-started getting-started.c -lm
```

The compiler should do its job without complaining, and output an executable file called `getting-started` in your current directory.<sup>[Exs 2]</sup> In the above line

- `c99` is the compiler program.
- `-Wall` tells it to warn us about anything that it finds unusual.

<sup>1</sup>In fact, the *translation* itself is done in several steps that goes from textual replacement, over proper compilation to linking. Nevertheless, the tool that bundles all this is traditionally called *compiler* and not *translator*, which would be more accurate.

[Exs 2] Try the compilation command in your terminal.

- `-o getting-started` tells it to store the *compiler output*<sup>C</sup> in a file named `getting-started`.
- `getting-started.c` names the *source file*<sup>C</sup>, namely the file that contains the C code that we have written. Note that the `.c` extension at the end of the file name refers to the C programming language.
- `-lm` tells it to add some standard mathematical functions if necessary, we will need those later on.

Now we can *execute*<sup>C</sup> our newly created *executable*<sup>C</sup>. Type in:

```

0  > ./getting-started

```

and you should see exactly the same output as I have given you above. That's what portable means, wherever you run that program its *behavior*<sup>C</sup> should be the same.

If you are not lucky and the compilation command above didn't work, you'd have to look up the name of your *compiler*<sup>C</sup> in your system documentation. You might even have to install a compiler if one is not available. The names of compilers vary. Here are some common alternatives that might do the trick:

```

0  > clang -Wall -lm -o getting-started getting-started.c
1  > gcc -std=c99 -Wall -lm -o getting-started getting-started.c
2  > icc -std=c99 -Wall -lm -o getting-started getting-started.c

```

Some of these, even if they are present on your computer, might not compile the program without complaining.<sup>[Exs 3]</sup>

With the program in Listing 1 we presented an ideal world — a program that works and produces the same result on all platforms. Unfortunately, when programming yourself very often you will have a program that only works partially and that maybe produces wrong or unreliable results. Therefore, let us look at the program in Listing 2. It looks quite similar to the previous one.

If you run your compiler on that one, it should give you some *diagnostic*<sup>C</sup>, something similar to this

```

0  > c99 -Wall -o getting-started-badly getting-started-badly.c
1  getting-started-badly.c:4:6: warning: return type of 'main' is not 'int' [-Wmain]
2  getting-started-badly.c: In function 'main':
3  getting-started-badly.c:16:6: warning: implicit declaration of function 'printf' [-Wimplicit-function-declaration]
4  getting-started-badly.c:16:6: warning: incompatible implicit declaration of built-in function 'printf' [-Wbuiltin-declaration-mismatch]
5  getting-started-badly.c:22:3: warning: 'return' with a value, in function returning void [-Wreturn-type]

```

Here we had a lot of long “warning” lines that are even too long to fit on a terminal screen. In the end the compiler produced an executable. Unfortunately, the output when we run the program is different. This is a sign that we have to be careful and pay attention to details.

`clang` is even more picky than `gcc` and gives us even longer diagnostic lines:

<sup>[Exs 3]</sup> Start writing a textual report about your tests with this book. Note down which command worked for you.

LISTING 2. An example of a C program with flaws

```

1  /* This may look like nonsense, but really is -*- mode: C -*- */
2
3  /* The main thing that this program does. */
4  void main() {
5      // Declarations
6      int i;
7      double A[5] = {
8          9.0,
9          2.9,
10         3.E+25,
11         .00007,
12     };
13
14     // Doing some work
15     for (i = 0; i < 5; ++i) {
16         printf("element %d is %g, \ tits square is %g\n",
17             i,
18             A[i],
19             A[i]*A[i]);
20     }
21
22     return 0;
23 }

```

Terminal

```

0  > clang -Wall -o getting-started-badly getting-started-badly.c
1  getting-started-badly.c:4:1: warning: return type of 'main' is not 'int' [-Wmain-return-type]
2  void main() {
3      ^
4  getting-started-badly.c:16:6: warning: implicitly declaring library function 'printf' with type
5      'int (const char *, ...)'
6      printf("element %d is %g, \ tits square is %g\n", /*@\label{printf start-badly}*/
7      ^
8  getting-started-badly.c:16:6: note: please include the header <stdio.h> or explicitly provide a d
9      'printf'
10 getting-started-badly.c:22:3: error: void function 'main' should not return a value [-Wreturn-type]
11     return 0;
12     ^      ~
13 2 warnings and 1 error generated.

```

This is a good thing! Its *diagnostic output*<sup>C</sup> is much more informative. In particular it gave us two hints: it expected a different return type for `main` and it expected us to have a line such as Line 3 of Listing 1 to specify where the `printf` function comes from. Notice how `clang`, unlike `gcc`, did not produce an executable. It considers the problem in Line 22 fatal. Consider this to be a feature.

In fact depending on your platform you may force your compiler to reject programs that produce such diagnostics. For `gcc` such a command line option would be `-Werror`.

**Rule 0.1.2.3** A C program should compile cleanly without warnings.

So we have seen two of the points in which Listings 1 and 2 differed, and these two modifications turned a good, standard conforming, portable program into a bad one. We also have seen that the compiler is there to help us. It nailed the problem down to the lines in the program that cause trouble, and with a bit of experience you will be able to understand what it is telling you.<sup>[Exs 4]</sup> <sup>[Exs 5]</sup>

## 2. The principal structure of a program

Compared to our little examples from above, real programs will be more complicated and contain additional constructs, but their structure will be very similar. Listing 1 already has most of the structural elements of a C program.

There are two categories of aspects to consider in a C program: syntactical aspects (how do we specify the program so the compiler understands it) and semantic aspects (what do we specify so that the program does what we want it to do). In the following subsections we will introduce the syntactical aspects (“grammar”) and three different semantic aspects, namely declarative parts (what things are), definitions of objects (where things are) and statements (what are things supposed to do).

**2.1. Grammar.** Looking at its overall structure, we can see that a C program is composed of different types of text elements that are assembled in a kind of grammar. These elements are:

**special words:** In Listing 1 we have used the following special words<sup>6</sup>: **#include**, **int**, **void**, **double**, **for**, and **return**. In our program text, here, they will usually be printed in bold face. These special words represent concepts and features that the C language imposes and that cannot be changed.

**punctuations<sup>C</sup>:** There are several punctuation concepts that C uses to structure the program text.

- There are five sorts of parenthesis: { ... }, ( ... ), [ ... ], /\* ... \*/ and < ... >. Parenthesis *group* certain parts of the program together and should always come in pairs. Fortunately, the < ... > parenthesis are rare in C, and only used as shown in our example, on the same logical line of text. The other four are not limited to a single line, their contents might span several lines, like they did when we used **printf** earlier.
- There are two different separators or terminators, comma and semicolon. When we used **printf** we saw that commas *separated* the four arguments to that function, in line 12 we saw that a comma also can follow the last element of a list of elements.

12 [3] = .00007, getting-started.c

One of the difficulties for newcomers in C is that the same punctuation characters are used to express different concepts. For example, { } and [ ] are each used for two different purposes in our program.

**Rule 0.2.1.1** *Punctuation characters can be used with several different meanings.*

**comments<sup>C</sup>:** The construct */\* ... \*/* that we saw as above tells the compiler that everything inside it is a *comment*, see e.g Line 5.

<sup>[Exs 4]</sup> Correct Listing 2 step by step. Start from the first diagnostic line, fix the code that is mentioned there, recompile and so on, until you have a flawless program.

<sup>[Exs 5]</sup> There is a third difference between the two programs that we didn’t mention, yet. Find it.

<sup>6</sup>In the C jargon these are *directives<sup>C</sup>*, *keywords<sup>C</sup>* and *reserved<sup>C</sup>* identifiers

getting-started.c

```
5  /* The main thing that this program does. */
```

Comments are ignored by the compiler. It is the perfect place to explain and document your code. Such “in-place” documentation can (and should) improve the readability and comprehensibility of your code a lot. Another form of comment is the so-called C++-style comment as in Line 15. These are marked by `/*`. C++-style comments extend from the `/*` to the end of the line.

**literals<sup>C</sup>**: Our program contains several items that refer to fixed values that are part of the program: 0, 1, 3, 4, 5, 9.0, 2.9, 3.E+25, .00007, and `"element_zu_is_g, \tits_square_is_g\n"`. These are called **literals<sup>C</sup>**.

**identifiers<sup>C</sup>**: These are “names” that we (or the C standard) give to certain entities in the program. Here we have: A, i, `main`, `printf`, `size_t`, and `EXIT_SUCCESS`. Identifiers can play different roles in a program. Amongst others they may refer to:

- **data objects<sup>C</sup>** (such as A and i), these are also referred to as **variables<sup>C</sup>**
- **type<sup>C</sup>** aliases, `size_t`, that specify the “sort” of a new object, here of i. Observe the trailing `_t` in the name. This naming convention is used by the C standard to remind you that the identifier refers to a type.
- functions (`main` and `printf`),
- constants (`EXIT_SUCCESS`).

**functions<sup>C</sup>**: Two of the identifiers refer to functions: `main` and `printf`. As we have already seen `printf` is *used* by the program to produce some output. The function `main` in turn is *defined<sup>C</sup>*, that is its *declaration<sup>C</sup>* `int main(void)` is followed by a *block<sup>C</sup>* enclosed in `{ ... }` that describes what that function is supposed to do. In our example this function *definition<sup>C</sup>* goes from Line 6 to 24. `main` has a special role in C programs as we will encounter them, it must always be present since it is the starting point of the program’s execution.

**operators<sup>C</sup>**: Of the numerous C operators our program only uses a few:

- `=` for *initialization<sup>C</sup>* and *assignment<sup>C</sup>*,
- `<` for comparison,
- `++` to increment a variable, that is to increase its value by 1
- `*` to perform the multiplication of two values.

**2.2. Declarations.** Declarations have to do with the **identifiers<sup>C</sup>** that we encountered above. As a general rule:

**Rule 0.2.2.1** *All identifiers of a program have to be declared.*

That is, before we use an identifier we have to give the compiler a **declaration<sup>C</sup>** that tells it what that identifier is supposed to be. This is where identifiers differ from **keywords<sup>C</sup>**; keywords are predefined by the language, and must not be declared or redefined.

Three of the identifiers we use are effectively declared in our program: `main`, A and i. Later on, we will see where the other identifiers (`printf`, `size_t`, and `EXIT_SUCCESS`) come from.

Above, we already mentioned the declaration of the `main` function. All three declarations, in isolation as “declarations only”, look like this:

```
1  int main(void);
2  double A[5];
3  size_t i;
```

These three follow a pattern. Each has an identifier (**main**, **A** or **i**) and a specification of certain properties that are associated with that identifier.

- **i** is of *type<sup>C</sup>* **size\_t**.
- **main** is additionally followed by parenthesis, ( ... ), and thus declares a function of type **int**.
- **A** is followed by brackets, [ ... ], and thus declares an *array<sup>C</sup>*. An array is an aggregate of several items of the same type, here it consists of 5 items of type **double**. These 5 items are ordered and can be referred to by numbers, called *indices<sup>C</sup>*, from 0 to 4.

Each of these declarations starts with a *type<sup>C</sup>*, here **int**, **double** and **size\_t**. We will see later what that represents. For the moment it is sufficient to know that this specifies that all three identifiers, when used in the context of a statement, will act as some sort of “numbers”.

For the other three identifiers, **printf**, **size\_t** and **EXIT\_SUCCESS**, we don’t see any declaration. In fact they are pre-declared identifiers, but as we saw when we tried to compile Listing 2, the information about these identifiers doesn’t come out of nowhere. We have to tell the compiler where it can obtain information about them. This is done right at the start of the program, in the Lines 2 and 3: **printf** is provided by **stdio.h**, whereas **size\_t** and **EXIT\_SUCCESS** come from **stdlib.h**. The real declarations of these identifiers are specified in .h files with these names somewhere on your computer. They could be something like:

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

```
1 int printf(char const format[static 1], ...);
2 typedef unsigned long size_t;
3 #define EXIT_SUCCESS 0
```

but this is not important for the moment. This information is normally hidden from you in these *include files<sup>C</sup>* or *header files<sup>C</sup>*. If you need to know the semantics of these, it’s usually a bad idea to look them up in the corresponding files, as they tend to be barely readable. Instead, search in the documentation that comes with your platform. For the brave, I always recommend a look into the current C standard, as that is where they all come from. For the less courageous the following commands may help:

```
Terminal
0 > apropos printf
1 > man printf
2 > man 3 printf
```

Declarations may be repeated, but only if they specify exactly the same thing.

**Rule 0.2.2.2** *Identifiers may have several consistent declarations.*

Another property of declarations is that they might only be valid (*visible<sup>C</sup>*) in some part of the program, not everywhere. A *scope<sup>C</sup>* is a part of the program where an identifier is valid.

**Rule 0.2.2.3** *Declarations are bound to the scope in which they appear.*

In Listing 1 we have declarations in different scopes.



- A is visible inside the definition of **main**, starting at its very declaration on Line 8 and ending at the closing `}` on Line 24 of the innermost `{ ... }` block that contains that declaration.
- `i` has a more restricted visibility. It is bound to the **for** construct in which it is declared. Its visibility reaches from that declaration in Line 16 to the end of the `{ ... }` block that is associated with the **for** in Line 21.
- **main** is not enclosed in any `{ ... }` block, so it is visible from its declaration onwards until the end of the file.

In a slight abuse of terminology, the first two types of scope are called *block scope*<sup>C</sup>. The third type, as used for **main** is called *file scope*<sup>C</sup>. Identifiers in file scope are often referred to as *globals*.

**2.3. Definitions.** Generally, declarations only specify the kind of object an identifier refers to, not what the concrete value of an identifier is, nor where the object it refers to can be found. This important role is filled by a *definition*<sup>C</sup>.

**Rule 0.2.3.1** *Declarations specify identifiers whereas definitions specify objects.*

We will later see that things are a little bit more complicated in real life, but for now we can make a simplification

**Rule 0.2.3.2** *An object is defined at the same time as it is initialized.*

Initializations augment the declarations and give an object its initial value. For instance:

```
1  size_t i = 0;
```

is a declaration of `i` that is also a definition with initial *value*<sup>C</sup> 0.

A is a bit more complex

```

.
8  double A[5] = {
9    [0] = 9.0,
10   [1] = 2.9,
11   [4] = 3.E+25,
12   [3] = .00007,
13   };

```

getting-started.c

this initializes the 5 items in A to the values 9.0, 2.9, 0.0, 0.00007 and 3.0E+25, in that order. The form of an initializer we see here is called *designated*<sup>C</sup>: a pair of brackets with an integer *designate* which item of the array is initialized with the corresponding value. E.g. `[4] = 3.E+25` sets the last item of the array A to the value 3.E+25. As a special rule, any position that is not listed in the initializer is set to 0. In our example the missing `[2]` is filled with 0.0.<sup>7</sup>

**Rule 0.2.3.3** *Missing elements in initializers default to 0.*

You might have noticed that array positions, *indices*<sup>C</sup>, above are not starting at 1 for the first element, but with 0. Think of an array position as the “distance” of the corresponding array element from the start of the array.

**Rule 0.2.3.4** *For an array with  $n$  the first element has index 0, the last has index  $n-1$ .*

<sup>7</sup>We will see later how these number literals with dots . and exponents E+25 work.



A **for** statement can be written in several ways other than what we just saw. Often people place the definition of the loop variable somewhere before the **for** or even reuse the same variable for several loops. Don't do that.

**Rule 0.2.4.2** *The loop variable should be defined in the initial part of a **for**.*

2.4.2. *Function return.* The last statement in **main** is a **return**. It tells the **main** function, to *return* to the statement that it was called from once it's done. Here, since **main** has **int** in its declaration, a **return** must send back a value of type **int** to the calling statement. In this case that value is **EXIT\_SUCCESS**.

Even though we can't see its definition, the **printf** function must contain a similar **return** statement. At the point where we call the function in Line 17, execution of the statements in **main** is temporarily suspended. Execution continues in the **printf** function until a **return** is encountered. After the return from **printf**, execution of the statements in **main** continues from where it stopped.

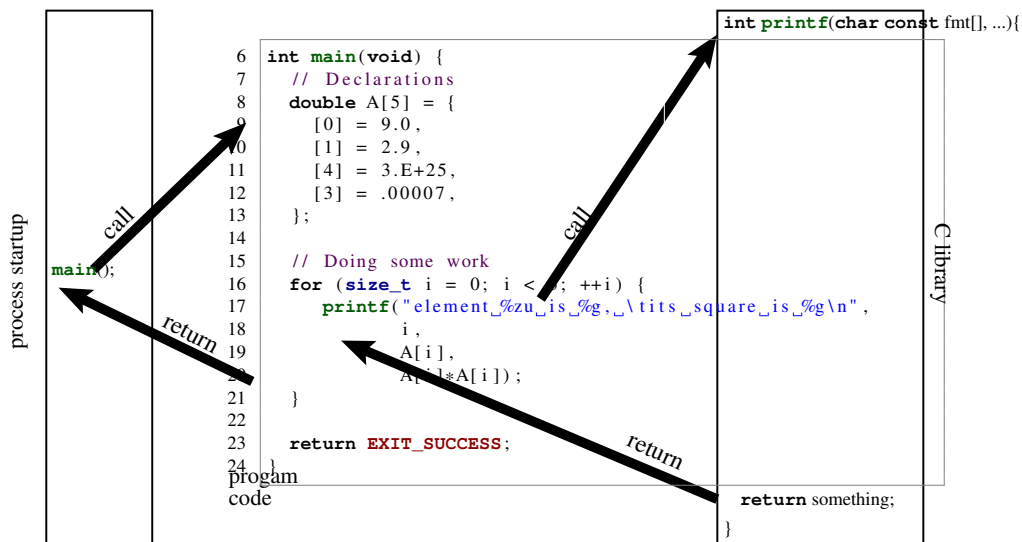


FIGURE 1. Execution of a small program

In Figure 1 we have a schematic view of the execution of our little program, its *control flow*. First, a process startup routine (on the left) that is provided by our platform calls the user-provided function **main** (middle). That in turn calls **printf**, a function that is part of the *C library*<sup>C</sup>, on the right. Once a **return** is encountered there, control returns back to **main**, and when we reach the **return** in **main**, it passes back to the startup routine. The latter transfer of control, from a programmer's point of view, is the end of the program's execution.



## LEVEL 1



# Acquaintance

This chapter is supposed to get you acquainted with the C programming language, that is to provide you with enough knowledge to write and use good C programs. “Good” here refers to a modern understanding of the language, avoiding most of the pitfalls of early dialects of C, offering you some constructs that were not present before, and that are portable across the vast majority of modern computer architectures, from your cell phone to a mainframe computer.

Having worked through this you should be able to write short code for everyday needs, not extremely sophisticated, but useful and portable. In many ways, C is a permissive language, a programmer is allowed to shoot themselves in the foot or other body parts if they choose to, and C will make no effort to stop them. Therefore, just for the moment, we will introduce some restrictions. We’ll try to avoid handing out guns in this chapter, and place the key to the gun safe out of your reach for the moment, marking its location with big and visible exclamation marks.

The most dangerous constructs in C are the so-called *casts*<sup>C</sup>, so we’ll skip them at this level. However, there are many other pitfalls that are less easy to avoid. We will approach some of them in a way that might look unfamiliar to you, in particular if you have learned your C basics in the last millennium or if you have been initiated to C on a platform that wasn’t upgraded to current ISO C for years.

- We will focus primarily on the *unsigned*<sup>C</sup> versions of integer types.
- We will introduce pointers in steps: first, in disguise as parameters to functions (6.1.4), then with their state (being valid or not, 6.2) and then, only when we really can’t delay it any further (11), using their entire potential.
- We will focus on the use of arrays whenever possible, instead.

**Warning to experienced C programmers.** If you already have some experience with C programming, this may need some getting used to. Here are some of the things that may provoke allergic reactions. If you happen to break out in spots when you read some code here, try to take a deep breath and let it go.

*We bind type modifiers and qualifiers to the left.* We want to separate identifiers visually from their type. So we will typically write things as

```
1 char* name;
```

where **char\*** is the type and **name** is the identifier. We also apply the left binding rule to qualifiers and write

```
1 char const* const path_name;
```

Here the first **const** qualifies the **char** to its left, the **\*** makes it to a pointer and the second **const** again qualifies what is to its left.

*We use array or function notation for pointer parameters to functions.* wherever these assume that the pointer can’t be null. Examples

```
1 size_t strlen(char const string[static 1]);
```

```

2  int main(int argc, char* argv[argc+1]);
3  int atexit(void function(void));

```

The first stresses the fact that `strlen` must receive a valid (non-null) pointer and will access at least one element of `string`. The second summarizes the fact that `main` receives an array of pointers to `char`: the program name, `argc-1` program arguments and one null pointer that terminates the array. The third emphasizes that semantically `atexit` receives a function as an argument. The fact that technically this function is passed on as a *function pointer* is usually of minor interest, and the commonly used pointer-to-function syntax is barely readable. Here are syntactically equivalent declarations for the three functions above as they would be written by many:

```

1  size_t strlen(const char *string);
2  int main(int argc, char **argv);
3  int atexit(void (*function) (void));

```

As you now hopefully see, this is less informative and more difficult to comprehend visually.

*We define variables as close to their first use as possible.* Lack of variable initialization, especially for pointers, is one of the major pitfalls for novice C programmers. This is why we should, whenever possible, combine the declaration of a variable with the first assignment to it: the tool that C gives us for this purpose is a definition - a declaration together with an initialization. This gives a name to a value, and introduces this name at the first place where it is used.

This is particularly convenient for `for`-loops. The iterator variable of one loop is semantically a different object from the one in another loop, so we declare the variable within the `for` to ensure it stays within the loop's scope.

*We use prefix notation for code blocks.* To be able to read a code block it is important to capture two things about it easily: its purpose and its extent. Therefore:

- All `{` are prefixed on the same line with the statement or declaration that introduces them.
- The code inside is indented by one level.
- The terminating `}` starts a new line on the same level as the statement that introduced the block.
- Block statements that have a continuation after the `}` continue on the same line.

Examples:

```

1  int main(int argc, char* argv[argc+1]) {
2      puts("Hello_world!");
3      if (argc > 1) {
4          while (true) {
5              puts("some_programs_never_stop");
6          }
7      } else {
8          do {
9              puts("but_this_one_does");
10         } while (false);
11     }
12     return EXIT_SUCCESS;
13 }

```

### 3. Everything is about control

In our introductory example we saw two different constructs that allowed us to control the flow of a program execution: functions and the `for`-iteration. Functions are a way to

transfer control unconditionally. The call transfers control unconditionally *to* the function and a **return**-statement unconditionally transfers it *back* to the caller. We will come back to functions in Section 7.

The **for** statement is different in that it has a controlling condition ( $i < 5$  in the example) that regulates if and when the dependent block or statement (`{ printf(...) }`) is executed. C has five conditional *control statements*: **if**, **for**, **do**, **while** and **switch**. We will look at these statements in this section.

There are several other kinds of conditional expressions we will look at later on: the *ternary operator*<sup>C</sup>, denoted by an expression in the form “cond ? A : B”, and the compile-time preprocessor conditionals (**#if**–**#else**) and type generic expressions (noted with the keyword **\_Generic**). We will visit these in Sections 4.4 and 17.5, respectively.

**3.1. Conditional execution.** The first construct that we will look at is specified by the keyword **if**. It looks like this:

```
1  if (i > 25) {
2      j = i - 25;
3  }
```

Here we compare  $i$  against the value 25. If it is larger than 25,  $j$  is set to the value  $i - 25$ . In that example  $i > 25$  is called the *controlling expression*<sup>C</sup>, and the part in `{ ... }` is called the *dependent block*<sup>C</sup>.

This form of an **if** statement is syntactically quite similar to the **for** statement that we already have encountered. It is a bit simpler, the part inside the parenthesis has only one part that determines whether the dependent statement or block is run.

There is a more general form of the **if** construct:

```
1  if (i > 25) {
2      j = i - 25;
3  } else {
4      j = i;
5  }
```

It has a second dependent statement or block that is executed if the controlling condition is not fulfilled. Syntactically, this is done by introducing another keyword **else** that separates the two statements or blocks.

The **if (...)**... **else ...** is a *selection statement*<sup>C</sup>. It selects one of the two possible *code paths*<sup>C</sup> according to the contents of `( ... )`. The general form is

```
1  if (condition) statement0-or-block0
2  else statement1-or-block1
```

The possibilities for the controlling expression “condition” are numerous. They can range from simple comparisons as in this example to very complex nested expressions. We will present all the primitives that can be used in Section 4.3.2.

The simplest of such “condition” specifications in an **if** statement can be seen in the following example, in a variation of the **for** loop from Listing 1.

```
1  for (size_t i = 0; i < 5; ++i) {
2      if (i) {
3          printf("element_%zu_is_%g, \ tits_square_is_%g\n",
4              i,
5              A[i],
6              A[i]*A[i]);
7      }
```

8    }

Here the condition that determines whether **printf** is executed or not is just **i**: a numerical value by itself can be interpreted as a condition. The text will only be printed when the value of **i** is not 0.<sup>[Exs 1]</sup>

There are two simple rules for the evaluation of a numerical “condition”:

**Rule 1.3.1.1** *The value 0 represents logical false.*

**Rule 1.3.1.2** *Any value different from 0 represents logical true.*

The operators **==** and **!=** allow us to test for equality and inequality, respectively. **a == b** is true if the value of **a** is equal to the value of **b** and false otherwise; **a != b** is false if **a** is equal to **b** and true otherwise. Knowing how numerical values are evaluated as conditions, we can avoid redundancy. For example, we can rewrite

```
1  if (i != 0) {
2      ...
3  }
```

as:

```
1  if (i) {
2      ...
3  }
```

**#include <stdbool.h>**

The type **bool**, specified in **stdbool.h**, is what we should be using if we want to store truth values. Its values are **false** and **true**. Technically, **false** is just another name for 0 and **true** for 1. It’s important to use **false** and **true** (and not the numbers) to emphasize that a value is to be interpreted as a condition. We will learn more about the **bool** type in Section 5.5.4.

Redundant comparisons quickly become unreadable and clutter your code. If you have a conditional that depends on a truth value, use that truth value directly as the condition. Again, we can avoid redundancy by rewriting something like:

```
1  bool b = ...;
2      ...
3  if ((b != false) == true) {
4      ...
5  }
```

as

```
1  bool b = ...;
2      ...
3  if (b) {
4      ...
5  }
```

Generally:

**Rule 1.3.1.3** *Don’t compare to 0, **false** or **true**.*

Using the truth value directly makes your code clearer, and illustrates one of the basic concepts of the C language:

<sup>[Exs 1]</sup> Add the **if (i)** condition to the program and compare the output to the previous.



TABLE 1. Scalar types used in this book

level	name	other	category	where	printf
0	<code>size_t</code>		unsigned	<stddef.h>	"%zu" "%zx"
0	<code>double</code>		floating	builtin	"%e" "%f" "%g" "%a"
0	<code>signed</code>	<code>int</code>	signed	builtin	"%d"
0	<code>unsigned</code>		unsigned	builtin	"%u" "%x"
0	<code>bool</code>	<code>_Bool</code>	unsigned	<stdbool.h>	"%d" as 0 or 1
1	<code>ptrdiff_t</code>		signed	<stddef.h>	"%td"
1	<code>char const*</code>		string	builtin	"%s"
1	<code>char</code>		character	builtin	"%c"
1	<code>void*</code>		pointer	builtin	"%p"
2	<code>unsigned char</code>		unsigned	builtin	"%hhu" "%02hhx"

**Rule 1.3.1.4** *All scalars have a truth value.*

Here *scalar*<sup>C</sup> types include all the numerical types such as `size_t`, `bool` or `int` that we already encountered, and *pointer*<sup>C</sup> types, that we will come back to in Section 6.2.

**3.2. Iterations.** Previously, we encountered the **for** statement that allows us to iterate over a domain; in our introductory example it declared a variable `i` that was set to the values 0, 1, 2, 3 and 4. The general form of this statement is

```
1  for (clause1; condition2; expression3) statement-or-block
```

This statement is actually quite generic. Usually “clause1” is an assignment expression or a variable definition. It serves to state an initial value for the iteration domain. “condition2” tests if the iteration should continue. Then, “expression3” updates the iteration variable that had been used in “clause1”. It is performed at the end of each iteration. Some advice

- In view of Rule 0.2.4.2 “clause1” should in most cases be a variable definition.
- Because **for** is relatively complex with its four different parts and not so easy to capture visually, “statement-or-block” should usually be a { ... } block.

Let’s see some more examples:

```
1  for (size_t i = 10; i; --i) {
2      something(i);
3  }
4  for (size_t i = 0, stop = upper_bound(); i < stop; ++i) {
5      something_else(i);
6  }
7  for (size_t i = 9; i <= 9; --i) {
8      something_else(i);
9  }
```

The first **for** counts `i` down from 10 to 1, inclusive. The condition is again just the evaluation of the variable `i`, no redundant test against value 0 is required. When `i` becomes 0, it will evaluate to false and the loop will stop. The second **for** declares two variables, `i` and `stop`. As before `i` is the loop variable, `stop` is what we compare against in the condition, and when `i` becomes greater than or equal to `stop`, the loop terminates.

The third **for** appears like it would go on forever, but actually counts down from 9 to 0. In fact, in the next section we will see that “sizes” in C, that is numbers that have type **size\_t**, are never negative.<sup>[Exs 2]</sup>

Observe that all three **for** statements declare variables named *i*. These three variables with the same name happily live side by side, as long as their scopes don’t overlap.

There are two more iterative statements in C, namely **while** and **do**.

```
1 while (condition) statement-or-block
2 do statement-or-block while(condition);
```

The following example shows a typical use of the first:

```
1 #include <tgmath.h>
2
3 double const eps = 1E-9;           // desired precision
4 ...
5 double const a = 34.0;
6 double x = 0.5;
7 while (fabs(1.0 - a*x) >= eps) {   // iterate until close
8     x *= (2.0 - a*x);              // Heron approximation
9 }
```

It iterates as long as the given condition evaluates true. The **do** loop is very similar, except that it checks the condition *after* the dependent block:

```
1 do {                               // iterate
2     x *= (2.0 - a*x);              // Heron approximation
3 } while (fabs(1.0 - a*x) >= eps); // iterate until close
```

This means that if the condition evaluates to false, a **while**-loop will not run its dependent block at all, and a **do**-loop will run it once before terminating.

As with the **for** statement, for **do** and **while** it is advisable to use the { ... } block variants. There is also a subtle syntactical difference between the two, **do** always needs a semicolon ; after the **while** (condition) to terminate the statement. Later we will see that this is a syntactic feature that turns out to be quite useful in the context of multiple nested statements, see Section 10.3.

All three iteration statements become even more flexible with **break** and **continue** statements. A **break** statement stops the loop without re-evaluating the termination condition or executing the part of the dependent block after the **break** statement:

```
1 while (true) {
2     double prod = a*x;
3     if (fabs(1.0 - prod) < eps)    // stop if close enough
4         break;
5     x *= (2.0 - prod);             // Heron approximation
6 }
```

This way, we can separate the computation of the product *a\*x*, the evaluation of the stop condition and the update of *x*. The condition of the **while** then becomes trivial. The same can be done using a **for**, and there is a tradition among C programmers to write it in as follows:

```
1 for (;;) {
2     double prod = a*x;
3     if (fabs(1.0 - prod) < eps)    // stop if close enough
```

[Exs 2] Try to imagine what happens when *i* has value 0 and is decremented by means of operator --.

```

4     break;
5     x *= (2.0 - prod);           // Heron approximation
6 }

```

**for** ( ; ; ) here is equivalent to **while** (**true**). The fact that the controlling expression of a **for** (the middle part between the ; ; ) can be omitted and is interpreted as “always **true**” is just an historic artifact in the rules of C and has no other special reason.

The **continue** statement is less frequently used. Like **break**, it skips the execution of the rest of the dependent block, so all statements in the block after the **continue** are not executed for the current iteration. However, it then re-evaluates the condition and continues from the start of the dependent block if the condition is true.



```

1 for (size_t i =0; i < max_iterations; ++i) {
2     if (x > 1.0) { // check if we are on the correct side of 1
3         x = 1.0/x;
4         continue;
5     }
6     double prod = a*x;
7     if (fabs(1.0 - prod) < eps) // stop if close enough
8         break;
9     x *= (2.0 - prod);           // Heron approximation
10 }

```

In the examples above we made use of a standard macro **fabs**, that comes with the `tgmath.h` header<sup>3</sup>. It calculates the absolute value of a **double**. If you are interested in how this works, Listing 1.1 is a program that does the same thing without the use of **fabs**. In it, **fabs** has been replaced by several explicit comparisons.

**#include** <tgmath.h>

The task of the program is to compute the inverse of all numbers that are provided to it on the command line. An example of a program execution looks like:

	Terminal
0	> ./heron 0.07 5 6E+23
1	heron: a=7.00000e-02,      x=1.42857e+01,      a*x=0.999999999996
2	heron: a=5.00000e+00,      x=2.00000e-01,      a*x=0.999999999767
3	heron: a=6.00000e+23,      x=1.66667e-24,      a*x=0.999999997028

To process the numbers on the command line the program uses another library function **strtod** from `stdlib.h`.<sup>[Exs 4][Exs 5][Exs 6]</sup>

**#include** <stdlib.h>

LISTING 1.1. A program to compute inverses of numbers

```

1 #include <stdlib.h>
2 #include <stdio.h>
3
4 /* lower and upper iteration limits centered around 1.0 */
5 static double const epslm01 = 1.0 - 0x1P-01;
6 static double const epslp01 = 1.0 + 0x1P-01;
7 static double const epslm24 = 1.0 - 0x1P-24;
8 static double const epslp24 = 1.0 + 0x1P-24;
9
10 int main(int argc, char* argv[argc+1]) {

```

<sup>3</sup>“tgmath” stands for *type generic mathematical functions*.

[Exs 4] Analyse Listing 1.1 by adding **printf** calls for intermediate values of **x**.

[Exs 5] Describe the use of the parameters **argc** and **argv** in Listing 1.1.

[Exs 6] Print out the values of **epslm01** and observe the output when you change them slightly.

```

11  for (int i = 1; i < argc; ++i) {           // process args
12      double const a = strtod(argv[i], 0);   // arg -> double
13      double x = 1.0;
14      for (;;) {                             // by powers of 2
15          double prod = a*x;
16          if (prod < eps1m01)                 x *= 2.0;
17          else if (eps1p01 < prod) x *= 0.5;
18          else break;
19      }
20      for (;;) {                             // Heron approximation
21          double prod = a*x;
22          if ((prod < eps1m24) || (eps1p24 < prod))
23              x *= (2.0 - prod);
24          else break;
25      }
26      printf("heron:_a=%.5e,\tx=%.5e,\ta*x=%.12f\n",
27             a, x, a*x);
28  }
29  return EXIT_SUCCESS;
30  }

```

**3.3. Multiple selection.** The last control statement that C has to offer is called **switch** statement and is another *selection*<sup>C</sup> statement. It is mainly used when cascades of **if-else** constructs would be too tedious:

```

1  if (arg == 'm') {
2      puts("this_is_a_magpie");
3  } else if (arg == 'r') {
4      puts("this_is_a_raven");
5  } else if (arg == 'j') {
6      puts("this_is_a_jay");
7  } else if (arg == 'c') {
8      puts("this_is_a_chough");
9  } else {
10     puts("this_is_an_unknown_corvid");
11 }

```

In this case, we have a choice that is more complex than a **false-true** decision and that can have several outcomes. We can simplify this as follows:

```

1  switch (arg) {
2      case 'm': puts("this_is_a_magpie");
3                break;
4      case 'r': puts("this_is_a_raven");
5                break;
6      case 'j': puts("this_is_a_jay");
7                break;
8      case 'c': puts("this_is_a_chough");
9                break;
10     default: puts("this_is_an_unknown_corvid");
11 }

```

Here we select one of the **puts** calls according to the value of the **arg** variable. Like **printf**, the function **puts** is provided by `stdio.h`. It outputs a line with the string that is passed as an argument. We provide specific cases for characters **'m'**, **'r'**, **'j'**, **'c'**

**#include <stdio.h>**

and a *fallback*<sup>C</sup> case labeled **default**. The default case is triggered if `arg` doesn't match any of the **case** values.<sup>[Exs 7]</sup>

Syntactically, a **switch** is as simple as

```
1  switch (expression) statement-or-block
```

and the semantics of it are quite straightforward: the **case** and **default** labels serve as *jump targets*<sup>C</sup>. According to the value of the `expression`, control just continues at the statement that is labeled accordingly. If we hit a **break** statement, the whole **switch** under which it appears terminates and control is transferred to the next statement after the **switch**.

By that specification a **switch** statement can in fact be used much more widely than iterated **if-else** constructs.

```
1  switch (count) {
2      default: puts("+++...+++");
3      case 4: puts("+++");
4      case 3: puts("++");
5      case 2: puts("+");
6      case 1: puts("");
7      case 0:;
8  }
```

Once we have jumped into the block, the execution continues until it reaches a **break** or the end of the block. In this case, because there are no **break** statements, we end up running all subsequent **puts** statements. For example, the output when the value of `count` is 3 would be a triangle with three lines.

	Terminal
0	+++
1	++
2	+

The structure of a **switch** can be more flexible than **if-else**, but it is restricted in another way:

**Rule 1.3.3.1** *case values must be integer constant expressions.*

In Section 5.4.2 we will see what these expressions are in detail. For now it suffices to know that these have to be fixed values that we provide directly in the source such as the 4, 3, 2, 1, 0 above. In particular variables such as `count` above are only allowed in the **switch** part but not for the individual **case** s.

With the greater flexibility of the **switch** statement also comes a price: it is more error prone. In particular, we might accidentally skip variable definitions:

**Rule 1.3.3.2** *case labels must not jump beyond a variable definition.*

<sup>[Exs 7]</sup> Test the above **switch** statement in a program. See what happens if you leave out some of the **break** statements.

#### 4. Expressing computations

We’ve already made use of some simple examples of *expressions*<sup>C</sup>. These are code snippets that compute some value based on other values. The simplest such expressions are certainly arithmetic expressions that are similar to those that we learned in school. But there are others, notably comparison operators such as `==` and `!=` that we already saw earlier.

In this section, the values and objects on which we will do these computations will be mostly of the type `size_t` that we already met above. Such values correspond to “sizes”, so they are numbers that cannot be negative. Their range of possible values starts at 0. What we would like to represent are all the non-negative integers, often denoted as  $\mathbb{N}$ ,  $\mathbb{N}_0$ , or “natural” numbers in mathematics. Unfortunately computers are finite so we can’t directly represent all the natural numbers, but we can do a reasonable approximation. There is a big upper limit `SIZE_MAX` that is the upper bound of what we can represent in a `size_t`.

**Rule 1.4.0.1** The type `size_t` represents values in the range  $[0, \text{SIZE\_MAX}]$ .

The value of `SIZE_MAX` is quite large, depending on the platform it should be one of

$$\begin{aligned} 2^{16} - 1 &= 65535 \\ 2^{32} - 1 &= 4294967295 \\ 2^{64} - 1 &= 18446744073709551615 \end{aligned}$$

The first value is a minimal requirement, the other two values are much more commonly used today. They should be large enough for calculations that are not too sophisticated. The standard header `stdint.h` provides `SIZE_MAX` such that you don’t have to figure it out yourself to write portable code.

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

The concept of “numbers that cannot be negative” to which we referred for `size_t` corresponds to what C calls *unsigned integer types*<sup>C</sup>. The symbols and combinations like `+` or `!=` are called *operators*<sup>C</sup> and the things to which they are applied are called *operands*<sup>C</sup>, so in something like “`a + b`”, “`+`” is the operator and “`a`” and “`b`” are its operands.

For an overview of all C operators see the following tables; Table 2 lists the operators that operate on values, Table 3 those that operate on objects and Table 4 those that operate on types. To work with these, you may have to jump from one table to another. *E.g.* if you want to work out an expression such as `a + 5` where `a` is some variable of type **unsigned**, you’d first have to go to the third line in Table 3 to see that `a` is evaluated. Then, you may use the third line in Table 2 to deduce that the value of `a` and `5` are combined in an arithmetic operation, namely `a +`.

Don’t be frustrated if you don’t understand everything in these tables. A lot of the concepts that are mentioned there are not yet introduced. They are listed here to form a reference for the whole book.

**4.1. Arithmetic.** Arithmetic operators form the first group in Table 2 of operators that operate on values.

4.1.1. `+`, `-` and `*`. Arithmetic operators `+`, `-` and `*` mostly work as we would expect by computing the sum, the difference and the product of two values.

```
1  size_t a = 45;
2  size_t b = 7;
3  size_t c = (a - b) * 2;
4  size_t d = a - b * 2;
```

TABLE 2. Value operators: “form” gives the syntactic form of the operation where @ represents the operator and a and possibly b denote values that serve as operands. For arithmetic and bit operations the type of the result is a type that reconciles the types of a and b.

operator	nick	form	type restriction		result	
			a	b		
		a	narrow		wide	promotion
+ -		a@b	pointer	integer	pointer	arithmetic
+ - * /		a@b	arithmetic	arithmetic	arithmetic	arithmetic
+ -		@a	arithmetic		arithmetic	arithmetic
%		a@b	integer	integer	integer	arithmetic
~	<b>compl</b> <b>bitand</b> <b>bitor</b> <b>xor</b>	@a	integer		integer	bit
&		a@b	integer	integer	integer	bit
^						
<< >>		a@b	integer	positive	integer	bit
== != < > <= >=	<b>not_eq</b>	a@b	scalar	scalar	0, 1	comparison
!a	<b>not</b> <b>and or</b>	a	scalar		0, 1	logic
&&		@a	scalar		0, 1	logic
		a@b	scalar	scalar	0, 1	logic
.		a@m	<b>struct</b>		value	member
*		@a	pointer		object	reference
[]		a[b]	pointer	integer	object	member
->		a@m	<b>struct</b> pointer		object	member
()		a (b ...)	function pointer		value	call
<b>sizeof</b>		@ a	none		<b>size_t</b>	size, ICE
<b>_Alignof</b>	<b>alignof</b>	@ (a)	none		<b>size_t</b>	alignment, ICE

TABLE 3. Object operators: “form” gives the syntactic form of the operation where @ represents the operator, o denotes an object and a denotes a suitable additional *value* (if any) that serve as operands. An additional \* in “type” requires that the object o is addressable.

operator	nick	form	type	result	
		o	array*	pointer	array decay
		o	function	pointer	function decay
		o	other	value	evaluation
=		o@a	non-array	value	assignment
+= -= *= /= += -= %= ++ -- &=  = ^= <<= >>=	<b>and_eq</b> <b>or_eq</b> <b>xor_eq</b>	o@a	arithmetic	value	arithmetic
		o@a	pointer	value	arithmetic
		o@a	integer	value	arithmetic
		@o o@	arithmetic or pointer	value	arithmetic
		o@a	integer	value	bit
		o@a	integer	value	bit
.		o@m	<b>struct</b>	object	member
[]		o[a]	array*	object	member
&		@o	any*	pointer	address
<b>sizeof</b>		@ o	data object, non-VLA	<b>size_t</b>	size, ICE
<b>sizeof</b>		@ o	VLA	<b>size_t</b>	size
<b>_Alignof</b>	<b>alignof</b>	@ (o)	non-function	<b>size_t</b>	alignment, ICE

TABLE 4. Type operators: these operators return an integer constant (ICE) of type `size_t`. They have function-like syntax with the operands in parenthesis.

operator	nick	form	type of T	
<code>sizeof</code>		<code>sizeof(T)</code>	any	size
<code>_Alignof</code>	<code>alignof</code>	<code>_Alignof(T)</code>	any	alignment
	<code>offsetof</code>	<code>offsetof(T, m)</code>	<code>struct</code>	member offset

must result in `c` being equal to 76, and `d` to 31. As you can see from that little example, sub-expressions can be grouped together with parenthesis to enforce a preferred binding of the operator.

In addition, operators `+` and `-` also have unary variants. `-b` just gives the negative of `b`, namely a value `a` such that `b + a` is 0. `+a` simply provides the value of `a`. The following would give 76 as well.

```
3  size_t c = (+a + -b) * 2;
```

Even though we use an unsigned type for our computation, negation and difference by means of the operator `-` is well defined. In fact, one of the miraculous properties of `size_t` is that `+-*` arithmetic always works where it can. This means that as long as the final mathematical result is within the range `[0, SIZE_MAX]`, then that result will be the value of the expression.

**Rule 1.4.1.1** *Unsigned arithmetic is always well defined.*

**Rule 1.4.1.2** *Operations `+`, `-` and `*` on `size_t` provide the mathematically correct result if it is representable as a `size_t`.*

In case that we have a result that is not representable, we speak of arithmetic *overflow*<sup>C</sup>. Overflow can e.g. happen if we multiply two values that are so large that their mathematical product is greater than `SIZE_MAX`. We'll look how C deals with overflow in the next section.

**4.1.2. Division and remainder.** The operators `/` and `%` are a bit more complicated, because they correspond to integer division and remainder operation. You might not be as used to them as to the other three arithmetic operators. `a/b` evaluates to the number of times `b` fits into `a`, and `a%b` is the remaining value once the maximum number of `b` are removed from `a`. The operators `/` and `%` come in pair: if we have `z = a / b` the remainder `a % b` could be computed as `a - z*b`:

**Rule 1.4.1.3** *For unsigned values,  $a == (a/b) * b + (a \% b)$ .*

A familiar example for the `%` operator are the hours on a clock. Say we have a 12 hour clock: 6 hours after 8 o'clock is 2 o'clock. Most people are able to compute time differences on 12 hour or 24 hour clocks. This computation corresponds to `a % 12`, in our example `(8 + 6) % 12 == 2`.<sup>[Exs 8]</sup> Another similar use for `%` is computation with minutes in the hour, of the form `a % 60`.

There is only one exceptional value that is not allowed for these two operations: 0. Division by zero is forbidden.

**Rule 1.4.1.4** *Unsigned `/` and `%` are well defined only if the second operand is not 0.*

<sup>[Exs 8]</sup> Implement some computations using a 24 hour clock, e.g. 3 hours after ten, 8 hours after twenty.



The `%` operator can also be used to explain additive and multiplicative arithmetic on unsigned types a bit better. As already mentioned above, when an unsigned type is given a value outside its range, it is said to *overflow*<sup>C</sup>. In that case, the result is reduced as if the `%` operator had been used. The resulting value “wraps around” the range of the type. In the case of `size_t`, the range is 0 to `SIZE_MAX`, therefore

**Rule 1.4.1.5** *Arithmetic on `size_t` implicitly does computation `% (SIZE_MAX+1)`.*

**Rule 1.4.1.6** *In case of overflow, unsigned arithmetic wraps around.*

This means that for `size_t` values, `SIZE_MAX + 1` is equal to 0 and `0 - 1` is equal to `SIZE_MAX`.

This “wrapping around” is the magic that makes the `-` operators work for unsigned types. For example, the value `-1` interpreted as a `size_t` is equal to `SIZE_MAX` and so adding `-1` to a value `a`, just evaluates to `a + SIZE_MAX` which wraps around to `a + SIZE_MAX - (SIZE_MAX+1) = a - 1`.

Operators `/` and `%` have the nice property that their results are always smaller than or equal to their operands:

**Rule 1.4.1.7** *The result of unsigned `/` and `%` is always smaller than the operands.*

And thus

**Rule 1.4.1.8** *Unsigned `/` and `%` can’t overflow.*

**4.2. Operators that modify objects.** Another important operation that we already have seen is assignment, `a = 42`. As you can see from that example this operator is not symmetric, it has a value on the right and an object on the left. In a freaky abuse of language C jargon often refers to the right hand side as *rvalue*<sup>C</sup> (right value) and to the object on the left as *lvalue*<sup>C</sup> (left value). We will try to avoid that vocabulary whenever we can: speaking of a value and an object is completely sufficient.

C has other assignment operators. For any binary operator `@` from the five we have seen above all have the syntax

```
1 an_object @= some_expression;
```

They are just convenient abbreviations for combining the arithmetic operator `@` and assignment, see Table 3. A mostly equivalent form would be

```
1 an_object = (an_object @ (some_expression));
```

In other words there are operators `+=`, `-=`, `*=`, `/=`, and `%=`. For example in a **for** loop operator `+=` could be used:

```
1 for (size_t i = 0; i < 25; i += 7) {
2     ...
3 }
```

The syntax of these operators is a bit picky, you aren’t allowed to have blanks between the different characters, e.g. “`i + = 7`” instead of “`i += 7`” is a syntax error.

**Rule 1.4.2.1** *Operators must have all their characters directly attached to each other.*

We already have seen two other operators that modify objects, namely the *increment operator*<sup>C</sup> `++` and the *decrement operator*<sup>C</sup> `--`:

- `++i` is equivalent to `i += 1`,
- `--i` is equivalent to `i -= 1`.

All these assignment operators are real operators, they return a value (but not an object!). You could, if you were crazy enough, write something like

```
1 a = b = c += ++d;
2 a = (b = (c += (++d))); // same
```

But such combinations of modifications to several objects in one go is generally frowned upon. Don't do that unless you want to obfuscate your code. Such changes to objects that are involved in an expression are referred to as *side effects*<sup>C</sup>.

**Rule 1.4.2.2** *Side effects in value expressions are evil.*

**Rule 1.4.2.3** *Never modify more than one object in a statement.*

For the increment and decrement operators there are even two other forms, namely *postfix increment*<sup>C</sup> and *postfix decrement*<sup>C</sup>. They differ from the one that we have seen in the result when they are used inside a larger expression. But since you will nicely obey to Rule 1.4.2.2, you will not be tempted to use them.

**4.3. Boolean context.** Several operators yield a value 0 or 1 depending on whether some condition is verified or not, see Table 2. They can be grouped in two categories, comparisons and logical evaluation.

**4.3.1. Comparison.** In our examples we already have seen the comparison operators `==`, `!=`, `<`, and `>`. Whereas the later two perform strict comparison between their operands, operators `<=` and `>=` perform “less or equal” and “greater or equal” comparison, respectively. All these operators can be used in control statements as we have already seen, but they are actually more powerful than that.

**Rule 1.4.3.1** *Comparison operators return the values **false** or **true**.*

Remember that **false** and **true** are nothing else then fancy names for 0 and 1 respectively. So they can perfectly used in arithmetic or for array indexing. In the following code

```
1 size_t c = (a < b) + (a == b) + (a > b);
2 size_t d = (a <= b) + (a >= b) - 1;
```

we have that `c` will always be 1, and `d` will be 1 if `a` and `b` are equal and 0 otherwise. With

```
1 double largeA[N] = { 0 };
2 ...
3 /* fill largeA somehow */
4
5 size_t sign[2] = { 0, 0 };
6 for (size_t i = 0; i < N; ++i) {
7     sign[(largeA[i] < 1.0)] += 1;
8 }
```

the array element `sign[0]` will hold the number of values in `largeA` that are greater than or equal to 1.0 and `sign[1]` those that are strictly less.

Finally, let's mention that there also is an identifier “**not\_eq**” that may be used as a replacement for `!=`. This feature is rarely used. It dates back to the times where some characters were not properly present on all computer platforms. To be able to use it you'd have to include the file `iso646.h`.

```
#include <iso646.h>
```

4.3.2. *Logic.* Logic operators operate on values that are already supposed to represent values **false** or **true**. If they are not, the rules that we described for conditional execution with Rules 1.3.1.1 and 1.3.1.2 apply first. The operator **!** (**not**) logically negates its operand, operator **&&** (**and**) is logical and, operator **||** (**or**) is logical or. The results of these operators are summarized in the following table:

TABLE 5. Logical operators

a	<b>not</b> a	a <b>and</b> b	<b>false</b>	<b>true</b>	a <b>or</b> b	<b>false</b>	<b>true</b>
<b>false</b>	<b>true</b>	<b>false</b>	<b>false</b>	<b>false</b>	<b>false</b>	<b>false</b>	<b>true</b>
<b>true</b>	<b>false</b>	<b>true</b>	<b>false</b>	<b>true</b>	<b>true</b>	<b>true</b>	<b>true</b>

Similar as for the comparison operators we have

**Rule 1.4.3.2** *Logic operators return the values **false** or **true**.*

Again, remember that these values are nothing else than 0 and 1 and can thus be used as indices:

```

1 double largeA[N] = { 0 };
2 ...
3 /* fill largeA somehow */
4
5 size_t isset[2] = { 0, 0 };
6 for (size_t i = 0; i < N; ++i) {
7     isset[!!largeA[i]] += 1;
8 }

```

Here the expression `!!largeA[i]` applies the `!` operator twice and thus just ensures that `largeA[i]` is evaluated as a truth value according to the general Rule 1.3.1.4. As a result, the array elements `isset[0]` and `isset[1]` will hold the number of values that are equal to 0 and unequal, respectively.

Operators `&&` and `||` have a particular property that is called *short circuit evaluation*<sup>C</sup>. This barbaric term denotes the fact that the evaluation of the second operand is omitted, if it is not necessary for the result of the operation. Suppose `isgreat` and `issmall` are two functions that yield a scalar value. Then in this code

```

1 if (isgreat(a) && issmall(b))
2     ++x;
3 if (issmall(c) || issmall(d))
4     ++y;

```

the second function call on each line would conditionally be omitted during execution: `issmall(b)` if `isgreat(a)` was 0, `issmall(d)` if `issmall(c)` was not 0. Equivalent code would be

```

1 if (isgreat(a))
2     if (issmall(b))
3         ++x;
4 if (issmall(c)) ++y;
5 else if (issmall(d)) ++y;

```

**4.4. The ternary or conditional operator.** The *ternary operator* is much similar to an **if** statement, only that it is an expression that returns the value of the chosen branch:

```
1 size_t size_min(size_t a, size_t b) {
2     return (a < b) ? a : b;
3 }
```

```
#include <tgmath.h>
```

Similar to the operators **&&** and **||** the second and third operand are only evaluated if they are really needed. The macro **sqrt** from **tgmath.h** computes the square root of a non-negative value. Calling it with a negative value raises a *domain error*<sup>C</sup>.

```
1 #include <tgmath.h>
2
3 #ifdef __STDC_NO_COMPLEX__
4 # error "we_need_complex_arithmetic"
5 #endif
6
7 double complex sqrt_real(double x) {
8     return (x < 0) ? CMLX(0, sqrt(-x)) : CMLX(sqrt(x), 0);
9 }
```

In this function **sqrt** is only called once, and the argument to that call is never negative. So **sqrt\_real** is always well behaved, no bad values are ever passed to **sqrt**.

```
#include <complex.h>
```

```
#include <tgmath.h>
```

Complex arithmetic and the tools used for it need the header **complex.h** which is indirectly included by **tgmath.h**. They will be introduced later in Section 5.5.7.

In the example above we also see conditional compilation that is achieved with *pre-processor directives*<sup>C</sup>, the **#ifdef** construct ensures that we hit the **#error** condition only if the macro **\_\_STDC\_NO\_COMPLEX\_\_** isn't defined.

**4.5. Evaluation order.** Of the above operators we have seen that **&&**, **||** and **?:** condition the evaluation of some of their operands. This implies in particular that for these operators there is an evaluation order on the operands: the first operand, since it is a condition for the remaining ones is always evaluated first:

**Rule 1.4.5.1** ***&&**, **||**, **?:** and **,** evaluate their first operand first.*

Here, **,** is the only operator that we haven't introduced, yet. It evaluates its operands in order and the result is then the value of the right operand. *E.g.*  $(f(a), f(b))$  would first evaluate  $f(a)$ , then  $f(b)$  and the result would be the value of  $f(b)$ . This feature is rarely useful in clean code, and is a trap for beginners. *E.g.*  $A[i, j]$  is *not* a two dimension index for matrix  $A$ , but results just in  $A[j]$ .

**Rule 1.4.5.2** *Don't use the **,** operator.*

Other operators don't have an evaluation restriction. *E.g.* in an expression such as  $f(a) + g(b)$  there is no pre-established ordering specifying whether  $f(a)$  or  $g(b)$  is to be computed first. If any of functions  $f$  or  $g$  work with side effects, *e.g.* if  $f$  modifies  $b$  behind the scenes, the outcome of the expression will depend on the chosen order.

**Rule 1.4.5.3** *Most operators don't sequence their operands.*

That chosen order can depend on your compiler, on the particular version of that compiler, on compile time options or just on the code that surrounds the expression. Don't rely on any such particular sequencing, it will bite you.

The same holds for the arguments of functions. In something like

```
1 printf("%g_and_%g\n", f(a), f(b));
```

we wouldn't know which of the last two arguments is evaluated first.

**Rule 1.4.5.4** *Function calls don't sequence their argument expressions.*

The only reliable way not to depend on evaluation ordering of arithmetic expressions is to ban side effects:

**Rule 1.4.5.5** *Functions that are called inside expressions should not have side effects.*

## 5. Basic values and data

We will now change the angle of view from the way “how things are to be done” (statements and expressions) to the things on which C programs operate, *values*<sup>C</sup> and *data*<sup>C</sup>.

A concrete program at an instance in time has to *represent* values. Humans have a similar strategy: nowadays we use a decimal presentation to write numbers down on paper, a system that we inherited from the arabic culture. But we have other systems to write numbers: roman notation, *e.g.*, or textual notation. To know that the word “twelve” denotes the value 12 is a non trivial step, and reminds us that European languages are denoting numbers not entirely in decimal but also in other systems. English is mixing with base 12, French with bases 16 and 20. For non-natives in French such as myself, it may be difficult to spontaneously associate “*quatre vingt quinze*” (four times twenty and fifteen) with the number 95.

Similarly, representations of values in a computer can vary “culturally” from architecture to architecture or are determined by the type that the programmer gave to the value. What representation a particular value has should in most cases not be your concern; the compiler is there to organize the translation between values and representations back and forth.

Not all representations of values are even *observable* from within your program. They only are so, if they are stored in *addressable* memory or written to an output device. This is another assumptions that C makes: it supposes that all data is stored in some sort of storage called memory that allows to retrieve values from different parts of the program in different moments in time. For the moment only keep in mind that there is something like an *observable state*<sup>C</sup>, and that a C compiler is only obliged to produce an executable that reproduces that observable state.

5.0.1. *Values.* A *value* in C is an abstract entity that usually exists beyond your program, the particular implementation of that program and the representation of the value during a particular run of the program. As an example, the value and concept of 0 should and will always have the same effects on all C platforms: adding that value to another value *x* will again be *x*, evaluating a value 0 in a control expression will always trigger the **false** branch of the control statement. C has the very simple rule

**Rule 1.5.0.1** *All values are numbers or translate to such.*

This really concerns all values a C program is about, whether these are the characters or texts that we print, truth values, measures that we take, relations that we investigate. First of all, think of these numbers as of mathematical entities that are independent of your program and its concrete realization.

The *data* of a program execution are all the assembled values of all objects at a given moment. The *state* of the program execution is determined by:

- the executable
- the current point of execution
- the data
- outside intervention such as IO from the user.

If we abstract from the last point, an executable that runs with the same data from the same point of execution must give the same result. But since C programs should be portable between systems, we want more than that. We don’t want that the result of a computation depends on the executable (which is platform specific) but ideally that it only depends on the program specification itself.

5.0.2. *Types.* An important step in that direction is the concept of *types*<sup>C</sup>. A type is an additional property that C associates with values. Up to now we already have seen several such types, most prominently **size\_t**, but also **double** or **bool**.

**Rule 1.5.0.2** *All values have a type that is statically determined.*

**Rule 1.5.0.3** *Possible operations on a value are determined by its type.*

**Rule 1.5.0.4** *A value's type determines the results of all operations.*

5.0.3. *Binary representation and the abstract state machine.* Unfortunately, the variety of computer platforms is not such that the C standard can impose the results of the operations on a given type completely. Things that are not completely specified as such by the standard are *e.g.* how the sign of signed type is represented, the so-called *sign representation*, or to which precision a **double** floating point operation is performed, so-called *floating point representation*. C only imposes as much properties on all representations, such that the results of operations can be deduced *a priori* from two different sources:

- the values of the operands
- some characteristic values that describe the particular platform.

*E.g.* the operations on the type **size\_t** can be entirely determined when inspecting the value of **SIZE\_MAX** in addition to the operands. We call the model to represent values of a given type on a given platform the *binary representation*<sup>C</sup> of the type.

**Rule 1.5.0.5** *A type's binary representation determines the results of all operations.*

Generally, all information that we need to determine that model are in reach of any C program, the C library headers provide the necessary information through named values (such as **SIZE\_MAX**), operators and function calls.

**Rule 1.5.0.6** *A type's binary representation is observable.*

This binary representation is still a model and so an *abstract representation* in the sense that it doesn't completely determine how values are stored in the memory of a computer or on a disk or other persistent storage device. That representation would be the *object representation*. In contrast to the binary representation, the object representation usually is of not much concern to us, as long as we don't want to hack together values of objects in main memory or have to communicate between computers that have a different platform model. Much later, in Section 12.1, we will see that we may even observe the object representation *if* such an object is stored in memory *and* we know its address.

As a consequence all computation is fixed through the values, types and their binary representations that are specified in the program. The program text describes an *abstract state machine*<sup>C</sup> that regulates how the program switches from one state to the next. These transitions are determined by value, type and binary representation, only.

**Rule 1.5.0.7 (as-if)** *Programs execute as if following the abstract state machine.*

5.0.4. *Optimization.* How a concrete executable achieves this goal is left to the discretion of the compiler creators. Most modern C compilers produce code that *doesn't* follow the exact code prescription, they cheat wherever they can and only respect the observable states of the abstract state machine. For example a sequence of additions with constants values such as

```
1  x += 5;
2  /* do something else without x in the mean time */
3  x += 7;
```

may in many cases be done as if it were specified as either

```

1  /* do something without x */
2  x += 12;

```

or

```

1  x += 12;
2  /* do something without x */

```

The compiler may perform such changes to the execution order as long as there will be no observable difference in the result, *e.g.* as long we don't print the intermediate value of "x" and as long as we don't use that intermediate value in another computation.

But such an optimization can also be forbidden because the compiler can't prove that a certain operation will not force a program termination. In our example, much depends on the type of "x". If the current value of x could be close to the upper limit of the type, the innocent looking operation `x += 7` may produce an overflow. Such overflows are handled differently according to the type. As we have seen above, overflow of an unsigned type makes no problem and the result of the condensed operation will always be consistent with the two separated ones. For other types such as signed integer types (**signed**) or floating point types (**double**) an overflow may "raise an exception" and terminate the program. So in this cases the optimization cannot be performed.

This allowed slackness between program description and abstract state machine is a very valuable feature, commonly referred to as *optimization<sup>C</sup>*. Combined with the relative simplicity of its language description, this is actually one of the main features that allows C to outperform other programming languages that have a lot more knobs and whistles. An important consequence about the discussion above can be summarized as follows.

**Rule 1.5.0.8** *Type determines optimization opportunities.*

**5.1. Basic types.** C has a series of basic types and some means of constructing *derived types<sup>C</sup>* from them that we will describe later in Section 6.

Mainly for historical reasons, the system of basic types is a bit complicated and the syntax to specify such types is not completely straightforward. There is a first level of specification that is entirely done with keywords of the language, such as **signed**, **int** or **double**. This first level is mainly organized according to C internals. On top of that there is a second level of specification that comes through header files and for which we already have seen examples, too, namely **size\_t** or **bool**. This second level is organized by type semantic, that is by specifying what properties a particular type brings to the programmer.

We will start with the first level specification of such types. As we already discussed above in Rule 1.5.0.1, all basic values in C are numbers, but there are numbers of different kind. As a principal distinction we have two different classes of numbers, with two subclasses, each, namely *unsigned integers<sup>C</sup>*, *signed integers<sup>C</sup>*, *real floating point numbers<sup>C</sup>* and *complex floating point numbers<sup>C</sup>*.

All these classes contain several types. They differ according to their *precision<sup>C</sup>*, which determines the valid range of values that are allowed for a particular type.<sup>9</sup> Table 6 contains an overview of the 18 base types. As you can see from that table there are some types which we can't directly use for arithmetic, so-called *narrow types<sup>C</sup>*. As a rule of thumb we get

**Rule 1.5.1.1** *Each of the 4 classes of base types has 3 distinct unpromoted types.*

<sup>9</sup>The term *precision* is used here in a restricted sense as the C standard defines it. It is different from the *accuracy* of a floating point computation.





TABLE 6. Base types according to the four main type classes. Types with a grey background don't allow for arithmetic, they are *promoted* before doing arithmetic. Type **char** is special since it can be unsigned or signed, depending on the platform. *All* types in the table are considered to be distinct types, even if they have the same class and precision.

class		systematic name	other name
integers	unsigned	<b>_Bool</b>	bool
		unsigned char	
		unsigned short	
		unsigned int	unsigned
		unsigned long	
	[un]signed	unsigned long long	
		char	
		signed char	
		signed short	short
		signed int	signed or int
floating point	real	signed long	long
		signed long long	long long
		float	
	complex	double	
		long double	
		float _Complex	float complex
		double _Complex	double complex
		long double _Complex	long double complex

Contrary to what many people believe, the C standard doesn't even prescribe the precision of these 12 types, it only constrains them. They depend on a lot of factors that are *implementation dependent*<sup>C</sup>. Thus, to choose the “best” type for a given purpose in a portable way could be a tedious task, if we wouldn't get help from the compiler implementation.

Remember that unsigned types are the most convenient types, since they are the only types that have an arithmetic that is defined consistently with mathematical properties, namely modulo operation. They can't raise signals on overflow and can be optimized best. They are described in more detail in Section 5.5.1.

**Rule 1.5.1.2** Use **size\_t** for sizes, cardinalities or ordinal numbers.

**Rule 1.5.1.3** Use **unsigned** for small quantities that can't be negative.

If your program really needs values that may both be positive and negative but don't have fractions, use a signed type, see Section 5.5.5.

**Rule 1.5.1.4** Use **signed** for small quantities that bear a sign.

**Rule 1.5.1.5** Use **ptrdiff\_t** for large differences that bear a sign.

If you want to do fractional computation with values such as 0.5 or 3.77189E+89 use floating point types, see Section 5.5.7.

**Rule 1.5.1.6** Use **double** for floating point calculations.

**Rule 1.5.1.7** Use **double complex** for complex calculations.

TABLE 7. Some semantic arithmetic types for specialized use cases

type	header	context of definition	meaning
<code>size_t</code>	<code>stddef.h</code>		type for “sizes” and cardinalities
<code>ptrdiff_t</code>	<code>stddef.h</code>		type for size differences
<code>uintmax_t</code>	<code>stdint.h</code>		maximum width unsigned integer, preprocessor
<code>intmax_t</code>	<code>stdint.h</code>		maximum width signed integer, preprocessor
<code>errno_t</code>	<code>errno.h</code>	Appendix K	error return instead of <code>int</code>
<code>rsize_t</code>	<code>stddef.h</code>	Appendix K	size arguments with bounds checking
<code>time_t</code>	<code>time.h</code>	<code>time(0)</code> , <code>difftime(t1, t0)</code>	calendar time in seconds since epoch
<code>clock_t</code>	<code>time.h</code>	<code>clock()</code>	processor time

The C standard defines a lot of other types, among them other arithmetic types that model special use cases. Table 7 list some of them. The second pair represents the types with maximal width that the platform supports. This is also the type in which the preprocessor does any of its arithmetic or comparison.

The third pair are types that can replace `int` and `size_t` in certain context. The first, `errno_t`, is just another name for `int` to emphasize the fact that it encodes an error value; `rsize_t`, in turn, is used to indicate that an interface performs bounds checking on its “size” parameters.

The two types `time_t` and `clock_t` are used to handle times. They are semantic types, because the precision of the time computation can be different from platform to platform. The way to have a time in seconds that can be used in arithmetic is the function `difftime`: it computes the difference of two timestamps. `clock_t` values present the platforms model of processor clock cycles, so the unit of time here is usually much below the second; `CLOCKS_PER_SEC` can be used to convert such values to seconds.

**5.2. Specifying values.** We have already seen several ways in which numerical constants, so-called *literals*<sup>C</sup> can be specified:

- 123 **decimal integer constant**<sup>C</sup>. The most natural choice for most of us.
- 077 **octal integer constant**<sup>C</sup>. This is specified by a sequence of digits, the first being 0 and the following between 0 and 7, *e.g.* 077 has the value 63. This type of specification has merely historical value and is rarely used nowadays. There is only one octal literal that is commonly used, namely 0 itself.
- 0xFFFF **hexadecimal integer constant**<sup>C</sup>. This is specified by a start of 0x followed by a sequence of digits between 0, ..., 9, a ... f, *e.g.* 0xbeaf is value 48815. The a ... f and x can also be written in capitals, 0XBEAF.
- 1.7E-13 **decimal floating point constants**<sup>C</sup>. Quite familiar for the version that just has a decimal point. But there is also the “scientific” notation with an exponent. In the general form `mEe` is interpreted as  $m \cdot 10^e$ .
- 0x1.7aP-13 **hexadecimal floating point constants**<sup>C</sup>. Usually used to describe floating point values in a form that will ease to specify values that have exact representations. The general form `0XhPe` is interpreted as  $h \cdot 2^e$ . Here *h* is specified as an hexadecimal fraction. The exponent *e* is still specified as a decimal number.



'a' *integer character constant*<sup>C</sup>. These are characters put into ' apostrophs, such as 'a' or '?'. These have values that are only implicitly fixed by the C standard. E.g. 'a' corresponds to the integer code for the character “a” of the Latin alphabet.

Inside character constants a “\” character has a special meaning. E.g. we already have seen '\n' for the newline character.

"hello" *string literals*<sup>C</sup>. They specify text, e.g. as we needed it for the **printf** and **puts** functions. Again, the “\” character is special as in character constants.

All but the last are numerical constants, they specify numbers. String literals are an exception and can be used to specify text that is known at compile time. Integrating larger text into our could be tedious, if we weren't allowed to split string literals into chunks:

```
1 puts("first_line\n"
2     "another_line\n"
3     "first_and_"
4     "second_part_of_the_third_line");
```

**Rule 1.5.2.1** *Consecutive string literals are concatenated.*

For numerical literals, the first rule is:

**Rule 1.5.2.2** *Numerical literals are never negative.*

That is if we write something like  $-34$  or  $-1.5\text{E}-23$ , the leading sign is not considered part of the number but is the *negation* operator applied to the number that comes after. We will see below where this is important. Bizarre as this may sound, the minus sign in the exponent is considered to be part of a floating point literal.

In view of Rule 1.5.0.2 we know that all literals must not only have a value but also a type. Don't mix up the fact of a constant having a positive value with its type, which can be **signed**.

**Rule 1.5.2.3** *Decimal integer constants are signed.*

This is an important feature, we'd probably expect the expression  $-1$  to be a signed, negative value.

To determine the exact type for integer literals we always have a “*first fit*” rule. For decimal integers this reads:

**Rule 1.5.2.4** *A decimal integer constant has the first of the 3 signed types that fits it.*

This rule can have surprising effects. Suppose that on a platform the minimal **signed** value is  $-2^{15} = -32768$  and the maximum value is  $2^{15} - 1 = 32767$ . The constant 32768 then doesn't fit into **signed** and is thus **signed long**. As a consequence the expression  $-32768$  has type **signed long**. Thus the minimal value of the type **signed** on such a platform cannot be written as a literal constant.<sup>[Exs 10]</sup>

**Rule 1.5.2.5** *The same value can have different types.*

Deducing the type of an octal or hexadecimal constant is a bit more complicated. These can also be of an unsigned type if the value doesn't fit for a signed one. In our example above the hexadecimal constant  $0\times7\text{FFF}$  has the value 32767 and thus type **signed**. Other than for the decimal constant, the constant  $0\times8000$  (value 32768 written in hexadecimal) then is an **unsigned** and expression  $-0\times8000$  again is **unsigned**.<sup>[Exs 11]</sup>

<sup>[Exs 10]</sup> Show that if the minimal and maximal values for **signed long long** have similar properties, the smallest integer value for the platform can't be written as a combination of one literal with a minus sign.

<sup>[Exs 11]</sup> Show that if in that case the maximum **unsigned** is  $2^{16} - 1$  that then  $-0\times8000$  has value 32768, too.

**Rule 1.5.2.6** *Don't use octal or hexadecimal constants to express negative values.*

Or if we formulate it positively

**Rule 1.5.2.7** *Use decimal constants to express negative values.*

Integer constants can be forced to be unsigned or to be of a type of minimal width. This done by appending “U”, “L” or “LL” to the literal. *E.g.* 1U has value 1 and type **unsigned**, 1L is **signed long** and 1ULL has the same value but type **unsigned long long**.<sup>[Exs 12]</sup>

TABLE 8. Examples for constants and their types, under the supposition that **signed** and **unsigned** have the commonly used representation with 32 bit.

constant $x$	value	type	value of $-x$
2147483647	+2147483647	<b>signed</b>	-2147483647
2147483648	+2147483648	<b>signed long</b>	-2147483648
4294967295	+4294967295	<b>signed long</b>	-4294967295
0x7FFFFFFF	+2147483647	<b>signed</b>	-2147483647
0x80000000	+2147483648	<b>unsigned</b>	+2147483648
0xFFFFFFFF	+4294967295	<b>unsigned</b>	+1
1	+1	<b>signed</b>	-1
1U	+1	<b>unsigned</b>	+4294967295

A common error is to try to assign a hexadecimal constant to a **signed** under the expectation that it will represent a negative value. Consider something like `int x = 0xFFFFFFFF`. This is done under the assumption that the hexadecimal value has the same *binary representation* as the signed value -1. On most architectures with 32 bit signed this will be true (but not on all of them) but then nothing guarantees that the effective value +4294967295 is converted to the value -1.

You remember that value 0 is important. It is so important that it has a lot of equivalent spellings: 0, 0x0 and '\0' are all the same value, a 0 of type **signed int**. 0 has no decimal integer spelling: 0.0 is a decimal spelling for the value 0 but seen as a floating point value, namely with type **double**.

**Rule 1.5.2.8** *Different literals can have the same value.*

For integers this rule looks almost trivial, for floating point constants this is less obvious. Floating point values are only an *approximation* of the value they present literally, because binary digits of the fractional part may be truncated or rounded.

**Rule 1.5.2.9** *The effective value of a decimal floating point constant may be different from its literal value.*

*E.g.* on my machine the constant 0.2 has in fact the value 0.2000000000000000111, and as a consequence constants 0.2 and 0.2000000000000000111 have the same value.

Hexadecimal floating point constants have been designed because they better correspond to binary representations of floating point values. In fact, on most modern architectures such a constant (that has not too many digits) will exactly correspond to the literal value. Unfortunately, these beasts are almost unreadable for mere humans.

<sup>[Exs 12]</sup> Show that the expressions -1U, -1UL and -1ULL have the maximum values and type of the three usable unsigned types, respectively.

Finally, floating point constants can be followed by the letters `f` or `F` to denote a **float** or by `l` or `L` to denote a **long double**. Otherwise they are of type **double**. Beware that different types of constants generally lead to different values for the same literal. A typical example:

	<b>float</b>	<b>double</b>	<b>long double</b>
literal	0.2F	0.2	0.2L
value	0x1.999999AP-3F	0x1.999999999999AP-3	0xC.CCCCCCCCCCCCCCDP-6L

**Rule 1.5.2.10** *Literals have value, type and binary representation.*

**5.3. Initializers.** We already have seen (Section 2.3) that the initializer is an important part of an object definition. Accessing uninitialized objects has undefined behavior, the easiest way out is to avoid that situation systematically:

**Rule 1.5.3.1** *All variables should be initialized.*

There are only few exception to that rule, VLA, see Section 6.1.3, that don't allow for an initializer, or code that must be highly optimized. The latter mainly occurs in situations that use pointers, so this is not yet relevant to us. For most code that we are able to write so far, a modern compiler will be able to trace the origin of a value to the last assignment or the initialization. Superfluous assignments will simply be optimized out.

For scalar types such as integers or floating point, an initializer just contains an expression that can be converted to that type. We already have seen a lot of examples for that. Optionally, such an initializer expression may be surrounded with `{ }`. Examples:

```
1 double a = 7.8;
2 double b = 2 * a;
3 double c = { 7.8 };
4 double d = { 0 };
```

Initializers for other types *must* have these `{ }`. E.g. array initializers contain initializers for the different elements, separated by a comma.

```
1 double A[] = { 7.8, };
2 double B[3] = { 2 * A[0], 7, 33, };
3 double C[] = { [0] = 7.8, [7] = 0, };
```

As we have already seen above, arrays that have an *incomplete type*<sup>C</sup> because there is no length specification are completed by the initializer to fully specify the length. Here A only has one element, whereas C has eight. For the first two initializers the element to which the scalar initialization applies is deduced from the position of the scalar in the list: e.g. `B[1]` is initialized to 7. Designated initializers as for C are by far preferable, since they make the code more robust against small changes in declaration.

**Rule 1.5.3.2** *Use designated initializers for all aggregate data types.*

If you don't know how to initialize a variable of type T, the *default initializer*<sup>C</sup> `a = {0}` will almost<sup>13</sup> always do.

**Rule 1.5.3.3** *{0} is a valid initializer for all object types that are not VLA.*

<sup>13</sup>The exception are variable length arrays, see Section 6.1.3.

There are several things, that ensure that this works. First, if we omit the designation (the `.fieldname` for **struct**, see Section 6.3 or `[n]` for arrays, see Section 6.1) initialization is just done in *declaration order*<sup>C</sup>, that is the 0 in the default initializer designates the very first field that is declared, and all other fields then are initialized per default to 0 as well. Then, the `{ }` form of initializers for scalars ensures that `{ 0 }` is also valid for these.

Maybe your compiler warns you about this: annoyingly some compiler implementers don't know about this special rule. It is explicitly designed as catch-all initializer in the C standard, so this is one of the rare cases where I would switch off a compiler warning.

**5.4. Named constants.** A common issue even in small programs is that they use special values for some purpose that are textually repeated all over. If for one reason or another this value changes, the program falls apart. Take an artificial setting as an example where we have arrays of strings, on which we would like to perform some operations:

```

1 char const*const animal[3] = {
2     "raven",
3     "magpie",
4     "jay",
5 };
6 char const*const pronoun[3] = {
7     "we",
8     "you",
9     "they",
10 };
11 char const*const ordinal[3] = {
12     "first",
13     "second",
14     "third",
15 };
16 ...
17 for (unsigned i = 0; i < 3; ++i)
18     printf("Corvid_%u_is_the_%s\n", i, animal[i]);
19 ...
20 for (unsigned i = 0; i < 3; ++i)
21     printf("%s_plural_pronoun_is_%s\n", ordinal[i], pronoun[i]);

```

Here we use the constant 3 at several places, and with three different “meanings” that are not much correlated. E.g. an addition to our set of corvids would need two independent code changes. In a real setting there could be much more places in the code that that would depend on this particular value, and in a large code base it can be very tedious to maintain.

**Rule 1.5.4.1** *All constants with particular meaning must be named.*

But it is equally important to distinguish constants that are equal, but for which equality is just a coincidence.

**Rule 1.5.4.2** *All constants with different meaning must be distinguished.*

**5.4.1. Read-only objects.** Don't mix the term “constant” which has a very specific meaning in C with objects that can't be modified. E.g. in the above code, `animal`, `pronoun` and `ordinal` are not constants according to our terminology, they are **const**-qualified objects. This *qualifier*<sup>C</sup> specifies that we don't have the right to change this object. For `animal` neither the array entries nor the actual strings can be modified and your compiler should give you a diagnostic if you try to do so.

**Rule 1.5.4.3** *An object of **const**-qualified type is read-only.*

That doesn't mean that the compiler or run-time system may not perhaps change the value of such an object: other parts of the program may see that object without the qualification and change it. The fact that you cannot write the summary of your bank account directly (but only read it), doesn't mean that it will remain constant over time.

There is another family of read-only objects, that unfortunately are not protected by their type from being modified: string literals:

**Rule 1.5.4.4** *String literals are read-only.*



If introduced today, the type of string literals would certainly be `char const []`, an array of **const**-qualified characters. Unfortunately, the **const** keyword had only been introduced much later than string literals to the C language, and therefore remained as it is for backward compatibility.<sup>14</sup>

5.4.2. *Enumerations.* C has a simple mechanism for such cases as we see them in the example, namely *enumerations*<sup>C</sup>:

```

1 enum corvid { magpie, raven, jay, corvid_num, };
2 char const*const animal[corvid_num] = {
3     [raven] = "raven",
4     [magpie] = "magpie",
5     [jay] = "jay",
6 };
7 ...
8 for (unsigned i = 0; i < corvid_num; ++i)
9     printf("Corvid_%u is the_%s\n", i, animal[i]);

```

This declares a new integer type **enum** `corvid` for which we know four different values. The rules for these values are simple:

**Rule 1.5.4.5** *Enumeration constants have either an explicit or positional value.*

As you might have guessed, positional values start from 0 onward, so in our example we have `raven` with value 0, `magpie` with 1, `jay` with 2 and `corvid_num` with 3. This last 3 is obviously the 3 we are interested in.

Now if we want to add another `corvid`, we just put it in the list, anywhere before `corvid_num`:

```

1 enum corvid { magpie, raven, jay, chough, corvid_num, };
2 char const*const animal[corvid_num] = {
3     [chough] = "chough",
4     [raven] = "raven",
5     [magpie] = "magpie",
6     [jay] = "jay",
7 };

```

As for most other narrow types, there is not really much interest of declaring variables of an enumeration type, for indexing and arithmetic they would be converted to a wider integer, anyhow. Even the enumeration constants themselves aren't of the enumeration type:

**Rule 1.5.4.6** *Enumeration constants are of type **signed int**.*

<sup>14</sup>A third class of read-only objects exist, temporary objects. We will see them later in Section 13.2.2.

So the interest really lies in the constants, not in the newly created type. We may thus name any **signed int** constant that we need, without even providing a *tag<sup>C</sup>* for the type name:

```
1 enum { p0 = 1, p1 = 2*p1, p2 = 2*p1, p3 = 2*p2, };
```

To define these constants we can use *integer constant expressions<sup>C</sup>*, *ICE*. Such an ICE provides a compile time integer value and is much restricted. Not only that its value must be determinable at compile time (no function call allowed), also no evaluation of an object must participate as an operand to the value.

```
1 signed const o42 = 42;
2 enum {
3     b42 = 42,           // ok, 42 is a literal
4     c52 = o42 + 10,    // error, o42 is an object
5     b52 = b42 + 10,    // ok, b42 is not an object
6 };
```

Here, `o52` is an object, **const**-qualified but still, so the expression for `c52` is not an “integer constant expression”.

**Rule 1.5.4.7** *An integer constant expression doesn’t evaluate any object.*

So principally an ICE may consist of any operations with integer literals, enumeration constants, **\_Alignof** and **offsetof** sub-expressions and eventually some **sizeof** sub-expressions.<sup>15</sup>

Still, even when the value is an ICE to be able to use it to define an enumeration constant you’d have to ensure that the value fits into a **signed**.

5.4.3. *Macros*. Unfortunately there is no other mechanism to declare constants of other type than **signed int** in the strict sense of the C language. Instead, C proposes another powerful mechanism that introduces textual replacement of the program code, *macros<sup>C</sup>*. A macro is introduced by a *preprocessor<sup>C</sup>* **#define**:

```
1 # define M_PI 3.14159265358979323846
```

This macro definition has an effect that the identifier `M_PI` is replaced in the then following program code by the **double** constant. Such a macro definition consists of 5 different parts:

- (1) A starting **#** character that must be the first non-blank character on the line.
- (2) The keyword **define**.
- (3) An identifier that is to be declared, here `M_PI`.
- (4) The replacement text, here `3.14159265358979323846`.
- (5) A terminating newline character.

With this trick we can declare textual replacement for constants of **unsigned**, **size\_t** or **double**. In fact the implementation imposed bound of **size\_t**, **SIZE\_MAX**, is such defined, but also many of the other system features that we already have seen: **EXIT\_SUCCESS**, **false**, **true**, **not\_eq**, **bool**, **complex**... Here, in this book such C standard macros are all printed in **dark red**.

These examples should not distract you from the general style requirement that is almost universally followed in production code:

**Rule 1.5.4.8** *Macro names are in all caps.*

Only deviate from that rule if you have good reasons, in particular not before you reached Level 3.

<sup>15</sup>We will handle the latter two concepts in Sections 12.7 and 12.1.



5.4.4. *Compound literals.* For types that don't have literals that describe their constants, things get even a bit more complicated. We have to use *compound literals*<sup>C</sup> on the replacement side of the macro. Such a compound literal has the form

```
1 (T) { INIT }
```

that is a type, in parenthesis, followed by an initializer. Example:

```
1 # define CORVID_NAME /**/ \
2 (char const*const[corvid_num]) { \
3     [chough] = "chough", \
4     [raven] = "raven", \
5     [magpie] = "magpie", \
6     [jay] = "jay", \
7 }
```

With that we could leave out the “animal” array from above and rewrite our **for**-loop:

```
1 for (unsigned i = 0; i < corvid_num; ++i)
2     printf("Corvid_%u is the_%s\n", i, CORVID_NAME[i]);
```

Whereas compound literals in macro definitions can help us to declare something that behaves similar to a constant of a chosen type, it ain't a constant in the narrow sense of C.

**Rule 1.5.4.9** *A compound literal defines an object.*

Over all, this form of macro has some pitfalls

- Compound literals aren't suitable for ICE.
- For our purpose here to declare “named constants” the type T should be **const-qualified**<sup>C</sup>. This ensures that the optimizer has a bit more slackness to generate good binary code for such a macro replacement.
- There *must* be space between the macro name and the ( ) of the compound literal, here indicated by the /\*\*/ comment. Otherwise this would be interpreted as the start of a definition of a *function-like macro*. We will see these much later.
- A backspace character \ at the *very end* of the line can be used to continue the macro definition to the next line.
- There must be no ; at the end of the macro definition. Remember it is all just text replacement.

**Rule 1.5.4.10** *Don't hide a terminating semicolon inside a macro.*

Also for readability of macros, please pity the poor occasional reader of your code:

**Rule 1.5.4.11** *Right-indent continuation markers for macros to the same column.*

As you can see above this helps to visualize the whole spread of the macro definition easily.

5.4.5. *Complex constants.* Complex types are not necessarily supported by all C platforms. The fact can be checked by inspecting **\_\_STDC\_NO\_COMPLEX\_\_**. To have full support of complex types, the header `complex.h` should be included. If you use `tgmath.h` for mathematical functions, this is already done implicitly.

```
#include <complex.h>
#include <tgmath.h>
```

It has several macros that may ease the manipulation of these types, in particular **I**, a constant value such that **I\*I** == -1.0F. This can be used to specify constants of complex types similar to the usual mathematical notation. E.g. `0.5 + 0.5*I` would be of type **double complex**, `0.5F + 0.5F*I` of **float complex**.

One character macro names in capital are often used in programs for numbers that are fixed for the whole program. By itself it is not a brilliant idea, the resource of one character names is limited, but you should definitely leave **I** alone.

TABLE 9. Bounds for scalar types used in this book

name	[min, max]	where	typical
<code>size_t</code>	<code>[0, SIZE_MAX]</code>	<code>&lt;stdint.h&gt;</code>	$[0, 2^w - 1]$ , $w = 32, 64$
<code>double</code>	<code>[±DBL_MIN, ±DBL_MAX]</code>	<code>&lt;float.h&gt;</code>	$[\pm 2^{-w-2}, \pm 2^w]$ , $w = 1024$
<code>signed</code>	<code>[INT_MIN, INT_MAX]</code>	<code>&lt;limits.h&gt;</code>	$[-2^w, 2^w - 1]$ , $w = 31$
<code>unsigned</code>	<code>[0, UINT_MAX]</code>	<code>&lt;limits.h&gt;</code>	$[0, 2^w - 1]$ , $w = 32$
<code>bool</code>	<code>[false, true]</code>	<code>&lt;stdbool.h&gt;</code>	$[0, 1]$
<code>ptrdiff_t</code>	<code>[PTRDIFF_MIN, PTRDIFF_MAX]</code>	<code>&lt;stdint.h&gt;</code>	$[-2^w, 2^w - 1]$ , $w = 31, 63$
<code>char</code>	<code>[CHAR_MIN, CHAR_MAX]</code>	<code>&lt;limits.h&gt;</code>	$[0, 2^w - 1]$ , $w = 7, 8$
<code>unsigned char</code>	<code>[0, UCHAR_MAX]</code>	<code>&lt;limits.h&gt;</code>	$[0, 255]$

**Rule 1.5.4.12** ***I** is reserved for the imaginary unit.*

Another possibility to specify complex values is **CMPLX**, e.g. **CMPLX**(0.5, 0.5) is the same **double complex** value as above, and using **CMPLXF** is similar for **float complex**. But using **I** as above is probably more convenient since the type of the constant is then automatically adapted from the two floating point constants that are used for the real and imaginary part.

### 5.5. Binary representations.

**Rule 1.5.5.1** *The same value may have different binary representations.*

5.5.1. *Unsigned integers.* We already have seen that unsigned integer types are those arithmetic types for which the standard arithmetic operations have a nice and closed mathematical description. Namely they are closed to these operations:

**Rule 1.5.5.2** *Unsigned arithmetic wraps nicely.*

In mathematical terms they implement a *ring*,  $\mathbb{Z}_N$ , the set of integers modulo some number  $N$ . The values that are representable are  $0, \dots, N - 1$ . The maximum value  $N - 1$  completely determines such an unsigned integer type and is made available through a macro with terminating **\_MAX** in the name. For the basic unsigned integer types these are **UINT\_MAX**, **ULONG\_MAX** and **ULLONG\_MAX** and they are provided through `limits.h`. As we already have seen the one for `size_t` is **SIZE\_MAX** from `stdint.h`.

The binary representation for non-negative integer values is always exactly what the term indicates: such a number is represented by binary digits  $b_0, b_1, \dots, b_{p-1}$  called *bits*<sup>C</sup>. Each of the bits has a value of 0 or 1. The value of such a number is computed as

$$(1) \quad \sum_{i=0}^{p-1} b_i 2^i.$$

The value  $p$  in that binary representation is called the *precision*<sup>C</sup> of the underlying type. Of the bits  $b_i$  that are 1 the one with minimal index  $i$  is called the *least significant bit*<sup>C</sup>, *LSB*, the one with the highest index is the *most significant bit*<sup>C</sup>, *MSB*. E.g. for an unsigned type with  $p = 16$ , the value 240 would have  $b_4 = 1$ ,  $b_5 = 1$ ,  $b_6 = 1$  and  $b_7 = 1$ . All other bits of the binary representation are 0, the LSB is  $b_4$  the MSB is  $b_7$ . From (1) we see immediately that  $2^p$  is the first value that cannot be represented with the type. Thus we have that  $N = 2^p$  and

**Rule 1.5.5.3** *The maximum value of any integer type is of the form  $2^p - 1$ .*

Observe that for this discussion of the representation of non-negative values we hadn't argued about the signedness of the type. These rules apply equally to signed or unsigned types. Only for unsigned types we are lucky and what is said so far completely suffices to describe such an unsigned type.

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

**Rule 1.5.5.4** *Arithmetic on an unsigned integer type is determined by its precision.*

Finally, Tab. 9 shows the bounds of some of the commonly used scalars throughout this book.

**5.5.2. Bit sets and bitwise operators.** This simple binary representation of unsigned types allows us to use them for another purpose that is not directly related to arithmetic, namely as bit sets. A bit set is a different interpretation of an unsigned value, where we assume that it represents a subset of the base set  $V = \{0, \dots, p-1\}$  and where we take element  $i$  to be member of the set, if the bit  $b_i$  is present.

There are three binary operators that operate on bitsets:  $|$ ,  $\&$ , and  $\wedge$ . They represent the *set union*  $A \cup B$ , *set intersection*  $A \cap B$  and *symmetric difference*  $A \Delta B$ , respectively. For an example let us chose  $A = 240$ , representing  $\{4, 5, 6, 7\}$ , and  $B = 287$ , the bit set  $\{0, 1, 2, 3, 4, 8\}$ . We then have

bit op	value	hex	$b_{15}$	...	$b_0$	set op	set
V	65535	0xFFFF	1	1	1		$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$
A	240	0x00F0	0	0	0		$\{4, 5, 6, 7\}$
$\sim A$	65295	0xFF0F	1	1	1	$V \setminus A$	$\{0, 1, 2, 3, 8, 9, 10, 11, 12, 13, 14, 15\}$
$-A$	65296	0xFF10	1	1	1		$\{4, 8, 9, 10, 11, 12, 13, 14, 15\}$
B	287	0x011F	0	0	0		$\{0, 1, 2, 3, 4, 8\}$
$A B$	511	0x01FF	0	0	0	$A \cup B$	$\{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
$A\&B$	16	0x0010	0	0	0	$A \cap B$	$\{4\}$
$A\wedge B$	495	0x01EF	0	0	0	$A \Delta B$	$\{0, 1, 2, 3, 5, 6, 7, 8\}$

For the result of these operations the total size of the base set, and thus the precision  $p$  is not needed. As for the arithmetic operators, there are corresponding assignment operators  $\&=$ ,  $/=$ , and  $\wedge=$ , respectively.<sup>[Exs 16][Exs 17][Exs 18][Exs 19]</sup>

There is yet another operator that operates on the bits of the value, the complement operator  $\sim$ . The complement  $\sim A$  would have value 65295 and would correspond to the set  $\{0, 1, 2, 3, 8, 9, 10, 11, 12, 13, 14, 15\}$ . This bit complement always depends on the precision  $p$  of the type.<sup>[Exs 20][Exs 21]</sup>

All these operator can be written with identifiers, namely **bitor**, **bitand**, **xor**, **or\_eq**, **and\_eq**, **xor\_eq**, and **compl** if you include header `iso646.h`.

`#include <iso646.h>`

A typical usage of bit sets is for “flags”, variables that control certain settings of a program.

```

1 enum corvid { magpie, raven, jay, chough, corvid_num, };
2 #define FLOCK_MAGPIE 1U
3 #define FLOCK_RAVEN 2U
4 #define FLOCK_JAY 4U
5 #define FLOCK_CHOUGH 8U
6 #define FLOCK_EMPTY 0U
7 #define FLOCK_FULL 15U
8
9 int main(void) {
10     unsigned flock = FLOCK_EMPTY;
11
12     ...

```

[Exs 16] Show that  $A \setminus B$  can be computed by  $A - (A \& B)$

[Exs 17] Show that  $V + 1$  is 0.

[Exs 18] Show that  $A \wedge B$  is equivalent to  $(A - (A \& B)) + (B - (A \& B))$  and  $A + B - 2 * (A \& B)$

[Exs 19] Show that  $A|B$  is equivalent to  $A + B - (A \& B)$

[Exs 20] Show that  $\sim B$  can be computed by  $V - B$

[Exs 21] Show that  $\sim B = \sim B + 1$

```

13
14     if (something) flock |= FLOCK_JAY;
15
16     ...
17
18     if (flock & FLOCK_CHOUGH)
19         do_something_chough_specific(flock);
20
21 }

```

Here the constants for each type of corvid are a power of two, and so they have exactly one bit set in their binary representation. Membership in a “flock” can then be handled through the operators: `|=` adds a corvid to `flock` and `&` with one of the constants tests if a particular corvid is present.

Observe the similarity between operators `&` and `&&` or `|` and `||`: if we see each of the bits  $b_i$  of an unsigned as a truth value, `&` performs the “logical and” of all bits of its arguments simultaneously. This is a nice analogy that should help to memorize the particular spelling of these operators. On the other hand have in mind that operators `||` and `&&` have short circuit evaluation, so be sure to distinguish them clearly from the bit operators.

**5.5.3. Shift operators.** The next set of operators builds a bridge between interpretation of unsigned values as numbers and as bit sets. A left shift operation `<<` corresponds to the multiplication of the numerical value by the corresponding power of two. *E.g.* for  $A = 240$ ,  $A << 2$  is  $240 \cdot 2^2 = 240 \cdot 4 = 960$ , which represents the set  $\{6, 7, 8, 9\}$ . Resulting bits that don’t fit into the binary representation for the type are simply omitted. In our example,  $A << 9$  would correspond to set  $\{13, 14, 15, 16\}$  (and value 122880), but since there is no bit 16 the resulting set is  $\{13, 14, 15\}$ , value 57344.

Thus for such a shift operation the precision  $p$  is important, again. Not only that bits that don’t fit are dropped it also restricts the possible values of the operand on the right:

**Rule 1.5.5.5** *The second operand of a shift operation must be less than the precision.*

There is an analogous right shift operation `>>` that shifts the binary representation towards the less significant bits. Analogously this corresponds to an integer division by a power of two. Bits in positions less or equal to the shift value are omitted for the result. Observe that for this operation, the precision of the type isn’t important.<sup>[Exs 22]</sup>

Again, there are also corresponding assignment operators `<<=` and `>>=`.

The left shift operator `<<` has a primary use for specifying powers of two. In our example from above we may now replace the **#define**s by:

```

1 #define FLOCK_MAGPIE (1U << magpie)
2 #define FLOCK_RAVEN (1U << raven)
3 #define FLOCK_JAY (1U << jay)
4 #define FLOCK_CHOUGH (1U << chough)
5 #define FLOCK_EMPTY 0U
6 #define FLOCK_FULL ((1U << corvid_num)-1)

```

This makes the example more robust against changes to the enumeration.

**5.5.4. Boolean values.** The Boolean data type in C is also considered to be an unsigned type. Remember that it only has values 0 and 1, so there are no negative values. For historical reasons the basic type is called `_Bool`. The name `bool` as well as the constants **false** and **true** only come through the inclusion of `stdbool.h`. But it probably is a good idea to use the latter to make the semantics of the types and values that you are using clear.



**#include** <stdbool.h>

[Exs 22] Show that the bits that are “lost” in an operation  $x \gg n$  correspond to the remainder  $x \% (1ULL \ll n)$ .

Treating `bool` as an unsigned type is a certain stretch of the concept. Assignment to a variable of that type doesn't follow the Modulus Rule 1.4.1.5, but the special Rule 1.3.1.1.

You'd probably need `bool` variables rarely. They are only useful if you'd want to ensure that the value is always reduced to **false** or **true** on assignment. Early versions of C didn't have a Boolean type and many experienced C programmers still don't use it.

5.5.5. *Signed integers.* Signed types are a bit more complicated to handle than unsigned types, because a C implementation has to decide on two points

- What happens on arithmetic overflow?
- How is the sign of a signed type represented?

Signed and unsigned types come in pairs, with the notable two exceptions from Table 6 **char** and **bool**. The binary representation of the signed type is constrained by corresponding unsigned type:

**Rule 1.5.5.6** *Positive values are represented independently from signedness.*

Or stated otherwise, a positive value with a signed type has the same representation as in the corresponding unsigned type. That is the reason why we were able to express Rule 1.5.5.3 for *all* integer types. These also have a precision, *p* that determines the maximum value of the type.

The next thing that the standard prescribes is that signed types have exactly one additional bit, the *sign bit*<sup>C</sup>. If it is 0 we have a positive value, if it is 1 the value is negative. Unfortunately there are different concepts how such a sign bit can be used to obtain a negative number. C allows three different *sign representations*<sup>C</sup>

- *sign and magnitude*<sup>C</sup>
- *ones' complement*<sup>C</sup>
- *two's complement*<sup>C</sup>

The first two nowadays probably only have historic or exotic relevance: for sign and magnitude, the magnitude is taken as for positive values, and the sign bit simply specifies that there is a minus sign. Ones' complement takes the corresponding positive value and complements all bits. Both representations have the disadvantage that two values evaluate to 0, there is a positive and a negative 0.



Commonly used on modern platforms is the two's complement representation. It performs exactly the same arithmetic as we have already seen for unsigned types, only that the upper half of the unsigned values is interpreted as being negative. The following two functions are basically all that is needed to interpret unsigned values as signed ones:

```

1 bool is_negative(unsigned a) {
2     unsigned const int_max = UINT_MAX/2;
3     return a > int_max;
4 }
5 bool is_signed_less(unsigned a, unsigned b) {
6     if (is_negative(b) && !is_negative(a)) return false;
7     else return a < b;
8 }

```

When realized like that, signed integer arithmetic will again behave more or less nicely. Unfortunately, there is a pitfall that makes the outcome of signed arithmetic difficult to predict: overflow. Where unsigned values are forced to wrap around, the behavior of a signed overflow is *undefined*<sup>C</sup>. The following two loops look quite the same:

```

1 for (unsigned i = 1; i; ++i) do_something();
2 for ( signed i = 1; i; ++i) do_something();

```

We know what happens for the first one: the counter is increment up to `UINT_MAX`, then wraps around to 0. All of this may take some time, but after `UINT_MAX-1` iterations the loop stops because `i` will have reached 0.

For the second, all looks similar. But because here the behavior of overflow is undefined the compiler is allowed to *pretend* that it will never happen. Since it also knows that the value at start is positive it may assume that `i`, as long as the program has defined behavior, is never negative or 0. The *as-if* Rule 1.5.0.7 allows it to optimize the second loop to

```
1 while (true) do_something();
```

That's right, an *infinite loop*. This is a general feature of undefined behavior in C code:

**Rule 1.5.5.7** *Once the abstract state machine reaches an undefined state no further assumption about the continuation of the execution can be made.*

Not only that the compiler is allowed to do what pleases for the operation itself (“*undefined? so let's define it*”), but it may also assume that it will never reach such a state and draw conclusions from that.

**Rule 1.5.5.8** *It is your responsibility to avoid undefined behavior of all operations.*

What makes things even worse is that on some platforms with some standard compiler options all will just *look* right. Since the behavior is undefined, on a given platform signed integer arithmetic might turn out basically the same as unsigned. But changing the platform, the compiler or just some options can change that. All of a sudden your program that worked for years crashes out of nowhere.

Basically what we discussed up to this section always had well defined behavior, so the abstract state machine is always in a well defined state. Signed arithmetic changes this, so as long as you mustn't, avoid it.

**Rule 1.5.5.9** *Signed arithmetic may trap badly.*

One of the things that might already overflow for signed types is negation. We have seen above that `INT_MAX` has all bits but the sign bit set to 1. `INT_MIN` has then the “next” representation, namely the sign bit set to 1 and all other values set to 0. The corresponding value is not `-INT_MAX`.<sup>[Exs 23]</sup>

We then have

**Rule 1.5.5.10** *In twos' complement representation `INT_MIN` < `-INT_MAX`.*

Or stated otherwise, in twos' complement representation the positive value `-INT_MIN` is out of bounds since the *value* of the operation is larger than `INT_MAX`.

**Rule 1.5.5.11** *Negation may overflow for signed arithmetic.*

For signed types, bit operations work with the binary representation. So the value of a bit operation depends in particular on the sign representation. In fact bit operations even allow to detect the sign representation.

<sup>[Exs 23]</sup> Show that `INT_MIN+INT_MAX` is `-1`.

```

1 char const* sign_rep[4] =
2 {
3     [1] = "sign_and_magnitude",
4     [2] = "ones'_complement",
5     [3] = "two's_complement",
6     [0] = "weird",
7 };
8 enum { sign_magic = -1&3, };
9 ...
10 printf("Sign_representation:_%s.\n", sign_rep[sign_magic]);

```

The shift operations then become really messy. The semantics of what such an operation is for a negative value is not clear.

**Rule 1.5.5.12** *Use unsigned types for bit operations.*

5.5.6. *Fixed width integer types.* The precision for the integer types that we have seen so far can be inspected indirectly by using macros from `limits.h`, such as **UINT\_MAX** or **LONG\_MIN**. The C standard only gives us a minimal precision for them. For the unsigned types these are:

`#include <limits.h>`

type	minimal precision
<code>bool</code>	1
<code>unsigned char</code>	8
<code>unsigned short</code>	16
<code>unsigned</code>	16
<code>unsigned long</code>	32
<code>unsigned long long</code>	64

Under some technical constraints such guarantees might not be sufficient and the C standard provides names for *exact-width integer types* in `stdint.h`. As the name indicates they are of an exact prescribed “width”, which for provided unsigned types is guaranteed to be the same as their precision.

`#include <stdint.h>`

**Rule 1.5.5.13** *If the type `uintN_t` is provided it is an unsigned integer type with exactly  $N$  bits width and precision.*

For signed types this mechanism is even more restrictive:

**Rule 1.5.5.14** *If the type `intN_t` is provided it is signed, with two’s complement representation, has a width of exactly  $N$  bits and a precision of  $N - 1$ .*

None of these types is guaranteed to exist, but

**Rule 1.5.5.15** *If types with the required properties exist for values of 8, 16, 32 or 64, types `uintN_t` and `intN_t` respectively must be provided.*

And in fact, nowadays platforms usually provide `uint8_t`, `uint16_t`, `uint32_t` and `uint64_t` unsigned types and `int8_t`, `int16_t`, `int32_t` and `int64_t` signed types. Their presence and bounds can be tested with macros **UINT8\_MAX**, **UINT16\_MAX**, **UINT32\_MAX** and **UINT64\_MAX** for unsigned types and **INT8\_MIN**, **INT8\_MAX**, **INT16\_MIN**, **INT16\_MAX**, **INT32\_MIN**, **INT32\_MAX**, **INT64\_MIN** and **INT64\_MAX**, respectively.<sup>[Exs 24]</sup>

To encode literals of the requested type there are macros **UINT8\_C**, **UINT16\_C**, **UINT32\_C** **UINT64\_C**, **INT8\_C**, **INT16\_C**, **INT32\_C** and **INT64\_C**, respectively. E.g on platforms where `uint64_t` is **unsigned long**, **INT64\_C**(1) would expand to 1UL.

[Exs 24] If they exist, the value of all these macros is prescribed by the properties of the types. Think of a closed formulas in  $N$  for these values.



**Rule 1.5.5.16** For any of the fixed-width types that are provided, **\_MIN** (only signed), maximum **\_MAX** and literals **\_C** macros are provided, too.

Since we cannot know the type behind such fixed-width type, it would be difficult to guess the correct format specifier that we would have to use for **printf** and friends. The `#include <inttypes.h>` header `inttypes.h` provides us with macros for that. E.g for  $N = 64$  we are provided with **PRId64**, **PRi64**, **PRo64**, **PRu64**, **PRx64** and **PRIX64**, for **printf** formats **%d**, **%i**, **%o**, **%u**, **%x** and **%X**, respectively:

```
uint32_t n = 78;
int64_t max = (-UINT64_C(1)) >> 1; // same value as INT64_MAX
printf("n is %PRu32 and max is %PRId64\n", n, max);
```

As you can see, these macros expand to string literals that are combined with other string literals into the format string. This is certainly not the best candidate for a C coding beauty contest.

**5.5.7. Floating point data.** Where integers come near the mathematical concepts of  $\mathbb{N}$  (unsigned) or  $\mathbb{Z}$  (signed), floating point types are close to  $\mathbb{R}$  (non-complex) or  $\mathbb{C}$  (complex).

The way they differ from these mathematical concepts is twofold. First there is a size restriction of what is presentable. This is similar to what we have seen for integer types. The include file `float.h` has e.g. constants **DBL\_MIN** and **DBL\_MAX** that provides us with the minimal and maximal values for **double**. But beware, here **DBL\_MIN** is the smallest number that is strictly greater than 0.0; the smallest negative **double** value is **-DBL\_MAX**.

But real numbers ( $\mathbb{R}$ ) have another difficulty when we want to represent them on a physical system: they can have an unlimited expansion, such as the value  $\frac{1}{3}$  which has an endless repetition of digit 3 in decimal representation or such as the value of  $\pi$  which is “transcendent” and so has an endless expansion in any representation, and which even doesn’t repeat in any way.

C and other programming languages deal with these difficulties by cutting off the expansion. The position where the expansion is cut is “floating”, thus the name, and depends on the magnitude of the number in question.

In a view that is a bit simplified a floating point value is computed from the following values:

$s$  sign ( $\pm 1$ )  
 $e$  exponent, an integer  
 $f_1, \dots, f_p$  values 0 or 1, the mantissa bits.

For the exponent we have  $e_{min} \leq e \leq e_{max}$ .  $p$ , the number of bits in the mantissa is called *precision*. The floating point value is then given by the formula:

$$s \cdot 2^e \cdot \sum_{k=1}^p f_k 2^{-k}.$$

The values  $p$ ,  $e_{min}$  and  $e_{max}$  are type dependent, and therefore not represented explicitly in each number. They can be obtained through macros such as **DBL\_MANT\_DIG** (for  $p$ , typically 53) **DBL\_MIN\_EXP** ( $e_{min}$ , -1021) and **DBL\_MAX\_EXP** ( $e_{max}$ , 1024).

If we have e.g. a number that has  $s = -1$ ,  $e = -2$ ,  $f_1 = 1$ ,  $f_2 = 0$  and  $f_3 = 1$ , its value is

$$-1 \cdot 2^{-2} \cdot (f_1 2^{-1} + f_2 2^{-2} + f_3 2^{-3}) = -1 \cdot \frac{1}{4} \cdot \left(\frac{1}{2} + \frac{1}{8}\right) = -1 \cdot \frac{1}{4} \cdot \frac{4+1}{8} = \frac{-5}{32}$$

which corresponds to the decimal value  $-0.15625$ . From that calculation we see also that floating point values are always representable as a fraction that has some power of two in the denominator.<sup>[Exs 25]</sup>

[Exs 25] Show that all representable floating point values with  $e > p$  are multiples of  $2^{e-p}$ .



An important thing that we have to have in mind with such floating point representations is that values can be cut off during intermediate computations.

**Rule 1.5.5.17** *Floating point operations are neither associative, commutative or distributive.*

So basically they lose all nice algebraic properties that we are used to when doing pure math. The problems that arise from that are particularly pronounced if we operate with values that have very different orders of magnitude.<sup>[Exs 26]</sup> E.g. adding a very small floating point value  $x$  with an exponent that is less than  $-p$  to a value  $y > 1$  just returns  $y$ , again. As a consequence it is really difficult to assert without further investigation if two computations have had the “same” result. Such investigations are often cutting edge research questions, so we cannot expect to be able to assert equality or not. We are only able to tell that they are “close”:

**Rule 1.5.5.18** *Never compare floating point values for equality.*

The representation of the complex types is straightforward, and identical to an array of two elements of the corresponding real floating point type. To access the real and imaginary part of a complex number we have two type generic macros that also come with the header `tgmath.h`, namely `creal` and `cimag`. For any  $z$  of one of the three complex types we have that  $z == \text{creal}(z) + \text{cimag}(z) * \mathbf{I}$ .<sup>27</sup>

`#include <tgmath.h>`

<sup>[Exs 26]</sup> Print the results of the following expressions: `1.0E-13 + 1.0E-13` and `(1.0E-13 + (1.0E-13 + 1.0)) - 1.0`

<sup>27</sup>We will learn about such function-like macros in Section 8.

## 6. Aggregate data types

All other data types in C are derived from the basic types that we know now. There are four different strategies for combining types:

- arrays These combine items that all have the same base type.
- pointers Entities that refer to an object in memory.
- structures These combine items that may have different base types.
- unions These overlay items of different base types in the same memory location.

Of these four, pointers are by far the most involved concept, and we will delay the full discussion on them to Section 11. Below, in Section 6.2, we only will present them as opaque data type, without even mentioning the real purpose they fulfill.

Unions also need a deeper understanding of C's memory model and are not of much use in every day's programmers life, so they are only introduced in Section 12.2. Here, for this level, we will introduce *aggregate data types*<sup>C</sup>, data types that group together several data to form one unit.

**6.1. Arrays.** Arrays allow us to group objects of the same type into an encapsulating object. We only will see pointer types later (Section 11) but many people come to C with a confusion of arrays and pointers. And this is completely normal, arrays and pointers are closely related in C and to explain them we face a *hen and egg* problem: arrays *look like* pointers in many contexts, and pointers refer to array objects. We chose an order of introduction that is perhaps unusual, namely we start with arrays and try to stay with them as long as possible before introducing pointers. This may seem “wrong” for some of you, but remember that everything which stated here has to be taken with the *as-if* rule: we will first describe arrays in a way that will be consistent with C's assumptions about the abstract state machine.

We start with a very important rule:

**Rule 1.6.1.1** *Arrays are not pointers.*

We will later see on how these two concepts relate, but for the moment it is important to enter this section without prejudice about arrays, otherwise you will block your ascent to a better understanding of C for a while.

**6.1.1. Array declaration.** We have already seen how arrays are declared, namely by placing something like *[N]* after another declaration. Examples:

```
1 double a[16];
2 signed b[N];
```

Here *a* comprises 16 subobjects of type **double** and *b* *N* of type **signed**.

The type that composes an array may itself again be an array to form a so-called *multidimensional array*<sup>C</sup>. For those, declarations become a bit more difficult to read since *[]* binds to the left. The following two declarations declare variables of exactly the same type:

```
1 double C[M][N];
2 double (D[M])[N];
```

Both, *C* and *D* are *M* objects of array type **double** *[N]*, we have to read an array declaration from inside out.

We also already have seen how array elements are accessed and initialized, again with a pair of *[]*. With the above *a[0]* is an object of **double** and can be used wherever we want to use *e.g.* a simple variable. As we have seen *C[0]* is itself an array, and so *C[0][0]* which is the same as *(C[0])[0]* is again an object of type **double**.

Initializers can use *designated initializers* to pick the specific position to which an initialization applies. Example code on pages 39 and 47 contains such initializers.

6.1.2. *Array operations.* Arrays are really objects of a different type than we have seen so far. First an array in a logical operator make not much sense, what would the truth value of such array be?

**Rule 1.6.1.2** *An array in a condition evaluates to **true**.*

The “truth” of that comes from the “array decay” operation, that we will see later. Another important property is that we can’t evaluate arrays like other objects.

**Rule 1.6.1.3** *There are array objects but no array values.*

So arrays can’t be operands for value operators in Table 2, there is no arithmetic declared on arrays (themselves) and also

**Rule 1.6.1.4** *Arrays can’t be compared.*

Arrays also can’t be on the value side of object operators in Table 3. Most of the object operators are likewise ruled out to have arrays as object operands, either because they assume arithmetic or because they have a second value operand that would have to be an array, too. In particular,

**Rule 1.6.1.5** *Arrays can’t be assigned to.*

From that table we see that there are only four operators left that work on arrays as object operator. We already know the operator `[]`<sup>28</sup>. The “array decay” operation, the address operator `&` and the `sizeof` operator will be introduced later.

6.1.3. *Array length.* There are two different categories of arrays, *fixed length arrays*<sup>C</sup>, FLA, and *variable length arrays*<sup>C</sup>, VLA. The first are a concept that has been present in C since the beginnings and this feature is shared with many other programming languages. The second, was introduced in C99 and is relatively unique to C, and has some restrictions to its usage.

**Rule 1.6.1.6** *VLA can’t have initializers.*

**Rule 1.6.1.7** *VLA can’t be declared outside functions.*

So let’s start at the other end and see which arrays are in fact FLA, such that they don’t fall under these restrictions.

**Rule 1.6.1.8** *The length of an FLA is determined by an ICE or an initializer.*

In the first case, their length is know at compile time through a integer constant expression, ICE, as we have introduced them in Section 5.4.2. There is no type restriction for the ICE, any integer type would do. The only restriction is

**Rule 1.6.1.9** *An array length specification must be strictly positive.*

There is another important special case that leads to a FLA, when there is no length specification at all. If the `[]` is left empty, the length of the array is determined from its initializer, if any:

```
1  double C[] = { [3] = 42.0, [2] = 37.0, };
2  double D[] = { 22.0, 17.0, 1, 0.5, };
```

<sup>28</sup>The real C jargon story about arrays and `[]` is a bit more complicated. Let us apply the **as-if** Rule 1.5.0.7 to our explanation. All C program behaves as if the `[]` are directly applied to an array object.

Here C and D both are of type `double[4]`. Since such an initializer's structure can always be determined at compile time without necessarily knowing the values of the items, the array then still is an FLA.

All other array variable declarations lead to VLA.

**Rule 1.6.1.10** *An array with a length not an integer constant expression is an VLA.*

The length of an array can be computed with the `sizeof` operator. That operator provides the “size” of any object<sup>29</sup> and so the length of an array can be calculate by a simple division.

**Rule 1.6.1.11** *The length of an array A is  $(\text{sizeof } A) / (\text{sizeof } A[0])$ .*

Namely the size of the array object itself divided by the size of any of the array elements.

6.1.4. *Arrays as parameters.* Yet another special case occurs for arrays as parameters to functions. As we have seen for the prototype of `printf` above, such parameters may have empty `[]`. Since there is no initializer possible for such a parameter, the array dimension can't be determined.

**Rule 1.6.1.12** *The innermost dimension of an array parameter to a function is lost.*

**Rule 1.6.1.13** *Don't use the `sizeof` operator on array parameters to functions.*

Array parameter are even more bizarre, because of Rule 1.6.1.3 array parameters cannot be passed by value; there is no such thing as an array value in C. As a first approximation of what is happening for array parameters to functions we have:

**Rule 1.6.1.14** *Array parameters behave as-if the array is **passed by reference**<sup>C</sup>.*

Take the following as an example:

```

1  #include <stdio.h>
2
3  void swap_double(double a[static 2]) {
4      double tmp = a[0];
5      a[0] = a[1];
6      a[1] = tmp;
7  }
8
9  int main(void) {
10     double A[2] = { 1.0, 2.0, };
11     swap_double(A);
12     printf("A[0]=%g, A[1]=%g\n", A[0], A[1]);
13 }
```

Here, `swap_double(A)` will act directly on array A and not on a copy. Therefore the program will swap the values of the two elements of A.

<sup>29</sup>Later we will see what the unit of measure for such sizes is.

6.1.5. *Strings are special.* There is a special kind of arrays that we already encountered several times and that in contrast to other arrays even has literals, *strings*<sup>C</sup>.

**Rule 1.6.1.15** A string is a 0-terminated array of **char**.

That is a string as "hello" always has one more element than is visible that contains the value 0, so here the array has length 6.

As all arrays, strings can't be assigned to, but they can be initialized from string literals:

```
1 char chough0[] = "chough";
2 char chough1[] = { "chough" };
3 char chough2[] = { 'c', 'h', 'o', 'u', 'g', 'h', 0, };
4 char chough3[7] = { 'c', 'h', 'o', 'u', 'g', 'h', };

```

These are all equivalent declarations. Beware that not all arrays of **char** are strings, such as

```
1 char chough4[6] = { 'c', 'h', 'o', 'u', 'g', 'h', };

```

because it is not 0-terminated.

We already briefly have seen the base type **char** of strings among the integer types. It is a narrow integer type that can be used to encode all characters of the *basic character set*<sup>C</sup>. This character set contains all characters of the Latin alphabet, Arabic digits and punctuation characters that we use for coding in C. It usually doesn't contain special characters (e.g. "ä", "á") or characters from completely different writing systems.

The vast majority of platforms nowadays uses the so-called ASCII<sup>30</sup> to encode characters in the type **char**. We don't have to know how the particular encoding works as long as we stay in the basic character set, everything is done in C and its standard library that this encoding is used transparently.

To deal with **char** arrays and strings, there are a bunch of functions in the standard library that come with the header `string.h`. Those that just suppose an array start their names with "mem" and those that in addition suppose that their arguments are strings start with "str".

`#include <string.h>`

Functions that operate on **char**-arrays:

- **memcpy**(target, source, len) can be used to copy one array to another. These have to be known to be distinct arrays. The number of **char** to be copied must be given as a third argument len.
- **memcmp**(s0, s1, len) compares two arrays in the lexicographic ordering. That is it first scans initial parts of the two arrays that happen to be equal and then returns the difference between the two first characters that are distinct. If no differing elements are found up to len, 0 is returned.
- **memchr**(s, c, len) searches array s for the appearance of character c.

String functions:

- **strlen**(s) returns the length of the string s. This is simply the position of the first 0-character and *not* the length of the array. It is your duty to ensure that s is indeed a string, that is that it is 0-terminated.
- **strcpy**(target, source) works similar to **memcpy**. It only copies up to the string length of the source, and therefore it doesn't need a len parameter. Again, source must be 0-terminated. Also, target must be big enough to hold the copy.

<sup>30</sup>American Standard Code for Information Interchange

- **strcmp**(s0, s1) compares two arrays in the lexicographic ordering similar to **memcmp**, but may not take some language specialties into account. The comparison stops at the first 0-character that is encountered in either s0 or s1. Again, both parameters have to be 0-terminated.
- **strcoll**(s0, s1) compares two arrays in the lexicographic ordering respecting language specific environment settings. We will learn how to properly ensure to set this in Section 8.5.
- **strchr**(s, c) is similar to **memchr**, only that the string s must be 0-terminated.
- **strspn**(s0, s1) returns the number of initial characters in s0 that also appear in s1.
- **strcspn**(s0, s1) returns the number of initial characters in s0 that do not appear in s1.

**Rule 1.6.1.16** *Using a string function with a non-string has undefined behavior.*

In real life, common symptoms for such a misuse may be:

- long times for **strlen** or similar scanning functions because they don't encounter a 0-character
- segmentation violations because such functions try to access elements after the boundary of the array object
- seemingly random corruption of data because the functions write data in places where they are not supposed to.

In other words, be careful, and make sure that all your strings really are strings. If your platform already supports this, use the functions with bounds checking that were introduced with C11. There are **strlen\_s** and **strcpy\_s** that additionally deal with the maximum length of their string parameters.<sup>[Exs 31]</sup>

The following is an example that uses many of the features that we talked about.

LISTING 1.2. copying a string

```

1  #include <string.h>
2  #include <stdio.h>
3  int main(int argc, char* argv[argc+1]) {
4      size_t const len = strlen(argv[0]); // compute the length
5      char name[len+1];                  // create VLA
6                                          // ensure place for 0
7      memcpy(name, argv[0], len);        // copy the name
8      name[len] = 0;                     // ensure 0 character
9      if (!strcmp(name, argv[0])) {
10         printf("program_name_\\"%s\\"_successfully_copied\\n",
11              name);
12     } else {
13         printf("copying_%s_leads_to_different_string_%s\\n",
14              argv[0], name);
15     }
16 }
```

In the above discussion I have been hiding an important detail to you: the prototypes of the functions. For the string functions they can be written as

```

1  size_t strlen(char const s[static 1]);
2  char* strcpy(char target[static 1], char const source[static 1])
3      ;
4  signed strcmp(char const s0[static 1], char const s1[static 1]);
```

[Exs 31] Use **memchr** and **memcmp** to implement a bounds checking version of **strcmp**.

```

4 signed strcoll(char const s0[static 1], char const s1[static 1]);
5 char* strchr(const char s[static 1], int c);
6 size_t strspn(const char s1[static 1], const char s2[static 1]);
7 size_t strcspn(const char s1[static 1], const char s2[static 1]);

```

Besides the bizarre return type of `strcpy`, this looks reasonable. The parameter arrays are arrays of “unknown” length, so the `[static 1]`, they correspond to arrays of at least one `char`. `strlen`, `strspn` and `strcspn` are to return a “size” and `strcmp` a negative, 0 or positive value according to the sort order of the arguments.

The picture darkens when we look at the declarations of the array functions:

```

1 void* memcpy(void* target, void const* source, size_t len);
2 signed memcmp(void const* s0, void const* s1, size_t len);
3 void* memchr(const void *s, int c, size_t n);

```

You are missing knowledge about entities that are specified as `void*`. These are “pointers” to objects of unknown type. It is only on Level 2, Section 11, that we will see why and how these new concept of pointers and `void`-type occur.

**6.2. Pointers as opaque types.** As a first approach we only need to know some simple properties of pointers.

The binary representation of pointer is completely up to the platform and not our business.

**Rule 1.6.2.1** *Pointers are opaque objects.*

This means that we will only be able to deal with pointers through the operations that the C language allows for them. As said, most of these operations will be introduced later. Here we only need one particular property of pointers, they have a state:

**Rule 1.6.2.2** *Pointers are valid, null or indeterminate.*

In fact, the null state of any pointer type corresponds to our old friend 0, sometimes known under its pseudo **false**.

**Rule 1.6.2.3** *Initialization or assignment with 0 makes a pointer null.*

Usually we refer to a pointer in the null state as *null pointer*<sup>C</sup>. Surprisingly, disposing of null pointers is really a feature.

**Rule 1.6.2.4** *In logical expressions, pointers evaluate to **false** iff they are null.*

Note that such test can’t distinguish valid pointers from indeterminate ones. So, the really “bad” state of a pointer is “indeterminate”, since this state is not observable.

**Rule 1.6.2.5** *Indeterminate pointers lead to undefined behavior.*

In view of Rule 1.5.5.7, we need to make sure that pointers never reach an intermediate state. Thus, if we can’t ensure that a pointer is valid, we *must* at least ensure that it is set to null:

**Rule 1.6.2.6** *Always initialize pointers.*

**6.3. Structures.** As we now have seen, arrays combine several objects of the same base type into a larger object. This makes perfect sense where we want to combine information for which the notion of a first, second etc. element is acceptable. If it is not, or if we have to combine objects of different type, structures, introduced by the keyword **struct** come into play.

As a first example, let us revisit the corvids on page 39. There, we used a trick with an enumeration type to keep track of our interpretation of the individual elements of an array name. C structures allow for a more systematic approach by giving names to so-called fields in an aggregate:

```

1 struct animalStruct {
2     const char* jay;
3     const char* magpie;
4     const char* raven;
5     const char* chough;
6 };
7 struct animalStruct const animal = {
8     .chough = "chough",
9     .raven = "raven",
10    .magpie = "magpie",
11    .jay = "jay",
12 };

```

That is, from Line 1 to 6 we have the declaration of a new type, denoted as **struct** `animalStruct`. This structure has four *fields*<sup>C</sup>, who's declaration look exactly like normal variable declarations. So instead of declaring four elements that are bound together in an array, here we name the different fields and declare types for them. Such declaration of a structure type only explains the type, it is not (yet) the declaration of an object of that type and even less a definition for such an object.

Then, starting in Line 7 we declare and define a variable (called `animal`) of the new type. In the initializer and in later usage, the individual fields are designated in a notation with a dot (`.`). Instead of `animal[chough]` for the array we have `animal.chough` for the structure.

Now, for a second example, let us look at a way to organize time stamps. Calendar time is an complicated way of counting, in years, month, days, minutes and seconds; the different time periods such as month or years can have different length etc. One possible way would be to organize such data in an array, say:

```

1 typedef signed calArray[6];

```

The use of this array type would be ambiguous: would we store the year in element `[0]` or `[5]`? To avoid ambiguities we could again use our trick with an **enum**. But the C standard has chosen a different way, in `time.h` it uses a **struct** that looks similar to the following:

```
#include <time.h>
```

```

1 struct tm {
2     int tm_sec; // seconds after the minute    [0, 60]
3     int tm_min; // minutes after the hour      [0, 59]
4     int tm_hour; // hours since midnight       [0, 23]
5     int tm_mday; // day of the month          [1, 31]
6     int tm_mon;  // months since January       [0, 11]
7     int tm_year; // years since 1900
8     int tm_wday; // days since Sunday          [0, 6]
9     int tm_yday; // days since January         [0, 365]
10    int tm_isdst; // Daylight Saving Time flag
11 };

```



This **struct** has *named fields*, such as **tm\_sec** for the seconds or **tm\_year** for the year. Encoding a date, such as the date of this writing

```

0      > LC_TIME=C date -u
1      Sat Mar 29 16:07:05 UTC 2014

```

is simple:

```

29      struct tm today = {
30          .tm_year = 2014,
31          .tm_mon  = 2,
32          .tm_mday = 29,
33          .tm_hour = 16,
34          .tm_min  = 7,
35          .tm_sec  = 5,
36      };

```

This creates a variable of type **struct tm** and initializes its fields with the appropriate values. The order or position of the fields in the structure is usually not important: using the name of the field preceded with a dot **.** suffices to specify where the corresponding data should go.

Accessing the fields of the structure is as simple and has similar “.” syntax:

```

37      printf("this_year_is_%d, next_year_will_be_%d\n",
38          today.tm_year, today.tm_year+1);

```

A reference to a field such as `today.tm_year` can appear in expression just as any variable of the same base type would.

There are three other fields in **struct tm** that we didn’t even mention in our initializer list, namely **tm\_wday**, **tm\_yday** and **tm\_isdst**. Since we don’t mention them, they are automatically set to 0.

**Rule 1.6.3.1** *Omitted **struct** initializers force the corresponding field to 0.*

This can even go to the extreme that all but one of the fields are initialized.

**Rule 1.6.3.2** *A **struct** initializer must initialize at least one field.*

In fact, in Rule 1.5.3.3 we have already seen that there is a default initializer that works for all data types: `{0}`.

So when we initialize **struct tm** as above, the data structure is not consistent; the **tm\_wday** and **tm\_yday** fields don’t have values that would correspond to the values of the remaining fields. A function that sets this field to a value that is consistent with the others could be such as

```

19      struct tm time_set_yday(struct tm t) {
20          // tm_mday starts at 1
21          t.tm_yday += DAYS_BEFORE[t.tm_mon] + t.tm_mday - 1;
22          // take care of leap years

```

```
23     if ((t.tm_mon > 1) && leapyear(t.tm_year))
24         ++t.tm_yday;
25     return t;
26 }
```

It uses the number of days of the months preceding the actual one, the `tm_mday` field and an eventual corrective for leap years to compute the day in the year. This function has a particularity that is important at our current level, it modifies only the field of the parameter of the function, `tm`, and not of the original object.

**Rule 1.6.3.3** ***struct** parameters are passed by value.*

To keep track of the changes, we have to reassign the result of the function to the original.

```
39  today = time_set_yday(today);
```

Later, with pointer types we will see how to overcome that restriction for functions, but we are not there, yet. Here we see that the assignment operator “=” is well defined for all structure types. Unfortunately, its counterparts for comparisons are not:

**Rule 1.6.3.4** Structures can be assigned with = but not compared with == or !=.

Listing 1.3 shows the completed example code for the use of `struct tm`. It doesn't contain a declaration of the historical `struct tm` since this is provided through the standard header `time.h`. Nowadays, the types for the individual fields would probably be chosen differently. But many times in C we have to stick with design decisions that have been done many years ago.

**Rule 1.6.3.5** *A structure layout is an important design decision.*

You may regret your design after some years, when all the existing code that uses it makes it almost impossible to adapt it to new situations.

Another use of **struct** is to group objects of different types together in one larger enclosing object. Again, for manipulating times with a nanosecond granularity the C standard already has made that choice:

```
1 struct timespec {
2     time_t tv_sec; // whole seconds    ≥ 0
3     long tv_nsec; // nanoseconds       [0, 999999999]
4 };
```

So here we see the opaque type `time_t` that we already know from Table 7 for the seconds, and a `long` for the nanoseconds.<sup>32</sup> Again, the reasons for this choice are just historic, nowadays the chosen types would perhaps be a bit different. To compute the difference between two `struct timespec` times, we can easily define a function.

Whereas the function `difftime` is part of the C standard, such a functionality here is very simple and isn't based on platform specific properties. So it can easily be implemented by anyone who needs it.<sup>[Exs 33]</sup>

Any data type besides VLA is allowed as a field in a structure. So structures can also be nested in the sense that a field of a **struct** can again of (another) **struct** type, and the smaller enclosed structure may even be declared inside the larger one:

<sup>32</sup>Unfortunately even the semantic of `time_t` is different, here. In particular, `tv_sec` may be used in arithmetic.

[Exs 33] Write a function `timespec_diff` that computes the difference between two `timespec` values.



LISTING 1.3. A sample program manipulating **struct tm**

```

1  #include <time.h>
2  #include <stdbool.h>
3  #include <stdio.h>
4
5  int leapyear(unsigned year) {
6      /* All years that are divisible by 4 are leap years, unless
7       * they start a new century, provided they don't start a new
8       * millennium. */
9      return !(year % 4) && ((year % 100) || !(year % 1000));
10 }
11
12 #define DAYS_BEFORE \
13 (int const[12]){ \
14     [0] = 0, [1] = 31, [2] = 59, [3] = 90, \
15     [4] = 120, [5] = 151, [6] = 181, [7] = 212, \
16     [8] = 243, [9] = 273, [10] = 304, [11] = 334, \
17 }
18
19 struct tm time_set_yday(struct tm t) {
20     // tm_mday starts at 1
21     t.tm_yday += DAYS_BEFORE[t.tm_mon] + t.tm_mday - 1;
22     // take care of leap years
23     if ((t.tm_mon > 1) && leapyear(t.tm_year))
24         ++t.tm_yday;
25     return t;
26 }
27
28 int main(void) {
29     struct tm today = {
30         .tm_year = 2014,
31         .tm_mon = 2,
32         .tm_mday = 29,
33         .tm_hour = 16,
34         .tm_min = 7,
35         .tm_sec = 5,
36     };
37     printf("this_year_is_%d,_next_year_will_be_%d\n",
38         today.tm_year, today.tm_year+1);
39     today = time_set_yday(today);
40     printf("day_of_the_year_is_%d\n", today.tm_yday);
41 }

```

```

1  struct person {
2      char name[256];
3      struct stardate {
4          struct tm date;
5          struct timespec precision;
6      } bdate;
7  };

```

The visibility of declaration **struct stardate** is the same as for **struct person**, there is no “namespace” associated to a **struct** such as it were *e.g.* in C++.

**Rule 1.6.3.6** All **struct** declarations in a nested declaration have the same scope of visibility.

So a more realistic version of the above would be as follows.

```
1 struct stardate {
2     struct tm date;
3     struct timespec precision;
4 };
5 struct person {
6     char name[256];
7     struct stardate bdate;
8 };
```

This version places all **struct** on the same level, as they end up there, anyhow.

**6.4. New names for types: typedef.** As we have seen in the previous section, a structure not only introduces a way to aggregate differing information into one unit, it also introduces a new type name for the beast. For historical reasons (again!) the name that we introduce for the structure always has to be preceded by the keyword **struct**, which makes the use of it a bit clumsy. Also many C beginners run into difficulties with this when they forget the **struct** keyword and the compiler throws an incomprehensible error at them.

There is a general tool that can help us avoid that, by giving a symbolic name to an otherwise existing type: **typedef**. By that a type can have several names, and we can even reuse the *tag name*<sup>C</sup> that we used in the structure declaration:

```
1 typedef struct animalStruct animalStructure;
2 typedef struct animalStruct animalStruct;
```

Then, “**struct** animalStruct”, “animalStruct” or “animalStructure” can all be used interchangeably. My favorite use of this feature is the following idiom:

```
1 typedef struct animalStruct animalStruct;
2 struct animalStruct {
3     ...
4 };
```

That is to *precede* the proper **struct** declaration by a **typedef** using exactly the same name. This works because the combination of **struct** with a following name, the *tag*<sup>C</sup> is always valid, a so-called *forward declaration*<sup>C</sup> of the structure.

**Rule 1.6.4.1** Forward-declare a **struct** within a **typedef** using the same identifier as the tag name.

C++ follows a similar approach by default, so this strategy will make your code easier to read for people coming from there.

The **typedef** mechanism can also be used for other types than structures. For arrays this could look like:

```
1 typedef double vector[64];
2 typedef vector vecvec[16];
3 vecvec A;
4 typedef double matrix[16][64];
5 matrix B;
6 double C[16][64];
```



Here **typedef** only introduces a new name for an existing type, so A, B and C have exactly the same type, namely **double** [16] [64].

The C standard also uses **typedef** a lot internally. The semantic integer types such as **size\_t** that we have seen in Section 5.1 are declared with this mechanism. Here the standard often uses names that terminate with “**\_t**” for “**typedef**”. This naming convention ensures that the introduction of such a name by an upgraded version of the standard will not conflict with existing code. So you shouldn’t introduce such names yourself in your code.

**Rule 1.6.4.2** *Identifier names terminating with **\_t** are reserved.*

## 7. Functions

We have already seen the different means that C offers for conditional execution, that is execution which according to some value will choose one branch of the program over another to continue. So there, the reason for a potential “jump” to another part of the program code (e.g. to an **else** branch) is a runtime decision that depends on runtime data.

This section handles other forms of transfer of control to other parts of our code, that is unconditional, i.e. that doesn’t (by itself) require any runtime data to come up with the decision where to go. The main reason motivating this kind of tools is *modularity*.

- Avoid code repetition.
  - Avoid copy and paste errors.
  - Increase readability and maintainability.
  - Decrease compilation times.
- Provide clear interfaces.
  - Specify the origin and type of data that flows into a computation.
  - Specify the type and value of the result of a computation.
  - Specify invariants for a computation, namely pre- and post-conditions.
- Dispose a natural way to formulate algorithms that use a “stack” of intermediate values.

Besides the concept of functions, C has other means of unconditional transfer of control, that are mostly used to handle error conditions or other forms of exceptions from the usual control flow:

- **exit**, **\_Exit**, **quick\_exit** and **abort** terminate the program execution, see Section 8.6.
- **goto** transfers control within a function body, see Sections 13.2.2 and 15.
- **setjmp** and **longjmp** can be used to return unconditionally to a calling context, see Section 18.4.
- Certain events of the execution environment or calls to the function **raise** may raise so-called signals and that pass control to a specialized function, a *signal handler*.

**7.1. Simple functions.** We already have seen a lot of functions for now, so the basic concept should hopefully be clear: parenthesis `()` play an important syntactical role for functions. They are used for function declaration and definition to encapsulate the list of parameter declaration. For function calls they hold the list of actual parameters for that concrete call. This syntactic role is similar to `[]` for arrays: in declaration and definition they contain the size of the corresponding dimension. In a designation `A[i]` they are used to indicate the position of the accessed element in the array.

All the functions that we have seen so far have a *prototype*<sup>C</sup>, i.e. their declaration and definition included a parameter type-list and a return type. There are two special conventions that use the keyword **void**:

- If the function is to be called with no parameter, the list is replaced by the keyword **void**.
- If the function doesn’t return a value, the return type is given as **void**.

Such a prototype helps the compiler in places where the function is to be called. It only has to know about the parameters that the function expects. Have a look at the following:

```

1  extern double fbar(double x);
2
3  ...
4  double fbar2 = fbar(2)/2;
```

Here the call `fbar(2)` is not directly compatible with the expectation of function `fbar`: it wants a **double** but receives a **signed int**. But since the calling code knows this, it can convert the **signed int** argument 2 to the **double** value 2.0 before calling the function. The same holds for the return: the caller knows that the return is a **double**, so floating point division is applied for the result expression.

In C, there are ways to declare functions without prototype, but you will not see them here. You shouldn't use them, they should be retired. There were even ways in previous versions of C that allowed to use functions without any knowledge about them at all. Don't even think of using functions that way, nowadays:

**Rule 1.7.1.1** *All functions must have prototypes.*

A notable exception from that rule are functions that can receive a varying number of parameters, such as `printf`. This uses a mechanism for parameter handling that is called *variable argument list*<sup>C</sup> that comes with the header `stdarg.h`.

`#include <stdarg.h>`

We will see later (17.4.2) how this works, but this feature is to be avoided in any case. Already from your experience with `printf` you can imagine why such an interface poses difficulties. You as a programmer of the calling code have to ensure consistency by providing the correct `"%XX"` format specifiers.

In the implementation of a function we must watch that we provide return values for all functions that have a non-**void** return type. There can be several **return** statements in a function,

**Rule 1.7.1.2** *Functions only have one entry but several **return**.*

but all must be consistent with the function declaration. For a function that expects a return value, all **return** statements must contain an expression; functions that expect none, mustn't contain expressions.

**Rule 1.7.1.3** *A function **return** must be consistent with its type.*

But the same rule as for the parameters on the calling side holds for the return value. A value with a type that can be converted to the expected return type will be converted before the return happens.

If the type of the function is **void** the **return** (without expression) can even be omitted:

**Rule 1.7.1.4** *Reaching the end of the `{ }` block of a function is equivalent to a **return** statement without expression.*

This implies

**Rule 1.7.1.5** *Reaching the end of the `{ }` block of a function is only allowed for **void** functions.*

**7.2. `main` is special.** Perhaps you already have noted some particularities about `main`. It has a very special role as the entry point into your program: its prototype is enforced by the C standard, but it is implemented by the programmer. Being such a pivot between the runtime system and the application, it has to obey some special rules.

First, to suit different needs it has several prototypes, one of which must be implemented. Two should always be possible:

```
1 int main(void);
2 int main(int argc, char* argv[argc+1]);
```

Then, any specific C platform may provide other interfaces. There are two variations that are relatively common:

- On some embedded platforms where **main** is not expected to return to the runtime system the return type may be **void**.
- On many platforms a third parameter can give access to the “environment”.

You should better not rely on the existence of such other forms. If you want to write portable code (which you do) stick to the two “official” forms. For these the return value of **int** gives an indication to the runtime system if the execution was successful: values of **EXIT\_SUCCESS** or **EXIT\_FAILURE** indicate success or failure of the execution from the programmers point of view. These are the only two values that are guaranteed to work on all platforms.

**Rule 1.7.2.1** Use **EXIT\_SUCCESS** or **EXIT\_FAILURE** as return values of **main**.

For **main** as an exception of Rule 1.7.2.2

**Rule 1.7.2.2** Reaching the end of the `{ }` block of **main** is equivalent to a **return EXIT\_SUCCESS**;

The library function **exit** holds a special relationship with **main**. As the name indicates, a call to **exit** terminates the program; the prototype is

```
1 _Noreturn void exit(int status);
```

In fact, this functions terminates the program exactly as a **return** from **main** would. The parameter `status` has the role that the return expression in **main** would have.

**Rule 1.7.2.3** Calling **exit** (*s*) is equivalent evaluation of **return** *s* in **main**.

We also see that the prototype of **exit** is special because it has a **void** type. Just as the **return** statement in **main**:

**Rule 1.7.2.4** **exit** never fails and never returns to its caller.

The later is indicated by the special keyword **\_Noreturn**. This keyword should only be used for such special functions. There is even a pretty printed version of it, the macro **#include <stdnoreturn.h>** **\_\_noreturn**, that comes with the header `stdnoreturn.h`.

There is another feature in the second prototype of **main**, namely `argv`, the vector of commandline arguments. We already have seen some examples where we used this vector to communicate some values from the commandline to the program. E.g. in Listing 1.1 these commandline arguments were interpreted as **double** data for the program.

Strictly spoken, each of the `argv[i]` for  $i = 0, \dots, argc$  is a pointer, but since we don't know yet what that is, as an easy first approximation we can see them as strings:

**Rule 1.7.2.5** All commandline arguments are transferred as strings.

It is up to us to interpret them. In the example we chose the function **strtod** to decode a double value that was stored in the string.

Of the strings of `argv` two elements hold special values:

**Rule 1.7.2.6** Of the arguments to **main**, `argv[0]` holds the name of the program invocation.

There is no strict rule of what that “program name” should be, but usually this is just taken as the name of the program executable.

**Rule 1.7.2.7** Of the arguments to **main**, `argv[argc]` is 0.

In the `argv` array, the last argument could always be identified by that property, but this feature isn't too useful: we have `argc` to process that array.



**7.3. Recursion.** An important feature of functions is encapsulation: local variables are only visible and alive until we leave the function, either via an explicit **return** or because execution falls out of the last enclosing brace of the function's block. Their identifiers ("names") don't conflict with other similar identifiers in other functions and once we leave the function all the mess that we leave behind is cleaned up.

Even better: whenever we call a function, even one that we have called before, a new set of local variables (including function parameters) is created and these are newly initialized. This holds also, if we newly call a function for which another call is still active in the hierarchy of calling functions. A function that directly or indirectly calls itself is called *recursive*, the concept itself is called *recursion*.

Recursive functions are crucial for understanding C functions: they demonstrate and use main features of the function call model and they are only fully functional with these features. As a first example we show an implementation of Euclid's algorithm to compute the *greatest common divisor*, gcd, of two numbers.

```

euclid.h
8  size_t gcd2(size_t a, size_t b) {
9      assert(a <= b);
10     if (!a) return b;
11     size_t rem = b % a;
12     return gcd2(rem, a);
13 }
```

As you can see, this function is short and seemingly nice. But to understand how it works we need to understand well how functions work, and how we transform mathematical statements into algorithms.

Given two integers  $a, b > 0$  the gcd is defined to be the greatest integer  $c > 0$  that divides both  $a$  and  $b$  or as formulas:

$$\text{gcd}(a, b) = \max\{c \in \mathbb{N} \mid c|a \text{ and } c|b\}$$

If we also assume that  $a < b$ , we can easily see that two *recursive* formula hold:

$$(2) \quad \text{gcd}(a, b) = \text{gcd}(a, b - a)$$

$$(3) \quad \text{gcd}(a, b) = \text{gcd}(a, b \% a)$$

That is the gcd doesn't change if we subtract the smaller one or if we replace the larger of the two by the modulus of the other. These formulas are used since antique Greek mathematics to compute the gcd. They are commonly attributed to Euclid (Εὐκλείδης, around 300 B.C) but may have been known even before him.

Our C function gcd2 above uses Equation (3). First (Line 9) it checks if a precondition for the execution of this function is satisfied, namely if the first argument is less or equal to the second. It does this by using the macro **assert** from `assert.h`. This would abort the program with some informative message in case the function would be called with arguments that don't satisfy that condition, we will see more explanations of **assert** in Section 8.6.

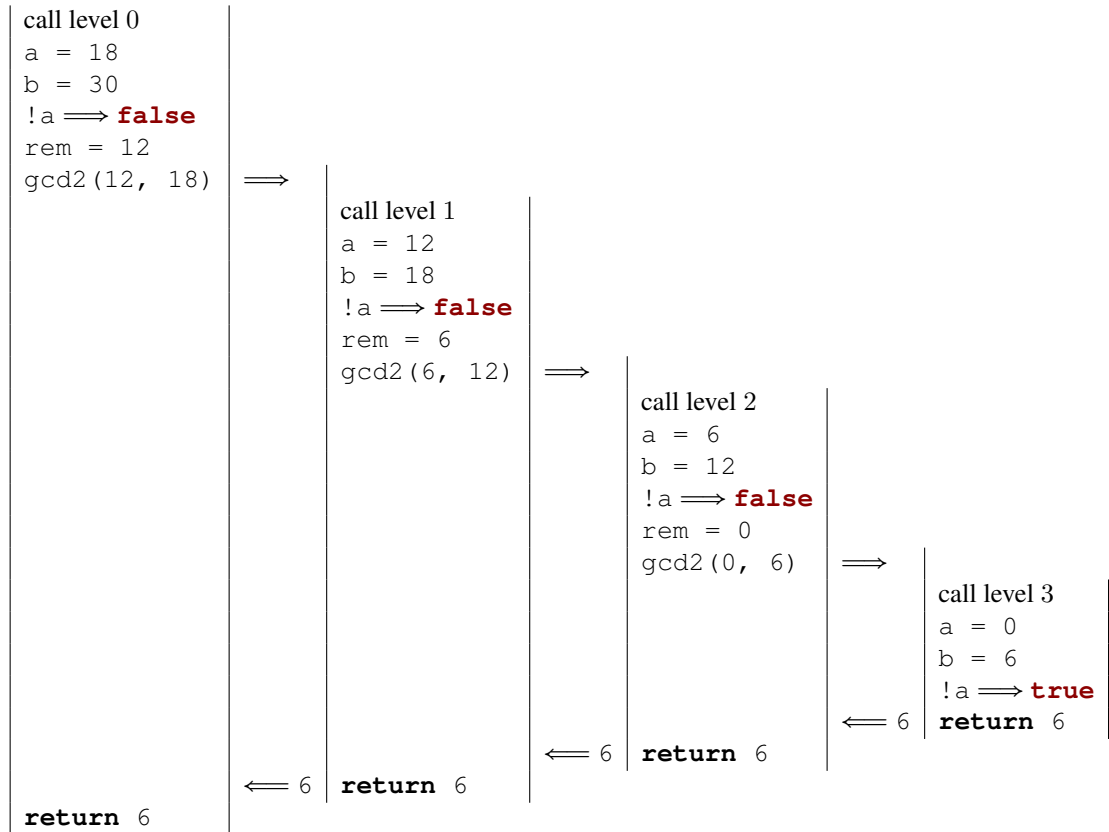
**#include <assert.h>**

**Rule 1.7.3.1** *Make all preconditions for a function explicit.*

Then, Line 10 checks if  $a$  is 0, in which case it returns  $b$ . This is an important step in a recursive algorithm:

**Rule 1.7.3.2** *In a recursive function, first check the termination condition.*

A missing termination check leads to *infinite recursion*; the function repeatedly calls new copies of itself until all system resources are exhausted and the program crashes. On

TABLE 10. Recursive call `gcd2(18, 30)`

modern systems with large amounts of memory this may take some time, during which the system will be completely unresponsive. You'd better not try it.

Otherwise, we compute the remainder `rem` of `b` modulo `a` (Line 11) and then the function is called recursively with `rem` and `a` and the return value of that, is directly returned.

Table 10 shows an example for the different recursive calls that are issued from an initial call `gcd2(18, 30)`. Here, the recursion goes 4 levels deep. Each level implements its own copies of the variables `a`, `b` and `rem`.

For each recursive call, Rule 1.4.1.7, guarantees that the precondition is always fulfilled automatically. For the initial call, we have to ensure this ourselves. This is best done by using a different function, a *wrapper*<sup>C</sup>:

euclid.h

```

15 size_t gcd(size_t a, size_t b) {
16     assert(a);
17     assert(b);
18     if (a < b)
19         return gcd2(a, b);
20     else
21         return gcd2(b, a);
22 }

```

**Rule 1.7.3.3** *Ensure the preconditions of a recursive function in a wrapper function.*

This avoids that the precondition has to be checked at each recursive call: the **assert** macro is such that it can be disabled in the final “production” object file.

Another famous example for a recursive definition of an integer sequence are *Fibonacci numbers*, a sequence of numbers that appears as early as 200 B.C in Indian texts. In modern terms the sequence can be defined as

$$\begin{aligned} (4) \quad & F_1 = 1 \\ (5) \quad & F_2 = 1 \\ (6) \quad & F_i = F_{i-1} + F_{i-2} \quad \text{for all } i > 2 \end{aligned}$$

The sequence of Fibonacci numbers is fast growing, its first elements are 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 377, 610, 987.

With the golden ratio

$$(7) \quad \varphi = \frac{1 + \sqrt{5}}{2} = 1.61803\dots$$

asymptotically we have

$$(8) \quad F_n = \frac{\varphi^n}{\sqrt{5}}$$

So the growth of  $F_n$  is exponential.

The recursive mathematical definition can be translated straightforward into a C function:

```

fibonacci.c
4  size_t fib(size_t n) {
5      if (n < 3)
6          return 1;
7      else
8          return fib(n-1) + fib(n-2);
9  }
```

Here, again, we first check for the termination condition, namely if the argument to the call,  $n$ , is less than 3. If it is the return value is 1, otherwise we return the sum of calls with argument values  $n-1$  and  $n-2$ .

Table 11 shows an example of a call to `fib` with a small argument value. We see that this leads to 3 levels of “stacked” calls to the same function with different arguments. Because Equation (6) uses two different values of the sequence, the scheme of the recursive calls is much more involved than the one for `gcd2`. In particular, there are 3 so-called *leaf calls*, calls to the function that fulfill the termination condition, and thus do by themselves not go into recursion. <sup>[Exs 34]</sup>

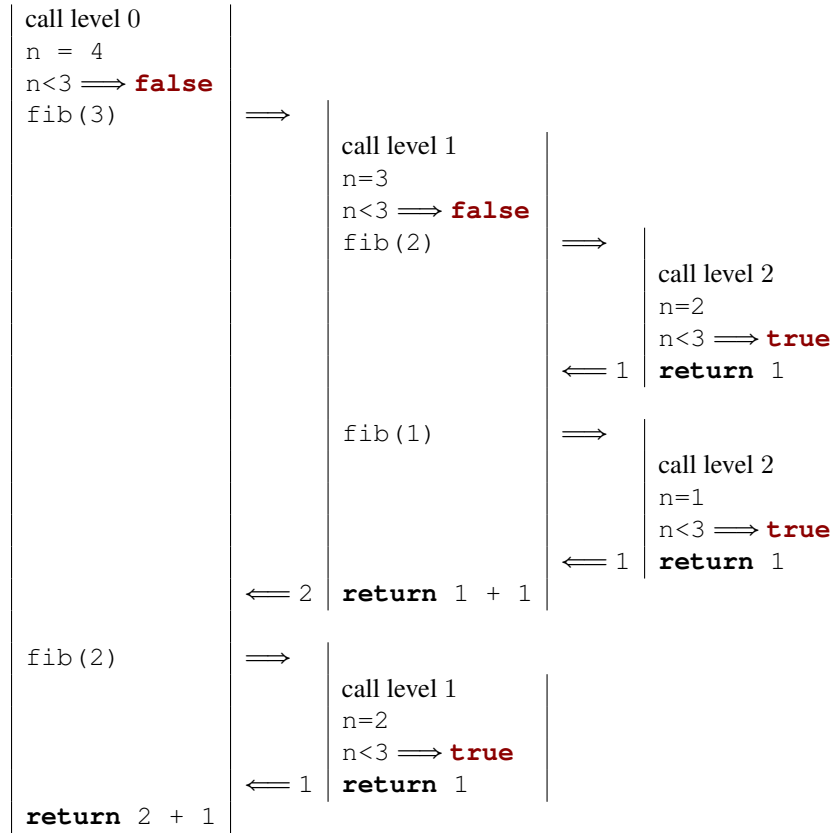
Implemented like that, the computation of the Fibonacci numbers is quite slow. <sup>[Exs 35]</sup> In fact it is easy to see that the recursive formula for the function itself also leads to an analogous formula for the execution time of the function:

$$\begin{aligned} (9) \quad & T_{\text{fib}(1)} = C_0 \\ (10) \quad & T_{\text{fib}(2)} = C_0 \\ (11) \quad & T_{\text{fib}(i)} = T_{\text{fib}(i-1)} + T_{\text{fib}(i-2)} + C_1 \quad \text{for all } i > 3 \end{aligned}$$

where  $C_0$  and  $C_1$  are some constants that depend on the platform.

[Exs 34] Show that a call `fib(n)` induces  $F_n$  leaf calls.

[Exs 35] Measure times for calls `fib(n)` with  $n$  set to different values. On POSIX systems you can use `/bin/time` to measure the run time of a program execution.

TABLE 11. Recursive call `fib(4)`

It follows that regardless of the platform and our cleverness of the implementation the execution time of the function will always be something like

$$(12) \quad T_{\text{fib}(i)} = F_i(C_0 + C_1) \approx \varphi^n \cdot \frac{C_0 + C_1}{\sqrt{5}} = \varphi^n \cdot C_2$$

with some other platform dependent constant  $C_2$ . So the execution time of `fib(n)` is exponential in  $n$ , which usually rules out such a function from being used in practice.

**Rule 1.7.3.4** *Multiple recursion may lead to exponential computation times.*

If we look at the nested calls in Table 11, we see that we have the same call `fib(2)`, twice, and thus all the effort to compute the value for `fib(2)` is repeated. The following function `fibCacheRec` avoids such repetitions. It receives an additional argument, `cache`, which is an array that holds all values that already have been computed.

```

fibonacciCache.c
1  .
2  /* Compute Fibonacci number n with help of a cache that may
3   * hold previously computed values. */
4  size_t fibCacheRec(size_t n, size_t cache[n]) {
5      if (!cache[n-1]) {
6          cache[n-1]
7              = fibCacheRec(n-1, cache) + fibCacheRec(n-2, cache);
8      }
9      return cache[n-1];
10 }
11
12 }
```

```

13
14 size_t fibCache(size_t n) {
15     if (n+1 <= 3) return 1;
16     /* Set up a VLA to cache the values. */
17     size_t cache[n];
18     /* A VLA must be initialized by assignment. */
19     cache[0] = 1; cache[1] = 1;
20     for (size_t i = 2; i < n; ++i)
21         cache[i] = 0;
22     /* Call the recursive function. */
23     return fibCacheRec(n, cache);
24 }

```

By trading storage against computation time, the recursive calls only are effected if the value has not yet been computed. By that, the call `fibCache(i)`, has an execution time that is linear in  $n$ , namely

$$(13) \quad T_{\text{fibCache}(n)} = n \cdot C_3$$

for some platform dependent parameter  $C_3$ .<sup>[Exs 36]</sup> Just by changing the algorithm that implements our sequence, we are able to reduce the execution time from exponential to linear! We didn't (and wouldn't) even discuss implementation details, nor did we perform concrete measurements of execution time: <sup>[Exs 37]</sup>

**Rule 1.7.3.5** *A bad algorithm will never lead to a performing implementation.*

**Rule 1.7.3.6** *Improving an algorithm can dramatically improve performance.*

For the fun of it, `fib2Rec` shows a third implemented algorithm for the Fibonacci sequence. It gets away with an FLA instead a VLA.

```

.
fibonacci2.c
4 void fib2rec(size_t n, size_t buf[2]) {
5     if (n > 2) {
6         size_t res = buf[0] + buf[1];
7         buf[1] = buf[0];
8         buf[0] = res;
9         fib2rec(n-1, buf);
10    }
11 }
12
13 size_t fib2(size_t n) {
14     size_t res[2] = { 1, 1, };
15     fib2rec(n, res);
16     return res[0];
17 }

```

Whether or not this is really faster, and how to prove that this version is still correct, is left as an exercise.<sup>[Exs 38] [Exs 39]</sup>

<sup>[Exs 36]</sup> Show Equation (13).

<sup>[Exs 37]</sup> Measure times for calls `fibCache(n)` with the same values as for `fib`.

<sup>[Exs 38]</sup> Measure times for calls `fib2(n)` with the same values as for `fib`.

<sup>[Exs 39]</sup> Use an iteration statement to transform `fib2rec` into a non-recursive function `fib2iter`.

## 8. C Library functions

The functionality that the C standard provides is separated into two big parts. One is the proper C language, the other is the *C library*. We already have seen several functions that come with the C library, e.g. `printf`, `puts` or `strtod`, so you should have a good idea what you may expect: basic tools that implement features that we need in every day's programming and for which we need clear interfaces and semantics to ensure portability.

On many platforms, the clear specification through an *application programmable interface*, *API*, also allows to separate the compiler implementation from the one of the library. E.g. on Linux systems we have the choice between different compilers, most commonly `gcc` and `clang`, and different C library implementations, e.g. the GNU C library (`glibc`), `dietlibc` or `musl`, and potentially any of the two choices can be used to produce an executable.

Roughly, library functions target one or two different purposes:

*Platform abstraction layer.* Functions that abstract from the specific properties and needs of the platform. These are functions that need platform specific bits to implement basic operations such as IO, that could not be implemented without deep knowledge of the platform. E.g. `puts` has to have some concept of a “terminal output” and how to address that. Implementing these functionalities by herself would exceed the knowledge of most C programmers, because it needs operating system or even processor specific magic. Be glad that some people did that job for you.

*Basic tools.* Functions that implement a task (such as e.g. `strtod`) that often occurs in programming in C for which it is important that the interface is fixed. These should be implemented relatively efficient, because they are used a lot, and they should be well tested and bug free such that we can rely safely on them. Implementing such functions should in principle be possible by any confirmed C programmer.<sup>[Exs 40]</sup>

A function as `printf` can be seen to target both purposes, it can effectively be separated in two, a formatting phase and an output phase. There is a function `snprintf` (explained much later in Section 14.1) that provides the same formatting functionalities as `printf` but stores the result in a string. This string could then be printed with `puts` to have the same output as `printf` as a whole.

The C library has a lot of functions, far more than we can handle in this book. Here on this level, we will discuss those functions that are necessary for a basic programming with the elements of the language that we have seen so far. We will complete this on higher levels, as soon as we discuss a particular concept. Table 12 has an overview of the different standard header files.

*Interfaces.* Most interfaces of the C library are specified as functions, but implementations are free to choose to implement them as macros, were this is appropriate. Compared to those that we have seen in Section 5.4.3 this uses a second form of macros that are syntactically similar to functions, *functionlike macros*<sup>C</sup>.

```
1 #define putchar(A) putc(A, stdout)
```

As before, these are just textual replacements, and since the replacement text may contain a macro argument several times, it would be quite bad to pass any expression with side effects to such a macro-or-function. Fortunately, because of Rule 1.4.2.2 you don't do that, anyhow.

Some of the interfaces that we will see below will have arguments or return values that are pointers. We can't handle these completely, yet, but in most cases we can get away by passing in some “known” pointers or 0 for pointer arguments. Pointers as return values will only occur in situations where they can be interpreted as an error condition.

<sup>[Exs 40]</sup> Write a function `my_strtod` that implements the functionality of `strtod` for decimal floating point constants.

TABLE 12. C library headers

<assert.h>	assert run time conditions	8.6
<complex.h>	complex numbers	5.5.7
<ctype.h>	character classification and conversion	8.3
<errno.h>	error codes	15
<fenv.h>	floating-point environment.	
<float.h>	properties of floating point types	5.5
<inttypes.h>	format conversion of integer types	5.5.6
<iso646.h>	alternative spellings for operators	4.1
<limits.h>	properties of integer types	5.0.3
<locale.h>	internationalization	8.5
<math.h>	type specific mathematical functions	8.1
<setjmp.h>	non-local jumps	18.4
<signal.h>	signal handling functions	18.5
<stdalign.h>	alignment of objects	12.7
<stdarg.h>	functions with varying number of arguments	17.4.2
<stdatomic.h>	atomic operations	18.5
<stdbool.h>	Booleans	3.1
<stddef.h>	basic types and macros	5.1
<stdint.h>	exact width integer types	5.5.6
<stdio.h>	input and output	8.2
<stdlib.h>	basic functions	2
<stdnoreturn.h>	non-returning functions	7
<string.h>	string handling	8.3
<tgmath.h>	type generic mathematical functions	8.1
<threads.h>	threads and control structures	19
<time.h>	time handling	8.4
<uchar.h>	Unicode characters	14.3
<wchar.h>	wide string	14.3
<wctype.h>	wide character classification and conversion	14.3

*Error checking.* C library functions usually indicate failure through a special return value. What value indicates the failure can be different and depends on the function itself. Generally you'd have to look up the specific convention in the manual page for the functions. Table 13 gives an rough overview of the different possibilities. There are three categories that apply: a special value that indicates an error, a special value that indicates success, and functions that return some sort of positive counter on success and a negative value on failure.

A typical error checking code in would look like the following

```

1 if (puts("hello_world") == EOF) {
2     perror("can't_output_to_terminal:");
3     exit(EXIT_FAILURE);
4 }
```

Here we see that `puts` falls into the category of functions that return a special value on error, `EOF`, “end-of-file”. The function `perror` from `stdio.h` is then used provide an additional diagnostic that depends on the specific error; `exit` ends the program execution. Don't wipe failures under the carpet, in programming

`#include <stdio.h>`

**Rule 1.8.0.1** *Failure is always an option.*

**Rule 1.8.0.2** *Check the return value of library functions for errors.*

TABLE 13. Error return strategies for C library functions. Some functions may also indicate a specific error condition through the value of the macro `errno`.

failure return	test	typical case	example
0	<code>!value</code>	other values are valid	<code>fopen</code>
special error code	<code>value == code</code>	other values are valid	<code>puts</code> , <code>clock</code> , <code>mktime</code> , <code>strtod</code> , <code>fclose</code>
non-zero value	<code>value</code>	value otherwise unneeded	<code>fgetpos</code> , <code>fsetpos</code>
special success code	<code>value != code</code>	case distinction for failure condition	<code>thr_create</code>
negative value	<code>value &lt; 0</code>	positive value is a “counter”	<code>printf</code>

An immediate failure of the program is often the best way to ensure that bugs are detected and get fixed in early development.

**Rule 1.8.0.3** *Fail fast, fail early and fail often.*

C has one major state “variable” that tracks errors of C library functions, a dinosaur called `errno`. The function `perror` uses this state under the hood, to provide its diagnostic. If a function fails in a way that allows us to recover, we have to ensure that the error state also is reset, otherwise library functions or error checking might get confused.

```

1 void puts_safe(char const s[static 1]) {
2     static bool failed = false;
3     if (!failed && puts(s) == EOF) {
4         perror("can't output to terminal:");
5         failed = true;
6         errno = 0;
7     }
8 }
```

**Bounds-checking interfaces.** Many of the functions in the C library are vulnerable to *buffer overflow*<sup>C</sup> if they are called with an inconsistent set of parameters. This led (and still leads) to a lot of security bugs and exploits and is generally something that should be handled very carefully.

C11 addressed this sort of problems by deprecating or removing some functions from the standard and by adding an optional series of new interfaces that check consistency of the parameters at runtime. These are the so-called bounds-checking interfaces of *Annex K* of the standard. Other than for most other features, this doesn't come with its own header file but adds interfaces to others. Two macros regulate access to these interfaces, `__STDC_LIB_EXT1__` tells if this optional interfaces is supported, and `__STDC_WANT_LIB_EXT1__` switches it on. The later must be set **before** any header files are included:

```

1 #if !__STDC_LIB_EXT1__
2 # error "This code needs bounds checking interface Annex_K"
3 #endif
4 #define __STDC_WANT_LIB_EXT1__ 1
5
6 #include <stdio.h>
```



```

7  /* Use printf_s from here on. */
8

```

Annex K

Optional features such as these are marked as this paragraph, here. The bounds-checking functions usually use the suffix `_s` to the name of the library function they replace, such as `printf_s` for `printf`. So you should not use that suffix for code of your own.

**Rule 1.8.0.4** *Identifier names terminating with `_s` are reserved.*

If such a function encounters an inconsistency, a *runtime constraint violation*<sup>C</sup>, it usually should end program execution after printing a diagnostic.

*Platform preconditions.* An important goal by programming with a standardized language such as C is portability. We should make as few assumptions about the execution platform as possible and leave it to the C compiler and library to fill out the gaps. Unfortunately this is not always possible, but if not, we should mark preconditions to our code as pronounced as possible.

**Rule 1.8.0.5** *Missed preconditions for the execution platform must abort compilation.*

The classical tool to achieve this are so-called *preprocessor conditionals*<sup>C</sup> as we have seen them above:

```

1  #if !__STDC_LIB_EXT1__
2  # error "This_code_needs_bounds_checking_interface_Annex_K"
3  #endif

```

As you can see, such a conditional starts with the token sequence `# if` on a line and terminates with another line containing the sequence `# endif`. The `# error` directive in the middle is only executed if the condition (here `!__STDC_LIB_EXT1__`) is true. It aborts the compilation process with an error message. The conditions that we can place in such a construct are limited.<sup>[Exs 41]</sup>

**Rule 1.8.0.6** *Only evaluate macros and integer literals in a preprocessor condition.*

As an extra feature in these conditions we have that identifiers that are unknown just evaluate to 0. So in the above example the expression is always valid, even if `__STDC_LIB_EXT1__` is completely unknown at that point.

**Rule 1.8.0.7** *In preprocessor conditions unknown identifiers evaluate to 0.*

If we want to test a more sophisticated condition we have `_Static_assert` and `static_assert` from the header `assert.h` with a similar effect at our disposal.

```
#include <assert.h>
```

```

1  #include <assert.h>
2  static_assert(sizeof(double) == sizeof(long double),
3  "Extra_precision_needed_for_convergence.");

```

**8.1. Mathematics.** Mathematical *functions* come with the `math.h` header, but it is much simpler to use the type generic macros that come with `tgmath.h`. Basically for all functions this has a macro that dispatches an invocation such as `sin(x)` or `pow(x, n)` to the function that inspects the type of `x` of its argument and for which the return value then is of that same type.

The type generic macros that are defined are `acos`, `acosh`, `asin`, `asinh`, `atan`, `atan2`, `atanh`, `carg`, `cbrt`, `ceil`, `cimag`, `conj`, `copysign`, `cos`, `cosh`, `cproj`, `creal`, `erf`, `erfc`, `exp`, `exp2`, `expm1`, `fabs`, `fdim`, `floor`, `fma`, `fmax`, `fmin`, `fmod`, `frexp`, `hypot`, `ilogb`, `ldexp`, `lgamma`, `llrint`, `llround`, `log`, `log10`, `log1p`, `log2`, `logb`, `lrint`, `lround`, `nearbyint`, `nextafter`, `nexttoward`, `pow`, `remainder`, `remquo`, `rint`, `round`, `scalbln`, `scalbn`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`, `tgamma`, `trunc`, far too many as we can describe them in detail, here. Table 14 gives an overview over the functions that are provided.

Nowadays, the implementations of numerical functions should be of high quality, efficient and with well controlled numerical precision. Although any of these functions could be implemented by a programmer with sufficient numerical knowledge, you should not try to replace or circumvent these functions. Many of them are not just implemented as C functions but can use processor specific instructions. E.g. processors may have fast approximations of `sqrt` or `sin` functions, or implement a *floating point multiply add*, `fma`, in one low level instruction. In particular, there are good chances that such low level instructions are used for all functions that inspect or modify floating point internals, such as `carg`, `creal`, `fabs`, `frexp`, `ldexp`, `llround`, `lround`, `nearbyint`, `rint`, `round`, `scalbn`, `trunc`. So replacing them or re-implementing them by handcrafted code is usually a bad idea.

**8.2. Input, output and file manipulation.** We have already seen some of the IO functions that come with the header file `stdio.h`, namely `puts` and `printf`. Where the second lets you format output in some convenient fashion, the first is more basic, it just outputs a string (its argument) and an end-of-line character.

**8.2.1. Unformatted output of text.** There is an even more basic function than that, namely `putchar`, that just outputs one single character. The interfaces of this two later functions are as follows:

```
1 int putchar(int c);
2 int puts(char const s[static 1]);
```

The type `int` as parameter for `putchar` is just a historical accident that shouldn't hurt you much. With this functions we could actually reimplement `puts` ourselves:

```
1 int puts_manually(char const s[static 1]) {
2     for (size_t i = 0; s[i]; ++i) {
3         if (putchar(s[i]) == EOF) return EOF;
4     }
5     if (putchar('\n') == EOF) return EOF;
6     return 0;
7 }
```

Just take this as an example, this is most probably less efficient than the `puts` that your platform provides.

Up to now we have only seen how we can output to the terminal. Often you'll want to write results to some permanent storage, the type `FILE*` for *streams*<sup>C</sup> provides an abstraction for this. There are two functions, `fputs` and `fputc`, that generalize the idea of unformatted output to streams.

[Exs 41] Write a preprocessor condition that tests if `int` has two's complement sign representation.

```
#include <math.h>
#include <tgmath.h>
```

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



TABLE 14. Mathematical functions. Type generic macros are printed in red, real functions in green.

<b>abs, labs, llabs</b>	$ x $ for integers
<b>acosh</b>	hyperbolic arc cosine
<b>acos</b>	arc cosine
<b>asinh</b>	hyperbolic arc sine
<b>asin</b>	arc sine
<b>atan2</b>	arc tangent, two arguments
<b>atanh</b>	hyperbolic arc tangent
<b>atan</b>	arc tangent
<b>cbrt</b>	$\sqrt[3]{x}$
<b>ceil</b>	$\lceil x \rceil$
<b>copysign</b>	copy the sign from $y$ to $x$
<b>cosh</b>	hyperbolic cosine
<b>cos</b>	cosine function, $\cos x$
<b>div, ldiv, lldiv</b>	quotient and remainder of integer division
<b>erfc</b>	complementary error function, $1 - \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$
<b>erf</b>	error function, $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$
<b>exp2</b>	$2^x$
<b>expm1</b>	$e^x - 1$
<b>exp</b>	$e^x$
<b>fabs</b>	$ x $ for floating point
<b>fdim</b>	positive difference
<b>floor</b>	$\lfloor x \rfloor$
<b>fmax</b>	floating point maximum
<b>fma</b>	$x \cdot y + z$
<b>fmin</b>	floating point minimum
<b>fmod</b>	remainder of the floating point division
<b>fpclassify</b>	classify floating point value
<b>frexp</b>	significand and exponent
<b>hypot</b>	$\sqrt{x^2 + y^2}$
<b>ilogb</b>	$\lfloor \log_{\text{FLT\_RADIX}} x \rfloor$ as integer
<b>isfinite</b>	check if finite
<b>isinf</b>	check if infinite
<b>isnan</b>	check if NaN
<b>isnormal</b>	checks if normal
<b>ldexp</b>	$x \cdot 2^y$
<b>lgamma</b>	$\log_e \Gamma(x)$
<b>log10</b>	$\log_{10} x$
<b>log1p</b>	$\log_e 1 + x$
<b>log2</b>	$\log_2 x$
<b>logb</b>	$\lfloor \log_{\text{FLT\_RADIX}} x \rfloor$ as floating point
<b>log</b>	$\log_e x$
<b>modf, modff, modfl</b>	integer and fractional parts
<b>nan, nanf, nanl</b>	not-a-number, NaN, of corresponding type
<b>nearbyint</b>	nearest integer using current rounding mode
<b>nextafter, nexttoward</b>	next representable floating point value
<b>pow</b>	$x^y$
<b>remainder</b>	signed remainder of division
<b>remquo</b>	signed remainder and the last bits of the division
<b>rint, lrint, llrint</b>	nearest integer using current rounding mode
<b>round, lround, llround</b>	$\text{sign}(x) \cdot \lfloor  x  + 0.5 \rfloor$
<b>scalbn, scalbln</b>	$x \cdot \text{FLT\_RADIX}^y$
<b>signbit</b>	checks if negative
<b>sinh</b>	hyperbolic sine
<b>sin</b>	sine function, $\sin x$
<b>sqrt</b>	$\sqrt{x}$
<b>tanh</b>	hyperbolic tangent
<b>tan</b>	tangent function, $\tan x$
<b>tgamma</b>	gamma function, $\Gamma(x)$
<b>trunc</b>	$\text{sign}(x) \cdot \lfloor  x  \rfloor$

```

1 int fputc(int c, FILE* stream);
2 int fputs(char const s[static 1], FILE* stream);

```

Here the `*` in the `FILE*` type again indicates that this is a pointer type, so we couldn't go into details. The only thing that we need for now is Rule 1.6.2.4; a pointer can be tested if it is null, and so we will be able to test if a stream is valid or not.

More than the fact that this is a pointer, the identifier `FILE` itself is a so-called *opaque type*<sup>C</sup>, for which we don't know more than is provided by the functional interfaces that we will see in this section. The facts that it is implemented as a macro and the misuse of the name "FILE" for a "stream" is a reminder that this is one of the historic interfaces that predate standardization.

**Rule 1.8.2.1** *Opaque types are specified through functional interfaces.*

**Rule 1.8.2.2** *Don't rely on implementation details of opaque types.*

If we don't do anything special, there are two streams that are available for output to us: `stdout` and `stderr`. We already have used `stdout` implicitly, this is what `putchar` and `puts` use under the hood, and this stream is usually connected to the terminal. `stderr` is similar, it also is linked to the terminal by default, with perhaps slightly different properties. In any case these two are closely related. The purpose to have two of them is to be able to distinguish "usual" output (`stdout`) from "urgent" one (`stderr`).

We can rewrite the former functions in terms of the more general ones:

```

1 int putchar_manually(int c) {
2     return fputc(c, stdout);
3 }
4 int puts_manually(char const s[static 1]) {
5     if (fputs(s[i], stdout) == EOF) return EOF;
6     if (fputc('\n', stdout) == EOF) return EOF;
7     return 0;
8 }

```

Observe that `fputs` differs from `puts` in that it doesn't append an end-of-line character to the string.

**Rule 1.8.2.3** *`puts` and `fputs` differ in their end of line handling.*

**8.2.2. Files and streams.** If we want to write some output to real files, we have to attach the files to our program execution by means of the function `fopen`.

```

1 FILE* fopen(char const path[static 1], char const mode[static 1])
2     ;
3 FILE* freopen(char const path[static 1], char const mode[static
4     1], FILE *stream);

```

This can be used as simple as here:

```

1 int main(int argc, char* argv[argc+1]) {
2     FILE* logfile = fopen("mylog.txt", "a");
3     if (!logfile) {
4         perror("fopen_failed");
5         return EXIT_FAILURE;
6     }
7     fputs("feeling_fine_today\n", logfile);
8     return EXIT_SUCCESS;
9 }

```



TABLE 15. Modes and modifiers for **fopen** and **freopen**. One of the first three must start the mode string, optionally followed by some of the other three. See Table 16 for all valid combinations.

mode	memo		file status after <b>fopen</b>
'a'	append	w	file unmodified, position at end
'w'	write	w	content of file wiped out, if any
'r'	read	r	file unmodified, position at start
modifier	memo		additional property
'+'	update	rw	open file for reading and writing
'b'	binary		view as binary file, otherwise text file
'x'	exclusive		create file for writing iff it does not yet exist

This *opens a file*<sup>C</sup> called "mylog.txt" in the file system and provides access to it through the variable logfile. The mode argument "a" has the file opened for appending, that is the contents of the file is preserved, if it exists, and writing begins at the current end of that file.

There are multiple reasons why opening a file might not succeed, *e.g* the file system might be full or the process might not have the permission to write at the indicated place. In accordance with Rule 1.8.0.2 we check for such conditions and exit the program if such a condition is present.

As we have seen above, the function **perror** is used to give a diagnostic of the error that occurred. It is equivalent to something like the following.

```
1  fputs("fopen_failed:_some-diagnostic\n", stderr);
```

This "some-diagnostic" might (but does not have to) contain more information that helps the user of the program to deal with the error.

Annex K

There are also bounds-checking replacements **fopen\_s** and **freopen\_s** that ensure that the arguments that are passed are valid pointers. Here **errno\_t** is a type that comes with **stdlib.h** and that encodes error returns. The **restrict** keyword that also newly appears only applies to pointer types and is out of our scope for the moment.

```
1  errno_t fopen_s(FILE* restrict streamptr[restrict],
2      char const filename[restrict], char const mode[
        restrict]);
3  errno_t freopen_s(FILE* restrict newstreamptr[restrict],
4      char const filename[restrict], char const mode[
        restrict]
5      FILE* restrict stream);
```

There are different modes to open a file, "a" is only one of several possibilities. Table 15 contains an overview of the different characters that may appear in that string. We have three different base modes that regulate what happens to a pre-existing file, if any, and where the stream is positioned. In addition, we have three modifiers that can be appended to these. Table 16 has a complete list of the possible combinations.

From these tables you see that a stream can not only be opened for writing but also for reading; we will see below how that can be done. To know which of the base modes opens for reading or writing just use your common sense. For 'a' and 'w' a file that is positioned at its end can't be read since there is nothing there, thus these open for writing.

TABLE 16. Mode strings for **fopen** and **freopen**. These are the valid combinations of the characters in Table 15.

"a"		create empty text file if necessary, open for writing at end-of-file
"w"		create empty text file or wipe out content, open for writing
"r"		open existing text file for reading
"a+"		create empty text file if necessary, open for reading and writing at end-of-file
"w+"		create empty text file or wipe out content, open for reading and writing
"r+"		open existing text file for reading and writing at beginning of file
"ab"	"rb"	same as above but for binary file instead of text file
"wb"	"a+b"	
"ab+"	"r+b"	
"rb+"	"w+b"	
"wb+"		
"wx"	"w+x"	same as above but error if file exists prior to call
"wbx"	"w+bx"	
"wb+x"		

For '**r**' a file contents that is preserved and that is positioned at the beginning should not be overwritten accidentally, so this is for reading.

The modifiers are used less commonly in everyday's coding. "Update" mode with '**+**' should be used carefully. Reading and writing at the same time is not so easy and needs some special care. For '**b**' we will discuss the difference between text and binary streams in some more detail in Section 14.4.

There are three other principal interfaces to handle streams, **freopen**, **fclose** and **fflush**.

```
1 int fclose(FILE* fp);
2 int fflush(FILE* stream);
```

The primary uses for **freopen** and **fclose** are straightforward: **freopen** can associate a given stream to a different file and eventually change the mode. This is particular useful to associate the standard streams to a file. *E.g* our little program from above could be rewritten as

```
1 int main(int argc, char* argv[argc+1]) {
2   if (!freopen("mylog.txt", "a", stdout)) {
3     perror("freopen_failed");
4     return EXIT_FAILURE;
5   }
6   puts("feeling_fine_today");
7   return EXIT_SUCCESS;
8 }
```

8.2.3. *Text IO*. Output to text streams is usually *buffered*<sup>C</sup>, that is to make more efficient use of its resources the IO system can delay the physical write of to a stream for some time. If we close the stream with **fclose** all buffers are guaranteed to be *flushed*<sup>C</sup> to where it is supposed to go. The function **fflush** is needed in places where we want to see an output immediately on the terminal or where don't want to close the file, yet, but where we want to ensure that all contents that we have written has properly reached its

LISTING 1.4. flushing buffered output

```

1  #include <stdio.h>
2
3  /* delay execution with some crude code,
4     should use thrd_sleep, once we have that */
5  void delay(double secs) {
6      double const magic = 4E8; // works just on my machine
7      unsigned long long const nano = secs * magic;
8      for (unsigned long volatile count = 0;
9           count < nano;
10          ++count) {
11          /* nothing here */
12      }
13  }
14
15  int main(int argc, char* argv[argc+1]) {
16      fputs("waiting_10_seconds_for_you_to_stop_me", stdout);
17      if (argc < 3) fflush(stdout);
18      for (unsigned i = 0; i < 10; ++i) {
19          fputc('.', stdout);
20          if (argc < 2) fflush(stdout);
21          delay(1.0);
22      }
23      fputs("\n", stdout);
24      fputs("You_did_ignore_me,_so_bye_bye\n", stdout);
25  }

```

destination. Listing 1.4 shows an example that writes 10 dots to `stdout` with a delay of approximately one second between all writes.<sup>[Exs 42]</sup>

The most common form of IO buffering for text files in *line buffering*<sup>C</sup>. In that mode, output is only physically written if the end of a text line is encountered. So usually text that is written with `puts` would appear immediately on the terminal, `fputs` would wait until it meets an `'\n'` in the output. Another interesting thing about text streams and files is that there is no one-to-one correspondence between characters that are written in the program and bytes that land on the console device or in the file.

**Rule 1.8.2.4** *Text input and output converts data.*

This is because internal and external representation of text characters are not necessarily the same. Unfortunately there are still many different character encodings around, the C library is in charge of doing the conversions correctly, if it may. Most notoriously the end-of-line encoding in files is platform depending:

**Rule 1.8.2.5** *There are three commonly used conversion to encode end-of-line.*

C here gives us a very suitable abstraction in using `'\n'` for this, regardless of the platform. Another modification that you should be aware of when doing text IO is that white space that precedes the end of line may be suppressed. Therefore presence of such *trailing white space*<sup>C</sup> such as blank or tabulator characters can not be relied upon and should be avoided:

**Rule 1.8.2.6** *Text lines should not contain trailing white space.*

[Exs 42] Observe the behavior of the program by running it with 0, 1 and 2 command line arguments.

TABLE 17. Format specifications for `printf` and similar functions, with the general syntax `"% [FF] [WW] [.PP] [LL] SS"`

FF	flags	special form of conversion
WW	field width	minimum width
PP	precision	
LL	modifier	select width of type
SS	specifier	select conversion

TABLE 18. Format specifiers for `printf` and similar functions

'd' or 'i'	decimal	signed integer
'u'	decimal	unsigned integer
'o'	octal	unsigned integer
'x' or 'X'	hexadecimal	unsigned integer
'e' or 'E'	[-]d.ddd e±dd, "scientific"	floating point
'f' or 'F'	[-]d.ddd	floating point
'g' or 'G'	generic e or f	floating point
'a' or 'A'	[-]0xh.hhhh p±d, hexadecimal	floating point
'%'	'%' character	no argument is converted
'c'	character	integer
's'	characters	string
'p'	address	<b>void*</b> pointer

The C library additionally also has very limited support to manipulate files within the file system:

```
1 int remove(char const pathname[static 1]);
2 int rename(char const oldpath[static 1], char const newpath[static 1]);
```

These basically do what their names indicate.

8.2.4. *Formatted output.* We have already seen how we can use `printf` for formatted output. The function `fprintf` is very similar to that, only that it has an additional parameter that allows to specify the stream to which the output is written:

```
1 int printf(char const format[static 1], ...);
2 int fprintf(FILE* stream, char const format[static 1], ...);
```

The syntax with the three dots `...` indicates that these functions may receive an arbitrary number of items that are to be printed. An important constraint is that this number must correspond exactly to the `'%'` specifiers, otherwise the behavior is undefined:

**Rule 1.8.2.7** *Parameters of `printf` must exactly correspond to the format specifiers.*

With the syntax `% [FF] [WW] [.PP] [LL] SS`, a complete format specification can be composed of 5 different parts, flags, width, precision, modifiers and specifier. See Table 17 for details.

The specifier is not optional and selects the type of output conversion that is performed. See Table 18 for an overview.

The modifier part is important to specify the exact type of the corresponding argument. Table 19 gives the codes for the types that we have encountered so far.

The flag can change the output variant, such as prefixing with signs (`"%+d"`), `0x` for hexadecimal conversion (`"%#X"`), `0` for octal (`"%#o"`), or padding with `0`. See Table 20.



TABLE 19. Format modifiers for `printf` and similar functions. `float` arguments are first converted to `double`.

character	type	conversion
"hh"	<b>char</b> types	integer
"h"	<b>short</b> types	integer
" "	<b>signed, unsigned</b>	integer
"l"	<b>long</b> integer types	integer
"ll"	<b>long long</b> integer types	integer
"j"	<b>intmax_t, uintmax_t</b>	integer
"z"	<b>size_t</b>	integer
"t"	<b>ptrdiff_t</b>	integer
"L"	<b>long double</b>	floating point

TABLE 20. Format flags for `printf` and similar functions.

character	meaning	conversion
"#"	alternate form, <i>e.g.</i> prefix 0x	"aAeEfFgGoxX"
"0"	zero padding	numeric
"-"	left adjustment	any
"_"	' ' for positive values '-' for negative	signed
"+"	'+' for positive values '-' for negative	signed

If we know that the numbers that we write will be read back in from a file, later, the forms `"%d"` for signed types, `"%X"` for unsigned types and `"%a"` for floating point are the most appropriate. They guarantee that the string to number conversions will detect the correct form and that the storage in file will be without loss of information.

**Rule 1.8.2.8** Use `"%d"`, `"%X"` and `"%a"` for conversions that have to be read, later.

Annex K

The optional interfaces `printf_s` and `fprintf_s` check that the stream, format and any string arguments are valid pointers. They **don't** check if the expressions in the list correspond to correct format specifiers.

```

1 int printf_s(char const format[restrict], ...);
2 int fprintf_s(FILE *restrict stream,
3               char const format[restrict], ...);

```

Here is a modified example for the re-opening of `stdout`.

```

1 int main(int argc, char* argv[argc+1]) {
2     int ret = EXIT_FAILURE;
3     fprintf_s(stderr, "freopen_of_%s:", argv[1]);
4     if (freopen(argv[1], "a", stdout)) {
5         ret = EXIT_SUCCESS;
6         puts("feeling_fine_today");
7     }
8     perror(0);
9     return ret;
10 }

```

This improves the diagnostic output by adding the file name to the output string. `fprintf_s` is used to check the validity of the stream, the format and the argument string. This function may mix the output to the two streams if they are both connected to the same terminal.

8.2.5. *Unformatted input of text.* Unformatted input is best done with `fgetc` for a single character and `fgets` for a string. There is one standard stream that is always defined that usually connects to terminal input: `stdin`.

```
1 int fgetc(FILE* stream);
2 char* fgets(char s[restrict], int n, FILE* restrict stream);
3 int getchar(void);
```

Annex K

In addition, there are also `getchar` and `gets_s` that read from `stdin` but they don't add much to the above interfaces that are more generic.

```
1 char* gets_s(char s[static 1], rsize_t n);
```

Historically, in the same spirit as `puts` specializes `fputs`, prior version of the C standard had a `gets` interface. This has been removed because it was inherently unsafe.

**Rule 1.8.2.9** *Don't use `gets`.*

The following listing shows a function that has an equivalent functionality as `fgets`.

LISTING 1.5. Implementing `fgets` in terms of `fgetc`

```
1 char* fgets_manually(char s[restrict], int n,
2 FILE* restrict stream) {
3     if (!stream) return 0;
4     if (!n) return s;
5     /* Read at most n-1 characters */
6     for (size_t pos = 0; pos < n-1; ++pos) {
7         int val = fgetc(stream);
8         switch (val) {
9             /* EOF signals end-of-file or error */
10            case EOF: if (feof(stream)) {
11                s[pos] = 0;
12                /* has been a valid call */
13                return s;
14            } else {
15                /* we are on error */
16                return 0;
17            }
18            /* stop at end-of-line */
19            case '\n': s[pos] = val; s[pos+1] = 0; return s;
20            /* otherwise just assign and continue */
21            default: s[pos] = val;
22        }
23    }
24    s[pos] = 0;
25    return s;
26 }
```

Again, such an example code is not meant to replace the function, but to illustrate properties of the functions in question, here the error handling strategy.

**Rule 1.8.2.10** *`fgetc` returns `int` to be capable to encode a special error status, `EOF`, in addition to all valid characters.*

Also, the detection of a return of `EOF` alone is not sufficient to conclude that the end of the stream has been reached. We have to call `feof` to test if a stream's position has reached its end-of-file marker.

**Rule 1.8.2.11** *End of file can only be detected after a failed read.*

Listing 1.6 presents an example that uses both, input and output functions.

LISTING 1.6. A program to concatenate text files

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <errno.h>
4
5  enum { buf_max = 32, };
6
7  int main(int argc, char* argv[argc+1]) {
8      int ret = EXIT_FAILURE;
9      char buffer[buf_max] = { 0 };
10     for (int i = 1; i < argc; ++i) {           // process args
11         FILE* instream = fopen(argv[i], "r"); // as file names
12         if (instream) {
13             while (fgetc(buffer, buf_max, instream)) {
14                 fputc(buffer, stdout);
15             }
16             fclose(instream);
17             ret = EXIT_SUCCESS;
18         } else {
19             /* Provide some error diagnostic. */
20             fprintf(stderr, "Could_not_open_%s:\n", argv[i]);
21             perror(0);
22             errno = 0;                          // reset error code
23         }
24     }
25     return ret;
26 }
```

Here, this presents a small implementation of `cat` that reads a number of files that are given on the command line, and dumps the contents to `stdout`.<sup>[Exs 43][Exs 44][Exs 45][Exs 46]</sup>

**8.3. String processing and conversion.** String processing in C has to deal with the fact that the source and execution environments may have different encodings. It is therefore crucial to have interfaces that work independent of the encoding. The most important tools are given by the language itself: integer character constants such as `'a'` or `'\n'` and string literals such as `"hello:\tx"` should always do the right thing on your platform. As you perhaps remember there are no constants for types that are narrower than `int` and as an historic artifact integer character constants such as `'a'` have type `int`, and not `char` as you would probably expect.

Handling such constants can become cumbersome if you have to deal with character classes.

[Exs 43] Under what circumstances this program finishes with success or failure return codes?

[Exs 44] Surprisingly this program even works for files with lines that have more than 31 characters. Why?

[Exs 45] Have the program read from `stdin` if no command line argument is given.

[Exs 46] Have the program precedes all output lines with line numbers if the first command line argument is `"-n"`.



TABLE 21. Character classifiers. The third column indicates if C implementations may extend these classes with platform specific characters, such as 'ä' as a lower case character or '€' as punctuation.

name	meaning	C locale	extended
<b>islower</b>	lower case	'a' ... 'z'	yes
<b>isupper</b>	upper case	'A' ... 'Z'	yes
<b>isblank</b>	blank	' ', '\t'	yes
<b>isspace</b>	space	' ', '\f', '\n', '\r', '\t', '\v'	yes
<b>isdigit</b>	decimal	'0' ... '9'	no
<b>isxdigit</b>	hexadecimal	'0' ... '9', 'a' ... 'f', 'A' ... 'F'	no
<b>iscntrl</b>	control	'\a', '\b', '\f', '\n', '\r', '\t', '\v'	yes
<b>isalnum</b>	alphanumeric	<b>isalpha</b> (x)    <b>isdigit</b> (x)	yes
<b>isalpha</b>	alphabet	<b>islower</b> (x)    <b>isupper</b> (x)	yes
<b>isgraph</b>	graphical	(! <b>iscntrl</b> (x)) && (x != ' ')	yes
<b>isprint</b>	printable	! <b>iscntrl</b> (x)	yes
<b>ispunct</b>	punctuation	<b>isprint</b> (x) && ! ( <b>isalnum</b> (x)    <b>isspace</b> (x))	yes

#include <ctype.h>

Therefore the C library provides functions or macros that deals with the most commonly used classes through the header `ctype.h`. It has classifiers **isalnum**, **isalpha**, **isblank**, **iscntrl**, **isdigit**, **isgraph**, **islower**, **isprint**, **ispunct**, **isspace**, **isupper**, **isxdigit**, and conversions **toupper** and **tolower**. Again, for historic reasons all of these take their arguments as **int** and also return **int**. See Table 21 for an overview over the classifiers. The functions **toupper** and **tolower** convert alphabetic characters to the corresponding case and leave all other characters as they are.

That table has some special characters such as '\n' for a new line character which we have encountered previously. All the special encodings and their meaning are given in Table 22. Integer character constants can also be encoded numerically, namely as octal

'\''	quote
'\"'	doublequote
'\?'	question mark
'\\'	backslash
'\a'	alert
'\b'	backspace
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab

TABLE 22. Special characters in character and string literals

value of the form '\037' or as hexadecimal value in the form '\xFFFF'. In the first form, up to three octal digits are used to represent the code. For the second, any sequence of characters after the x that can be interpreted as a hex digit is included in the code. Using these in strings needs special care to mark the end of such a character. "\xdeBruyn" is not the same as "\xde" "Bruyn"<sup>47</sup> but corresponds to "\xdeB" "ruyn", the character with code 3563 followed by the four characters 'r', 'u', 'y' and 'n'.

<sup>47</sup>But remember that consecutive string literals are concatenated, see Rule 1.5.2.1.

**Rule 1.8.3.1** *The interpretation of numerically encoded characters depends on the execution character set.*

So their use is not portable and should thus be avoided.

The following function `hexatridecimal` uses some of the functions from above to provide a numerical value for all alphanumerical characters base 36. This is analogous to hexadecimal constants, only that all other letters give a value in base 36, too. [Exs 48] [Exs 49] [Exs 50]

```

strtol.c
6  #include <string.h>
7
8  /* Supposes that lower case characters are contiguous. */
9  _Static_assert('z'-'a' == 25,
10               "alphabetic_characters_not_contiguous");
11 #include <ctype.h>
12 /* convert an alphanumeric digit to an unsigned */
13 /* '0' ... '9' => 0 .. 9u */
14 /* 'A' ... 'Z' => 10 .. 35u */
15 /* 'a' ... 'z' => 10 .. 35u */
16 /* other values => greater */
17 unsigned hexatridecimal(int a) {
18     if (isdigit(a)) {
19         /* This is guaranteed to work, decimal digits
20          * are consecutive and isdigit is not
21          * locale dependent */
22         return a - '0';
23     } else {
24         /* leaves a unchanged if it is not lower case */
25         a = toupper(a);
26         /* Returns value >= 36 if not latin upper case */
27         return (isupper(a)) ? 10 + (a - 'A') : -1;
28     }
29 }

```

Besides the function `strtod`, the C library has `strtoul`, `strtol`, `strtoumax`, `strtoimax`, `strtoull`, `strtolll`, `strtold`, and `strtolf` to convert a string to a numerical value. Here the characters at the end of the names correspond to the type, u for **unsigned**, l (the letter “el”) for **long**, d for **double**, f for **float**, and [i|u]max to **intmax\_t** and **uintmax\_t**.

The interfaces with an integral return type all have 3 parameters, such as e.g. `strtoul`

```

1  unsigned long int strtoul(char const nptr[restrict],
2                           char** restrict endptr,
3                           int base);

```

which interprets a string `nptr` as a number given in base `base`. Interesting values for base are 0, 8, 10 and 16. The later three correspond to octal, decimal and hexadecimal encoding, respectively. The first, 0, is a combination of these three, where the base is choosing according to the usually roles for the interpretation of text as numbers: “7” is decimal, “007” is octal and “0x7” is hexadecimal.

[Exs 48] The second `return` of `hexatridecimal` makes an assumption about the relation between `a` and `'A'`.

Which?

[Exs 49] Describe an error scenario in case that this assumption is not fulfilled.

[Exs 50] Fix this bug.

More precisely, the string is interpreted as potentially consisting of four different parts: white space, a sign, the number and some remaining data.

The second parameter can in fact be used to obtain the position of the remaining data, but this is still too involved for us. For the moment, it suffices to pass a 0 for that parameter to ensure that all works well. A convenient combination of parameters is often `strtoul(S, 0, 0)`, which will try to interpret *S* as representing a number, regardless of the input format.

The three functions that provide floating point values work similar, only that the number of function parameters is limited to two.

In the following we will demonstrate how the such functions can be implemented from more basic primitives. Let us first look into `Strtoul_inner`. It is the core of a `strtoul` implementation that uses hexadecimal in a loop to compute a large integer from a string.

```

                                                                    strtoul.c
31 unsigned long Strtoul_inner(char const s[static 1],
32                             size_t i,
33                             unsigned base) {
34     unsigned long ret = 0;
35     while (s[i]) {
36         unsigned c = hexadecimal(s[i]);
37         if (c >= base) break;
38         /* maximal representable value for 64 bit is
39            3w5e11264sgsf in base 36 */
40         if (ULONG_MAX/base < ret) {
41             ret = ULONG_MAX;
42             errno = ERANGE;
43             break;
44         }
45         ret *= base;
46         ret += c;
47         ++i;
48     }
49     return ret;
50 }

```

In case that the string represents a number that is too big for an `unsigned long`, this function returns `ULONG_MAX` and sets `errno` to `ERANGE`.

Now `Strtoul` gives a functional implementation of `strtoul`, as far as this can be done without pointers:

```

                                                                    strtoul.c
60 unsigned long Strtoul(char const s[static 1], unsigned base) {
61     if (base > 36u) { /* test if base */
62         errno = EINVAL; /* extends specification */
63         return ULONG_MAX;
64     }
65     size_t i = strspn(s, "\f\n\r\t\v"); /* skip spaces */
66     bool switchsign = false; /* look for a sign */
67     switch (s[i]) {
68     case '-': switchsign = true;
69     case '+': ++i;
70     }
71     if (!base || base == 16) { /* adjust the base */
72         size_t adj = find_prefix(s, i, "0x");
73         if (!base) base = (unsigned[]){ 10, 8, 16, }[adj];

```

```

74     i += adj;
75 }
76 /* now, start the real conversion */
77 unsigned long ret = Strtoul_inner(s, i, base);
78 return (switchsign) ? -ret : ret;
79 }

```

It wraps `strtoul_inner` and previously does the adjustments that are needed: it skips white space, looks for an optional sign, adjusts the base in case the `base` parameter was, 0, skips an eventual 0 or 0x prefix. Observe also that in case that a minus sign has been provided it does the correct negation of the result in terms of **unsigned long** arithmetic.<sup>[Exs 51]</sup>

To skip the spaces, `Strtoul` uses **strspn**, one of the string search functions that are provided by `string.h`. This function returns the length of the initial sequence in the first parameter that entirely consists of any character of the second parameter. The function **strcspn** (“c” for “complement”) works similarly, only that it looks for an initial sequence of characters **not** present in the second argument.

#include <string.h>

This header provides a lot more memory or string search functions: **memchr**, **strchr**, **strpbrk**, **strrchr**, **strstr**, and **strtok**. But to use them we would need pointers, so we can’t handle them, yet.

**8.4. Time.** The first class of “times” can be classified as calendar times, times with a granularity and range as it would typically appear in a human calendar, as for appointments, birthdays and so on. Here are some of the functional interfaces that deal with times and that are all provided by the `time.h` header:

#include <time.h>

```

1 time_t time(time_t *t);
2 double difftime(time_t time1, time_t time0);
3 time_t mktime(struct tm tm[1]);
4 size_t strftime(char s[static 1], size_t max,
5                 char const format[static 1],
6                 struct tm const tm[static 1]);
7 int timespec_get(struct timespec ts[static 1], int base);

```

The first simply provides us with a timestamp of type `time_t` of the current time. The simplest form to use the return value of `time(0)`. As we have already seen, two such times that we have taken at different moments in the program execution can then be used to express a time difference by means of **difftime**.

Let’s start to explain what all this is doing from the human perspective. As we already have seen, **struct tm** structures a calendar time mainly as you would expect. It has hierarchical date fields such as **tm\_year** for the year, **tm\_mon** for the month and so on, down to the granularity of a second. They have one pitfall, though, how the different fields are counted. All but one start with 0, e.g. **tm\_mon** set to 0 stands for January and **tm\_wday** 0 stands for Sunday.

Unfortunately, there are exceptions:

- **tm\_mday** starts counting days in the month at 1.
- **tm\_year** must add 1900 to get the year in the Gregorian calendar. Years represented in that way should lie between Gregorian years 0 and 9999.
- **tm\_sec** is in the range from 0 to 60, including. The latter is for the rare occasion of leap seconds.

There are three supplemental date fields that are used to supply additional information to a time value in a **struct tm**.

- **tm\_wday** for the week day,

[Exs 51] Implement a function `find_prefix` as needed by `Strtoul`.

- `tm_yday` for the day in the year, and
- `tm_isdst` a flag that informs if a date is considered being in DST of the local time zone or not.

The consistency of all these fields can be enforced with the function `mktime`. It can be seen to operate in three steps

- (1) The hierarchical date fields are normalized to their respective ranges.
- (2) `tm_wday` and `tm_yday` are set to the corresponding values.
- (3) If `tm_isday` has a negative value, this value is modified to 1 if the date falls into DST for the local platform, and to 0 otherwise.

`mktime` also serves an extra purpose. It returns the time as a `time_t`. `time_t` is thought to represent the same calendar times as `struct tm`, but is defined to be an arithmetic type, more suited to compute with them. It operates on a linear time scale. A `time_t` value of 0, the beginning of `time_t` is called *epoch*<sup>C</sup> in the C jargon. Often this corresponds to the beginning of Jan 1, 1970.

The granularity of `time_t` is usually to the second, but nothing guarantees that. Sometimes processor hardware has special registers for clocks that obey a different granularity. `difftime` translates the difference between two `time_t` values into seconds that are represented as a double value.

Annex K

Others of the traditional functions that manipulate time in C are a bit dangerous, they operate on global state, and we will not treat them here. So these interfaces have been reviewed in Annex K to a `_s` form:

```

1 errno_t asctime_s(char s[static 1], rsize_t maxsize,
2     struct tm const timeptr[static 1]);
3 errno_t ctime_s(char s[static 1], rsize_t maxsize,
4     const time_t timer[static 1]);
5 struct tm *gmtime_s(time_t const timer[restrict static 1],
6     struct tm result[restrict static 1]);
7 struct tm *localtime_s(time_t const timer[restrict static 1],
8     struct tm result[restrict static 1]);

```

The following picture shows how all these functions interact:

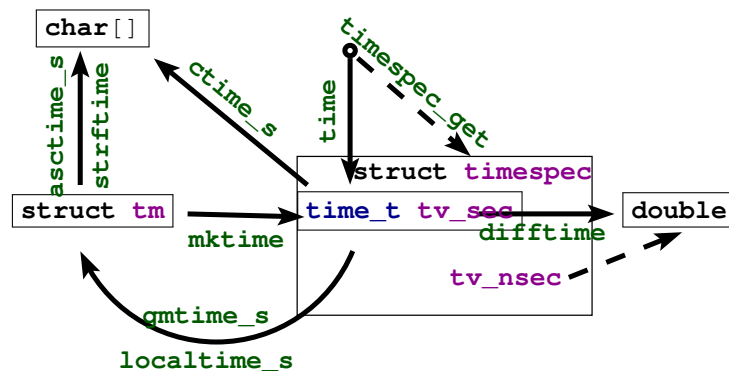


FIGURE 1. Time conversion functions

Two functions for the inverse operation from `time_t` into `struct tm` come into view:



- `localtime_s` stores the broken down local time
- `gmtime_s` stores the broken time, expressed as universal time, UTC.

As indicated, they differ in the time zone that they assume for the conversion. Under normal circumstances `localtime_s` and `mktime` should be inverse to each other, `gmtime_s` has no direct counterpart for the inverse direction.

Textual representations of calendar times are also available. `asctime_s` stores the date in a fixed format, independent of any locale, language (it uses English abbreviations) or platform dependency. The format is a string of the form

```
"Www_Mmm_DD_HH:MM:SS_YYYY\n"
```

`strftime` is more flexible and allows to compose a textual representation with format specifiers.

It works similar to the `printf` family, but has special %-codes for dates and times, see Table 23. Here “locale” indicates that different environment settings, such as preferred language or time zone may influence the output. How to access and eventually set these will be explained in Section 8.5. `strftime` receives three arrays: a `char[max]` array that is to be filled with the result string, another string that holds the format, and a `struct tm const [1]` that holds the time to be represented. The reason for passing in an array for the time will only become apparent when we know more about pointers.


The opaque type `time_t` and with that `time` only has a granularity of seconds.

If we need more precision than that, `struct timespec` and function `timespec_get` can be used. With that we have an additional field `tv_nsec` that provides nanosecond precision. The second argument `base` only has one value defined by the C standard, `TIME_UTC`. You should expect a call to `timespec_get` with that value to be consistent with calls to `time`. They both refer to Earth’s reference time. Specific platforms may provide additional values for `base` that would specify a “clock” that is different from that “walltime” clock. An example for such a clock could be relative to the planetary or other physical system that your computer system is involved with.<sup>52</sup> Relativity or other time adjustments could be avoided by using a so-called “monotonic clock” that would only be referring to the startup time of the system. A “cpu clock”, could refer to the time the program execution had been attributed processing resources.

For the later, there is an additional interface that is provided by the C standard library.

```
1 clock_t clock(void);
```

For historical reasons, this introduces yet another type, `clock_t`. It is an arithmetic time that gives the processor time in `CLOCKS_PER_SEC` units per second.

Having three different interfaces, `time`, `timespec_get` and `clock` is a bit unfortunate. It would have been beneficial to provide predefined constants such as `TIME_PROCESSOR_TIME` or `TIME_THREAD_TIME` for other forms of clocks. 

**8.5. Runtime environment settings.** A C program can have access to an *environment list*<sup>C</sup>: a list of name-value pairs of strings (often called *environment variables*<sup>C</sup>) that can transmit specific information from the runtime environment. There is a historic function `getenv` to access this list:

```
1 char* getenv(char const name[static 1]);
```

With our current knowledge, with this function we are only able to test if a name is present in the environment list:

```
1 bool havenv(char const name[static 1]) {
2     return getenv(name);
```

<sup>52</sup>Beware that objects that move fast relative to Earth such as satellites or space crafts may perceive relativistic time shifts compared to UTC.

TABLE 23. **strftime** format specifiers. Those marked as “locale” may differ dynamically according to locale runtime settings, see Section 8.5. Those marked with ISO 8601 are specified by that standard.

spec	meaning	locale	ISO 8601
"%S"	second ("00" to "60")		
"%M"	minute ("00" to "59")		
"%H"	hour ("00" to "23").		
"%I"	hour ("01" to "12").		
"%e"	day of the month ("1" to "31")		
"%d"	day of the month ("01" to "31")		
"%m"	month ("01" to "12")		
"%B"	full month name	X	
"%b"	abbreviated month name	X	
"%h"	equivalent to "%b"	X	
"%Y"	year		
"%y"	year ("00" to "99")		
"%C"	century number (year/100)		
"%G"	week-based year, same as "%Y", except if the ISO week number belongs another year		X
"%g"	like "%G", ("00" to "99")		X
"%u"	weekday ("1" to "7"), Monday being "1"		
"%w"	weekday ("0" to "6", Sunday being "0")		
"%A"	full weekday name	X	
"%a"	abbreviated weekday name	X	
"%j"	day of year ("001" to "366")		
"%U"	week number in the year ("00" to "53"), starting at Sunday		
"%W"	week number in the year ("00" to "53"), starting at Monday		
"%V"	week number in the year ("01" to "53"), starting with first 4 days in the new year		X
"%Z"	timezone name	X	
"%z"	" +hhmm" or "-hhmm", the hour and minute offset from UTC		
"%n"	newline		
"%t"	horizontal tabulator		
"%%"	literal "%"		
"%x"	date	X	
"%D"	equivalent to "%m/%d/%y"		
"%F"	equivalent to "%Y-%m-%d"		X
"%X"	time	X	
"%p"	either "AM" or "PM", noon is "PM", midnight is "AM"	X	
"%r"	equivalent to "%I:%M:%S_ %p".	X	
"%R"	equivalent to "%H:%M"		
"%T"	equivalent to "%H:%M:%S"		X
"%c"	preferred date and time representation	X	

3 | }

```

1 errno_t getenv_s(size_t * restrict len,
2               char value[restrict],
3               rsize_t maxsize,
4               char const name[restrict]);

```

If any, this function copies the value that corresponds to `name` from the environment into `value`, a `char[maxsize]`, provided that it fits. Printing such value can look as this:

```

1 void printenv(char const name[static 1]) {
2     if (getenv(name)) {
3         char value[256] = { 0, };
4         if (getenv_s(0, value, sizeof value, name)) {
5             fprintf(stderr,
6                 "%s:_value_is_longer_than_%zu\n",
7                 name, sizeof value);
8         } else {
9             printf("%s=%s\n", name, value);
10        }
11    } else {
12        fprintf(stderr, "%s_not_in_environment\n", name);
13    }
14 }

```

As you can see, that after detecting if the environment variable exists, `getenv_s` can safely be called with the first argument set to 0. Additionally, it is guaranteed that the `value` target buffer will only be written, if the intended result fits in, there. The `len` parameter could be used to detect the real length that is needed, and dynamic buffer allocation could be used to print out even large values. We have to refer to higher levels for such usages.

The environment variables that will be available to programs depend heavily on the operating system. Commonly provided environment variables include `"HOME"` for the user's home directory, `"PATH"` for the collection of standard paths to executables, `"LANG"` or `"LC_ALL"` for the language setting.

The language or *locale*<sup>C</sup> setting is another important part of the execution environment that a program execution inherits. At startup, C forces the locale setting to a normalized value, called the `"C"` locale. It has basically American English choices for numbers or times and dates.

The function `setlocale` from `locale.h` can be used to set or inspect the current value. `#include <locale.h>`

```

1 char* setlocale(int category, char const locale[static 1]);

```

Besides `"C"`, the C standard only prescribes the existence of one other valid value for `locale`, the empty string `"`". This can be used to set the effective locale to the systems default. The `category` argument can be used to address all or only parts of the language environment. Table 24 gives an overview over the possible values and the part of the C library they affect. Additional platform dependent categories may be available.

**8.6. Program termination and assertions.** We already have seen the simplest way of program termination: a regular return from `main`:

**Rule 1.8.6.1** *Regular program termination should use `return` from `main`.*

TABLE 24. Categories for the `setlocale` function

<code>LC_COLLATE</code>	string comparison through <code>strcoll</code> and <code>strxfrm</code>
<code>LC_CTYPE</code>	character classification and handling functions, see Section 8.3.
<code>LC_MONETARY</code>	monetary formatting information, <code>localeconv</code>
<code>LC_NUMERIC</code>	decimal-point character for formatted I/O, <code>localeconv</code>
<code>LC_TIME</code>	<code>strftime</code> , see Section 8.4
<code>LC_ALL</code>	all of the above

Using the function `exit` from within `main` is kind of senseless, it can be as easily done with a `return`.

**Rule 1.8.6.2** Use `exit` from a function that may terminate the regular control flow.

The C library has three other functions that terminate the program execution, in order of severity:

```
1 _Noreturn void quick_exit(int status);
2 _Noreturn void _Exit(int status);
3 _Noreturn void abort(void);
```

Now, `return` from `main` (or a call to `exit`) already provides the possibility to specify if the program execution is considered to be a success or not. Use the return value to specify that; as long as you have no other needs or you don't fully understand what these other functions do, don't use them, really don't.

**Rule 1.8.6.3** Don't use other functions for program termination than `exit`, unless you have to inhibit the execution of library cleanups.

Cleanup at program termination is important. The runtime system can flush and close files that are written or free other resources that the program occupied. This is a feature and should rarely be circumvented.

There even is a mechanism to install your own *handlers*<sup>C</sup> that are to be executed at program termination. Two functions can be used for that:

```
1 int atexit(void func(void));
2 int at_quick_exit(void func(void));
```

These have a syntax that we have not yet seen: *function parameters*<sup>C</sup>. E.g. the first reads “function `atexit` that returns an `int` and that receives a function `func` as a parameter”.<sup>53</sup>

We will not go into details here, an example just shows how this can be used:

```
1 void sayGoodBye(void) {
2     if (errno) perror("terminating_with_error_condition");
3     fputs("Good_Bye\n", stderr);
4 }
5
6 int main(int argc, char* argv[argc+1]) {
7     atexit(sayGoodBye);
8     ...
9 }
```

<sup>53</sup>In fact, in C such a notion of a function parameter `func` to a function `atexit` is equivalent to passing a *function pointer*<sup>C</sup>. In descriptions of such functions you will usually see that pointer variant. For us this distinction is not yet relevant and it is simpler to think of a function being passed by reference.

This uses the function **atexit** to establish the **exit**-handler `sayGoodBye`. After normal termination of the program code this function will be executed and give a status of the execution. This might be a nice thing to impress your fellow co-worker if you are in need of some respect. More seriously, this is the ideal place to put all kind of cleanup code, such as freeing memory, or such as writing a termination time stamp to a logfile. Observe that the syntax for calling is **atexit**(`sayGoodBye`), there are no `()` for `sayGoodBye` itself: here `sayGoodBye` is not called at that point, but only a reference to the function is passed to **atexit**.

Under rare circumstance you might want to circumvent these established **atexit** handlers. There is a second pair of functions, **quick\_exit** and **at\_quick\_exit**, that can be used to establish an alternative list of termination handlers. Such alternative list may be useful in case that the normal execution of the handlers is too time consuming. Use with care.

The next function, **\_Exit**, is even more sever than that: it inhibits both types of application specific handlers to be executed. The only thing that is still executed are the platform specific cleanups, such as file closure. Use this with even more care.

The last function, **abort**, is even more intrusive. Not only that it doesn't call the application handlers, it also inhibits the execution of some system cleanups. Use this with extreme care.

At the beginning of this section, we have already seen **\_Static\_assert** and **static\_assert** that should be used to make compile time assertions. This can test for any form of compile time Boolean expression. There are two other identifiers that come from `assert.h` that can be used for runtime assertions: **assert** and **NDEBUG**. The first can be used to test for an expression that must hold at a certain moment. It may contain any Boolean expression and it may be dynamic. If the **NDEBUG** macro is not defined during compilation, every time execution passes by the call to this macro the expression is evaluated. The functions `gcd` and `gcd2` from Section 7.3 show typical use cases of **assert**: a condition that is supposed to hold in *every* execution.

```
#include <assert.h>
```

If the condition doesn't hold, a diagnostic message is printed and **abort** is called. So all of this is not something that should make it through into a production executable. From the discussion above we know that the use of **abort** is harmful, in general, and also an error message such as

	Terminal	
0	assertion failed in file euclid.h, function gcd2(), line 6	

is not very helpful for your customers. It *is* helpful during the debugging phase where it can lead you to spots where you make false assumptions about the value of some variables.

**Rule 1.8.6.4** Use as many **assert** as you may to confirm runtime properties.

As mentioned **NDEBUG** inhibits the evaluation of the expression and the call to **abort**. Please use it to reduce the overhead.

**Rule 1.8.6.5** In production compilations, use **NDEBUG** to switch off all **assert**.



## LEVEL 2



# Cognition

## 9. Style

Programs serve both sides. First, as we have already seen, they serve to give instructions to the compiler and the final executable. But equally important, they document the intended behavior of a system for us people (users, customers, maintainers, lawyers, ...) that have to deal with it.

Therefore our prime directive is

**Rule C** *All C code must be readable.*

The difficulty with that directive is to know what constitutes “readable”. Not all experienced C programmers agree on all the points, so first of all we will try to establish a minimal list of necessities. The first things that we have to have in mind when discussing the human condition that it is constrained by two major factors: physical ability and cultural baggage.

**Rule 2.9.0.1** *Short term memory and the field of vision are small.*

Torvalds et al. [1996], the coding style for the Linux kernel, is a good example that insists on that aspect, and certainly is worth a detour, if you haven’t read it, yet. Its main assumptions are still valid: a programming text has to be represented in a relatively small “window” (be it a console or a graphical editor) that can be roughly be counted in 30 lines of 80 columns, a “surface” of 2400 characters. Everything that doesn’t fit there, has to be memorized. *E.g.* our very first program in Listing 1 fits into that constraint.

By its humorous reference to Kernighan and Ritchie [1978], the Linux coding style also refers to another fundamental fact, namely

**Rule 2.9.0.2** *Coding style is not a question of taste but of culture.*

Ignoring this, easily leads to endless and fruitless debates about not much at all.

**Rule 2.9.0.3** *When you enter an established project you enter a new cultural space.*

Try to adapt to the habits of the inhabitants. When you create your own project, you have a bit of freedom to establish your own rules. But be careful if you want others to adhere to it, still, you must not deviate too much from the common sense that reigns in the corresponding community.

**9.1. Formatting.** The C language itself is relatively tolerant to formatting issues. Under normal circumstances, a C compiler will dumbly parse an entire program that is written on a single line, with minimal white space and where all identifiers are composed of the letter `l` and the digit `1`: the need for code formatting originates in human incapacity.

**Rule 2.9.1.1** *Choose a consistent strategy for white space and other text formatting.*

This concerns indentation, placement of all kinds of parenthesis `{ }`, `[ ]` and `( )`, spaces before and after operators, trailing spaces or multiple new lines. The human eye and brain are quite peculiar in their habits, and to ensure that they work properly and efficiently we have to ensure to be in phase.

In the introduction for Level 1 you already have seen a lot of the coding style rules that are applied for the code in this book. Take them just as an example for one style, most probably you will encounter other styles as you go along. Let us recall some of the rules and introduce some others that had not yet been presented.

- We use prefix notation for code blocks, that is an opening `{` is on the end of a line.
- We bind type modifiers and qualifiers to the left.
- We bind function `()` to the left but `()` of conditions are separated from their keyword such as `if` or `for` by a space.
- A ternary expression has spaces around the `?` and the `:`.
- Punctuators `:`, `;` and `,` have no space before them but either one space or a new line after.

As you see, when written out these rules can appear quite cumbersome and arbitrary. They have no value as such, they are visual aids that help you and your collaborators capture a given code faster, with the blink of an eye. They are not meant to be meticulously typed by you directly, but you should acquire and learn the tools that help you with this.

**Rule 2.9.1.2** *Have your text editor automatically format your code correctly.*

I personally use Emacs<sup>1</sup> for that task (yes, I am that old). For *me*, it is ideal since it understands a lot of the structure of a C program by itself. Your mileage will probably vary, but don't use a tool in everyday's life that gives you less. Text editors, integrated development environments (IDE), code generators are there for us, not the other way around.

Then in bigger projects, you should enforce such a formatting policy for all the code that circulates and is read by others. Otherwise it will become difficult to track differences between versions of programming text. This can be automated by commandline tools that do the formatting. Here, I have a long time preference for `astyle` (artistic style)<sup>2</sup>. Again, your mileage may vary, chose anything that ensures the task.

**9.2. Naming.** The limit of such automatic formatting tools is reached when it comes to naming.

**Rule 2.9.2.1** *Choose a consistent naming policy for all identifiers.*

There are two different aspects to naming, technical restrictions on one hand and semantic conventions on the other. Unfortunately, they are often mixed up and subject of endless ideological debate.

For C, different technical restrictions apply, they are meant to help you, take them seriously. First of all Rule 2.9.2.1 says *all identifiers*: types (`struct` or not), `struct` and `union` fields, variables, enumerations, macros, functions, function-like macros. There are so many tangled "name spaces" you'd have to be careful.

In particular the interaction between header files and macro definitions can have surprising effects. A seemingly innocent example:

```
1 double memory_sum(size_t N, size_t I, double strip[N][I]);
```

- `N` is a capitalized identifier, thus your collaborator could be tempted to define a macro `N` as a big number.



```
#include <complex.h>
```

- **I** is used for the root of `-1` as soon as someone includes `complex.h`. (And you see that the automatic code annotation system of this book thinks that this refers to the macro.)
- The identifier `strip` might be used by some C implementation for a library function or macro.
- The identifier `memory_sum` might be used by the C standard for a type name in the future.

**Rule 2.9.2.2** *Any identifier that is visible in a header file must be conforming.*

Here conforming is a wide field. In the C jargon an identifier is *reserved*<sup>C</sup> if its meaning is fixed by the C standard and you may not redefine it otherwise.

- Names starting with an underscore and a second underscore or a capital letter are reserved for language extensions and other internal use.
- Names starting with an underscore are reserved in file scope and for **enum**, **struct** and **union** tags.
- Macros have all caps names.
- All identifiers that have a predefined meaning are reserved and cannot be used in file scope. These are lot of identifiers, *e.g.* all functions in the C library, all identifiers starting with **str** (as our `strip`, above), all identifiers starting with **E**, all identifiers ending in **\_t** and many more.

What makes all of these relatively difficult, is that you might not detect any violation for years and then all of a sudden on a new client machine, after the introduction of the next C standard and compiler or after a simple system upgrade your code explodes.

A simple strategy to keep the probability of naming conflicts low is to expose as few names as possible:

**Rule 2.9.2.3** *Don't pollute the global name space.*

So expose only types and functions as interfaces that are part of the *API*<sup>C</sup>, *application programmable interface*<sup>C</sup>, that is those that are supposed to be used by users of your code.

A good strategy for a library that has vocation of use by others or in other projects is to use naming prefixes that are unlikely to create conflicts. *E.g.* many functions and types in the POSIX thread API are prefixed with “`pthread_`”, or for my tool box P99, I use the prefixes “`p99_`”, “`P99_`”, for API interfaces and “`p00_`” and “`P00_`” for internals.

There are two sorts of names that may interact badly with macros that another programmer writes at which you might not think immediately:

- field names of **struct** and **union**
- parameter names in function interfaces.

The first point is the reason why the fields in standard structures usually have a prefix to their names: **struct timespec** has **tv\_sec** as a field name, because an uneducated user might declare a macro `sec` that would interfere in unpredictable way with when including `time.h`. For the second we already have seen the example from above. In P99 I would specify such a function similar to something like this:

```
#include <time.h>
```

```
1 double p99_memory_sum(size_t p00_n, size_t p00_i,
2                       double p00_strip[p00_n][p00_i]);
```

This problem gets worse when we are also exposing program internals to the public view. This happens in two cases:

<sup>1</sup><https://www.gnu.org/software/emacs/>

<sup>2</sup><http://sourceforge.net/projects/astyle/>

- So-called **inline** functions, that are functions that have their definition (and not only declaration) visible in a header file.
- Functional macros.

We will only discuss these in detail much later.

Now that we have cleared the technical aspects of naming, we will look at the semantic aspect.

**Rule 2.9.2.4** *Names must be recognizable and quickly distinguishable.*

That has two aspects, distinguishable *and* quickly. Compare

		recognizable	distinguishable	quickly
11111111011	11111111011	no	no	no
myLineNumber	myLimeNumber	yes	yes	no
n	m	yes	yes	yes
ffs	clz	no	yes	yes
lowBit	highBit	yes	yes	yes
p00Orb	p00Urb	no	yes	no
p00_orb	p00_urb	yes	yes	yes

For your personal taste, the answers on the right of this table may look different. This here reflects *mine*: an implicit context for such names is part of my personal expectation. The difference between `n` and `m` on one side and for `ffs` and `clz` on the other, is implicit semantic.

For me, having a heavily biased mathematical background, single letter variable names from `i` to `n` such as `n` and `m` are integer variables. These usually occur inside a quite restricted scope as loop variables or similar. Having a single letter identifier is fine (we always have the declaration in view) and they are quickly distinguished.

Function names `ffs` and `clz` are different because they compete with all other three letter acronyms that could potentially be used for function names. Accidentally here `ffs` is shorthand for *first bit set*, but this is not immediate to me. What that would mean is even less clear, are bits counted from higher or lower order?

There are several conventions that combine several words in one identifier, among the most commonly used are the following:

- **camel case**<sup>C</sup>, using `internalCapitalsToBreakWords`
- **snake case**<sup>C</sup>, using `internal_underscores_to_break_words`
- **Hungarian notation**<sup>C3</sup>, that encodes type information in the prefix of the identifiers, *e.g.* `szName`, where `sz` would stand for “string” and “zero terminated”.

As you might imagine, none of these is ideal. The first two tend to work against Rule 2.9.0.1, they easily clog up a whole, precious line of programming text with unreadable expressions:

```
1 return theVerySeldomlyUsedConstant*theVerySeldomlyUsedConstant/
   number_of_elements;
```

Hungarian notation, in turn, tends to use obscure abbreviations for types or concepts, produces unpronounceable identifiers, and completely breaks down if you have an API change.

So, to my opinion, none of these rules strategies have an absolute value, I encourage you to have a pragmatic approach to the question and to be aware that

**Rule 2.9.2.5** *Naming is a creative act.*

<sup>3</sup>Invented in Simonyi [1976], the phd thesis of Simonyi Károly

It is not easily subsumed by some simple technical rules.

Obviously, good naming is more important the more widely an identifier is used. So it is particular important for identifiers for which the declaration is generally out of view of the programmer, namely global names that constitute the API.

**Rule 2.9.2.6** *File scope identifiers must be comprehensive.*

What constitutes “comprehensive” here should be derived from the type of the identifier. Typenames, constants, variables or functions generally serve different purpose, so different strategies apply.

**Rule 2.9.2.7** *A type name identifies a concept.*

Examples for such concepts are “time” for `struct timespec`, “size” for `size_t`, a collection of corvidae for `enum corvid`, “person” for a data structure that collects data about people, “list” for a chained list of items, “dictionary” for a query data structure, and so on. If you have difficulties to come up with a concept for a data structure, an enumeration or an arithmetic type, you should most probably revisit your design.

**Rule 2.9.2.8** *A global constant identifies an artifact.*

That is a constant *stands out* for some reason from the other possible constants for the same type, it has a special meaning. It may have this meaning for some external reason beyond our control (`M_PI` for  $\pi$ ), because the C standard says so (`false`, `true`), because of a restriction of the execution platform (`SIZE_MAX`), be factual (`corvid_num`), culturally motivated (`fortytwo`) or a design decision.

Generally we will see below that file scope variables (*globals*) are much frowned upon. Nevertheless they are sometimes unavoidable, so we have to have an idea how to name them.

**Rule 2.9.2.9** *A global variable identifies state.*

Typical names for such variables would be `toto_initialized` to encode the fact that library *toto* has already been initialized, `onError` for a file scope but internal variable that is set in a library that must be torn down, or `visited_entries` for a hash table that collects some shared data.

**Rule 2.9.2.10** *A function or functional macro identifies an action.*

Not all but many of the functions in the C standard library follow that rule and use verbs as a component of their names. Some examples:

- A standard function that compares two strings is `strcmp`.
- A standard macro that queries for a property is `isless`.
- A function that accesses a data field could be called `toto_getFlag`.
- The corresponding one that sets such a field would be `toto_setFlag`.
- A function that multiplies two matrices `matrixMult`.

## 10. Organization and documentation

Being an important societal, cultural and economic activity, programming needs a certain form of organization to be successful. As for coding style, beginners tend to underestimate the effort that should be put into code and project organization and documentation: unfortunately many of us have to go to the experience of reading his or her own code and not have any clue what this is all about.

Documenting, or more generally explaining, program code is not an easy task. We have to find the right balance between providing context and necessary information and boringly stating the obvious. Let’s have a look at the two following lines:

```

1  u = fun4you(u, i, 33, 28); // ;)
2  ++i;                      // incrementing i

```

The first isn't good, because it uses magic constants, a function name that doesn't tell what is going on and a variable name that bares not much meaning, at least to me. The smiley comment indicates that the programmer had fun when writing this, but is not very helpful to the casual reader or maintainer.

In the second line, the comment is just superfluous and states just what any even not so experienced programmer knows about the ++ operator.

Compare this to

```

1  /* 33 and 28 are suitable because they are coprime. */
2  u = nextApprox(u, i, 33, 28);
3  /* Theorem 3 ensures that we may move to the next step. */
4  ++i;

```

Here we may deduce a lot more. With that, I'd expect `u` to be a floating point value, probably **double**, that is subject to an approximation procedure. That procedure runs in steps, indexed by `i`, and needs some additional arguments that are subject to a primality condition.

Generally we have the *what*, *what for*, *how* and *why* rules, by order of their importance.

**Rule 2.10.0.1 (what)** *Function interfaces describe what is done.*

**Rule 2.10.0.2 (what for)** *Interface comments document the purpose of a function.*

**Rule 2.10.0.3 (how)** *Function code tells how things are done.*

**Rule 2.10.0.4 (why)** *Code comments explain why things are done as they are.*

In fact, if you think of a larger library project that is used by others, you'd expect that all users will read the interface specification (e.g. in the synopsis part of a `man` page), most of them will read the explanation about these interfaces (the rest of the `man` page). Much less of them will look into the source code and read up *how* or *why* a particular interface implementation does things the way it does them.

A first consequence of these rules is that code structure and documentation go hand in hand. Especially the distinction between interface specification and implementation is important.

**Rule 2.10.0.5** *Separate interface and implementation.*

This rule is reflected in the use of two different kinds of C source files, **header files**<sup>C</sup>, usually ending with `".h"` and **translation units**<sup>C</sup>, `TU`, ending with `".c"`.

Syntactical comments have two distinct roles in those two kinds of source files that should well be separated.

**Rule 2.10.0.6** *Document the interface – Explain the implementation.*

**10.1. Interface documentation.** Other than more recent languages such as Java or perl, C has no “builtin” documentation standard. But in recent years a cross platform public domain tool has been widely adopted in many projects, doxygen<sup>4</sup>. It can be used to automatically produce web pages, pdf manuals, dependency graphs and a lot more. But even if you don't use doxygen or another equivalent tool, you should use its syntax to document interfaces.

<sup>4</sup><http://www.stack.nl/~dimitri/doxygen/>

**Rule 2.10.1.1** *Document interfaces thoroughly.*

Doxygen has a lot of categories that help for that, a extended discussion goes far beyond the scope of this book. Just consider the following example:

```

116  /**
117  ** @brief use the Heron process to approximate @a a to the
118  ** power of `1/k`
119  **
120  ** Or in other words this computes the @f$k^{th}@f$ root of @a a
121  **
122  ** As a special feature, if @a k is `-1` it computes the
123  ** multiplicative inverse of @a a.
124  **
125  ** @param a must be greater than `0.0`
126  ** @param k should not be `0` and otherwise be between
127  ** `DBL_MIN_EXP*FLT_RADIX` and
128  ** `DBL_MAX_EXP*FLT_RADIX`.
129  **
130  ** @see FLT_RADIX
131  */
double heron(double a, signed k);

```

Doxygen produces online documentation for that function that looks similar to Fig. 1 and also is able to produce formatted text that we can include in this book:

heron\_k.h

heron: use the Heron process to approximate  $a$  to the power of  $1/k$

Or in other words this computes the  $k^{th}$  root of  $a$ . As a special feature, if  $k$  is  $-1$  it computes the multiplicative inverse of  $a$ .

**Parameters:**

$a$	must be greater than $0.0$
$k$	should not be $0$ and otherwise be between $DBL\_MIN\_EXP * FLT\_RADIX$ and $DBL\_MAX\_EXP * FLT\_RADIX$ .

**See also:** FLT\_RADIX

---

```
double heron(double a, signed k);
```

heron\_k.h

FLT\_RADIX: the radix base 2 of **FLT\_RADIX**

This is needed internally for some of the code below.

---

```
# define FLT_RADIX something
```

As you have probably guessed, words starting with “@” have a special meaning for doxygen, they start its keywords. Here we have “@param”, “@a” and “@brief”. The first documents a function parameter, the second refers to such a parameter in the rest of the documentation and the last provides a brief synopsis for the function.

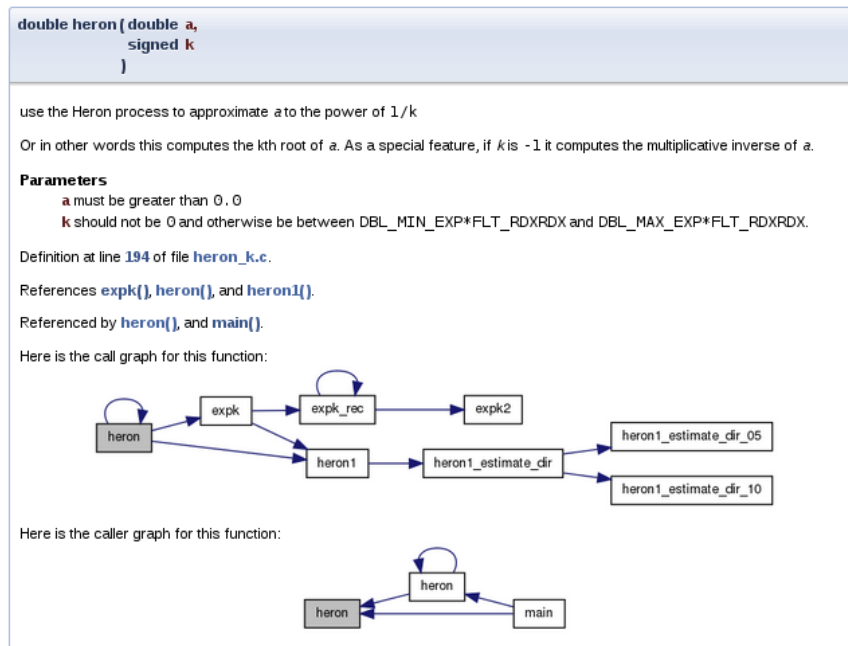


FIGURE 1. Documentation produced by doxygen

Additionally, we see that there is some “markup” capacity for in comment code, and also that doxygen was able to identify the place in compilation unit "`heron_k.c`" that defines the function and the call graph of the different functions involved in the implementation.

To make up for a good “project” organization, it is important that users of your code easily find connected pieces and don’t have to search all over the place.

#### Rule 2.10.1.2 Structure your code in units that have strong semantic connections.

Most often that is simply done by grouping all functions that treat a specific data type in one header file. A typical header file "`brian.h`" for `struct brian` would be like

```
1  #ifndef BRIAN_H
2  #define BRIAN_H 1
3  #include <time.h>
4
5  /** @file
6   ** @brief Following Brian the Jay
7   **/
8
9  typedef struct brian brian;
10 enum chap { sct, en, };
11 typedef enum chap chap;
12
13 struct brian {
14     struct timespec ts; /**< point in time */
15     unsigned counter; /**< wealth */
16     chap masterof; /**< occupation */
17 };
18
```

```

19  /**
20   ** @brief get the data for the next point in time
21   **/
22  brian brian_next(brian);
23
24  ...
25  #endif

```

That file comprises all the interfaces that are necessary to use that **struct**. It also includes other header files that might be needed to compile these interfaces and protect against multiple inclusion with *include guards*<sup>C</sup>, here the macro `BRIAN_H`.

**10.2. Implementation.** If you read code that is written by good programmers (and you should do that often!), you'll notice that it is often scarcely commented. Nevertheless it may end up to be quite readable, provided that the reader has basic knowledge of the C language. Good programming only needs to explain the ideas and prerequisites that are *not* obvious, the difficult part. Through the structure of the code, it shows what it does and how.

#### Rule 2.10.2.1 *Implement literally.*

A C program text is a descriptive text about what is to be done. The rules for naming entities that we introduced above play a crucial role to make that descriptive text readable and clear. The other is an obvious flow of control through visually clearly distinctive structuring in `{ }`-blocks that are linked together with comprehensive control statements.

#### Rule 2.10.2.2 *Control flow must be obvious.*

There are many possibilities to obfuscate control flow. The most important are

**burried jumps:** These are **break**, **continue**, **return** or **goto**<sup>5</sup> statements that are burried in a complicated nested structure of **if** or **switch** statements, eventually combined with loop structures.

**flyspeck expressions:** These are controlling expressions that combine a lot of operators in an unusual way (e.g. `!!++*p--` or `a --> 0`) such that they must be fly-specked with a looking glass to understand where the control flow goes from here.

**10.3. Macros.** We already happen to know one tools that can be very much abused to obfuscate control flow, macros. As you hopefully remember from Section 5.4.3 and 8, macros define textual replacements that can contain almost any C text. Because of the problems that we illustrate in the following, many projects ban macros completely. This is not the direction the evolution of the C standard goes, though. As we have seen, e.g. type-generic macros are *the* modern interface to mathematical functions (see 8.1), macros should be used for initialization constants ( 5.4.3) or are used to implement some compiler magic (**errno** p. 8).

So instead of denying it, we should try to tame the beast and setup some simple rules, that confine the possible damage.

#### Rule 2.10.3.1 *Macros should not change control flow in a surprising way.*

Notorious examples that pop up in discussion with beginners from time to time are things like:

<sup>5</sup>These will be discussed in Sections 13.2.2 and 15.

```

1 #define begin {
2 #define end }
3 #define forever for (;;)
4 #define ERRORCHECK(CODE) if (CODE) return -1
5
6 forever
7     begin
8     // do something
9     ERRORCHECK(x);
10    end

```

Don't do that. The visual habits of C programmers and of our tools don't easily work with that, and if you use such things in complicated code, it will almost certainly go wrong.

Here the `ERRORCHECK` macro is particularly dangerous. Its name doesn't suggest that a non-local jump such as a `return` might be hidden in there. And its implementation is even more dangerous. Consider the following two lines:

```

1 if (a) ERRORCHECK(x);
2 else puts("a_is_0!");

```

These lines are rewritten as

```

1 if (a) if (x) return -1;
2 else puts("a_is_0!");

```

The `else`-clause (so-called *dangling else*<sup>C</sup>) is attached to the innermost `if`, which we don't see. So this is equivalent to

```

1 if (a) {
2     if (x) return -1;
3     else puts("a_is_0!");
4 }

```

probably quite surprising for the casual reader.

This doesn't mean that control structures shouldn't be used in macros at all. They should just not be hidden and have no surprising effects.

```

1 #define ERROR_RETURN(CODE) \
2 do {                        \
3     if (CODE) return -1;    \
4 } while (false)

```

This macro by itself is probably not as obvious, but its *use* has no surprises.

```

1 if (a) ERROR_RETURN(x);
2 else puts("a_is_0!");

```

The name of the macro makes it explicit that there might be a `return`. The dangling `else` problem is handled because the replaced text

```

1 if (a) do {
2     if (CODE) return -1;
3 } while (false);
4 else puts("a_is_0!");

```

structures the code as expected: the `else` is associated to the first `if`.



The **do-while (false)**-trick is obviously ugly, and you shouldn't abuse it. But is a standard trick to surround one or several statements with a `{ }`-block without changing the block structure that is visible to the naked eye. Its intent is to fulfill

**Rule 2.10.3.2** *Function like macros should syntactically behave like function calls.*

Possible pitfalls are:

**if without else:** As we already have demonstrated above.

**trailing semicolon:** These can terminate an external control structure in a surprising way.

**comma operator:** The comma is an ambiguous fellow in C. In most contexts it is used as a list separator, *e.g.* for function calls, enumerator declarations or initializers. In the context of expressions it is a control operator. Avoid it.

**continuable expressions:** Expressions that will bind to operators in an unexpected way when put into a non-trivial context.<sup>[Exs 6]</sup> In the replacement text, put parentheses around parameters and expressions.

**multiple evaluation:** Macros are textual replacements. If a macro parameter is used twice (or more) its effects are done twice.<sup>[Exs 7]</sup>

**10.4. Pure functions.** Functions in C such as `size_min` (Section 4.4) or `gcd` (Section 7.3) that we declared ourselves have a limit in terms of what we are able to express: they don't operate on objects but on values. In a sense they are extensions of the value operators in Table 2 and not of the object operators of Table 3.

**Rule 2.10.4.1** *Function parameters are passed by value.*

That is, when we call a function all parameters are evaluated and the parameters, variables that are local to the function, receive the resulting values as initialization. The function then does whatever it has to do and then sends back the result of its computation through the return value.

For the moment, the only possibility that we have for two functions to manipulate the same *object* is to declare an object such that the declaration is visible to both functions. Such *global variables*<sup>C</sup> have a lot of disadvantages: they make code inflexible (the object to operate on is fixed), are difficult to predict (the places of modification are scattered all over) and difficult to maintain.

**Rule 2.10.4.2** *Global variables are frowned upon.*

A function with the following two properties is called *pure*<sup>C</sup>:

- The function has no other effects than returning a value.
- The function return value only depends on its parameters.

The only "interesting" about the execution of a pure function is its result, and that result only depends on the arguments that are passed. From the point of view of optimization, pure functions can such be moved around or even executed in parallel to other tasks. Execution can start at any point when its parameters are available and must be finished before the result is used.

Effects that would disqualify a function from being pure would be all those that change the abstract state machine other than by providing the return value. *E.g.*

- The functions reads part of the program's changeable state by other means than through its arguments.
- The functions modifies a global object.

[Exs 6] Consider a macro `sum(a, b)` that is implemented as `a+b`. What is the result of `sum(5, 2)*7?`

[Exs 7] Let `max(a, b)` be implemented as `((a) < (b) ? (b) : (a))`. What happens for `max(++i, 5)?`

LISTING 2.1. A type for computation with rational numbers.

```

1  #ifndef RATIONALS_H
2  # define RATIONALS_H 1
3  # include <stdbool.h>
4  # include "euclid.h"
5
6  typedef struct rat rat;
7
8  struct rat {
9      bool sign;
10     size_t num;
11     size_t denom;
12 };
13
14 /* Functions that return a value of type rat. */
15 rat rat_get(long long num, unsigned long long denom);
16 rat rat_get_normal(rat x);
17 rat rat_get_extended(rat x, size_t f);
18 rat rat_get_prod(rat x, rat y);
19 rat rat_get_sum(rat x, rat y);
20
21
22 /* Functions that operate on pointers to rat. */
23 void rat_destroy(rat * rp);
24 rat * rat_init(rat * rp,
25               long long num,
26               unsigned long long denom);
27 rat * rat_normalize(rat * rp);
28 rat * rat_extend(rat * rp, size_t f);
29 rat * rat_sumup(rat * rp, rat y);
30 rat * rat_rma(rat * rp, rat x, rat y);
31
32 #endif

```

- The function keeps a persistent internal state between calls<sup>8</sup>
- The function does IO<sup>9</sup>

Pure functions are a very good model for functions that perform small tasks, but they are pretty limited once we have to perform more complex ones. On the other hand, optimizers *love* pure functions, since their impact on the program state can simply be described by their parameters and return value. The influence on the abstract state machine that a pure function can have is very local and easy to describe.

**Rule 2.10.4.3** *Express small tasks as pure functions whenever possible.*

With pure functions we can be surprisingly far, even for some object-oriented programming style, if, for a first approach we are willing to accept a little bit of copying data around. Consider the structure type `rat` in Listing 2.1 that is supposed to be used for rational arithmetic.

Let us look into those functions that are named `rat_get_XXXXX`, to emphasize on the fact they return a value of type `rat`. The others, working with pointers to `rat` will be used later on in Section 11.3.

<sup>8</sup>Persistent state between calls to the same function can be established with local **static** variables, we will see this concept only in Section 13.2.

<sup>9</sup>Such an IO would *e.g.* occur by using `printf`.

This is a direct implementation of such a type, nothing that you should use as a library outside the scope of this learning experience. For simplicity it has numerator and denominator of identical type (`size_t`) and keeps track of the sign of the number in field `.sign`. We also include the file "`euclid.h`" to have access to the `gcd` function that we described earlier.

```

.
rational.c
3 rat rat_get(long long num, unsigned long long denom) {
4     rat ret = {
5         .sign = (num < 0),
6         .num = (num < 0) ? -num : num,
7         .denom = denom,
8     };
9     return ret;
10 }

```

```

.
rational.c
12 rat rat_get_normal(rat x) {
13     size_t c = gcd(x.num, x.denom);
14     x.num /= c;
15     x.denom /= c;
16     return x;
17 }

```

```

.
rational.c
19 rat rat_get_extended(rat x, size_t f) {
20     x.num *= f;
21     x.denom *= f;
22     return x;
23 }

```

With that we can define functions that do maintenance. The main idea is that such rational numbers should always be normalized. Not only is this easier to capture by humans, also it might avoid overflows while doing arithmetic.

Using these, we may define functions that are supposed to be used by others, namely `rat_get_prod` and `rat_get_sum`.

As you can see from all this, the fact that these are all pure functions ensures that they can easily be used, even in our own implementation, here. The only thing that we have to watch is to always assign the return values of our functions to some variable, such as in Line 38. Otherwise, since we don't operate on the object `x` but only on its value, changes during the function would be lost.<sup>[Exs 10] [Exs 11]</sup>

[Exs 10] The function `rat_get_prod` can produce intermediate values that may have it produce wrong results, even if the mathematical result of the multiplication is representable in `rat`. How is that?

[Exs 11] Reimplement the `rat_get_prod` function such that it produces a correct result all times that the mathematical result value is representable in a `rat`. This can be done with two calls to `rat_get_normal` instead of one.

rationals.c

```
25 rat rat_get_prod(rat x, rat y) {
26     rat ret = {
27         .sign = (x.sign != y.sign),
28         .num = x.num * y.num,
29         .denom = x.denom * y.denom,
30     };
31     return rat_get_normal(ret);
32 }
```

rationals.c

```
34 rat rat_get_sum(rat x, rat y) {
35     size_t c = gcd(x.denom, y.denom);
36     size_t ax = y.denom/c;
37     size_t bx = x.denom/c;
38     x = rat_get_extended(x, ax);
39     y = rat_get_extended(y, bx);
40     assert(x.denom == y.denom);
41
42     if (x.sign == y.sign) {
43         x.num += y.num;
44     } else if (x.sign > y.sign) {
45         x.num -= y.num;
46     } else {
47         x.num = y.num - x.num;
48         x.sign = !x.sign;
49     }
50     return rat_get_normal(x);
51 }
```

As mentioned earlier, because of the repeated copies this may result in compiled code that is not as efficient as it could. But this is not dramatic at all: the overhead by the copy operation can already be kept relatively low by good compilers. With optimization switched on, they usually can operate directly on the structure in place, as it is returned from such a function. Then such worries could be completely premature, because your program is short and sweet anyhow, or because the real performance problems of it lay elsewhere. Usually this should be completely sufficient for the level of programming skills that we have reach so far. Later, we will learn how to use that strategy efficiently by using so-called **inline** functions (Section 16.1) or *link time optimization* that many modern tool chains provide.

## 11. Pointers

Pointers are the first real hurdle to a deeper understanding of C. They are used in contexts where we have to be able to access objects from different points in the code, or where data is structured dynamically on the fly. The confusion of inexperienced programmers between pointers and arrays is notorious, so be warned that you might encounter difficulties in getting the terms correct.

On the other hand, pointers are one of the most important features of C. They are a big plus to help us in abstracting from the bits and odds of a particular platform and enable us to write portable code. So please, equip yourself with patience when you work through this section, it is crucial for the understanding of most of the rest of this book.

**11.1. Address-of and object-of operators.** If we have to perform tasks that can't be expressed with pure functions, things get more involved. We have to poke around in objects that are not locally defined in a function and need a suitable abstraction for doing so.

Let us take the task `double_swap` of swapping the contents of two `double` objects `d0` and `d1`. The unary *address-of*<sup>C</sup> operator `&` allows us to refer to an object through its *address*<sup>C</sup>. A call to our function could look as

```
1 double_swap(&d0, &d1);
```

The type that the address-of operator returns is a *pointer type*<sup>C</sup>, marked with a `*`. In

```
1 void double_swap(double* p0, double* p1) {
2     double tmp = *p0;
3     *p0 = *p1;
4     *p1 = tmp;
5 }
```

the function definition pointers `p0` and `p1` hold the addresses of the objects on which the function is supposed to operate. Inside we use the unary *object-of*<sup>C</sup> operator `*` to access the two objects: `*p0` then is the object corresponding to the first argument, with the call above that would be `d0`, and similarly `*p1` is the object `d1`.

In fact for the specification of the function `double_swap` we wouldn't even need the pointer notation. In the notation that we used so far it can equally be written as:

```
1 void double_swap(double p0[static 1], double p1[static 1]) {
2     double tmp = p0[0];
3     p0[0] = p1[0];
4     p1[0] = tmp;
5 }
```

Both, the use of array notation for the interface and the use of `[0]` to access the first element are simple rewrite operations that are built into the C language.

Remember from Section 6.2 that other than holding a valid address pointers may also be null or indeterminate.

**Rule 2.11.1.1** *Using `*` with an indeterminate or null pointer has undefined behavior.*

In practice, both cases will usually behave differently. The first might access some random object in memory and modify it. Often this leads to bugs that are difficult to trace. The second, null, will nicely crash your program. Consider this to be a feature.

**11.2. Pointer arithmetic.** We already have seen that a valid pointer holds the address of an object of its base type, but actually C is always willing to assume more:

**Rule 2.11.2.1** *A valid pointer addresses the first element of an array of the base type.*

Simple additive arithmetic allows to access the following elements of this array. The following functions are equivalent:

```
1 double sum1(size_t len, double const* a) {
2     double ret = 0.0;
3     for (double const* p = a; p < a+len; ++p) {
```

```

1 double sum0(size_t len, double const* a) {
2     double ret = 0.0;
3     for (size_t i = 0; i < len; ++i) {
4         ret += *(a + i);
5     }
6     return ret;
7 }

```

```

4     ret += *p;
5 }
6 return ret;
7 }

```

```

1 double sum2(size_t len, double const* a) {
2     double ret = 0.0;
3     for (double const*const aStop = a+len; a < aStop; ++a) {
4         ret += *a;
5     }
6     return ret;
7 }

```

These functions can then be called analogously to the following:

```

1 double A[7] = { 0, 1, 2, 3, 4, 5, 6, };
2 double s0_7 = sum0(7, &A[0]);    // for the whole
3 double s1_6 = sum0(6, &A[1]);    // for last 6
4 double s2_3 = sum0(3, &A[2]);    // for 3 in the middle

```

Unfortunately there is no way to know the length of the array that is hidden behind a pointer, therefore we have to pass it as a parameter into the function. The trick with **sizeof** as we have seen in Section 6.1.3 doesn't work.

**Rule 2.11.2.2** *The length an array object cannot be reconstructed from a pointer.*

So here we see a first important difference to arrays.

**Rule 2.11.2.3** *Pointers are not arrays.*

So if we pass arrays through pointers to a function, it is important to retain the real length that the array has. This is why we prefer the array notation for pointer interfaces throughout this book:

```

1 double sum0(size_t len, double const a[len]);
2 double sum1(size_t len, double const a[len]);
3 double sum2(size_t len, double const a[len]);

```

These specify exactly the same interfaces as above, but clarify to the casual reader of the code that `a` is expected to have `len` elements.

Another difference is that pointers have value and that that value can change. Thus they can be used by value operators as operands. This value is either the address of a valid object to which they are pointing or null, as we have already seen in Section 6.2.

Setting pointers to 0 if it hasn't some valid address is very important and should not be forgotten. It helps to check and keep track if a pointer has been set.

**Rule 2.11.2.4** *Pointers have truth.*

In application of Rule 1.3.1.3, you often will see code like

```

1 char const* name = 0;
2
3 // do something that eventually sets name
4
5 if (name) {
6     printf("today's_name_is_%s\n", name);
7 } else {
8     printf("today_we_are_anonymous\n");
9 }

```

A pointer must not only be set to an object (or null), such an object also must have the correct type.

**Rule 2.11.2.5** *A pointed-to object must be of the indicated type.*

As a direct consequence, a pointer that points beyond array bounds must not be dereferenced.

```

1 double A[2] = { 0.0, 1.0, };
2 double * p = &A[0];
3 printf("element_%g\n", *p); // referencing object
4 ++p; // valid pointer
5 printf("element_%g\n", *p); // referencing object
6 ++p; // valid pointer, no object
7 printf("element_%g\n", *p); // referencing non-object
8 // undefined behavior

```

Here, in the last line `p` has a value that is beyond the bounds of the array. Even if this might be the address of a valid object, we don't know anything about the object it is pointing to. So `p` is invalid at that point and accessing it as a type of `double` has no defined behavior.

In the above example, the pointer arithmetic itself is correct, as long as we don't access the object in the last line. The valid values of pointers are all addresses of array elements *and* the address beyond the array. Otherwise **for**-loops with pointer arithmetic as above wouldn't work reliably.

**Rule 2.11.2.6** *A pointer must point to a valid object, one position beyond a valid object or be null.*

So the example only worked up to the last line because the last `++p` left the pointer value just one element after the array. This version of the example still follows a similar pattern as the one before:

```

1 double A[2] = { 0.0, 1.0, };
2 double * p = &A[0];
3 printf("element_%g\n", *p); // referencing object
4 p += 2; // valid pointer, no object
5 printf("element_%g\n", *p); // referencing non-object
6 // undefined behavior

```

Whereas this last example may already crash at the increment operation:

```

1 double A[2] = { 0.0, 1.0, };
2 double * p = &A[0];

```

```

3 printf("element_%g\n", *p); // referencing object
4 p += 3;                      // invalid pointer arithmetic
5                               // undefined behavior

```

Pointer arithmetic as we discussed so far concerned operations between a pointer and an integer. There is also an inverse operation, *pointer difference*<sup>C</sup>, that takes to pointers and evaluates to some integer value. It is only allowed if the two pointers refer to elements of the same array object:

**Rule 2.11.2.7** *Only subtract pointers to elements of an array object.*

The value of such a difference then is simply the difference of the indices of the corresponding array elements:

```

1 double A[4] = { 0.0, 1.0, 2.0, -3.0, };
2 double * p = &A[1];
3 double * q = &A[3];
4 assert(p-q == -2);

```

We already have stressed the fact that the correct type for sizes of objects is `size_t`, an unsigned type that on many platforms is different from `unsigned`. This has its correspondence in the type of a pointer difference: in general we cannot assume that a simple `int` is wide enough to hold the possible values. Therefore the standard header `stddef.h` provides us with another type. On most architectures it will just be the signed integer type that corresponds to `size_t`, but we shouldn't care much.

**Rule 2.11.2.8** *All pointer differences have type `ptrdiff_t`.*

**Rule 2.11.2.9** *Use `ptrdiff_t` to encode signed differences of positions or sizes.*

**11.3. Pointers and `struct` s.** Pointers to structure types are crucial for most coding in C, so some specific rules and tools have been put in place to ease this typical usage. Let us e.g. consider the task of normalizing a `struct timespec` as we have encountered it previously. The use of a pointer parameter in the following function allows us to manipulate the object that we are interested directly:

```

timespec.c
10 /**
11  ** @brief compute a time difference
12  **
13  ** This uses a @c double to compute the time. If we want to
14  ** be able to track times without further loss of precision
15  ** and have @c double with 52 bit mantissa, this
16  ** corresponds to a maximal time difference of about 4.5E6
17  ** seconds, or 52 days.
18  **
19  **/
20 double timespec_diff(struct timespec const* later,
21                     struct timespec const* sooner){
22     /* be careful, tv_sec could be an unsigned type */
23     if (later->tv_sec < sooner->tv_sec)
24         return -timespec_diff(sooner, later);
25     else
26         return
27         (later->tv_sec - sooner->tv_sec)
28         /* tv_nsec is known to be a signed type. */
29         + (later->tv_nsec - sooner->tv_nsec) * 1E-9;
30 }

```



Here we use a new operator, `->`, that we had not previously introduced. It symbol is meant to represent a pointer as left operand that “points” to a field of the underlying **struct** as right operand. It is equivalent to a combination of “`*`” and “`.`”, namely `a->tv_sec` is nothing but `(*a).tv_sec`.

As another example, reconsider the type `rat` for rational numbers that we introduced in Section 10.4. The functions operating on pointers to that type of Listing 2.1 could be realized as follows.

```

.
rational.c
95 void rat_destroy(rat * rp) {
96     if (rp) *rp = (rat){ 0 };
97 }

```

```

.
rational.c
99 rat * rat_init(rat * rp,
100               long long num,
101               unsigned long long denom) {
102     if (rp) *rp = rat_get(num, denom);
103     return rp;
104 }

```

```

.
rational.c
106 rat * rat_normalize(rat * rp) {
107     if (rp) *rp = rat_get_normal(*rp);
108     return rp;
109 }

```

```

.
rational.c
111 rat * rat_extend(rat * rp, size_t f) {
112     if (rp) *rp = rat_get_extended(*rp, f);
113     return rp;
114 }

```

These first four functions are simple wrappers for the analogous functions that used copies instead of pointing to the object in question. The only pointer operations the use is to test validity and then, if the pointer is valid, to refer to the object in question. So these functions can be safely used, even if the pointer argument is null.

All four return their pointer argument. This is a convenient strategy to compose such functions, as we can see in the definitions of the two following arithmetic functions.

```

.
rational.c
116 rat * rat_sumup(rat * rp, rat y) {
117     size_t c = gcd(rp->denom, y.denom);
118     size_t ax = y.denom/c;
119     size_t bx = rp->denom/c;

```

```

120 rat_extend(rp, ax);
121 y = rat_get_extended(y, bx);
122 assert (rp->denom == y.denom);
123
124 if (rp->sign == y.sign) {
125     rp->num += y.num;
126 } else if (rp->num > y.num) {
127     rp->num -= y.num;
128 } else {
129     rp->num = y.num - rp->num;
130     rp->sign = !rp->sign;
131 }
132 return rat_normalize(rp);
133 }

```

rationals.c

```

135 rat * rat_rma(rat * rp, rat x, rat y) {
136     return rat_sumup(rp, rat_get_prod(x, y));
137 }

```

In particular the last, “rational multiply add”, comprehensively shows its purpose, to add the product of the two other function arguments to the object referred by `rp`.<sup>[Exs 12]</sup>

**11.4. Opaque structures.** Another special rule applies to pointers to structure types: they can be used even if the structure type itself is unknown. Such *opaque structures*<sup>C</sup> are often used to strictly separate the interface of some library and its implementation. *E.g.* a fictive type `toto` could just be presented in an include file as follows:

```

1 /* forward declaration of struct toto */
2 struct toto;
3 struct toto * toto_get(void);
4 void toto_destroy(struct toto *);
5 void toto_doit(struct toto *, unsigned);

```

Both, the programmer and the compiler, wouldn’t need more than that to use the type `struct toto`. The function `toto_get` could be used to obtain a pointer to an object of type `struct toto`, regardless how it might have been defined in the compilation unit that defines the functions. And the compiler gets away with it because he knows that all pointers to structures have the same representation, regardless of the specific definition of the underlying type.

Often such interfaces use the fact that null pointers are special. In the above example `toto_doit(0, 42)` could be a valid use case. This is why many C programmers don’t like it if pointers are hidden inside `typedef`.

```

1 /* forward declaration of struct toto_s and user type toto */
2 typedef struct toto_s * toto;
3 toto toto_get(void);
4 void toto_destroy(toto);
5 void toto_doit(toto, unsigned);

```

This is valid C, but hides the fact that 0 is a special value that `toto_doit` may receive.

#### Rule 2.11.4.1 Don’t hide pointers in a `typedef`.

[Exs 12] Write a function `rat * rat_dotproduct(rat * rp, size_t n, rat A[n], rat B[n])`; that computes  $\sum_{i=0}^{n-1} A[i] * B[i]$  and returns that value in `*rp`.

This is not the same as just introducing a **typedef** name for the **struct** as we have done before.

```
1  /* forward declaration of struct toto and typedef toto */
2  typedef struct toto toto;
3  toto * toto_get(void);
4  void toto_destroy(toto *);
5  void toto_doit(toto *, unsigned);
```

Here, the fact that the interface receive a pointer is still sufficiently visible.

**11.5. Array and pointer access are the same.** Now we are able to attack the major hurdle to understand the relationship between arrays and pointers.

**Rule 2.11.5.1** *The two expressions  $A[i]$  and  $*(A+i)$  are equivalent.*

This holds regardless whether  $A$  is an array or a pointer.

If it is a pointer we already understand the second expression. Here, Rule 2.11.5.1 just says that we may write the same expression as  $A[i]$ . Applying this notion of array access to pointers should just improve the readability of your code. The equivalence does not mean, that all of the sudden an array object appears where there was none. If  $A$  is null,  $A[i]$  should crash as nicely as should  $*(A+i)$ .

If  $A$  is an array,  $*(A+i)$  shows our first application of one of the most important rules in C, called *array-to-pointer decay*<sup>C</sup>:

**Rule 2.11.5.2** *Evaluation of an array  $A$  returns  $\&A[0]$ .*

In fact, this is the reason behind Rules 1.6.1.3 to 1.6.1.5. Whenever an array occurs that requires a value, it decays to a pointer and we loose all additional information.

**11.6. Array and pointer parameters are the same.** As a consequence of Rule 2.11.5.2, arrays can't be function arguments. There would be no way to call such a function with an array parameter; before any call to the function an array that we feed into it would decay into a pointer, and thus the argument type wouldn't match.

But we have seen declarations of functions with array parameters, so how did that work? The trick C gets away with it is to rewrite array parameters to pointers.

**Rule 2.11.6.1** *In a function declaration any array parameters rewrites to a pointer.*

Think of this and what it means for a while, understanding of that "chief feature" (or character flaw) is central for coding in C at ease.

To come back to our examples from Section 6.1.5 the functions that were written with array parameters could be declared

```
1  size_t strlen(char const* s);
2  char* strcpy(char* target, char const* source);
3  signed strcmp(char const* s0, char const* s1);
```

These are completely equivalent and any C compiler should be able to use both forms interchangeably.

Which one to use is a question of habits, culture or other social context. The rule that we follow in this book to use array notation if we suppose that this can't be null, and pointer notation if it corresponds to one single item of the base type that also can be null to indicate a special condition.

If semantically a parameter is an array, we also note what size we expect the array to be, if this is possible. And to make it possible, it is usually better to specify the length before the arrays/pointers. An interface such as tells a whole story. This becomes even more interesting if we handle two-dimensional arrays. A typical matrix multiplication could look as follows:

```

1 double double_copy(size_t len,
2                     double target[len],
3                     double const source[len]);

```

```

1 void matrix_mult(size_t n, size_t k, size_t m,
2                  double C[n][m],
3                  double A[n][k],
4                  double B[k][m]) {
5     for (size_t i = 0; i < n; ++i) {
6         for (size_t j = 0; j < m; ++j) {
7             C[i][j] = 0.0;
8             for (size_t l = 0; l < k; ++l) {
9                 C[i][j] += A[i][l]*B[l][j];
10            }
11        }
12    }
13 }

```

The prototype is equivalent to the less readable

```

1 void matrix_mult(size_t n, size_t k, size_t m,
2                  double (C[n])[m],
3                  double (A[n])[k],
4                  double (B[k])[m]);

```

and

```

1 void matrix_mult(size_t n, size_t k, size_t m,
2                  double (*C)[m],
3                  double (*A)[k],
4                  double (*B)[m]);

```

Observe that once we have rewritten the innermost dimension as a pointer, the parameter type is not an array anymore, but a “pointer to array”. So there is no need to rewrite the subsequent dimensions.

**Rule 2.11.6.2** *Only the innermost dimension of an array parameter is rewritten.*

So finally we have gained a lot by using the array notation. We have, without problems passed pointers to VLA into the function. Inside the function we could use conventional indexing to access the elements of the matrices. Not much acrobatics were required to keep track of the array lengths:

**Rule 2.11.6.3** *Declare length parameters before array parameters.*

They simply have to be known at the point where you use them first.

**11.7. Null pointers.** Some of you may have wondered that through all this discussion about pointers the macro **NULL** has not yet been used. The reason is that unfortunately the simple concept of a “generic pointer of value 0” didn’t succeed very well.

C has the concept of a *null pointer*<sup>C</sup> that corresponds to a 0 value of any pointer type.<sup>13</sup> Here

<sup>13</sup>Note the different capitalization of “null” versus **NULL**.

```

1 double const*const nix = 0;
2 double const*const nax = nix;

```

“nix” and “nax” would be such a pointer object of value 0. But unfortunately a *null pointer constant*<sup>C</sup> is then not what you’d expect.

First, here by *constant* the term refers to a compile time constant and not to a *const-qualified* object. So for that reason already, both pointer objects above *are not* null pointer constants. Second the permissible type for these constants is restricted, it may be any constant expression of integer type or of type **void\***. Other pointer types are not permitted, and we will only learn about pointers of that “type” below in Section 12.4.

The definition by the C standard of what could be the expansion of the macro **NULL** is quite loose, it just has to be a null pointer constant. Therefore a C compiler could chose any of the following for it:

expansion	type
0U	<b>unsigned</b>
0	<b>signed</b>
'\0'	
enumeration constant of value 0	
0UL	<b>unsigned long</b>
0L	<b>signed long</b>
0ULL	<b>unsigned long long</b>
0LL	<b>signed long</b>
(void*)0	<b>void*</b>

Commonly used values are 0, 0L and **(void\*) 0**.<sup>14</sup>

Here it is important that the type behind **NULL** is not prescribed by the C standard. Often people use it to emphasize that they talk about a pointer constant, which it simply isn’t on many platforms. Using **NULL** in context that we don’t master completely is even dangerous. This will in particular appear in the context of functions with variable number of arguments that will be discussed in Section 17.4.2. For the moment we go for the simplest solution:

**Rule 2.11.7.1** *Don’t use **NULL**.*

**NULL** hides more than it clarifies, either use 0, or if you really want to emphasize that the value is a pointer use the magic token sequence **(void\*) 0** directly.

**11.8. Function pointers.** There is yet another construct for which the address-of operator & can be used: functions. We already have seen this concept pop up when discussion the **atexit** function, see page 8.6, which is a function that receives a function argument. The rule is similar as for array decay that we described above:

**Rule 2.11.8.1** *A function f without following opening ( decays to a pointer to its start.*

Syntactically functions and function pointers are also similar to arrays in type declarations and as function parameters:

```

typedef void atexit_function(void);
// two equivalent definitions of the same type, that hides a pointer
typedef atexit_type* atexit_function_pointer;
typedef void (*atexit_function_pointer)(void);
// five equivalent declarations for the same function
void atexit(void f(void));

```

<sup>14</sup>In theory, there are even more possible expansions for **NULL**, such as **((char)+0)** or **((short)-0)**.

```
void atexit(void (*f) (void));
void atexit(atexit_function f);
void atexit(atexit_function* f);
void atexit(atexit_function_pointer f);
```

Which of the semantically equivalent writings for the function declaration is more readable, could certainly be subject of much debate. The second version with the `(*f)` parenthesis quickly gets difficult to read and the fifth is frowned upon because it hides a pointer in a type. Among the others, I personally slightly prefer the fourth over the first.

The C library has several functions that receive function parameters, we already have seen `atexit` and `at_quick_exit`. Another pair of functions in `stdlib.h` provides generic interfaces for searching (`bsearch`) and sorting (`qsort`).

```
typedef int compare_function(void const*, void const*);

void* bsearch(void const* key, void const* base,
              size_t n, size_t size,
              compare_function* compar);

void qsort(void* base,
           size_t n, size_t size,
           compare_function* compar);
```

Both receive an array `base` as argument on which they perform their task. The address to the first element is passed as a `void` pointer, so all type information is lost. To be able to handle the array properly, the functions have to know the size of the individual elements (`size`) and the number of elements (`n`).

In addition they receive a comparison function as a parameter that provides the information about the sort order between the elements. A simple version of such a function would look like this:

```
int compare_unsigned(void const* a, void const* b) {
    unsigned const* A = a;
    unsigned const* B = b;
    if (*A < *B) return -1;
    else if (*A > *B) return +1;
    else return 0;
}
```

The convention is that the two arguments point to elements that are to be compared and that the return value is strictly negative if `a` is considered less than `b`, 0 if they are equal and strictly positive otherwise.

The return type of `int` seems to suggest that `int` comparison could be done simpler:

```
/* An invalid example for integer comparison */
int compare_int(void const* a, void const* b) {
    int const* A = a;
    int const* B = b;
    return *A - *B;    // may overflow!
}
```

But this is not correct. *E.g* if `*A` is big, say `INT_MAX`, and `*B` is negative, the mathematical value of the difference can be larger than `INT_MAX`.

Because of the `void` pointers, a usage of this mechanism should always take care that the type conversions are encapsulated similar to the following:

```
/* A header that provides searching and sorting for unsigned. */

/* No use of inline here, we always use the function pointer. */
extern int compare_unsigned(void const*, void const*);
```

```

inline
unsigned const* bsearch_unsigned(unsigned const key[static 1],
                                size_t n, unsigned const base[nmemb]) {
    return bsearch(key, base, nmemb, sizeof base[0], compare_unsigned);
}

inline
void qsort_unsigned(size_t n, unsigned base[nmemb]) {
    qsort(base, nmemb, sizeof base[0], compare_unsigned);
}

```

Here **bsearch** (binary search) searches for an element that compares equal to `key[0]` or returns a null pointer if no such element is found. It supposes that array `base` is already sorted consistent to the ordering that is given by the comparison function. This assumption helps to speed up the search. Although this is not explicitly specified in the C standard, you can expect that a call to **bsearch** will never make more than  $\lceil \log_2(n) \rceil$  calls to `compare`.

If **bsearch** finds an array element that is equal to `*key` it returns the pointer to this element. Note that this drills a hole in C's type system, since this returns an unqualified pointer to an element whose effective type might be **const** qualified. Use with care. In our example we simply convert the return value to **unsigned const\***, such that we will never even see an unqualified pointer at the call side of `bsearch_unsigned`.

The name **qsort** is derived from the *quick sort* algorithm. The standard doesn't impose the choice the sorting algorithm, but the expected number of comparison calls should be of the magnitude of  $n \log_2(n)$ , just as quick sort would. There are no guarantees for upper bounds, you may assume that its worst case complexity is at most quadratic,  $O(n^2)$ .

Whereas there is some sort of catch-all pointer type, **void\***, that can be used as a generic pointer to object types, no such generic type or implicit conversion exists for function pointers.

#### Rule 2.11.8.2 *Function pointers must be used with their exact type.*

Such a strict rule is necessary because the calling conventions for functions with different prototypes may be quite different<sup>15</sup> and the pointer itself does not keep track of any of this.

The following function has a subtle problem because the types of the parameters are different than what we expect from a comparison function.

```

/* Another invalid example for an int comparison function */
int compare_int(int const* a, int const* b){
    if (*a < *b) return -1;
    else if (*a > *b) return +1;
    else return 0;
}

```

When you try to use this function with **qsort** your compiler should complain that the function has the wrong type. The variant that we gave above by using intermediate **void const\*** parameters should be almost as efficient as this invalid example, but it also can be guaranteed to be correct on all C platforms.

Calling functions and function pointers with the `( . . . )` operator also has similar rules as for arrays and pointers and the `[ . . . ]` operator:

#### Rule 2.11.8.3 *The function call operator ( . . . ) applies to function pointers.*

<sup>15</sup>The platform ABI may e.g. pass floating points in special hardware registers.

```
double f(double a);

// equivalent calls to f
f(3);           // decay to function pointer
(&f)(3);        // address of function
(*f)(3);        // decay to function pointer, then dereference, then decay
(*&f)(3);       // address of function, then dereference, then decay
(&*f)(3);       // decay, dereference, address of
```

So technically, in terms of the abstract state machine, the pointer decay is always performed and the function is called via a function pointer. The first, “natural”, call has a hidden evaluation of the `f` identifier that results in the function pointer.

With all that, we can use function pointers almost like functions.

```
// in a header
typedef int logger_function(char const*, ...);
extern logger_function* logger;
enum logs { log_pri, log_ign, log_ver, log_num };
```

This declares a global variable `logger` that will point to a function that prints out logging information. Using a function pointer will allow the user of this module to choose a particular function dynamically.

```
// in a TU
extern int logger_verbose(char const*, ...);
static
int logger_ignore(char const*, ...) {
    return 0;
}
logger_function* logger = logger_ignore;

static
logger_function* loggers = {
    [log_pri] = printf,
    [log_ign] = logger_ignore,
    [log_ver] = logger_verbose,
};
```

Here, we are defining tools that implement this. In particular, function pointers can be used as base type for array (here `loggers`). Observe that we use two external functions (`printf` and `logger_verbose`) and one `static` function (`logger_ignore`) for the array initialization: the storage class is not part of the function interface.

The `logger` variable can be assigned just as any other pointer type. Somewhere at startup we can have

```
if (LOGGER < log_num) logger = loggers[LOGGER];
```

And then this function pointer can be used anywhere to call the corresponding function:

```
logger("Do we ever see line %d of file %s?", __LINE__, __FILE__);
```

When using this feature you should always be aware that this introduces an indirection to the function call. The compiler first has to fetch the contents of `logger` and can only then call the function at the address that he found, there. This has a certain overhead and should be avoided in time critical code.



## 12. The C memory model

The operators that we have introduced above provide us with an abstraction, the C memory model. We may apply unary operator `&` to (almost) all objects<sup>16</sup> to retrieve their address. Seen from C, no distinction of the “real” location of an object is made. It could reside in your computers RAM, in a disk file or correspond to an IO port of some temperature sensor on the moon, you shouldn’t care. C is supposed to do right thing, regardless.

The only thing that C must care about is the *type* of the object which a pointer addresses. Each pointer type is derived from another type, its base type, and each such derived type is a distinct new type.

**Rule 2.12.0.1** *Pointer types with distinct base types are distinct.*

**12.1. A uniform memory model.** Still, even though generally all objects are typed, the memory model makes another simplification, namely that all objects are an assemblage of *bytes*<sup>C</sup>. The `sizeof` operator that we already introduced in the context of arrays measures the size of an object in terms of the bytes that it uses. There are three distinct types that by definition use exactly one byte of memory: the character types `char`, `unsigned char` and `signed char`.

**Rule 2.12.1.1** *`sizeof(char)` is 1 by definition.*

Not only can all objects be “accounted” in size as character typed on a lower level they can even be inspected and manipulated as if they were arrays of such character types. Below, we will see how this can be achieved, but for the moment we just note

**Rule 2.12.1.2** *Every object `A` can be viewed as `unsigned char[sizeof A]`.*

**Rule 2.12.1.3** *Pointers to character types are special.*

Unfortunately, the types that are used to “compose” all other object types is derived from `char`, the type that we already have seen for the characters of strings. This is merely an historical accident and you shouldn’t read too much into it. In particular, you should clearly distinguish the two different use cases.



**Rule 2.12.1.4** *Use the type `char` for character and string data.*

**Rule 2.12.1.5** *Use the type `unsigned char` as the atom of all object types.*

The type `signed char` is of much less importance than the two others.

From Rule 2.12.1.1 we also see another property of the `sizeof` operator, it also applies to object *types* if they are enclosed in `()`. This variant returns the value that an object of corresponding type would have.

**Rule 2.12.1.6** *The `sizeof` operator can be applied to objects and object types.*

<sup>16</sup>Only objects that are declared with keyword `register` don’t have an address, see Section 13.2.2 on Level 2

**12.2. Unions.** Let us now look into a way to look at the individual bytes of objects. Our preferred tool for this are **union**'s. These are similar in declaration to **struct** but have different semantic.

```

1
2 #include <inttypes.h>
3
4 typedef union unsignedInspect unsignedInspect;
5 union unsignedInspect {
6     unsigned val;
7     unsigned char bytes[sizeof(unsigned)];
8 };
9 unsignedInspect twofold = { .val = 0xAABBCCDD, };

```

The difference here is that such a **union** doesn't collect objects of different type into one bigger object, but it *overlays* an object with different type interpretation. By that it is the perfect tool to inspect the individual bytes of an object of another type.

Let us first try to figure out what values we would expect for the individual bytes. In a slight abuse of language let us speak of the different parts of an unsigned number that correspond to the bytes as *representation digits*. Since we view the bytes as being of type **unsigned char** they can have values 0 ... **UCHAR\_MAX**, including, and thus we interpret the number as written within a base of **UCHAR\_MAX+1**. In the example, on my machine, a value of type **unsigned** can be expressed with **sizeof(unsigned) == 4** such representation digits, and I chose the values 0xAA, 0xBB, 0xCC and 0xDD for the highest to lowest order representation digit. The complete **unsigned** value can be computed by the following expression, where **CHAR\_BIT** is the number of bits in a character type.

```

1 ((0xAA << (CHAR_BIT*3))
2  | (0xBB << (CHAR_BIT*2))
3  | (0xCC << CHAR_BIT)
4  | 0xDD)

```

With the **union** defined above, we have two different facets to look at the same object `twofold`: `twofold.val` presents it as being an **unsigned**, `twofold.bytes` presents it as array of **unsigned char**. Since we chose the length of `twofold.bytes` to be exactly the size of `twofold.val` it represents exactly its bytes, and thus gives us a way to inspect the *in memory representation*<sup>C</sup> of an **unsigned** value, namely all its representation digits.

```

12 printf("value_is_0x%.08X\n", twofold.val);
13 for (size_t i = 0; i < sizeof twofold.bytes; ++i)
14     printf("byte[%zu]:_0x%.02hhX\n", i, twofold.bytes[i]);

```

On my own computer, I receive a result as indicated below:<sup>17</sup>

<sup>17</sup>Test that code on your own machine.

```

Terminal
0  ~/build/modernC% ./code/endianess
1  value is 0xAABBCCDD
2  byte[0]: 0xDD
3  byte[1]: 0xCC
4  byte[2]: 0xBB
5  byte[3]: 0xAA

```

For my machine, we see that the output above had the low-order representation digits of the integer first, then the next-lower order digits and so on. At the end the highest order digits are printed. So the in-memory representation of such an integer on my machine is to have the low-order representation digits before the high-order ones.

All of this is *not* normalized by the standard, it is implementation defined behavior.

**Rule 2.12.2.1** *The in-memory order of the representation digits of a numerical type is implementation defined.*

That is, a platform provider might decide to provide a storage order that has the highest order digits first, then print lower order digits one by one. The storage order as given above is called *little Endian*<sup>C</sup>, the later one *big Endian*<sup>C</sup>. Both orders are commonly used by modern processor types. Some processors are even able to switch between the two orders on the fly.

The output above also shows another implementation defined behavior: I used the feature of my platform that one representation digit can nicely be printed by using two hexadecimal digits. Stated otherwise I assumed that `UCHAR_MAX+1` is 256 and that the number of value bits in an `unsigned char`, `CHAR_BIT`, is 8. As said, again this is implementation defined behavior: although the vast majority of platforms have these properties<sup>18</sup>, there are still some around that have wider character types.

**Rule 2.12.2.2** *On most architectures `CHAR_BIT` is 8 and `UCHAR_MAX` is 255.*

In the example we have investigated the in-memory representation of the simplest arithmetic base types, unsigned integers. Other base types have in-memory representations that are more complicated: signed integer types have to encode the sign, floating point types have to encode sign, mantissa and exponent and pointer types themselves may follow any internal convention that fits the underlying architecture.<sup>[Exs 19][Exs 20][Exs 21]</sup>

**12.3. Memory and state.** The value of all objects constitutes the state of the abstract state machine, and thus the state of a particular execution. C's memory model provides something like a unique location for (almost) all objects through the `&` operator, and that location can be accessed and modified from different parts of the program through pointers.

Doing so makes the determination of the abstract state of an execution much more difficult, if not impossible in many cases.

```

1  double blub(double const* a, double* b);
2
3  int main(void) {
4      double c = 35;
5      double d = 3.5;

```

<sup>18</sup>In particular all POSIX systems.

[Exs 19] Design a similar `union` type to investigate the bytes of a pointer type, `double*` say.

[Exs 20] With such a `union`, investigate the addresses of to consecutive elements of an array.

[Exs 21] Compare addresses of the same variable between different executions.

```

6  printf("blub_is_%g\n", blub(&c, &d));
7  printf("after_blub_the_sum_is_%g\n", c + d);
8  }

```

Here we (as well as the compiler) only see a declaration of function `blub`, no definition. So we cannot conclude much of what that function does to the objects its arguments point to. In particular we don't know if the variable `d` is modified and so the sum `c + d` could be anything. The program really has to inspect the object `d` in memory to find out what the values *after* the call to `blub` are.

Now let us look into such a function that receives two pointer arguments:

```

1  double blub(double const* a, double* b) {
2      double myA = *a;
3      *b = 2*myA;
4      return *a;          // may be myA or 2*myA
5  }

```

Such a function can operate under two different assumptions. First if called with two distinct addresses as arguments, `*a` will be unchanged, and the return value will be the same as `myA`. But if both argument are the same, *e.g.* the call is `blub(&c, &c)`, the assignement to `*b` would have changed `*a`, too.

The phenomenon of accessing the same object through different pointers is called *aliasing*<sup>C</sup>. It is a common cause for missed optimization, the knowledge about the abstract state of an execution can be much reduced. Therefore C forcibly restricts the possible aliasing to pointers of the same type:

**Rule 2.12.3.1 (Aliasing)** *With the exclusion of character types, only pointers of the same base type may alias.*

To see this rule in effect consider a slight modification of our example above:

```

1  size_t blob(size_t const* a, double* b) {
2      size_t myA = *a;
3      *b = 2*myA;
4      return *a;          // must be myA
5  }

```

Because here the two parameters have different types, C *assumes* that they don't address the same object. In fact, it would be an error to call that function as `blob(&e, &e)`, since this would never match the prototype of `blob`. So at the **return** statement we can be sure that the object `*a` hasn't changed and that we already hold the needed value in variable `myA`.

There are ways to fool the compiler and to call such a function with a pointer that addresses the same object. We will see some of these cheats later. Don't do this, this is a road to much grief and despair. *If* you do so the behavior of the program becomes undefined, so you'd have to guarantee (prove!) that no aliasing takes place.

In the contrary, we should try write our program such that to protect our variables from ever being aliased, and there is an easy way to achieve that:

**Rule 2.12.3.2** *Avoid the `&` operator.*

Depending on properties of a given variable, the compiler may then see that the address of the variable is never taken, and thus that the variable can't alias at all. In Section 13.2 we will see which properties of a variable or object may have influence on such decisions and how the **register** keyword can protect us from taking addresses inadvertently.

Later then, in Section 16.2, we will see how the **restrict** keyword allows to specify aliasing properties of pointer arguments, even if they have the same base type.

**12.4. Pointers to unspecific objects.** In some places we already have seen pointers that point to a valid address, but for which we do not control the type of the underlying object. C has invented some sort of *non-type* for such beasts, **void**. Its main purpose is to be used a fallback pointer type:

**Rule 2.12.4.1** *Any object pointer converts to and from **void\***.*

Observe that this only talks about pointers to objects, not pointers to functions. Not only that the conversion to **void\*** is well defined, it also is guaranteed to behave well with respect to the pointer value.

**Rule 2.12.4.2** *Converting an object pointer to **void\*** and then back to the same type is the identity operation.*

So the only thing that we loose when converting to **void\*** is the type information, the value remains intact.

**Rule 2.12.4.3 (avoid<sup>2</sup>\*)** Avoid **void\***.

It completely removes any type information that was associated to an address. Avoid it whenever you can. The other way around is much less critical, in particular if you have a C library call that returns a **void\***.

**void** as a type by itself shouldn't be used for variable declarations since it wouldn't lead to an object with which we could do anything.

**12.5. Implicit and explicit conversions.** We already have seen many places where a value of a certain type is *implicitly* converted to a value of a different type. *E.g.* in an initialization

```
1 unsigned big = -1;
```

The **int** value `-1` on the right is converted to an **unsigned** value, here **UINT\_MAX**. In most cases, C does the right thing, and we don't have to worry.

In other cases it is not as obvious (neither for you nor the compiler) so the intent must be made clearer. Notorious for such misunderstandings are narrow integer types. In the following let us assume that **int** has a width of 32 bit:

```
1 unsigned char highC = 1;
2 unsigned high highU = (highC << 31); // undefined behavior
```

Looks innocent, but isn't. The first line is ok, the value `1` is well in the range of **unsigned char**. In the second, the problem lays in the RHS. As we already had seen in Table 6, narrow types are converted before doing arithmetic on them. Here in particular `highC` is converted to **int** and the left shift operation is then performed on that type. By our assumption shifting by 31 bit shifts the `1` into the highest order bit, the sign bit. Thus the result of the expression on the right is undefined.

The details of all this are a bit arbitrary and more or less just historical artifacts. Don't dig too deep to understand them, just avoid them completely.

**Rule 2.12.5.1** *Chose your arithmetic types such that implicit conversions are harmless.*

In view of that, narrow types only make sense in very special circumstances:

- You have to save memory. You need to use a really big array of small values. Really big here means potentially millions or billions. In such a situation storing these values may gain you something.

- You use **char** for characters and strings. But then you wouldn't do arithmetic with them.
- You use **unsigned char** to inspect the bytes of an object. But then, again, you wouldn't do arithmetic with them.

**Rule 2.12.5.2** *Don't use narrow types in arithmetic.*

**Rule 2.12.5.3** *Use unsigned types whenever you may.*

Things become even more delicate if we move on to pointer types. There are only two forms of implicit conversions that are permitted for pointers to data, conversions from and to **void\*** and conversions that add a qualifier to the target type. Let's look at some examples.

```

1 float f = 37.0;           // conversion: to float
2 double a = f;             // conversion: back to double
3 float* pf = &f;          // exact type
4 float const* pdc = &f;    // conversion: adding a qualifier
5 void* pv = &f;            // conversion: pointer to void*
6 float* pfv = pv;          // conversion: pointer from void*
7 float* pd = &a;           // error: incompatible pointer type
8 double* pdv = pv;         // undefined behavior if used

```

The first two conversions that use **void\*** are already a bit tricky: we convert a pointer back and forth, but we watch that the target type of `pfv` is the same as `f` so everything works out fine.

Then comes the erroneous part. In the initialization of `pd` the compiler can protect us from a severe fault: assigning a pointer to a type that has a different size and interpretation can and will lead to serious damage. Any conforming compiler *must* give a diagnosis for this line. As you have by now well integrated Rule 0.1.2.3, you know that you should not continue until you have repaired such an error.

The last line is worse: it has an error, but that error is syntactically correct. The reason that this error might go undetected is that our first conversion for `pv` has striped the pointer from all type information. So in general the compiler can't know what is behind.

In addition to the implicit conversions that we have seen until now, C also allows to convert explicitly, so-called *casts*<sup>C.22</sup>. With a cast you are telling the compiler that you know better and that the type of the object behind the pointer is not what he might think and to shut up. In most use cases that I have come across in real life, the compiler was right and the programmer was wrong: even experienced programmers tend to abuse casts to hide poor design decisions concerning types.

**Rule 2.12.5.4** *Don't use casts.*

They deprive you of precious information and if you chose your types carefully you will only need them at very special occasions.

One such occasion is when you want to inspect the contents of an object on byte level. Constructing a **union** around an object as we have seen in Section 12.2 might not always be possible (or too complicated) so here we can go for a cast:

```

15 unsigned val = 0xAABBCCDD;
16 unsigned char* valp = (unsigned char*)&val;
17 for (size_t i = 0; i < sizeof val; ++i)
18     printf("byte[%zu]:_0x%.02hhX\n", i, valp[i]);

```

endianess.c

<sup>22</sup>A cast of an expression *X* to type *T* has the form `(T) X`. Think of it as in “to cast a spell”.

In that direction (from “pointer to object” to a “pointer to character type”) a cast is mostly harmless.

**12.6. Effective Type.** To cope with different views to the same object that pointers may provide, C has introduced the concept of *effective types*. It heavily restricts how an object can be accessed:

**Rule 2.12.6.1 (Effective Type)** *Objects must be accessed through their effective type or through a pointer to a character type.*

Because the effective type of a **union** variable is the **union** type and none of the member types, the rules for **union** members can be relaxed:

**Rule 2.12.6.2** *Any member of an object that has an effective **union** type can be accessed at any time, provided the byte representation amounts to a valid value of the access type.*

For all objects that have seen so far, it is easy to determine its effective type:

**Rule 2.12.6.3** *The effective type of a variable or compound literal is the type of its declaration.*

Later, we will see another category of objects that are a bit more involved.

Note that this rule makes no exception and that we can’t change the type of such a variable or compound literal.

**Rule 2.12.6.4** *Variables and compound literals must be accessed through their declared type or through a pointer to a character type.*

Also observe the asymmetry in all of this for character types. Any object can be seen as a composed of **unsigned char**, but not any array of **unsigned char** can be used through another type.

```
unsigned char A[sizeof(unsigned)] = { 9 };
// valid but useless, as most casts are
unsigned* p = (unsigned*)A;
// error, access with type that is neither the effective type nor a
// character type
printf("value_\\%u\\n", *p);
```

Here the access `*p` is an error and the program state is undefined afterwards. This is in strong contrast to our dealings with **union** above, see Section 12.2, where we actually could view a byte sequences as array of **unsigned char** or **unsigned**.

The reasons for such a strict rule are multiple. The very first motivation for introducing effective types in the C standard was to deal with aliasing as we have seen in Section 12.3. In fact, the Aliasing Rule 2.12.3.1 is derived from the Effective Type Rule 2.12.6.1. As long as there is no **union** involved, the compiler knows that we cannot access a **double** through a `size_t*`, and so he may *assume* that the objects are different.

**12.7. Alignment.** The inverse direction of pointer conversions (from “pointer to character type” to “pointer to object”) is not harmless at all, not only because of possible aliasing. This has to do with another property of C’s memory model, *alignment*<sup>C</sup>: objects of most non-character types can’t start at any arbitrary byte position, they usually start at a *word boundary*<sup>C</sup>. The alignment of a type then describes the possible byte positions at which an object of that type can start.

If we force some data to a false alignment, really bad things can happen. To see that have a look at the following code:

```

1 #include <stdio.h>
2 #include <inttypes.h>
3 #include <complex.h>
4 #include "crash.h"
5
6 void enable_alignment_check(void);
7 typedef complex double cdbl;
8
9 int main(void) {
10     enable_alignment_check();
11     /* An overlay of complex values and bytes. */
12     union {
13         cdbl val[2];
14         unsigned char buf[sizeof(cdbl[2])];
15     } toocomplex = {
16         .val = { 0.5 + 0.5*I, 0.75 + 0.75*I, },
17     };
18     printf("size/alignment: %zu/%zu\n",
19           sizeof(cdbl), _Alignof(cdbl));
20     /* Run over all offsets, and crash on misalignment. */
21     for (size_t offset = sizeof(cdbl); offset; offset /= 2) {
22         printf("offset\t%zu:\t", offset);
23         fflush(stdout);
24         cdbl* bp = (cdbl*)(&toocomplex.buf[offset]); // align!
25         printf("%g\t+%gI\t", creal(*bp), cimag(*bp));
26         fflush(stdout);
27         *bp *= *bp;
28         printf("%g\t+%gI", creal(*bp), cimag(*bp));
29         fputc('\n', stdout);
30     }
31 }

```

This starts with a declaration of a **union** similar to what we already have seen above. Again, we have a data object (of type **complex double** [2] in this case) that we overlay with an array of **unsigned char**. Besides that this part is a bit more complex, at a first glance there is no principal problem with it. But, if I execute this program on my machine I get:

Terminal					
0	~/.../modernC/code (master % u=) 14:45 <516>\$ ./crash				
1	size/alignment: 16/8				
2	offset	16:	0.75	+0.75I	0 +1.125I
3	offset	8:	0.5	+0I	0.25 +0I
4	offset	4:	Bus error		

The program crashes with an error indicated as *bus error*<sup>C</sup>, which is a shortcut for something like “data bus alignment error”. The real problem line is

		crash.c
24	cdbl* bp = (cdbl*)(&toocomplex.buf[offset]);	// align!

Here we see a pointer cast in the RHS, an **unsigned char\*** is converted to a **complex double\***. With the **for**-loop around it this cast is performed for byte offsets offset from the beginning of **toocomplex**. These are powers of 2: 16, 8, 4, 2, and 1. As you can see in the output above, it seems that **complex double** still works



well for alignments of half of its size, but then with an alignment of one fourth, the program crashes.

Some architectures are more tolerant to misalignment than others and we might have to force the system to error out on such a condition. We use the following function at the beginning to force crashing:

crash.c

```
enable_alignment_check: enable alignment check for i386 processors

Intel's i386 processor family is quite tolerant in accepting misalignment of data. This
can lead to irritating bugs when ported to other architectures that are not as tolerant.

This function enables a check for this problem also for this family or processors, such
that you can be sure to detect this problem early.

I found that code on Ygdrasil's blog: http://orchistro.tistory.com/206

void enable_alignment_check(void);
```

If you are interested in portable code (and if you are still here, you probably are) early errors in the development phase are really helpful.<sup>23</sup> So consider crashing a feature. See the mentioned blog entry for an interesting discussion on this topic.

In the above code example we also see a new operator, **alignof** (or **\_Alignof** if you don't include `stdalign.h`), that provides us with the alignment of a specific type. You will rarely find the occasion to use that in real live code.

`#include <stdalign.h>`

Another keyword can be used to force allocation at a specified alignment: **alignas** (respectively **\_Alignas**). Its argument can either be a type or expression. It can be useful where you know that your platform can perform certain operations more efficiently if the data is aligned in a certain way.

*E.g* to force alignment of a **complex** variable to its size and not half the size as we have seen above you could use

```
alignas(sizeof(complex double)) complex double z;
```

Or if you know that your platform has efficient vector instructions for **float**[4] arrays:

```
alignas(sizeof(float[4])) float fvec[4];
```

These operators don't help against the Effective Type Rule 2.12.6.1. Even with

```
alignas(unsigned) unsigned char A[sizeof(unsigned)] = { 9 };
```

the example on page 12.6 is invalid.

### 13. Allocation, initialization and destruction

So far, most objects that we handled in our programs have been *variables*, that is objects that are declared in a regular declaration with a specific type and an identifier that refers to it. Sometimes they then were defined at a different place in the code than they were declared, but even such a definition referred to them with type and identifier.

Another category of objects that we have seen less is only specified with a type but not with an identifier: *compound literals* as introduced in Section 5.4.4.

All such objects, variables or compound literals, have a *lifetime*<sup>C</sup> that depends on the syntactical structure of the program. They have an object lifetime and identifier visibility that either spans the whole program execution (global variables, global literals and

<sup>23</sup>For the code that is used inside that function, please consult the source code of `crash.h` to inspect it.

variables that are declared with **static**) or are bound to a block of statements inside a function.<sup>24</sup>

**13.1. malloc and friends.** For programs that have to handle growing collections of data these types of objects are too restrictive. To handle varying user input, web queries, large interaction graphs or other irregular data, big matrices or audio streams it is convenient to reclaim objects for storage on the fly and then release them, once they are not needed anymore. Such a scheme is called *dynamic memory allocation*<sup>C</sup>.

```
#include <stdlib.h>
```

The following set of functions, available with `stdlib.h`, has been designed to provide such an interface to allocatable objects:

```
1 #include <stdlib.h>
2 void* malloc(size_t size);
3 void free(void* ptr);
4 void* calloc(size_t nmemb, size_t size);
5 void* realloc(void* ptr, size_t size);
6 void* aligned_alloc(size_t alignment, size_t size);
```

Here the first two, **malloc** (memory allocate) and **free**, are by far the most prominent. As their names indicate, **malloc** creates an object for us on the fly, and **free** then annihilates it. The three other functions are specialized versions of **malloc**; **calloc** (clear allocate) sets all bits of the new object to 0, **realloc** may grow or shrink an object and **aligned\_alloc** ensures non-default alignment.

All these functions operate with **void\***, that is with pointers for which no type information is known. Being able to specify such a “non-type” for this series of functions is probably the *raison d’être* for the whole game with **void\*** pointers. By that, they become universally applicable to all types.

```
1 size_t length = livingPeople();
2 double* largeVec = malloc(length * sizeof *largeVec);
3 for (size_t i = 0; i < length; ++i) {
4     largeVec[i] = 0.0;
5 }
6 ...
7
8 free(largeVec);
```

Here we allocate a large vector of **double** one for each living person.<sup>[Exs 25]</sup> Because **malloc** knows nothing about the later use or type of the object, the size of the vector is specified in bytes. In the idiom given above we have specified the type information only once, as the pointer type for `largeVec`. By using **sizeof** `*largeVec` in the parameter for the **malloc** call, we ensure that we will allocate the right amount of bytes. Even if we change `largeVec` later to have type **size\_t\***, the allocation will adapt.

Another idiom that we will often encounter strictly takes the size of the type of the object that we want to create, namely an array of `length` elements of type **double**.

```
1 double* largeVec = malloc(sizeof(double[length]));
```

We already have been haunted by the introduction of “casts”, explicit conversions. It is important to note that the call to **malloc** stands as is, the conversion from **void\***, the return type of **malloc**, to the target type is automatic and doesn’t need any intervention.

**Rule 2.13.1.1** *Don’t cast the return of **malloc** and friends.*

<sup>24</sup>In fact, this is a bit of a simplification, we will see the gory details below.

[Exs 25] Don’t try this allocation but compute the size that would be needed on your platform. Is allocating such a vector feasible on your platform?

Not only is such a cast superfluous, doing an explicit conversion can even be counter-productive when we forget to include the header file `stdlib.h`. Older C compilers then `#include <stdlib.h>`

```

1  /* If we forget to include stdlib.h, many compilers
2     still assume: */
3  int malloc();           // wrong function interface!
4  ...
5  double* largeVec = (void*)malloc(sizeof(double[length]));
6                          |
7                          int <--
8                          |
9                          void* <--

```

suppose a return of `int` and trigger the wrong conversion from `int` to a pointer type. I have seen many crashes and subtle bugs triggered by that error, in particular in beginners code who's author have been subject to bad advice.

In the above code, as a next step we initialize the object that we just allocated. As from that point we view the large object that we allocated through a pointer to `double`, all actions that we can perform on this object are those that are allowed for this data type. But what we *must* do first, is initialize the object. Here, we do that by setting all elements to `0.0`.

**Rule 2.13.1.2** *Objects that are allocated through `malloc` are uninitialized.*

13.1.1. *A complete example with varying array size.* Let us now look at an example where using a dynamic array that is allocated with `malloc` brings us more flexibility than a simple array variable.

The following interface describes a circular buffer of `double` values called `circular`:

```

circular.h
circular: an opaque type for a circular buffer for double values

This data structure allows to add double values in rear and to take them out in front.
Each such structure has a maximal amount of elements that can be stored in it.

typedef struct circular circular;

```

```

circular.h
circular_append: Append a new element with value value to the buffer c.

Returns: c if the new element could be appended, 0 otherwise.

circular* circular_append(circular* c, double value);

```

```

circular.h
circular_pop: Remove the oldest element from c and return its value.

Returns: the removed element if it exists, 0.0 otherwise.

double circular_pop(circular* c);

```

The idea is that, starting with 0 elements, new elements can be appended to the buffer or dropped from the front, as long as the number of elements that are stored doesn't exceed a certain limit. The individual elements that are stored in the buffer can be accessed with the following function:

circular.h

`circular_element`: Return a pointer to position *pos* in buffer *c*.

**Returns:** a pointer to the *pos*' element of the buffer, 0 otherwise.

---

```
double* circular_element(circular* c, size_t pos);
```

Since our type `circular` will need to allocate and deallocate space for the circular buffer, we will need to provide consistent functions for initialization and destruction of instances of that type. This functionality is provided by two pairs functions:

circular.h

`circular_init`: Initialize a circular buffer *c* with maximally *max\_len* elements.

Only use this function on an uninitialized buffer.

Each buffer that is initialized with this function must be destroyed with a call to `circular_destroy`.

---

```
circular* circular_init(circular* c, size_t max_len);
```

circular.h

`circular_destroy`: Destroy circular buffer *c*.

*c* must have been initialized with a call to `circular_init`

---

```
void circular_destroy(circular* c);
```

circular.h

`circular_new`: Allocate and initialize a circular buffer with maximally *len* elements.

Each buffer that is allocated with this function must be deleted with a call to `circular_delete`.

---

```
circular* circular_new(size_t len);
```

circular.h

`circular_delete`: Delete circular buffer *c*.

*c* must have been allocated with a call to `circular_new`

---

```
void circular_delete(circular* c);
```

The first pair is to be applied to existing objects. They receive a pointer to such an object and ensure that space for the buffer is allocated or freed. The first of the second pair creates an object and initializes it; the last destroys such an object and then deallocates the memory space.

If we'd use regular array variables the maximum amount of elements that we can store in a `circular` would be fixed once we created such an object. We want to be more flexible such that this limit be raised or lowered by means of function `circular_resize` and the number of elements can be queried with `circular_getlength`.

`circular_resize`: Resize to capacity *max\_len*.

circular.h

```
circular* circular_resize(circular* c, size_t max_len);
```

`circular_getlength`: Return the number of elements stored.

circular.h

```
size_t circular_getlength(circular* c);
```

Then, with function `circular_element` it behaves like an array of double: calling it with a position within the current length, we obtain the address of the element that is stored in that position.

The hidden definition of the structure is as follows:

circular.c

```
5  /** @brief the hidden implementation of the circular buffer type
6  */
7  struct circular {
8      size_t start;    /**< position of element 0 */
9      size_t len;      /**< number of elements stored */
10     size_t max_len;   /**< maximum capacity */
11     double* tab;      /**< array holding the data */
12 };
```

The idea is that the pointer field `tab` will always point to an array object of length `max_len`. At a certain point in time the buffered elements will start at `start` and the number of elements stored in the buffer is maintained in field `len`. The position inside the table `tab` is computed modulo `max_len`.

The following table symbolizes one instance of this `circular` data structure, with `max_len=10`, `start=2` and `len=4`.

table index	0	1	2	3	4	5	6	7	8	9
buffer content	<del>tab</del>	<del>tab</del>	6.0	7.7	81.0	99.0	<del>tab</del>	<del>tab</del>	<del>tab</del>	<del>tab</del>
buffer position			0	1	2	3				

We see that the buffer content, the four numbers 6.0, 7.7, 81.0 and 99.0, are placed consecutively in the array object pointed to by `tab`.

The following scheme represents a circular buffer with the same four numbers, but the storage space for the elements wraps around.

table index	0	1	2	3	4	5	6	7	8	9
buffer content	81.0	99.0	<del>tab</del>	<del>tab</del>	<del>tab</del>	<del>tab</del>	<del>tab</del>	<del>tab</del>	6.0	7.7
buffer position	2	3							0	1

Initialization of such data structure needs to call `malloc` to provide memory for the `tab` field. Other than that it is straightforward:

circular.c

```

13 circular* circular_init(circular* c, size_t max_len) {
14     if (c) {
15         if (max_len) {
16             *c = (circular){
17                 .max_len = max_len,
18                 .tab = malloc(sizeof(double[max_len])),
19             };
20             // allocation failed
21             if (!c->tab) c->max_len = 0;
22         } else {
23             *c = (circular){ 0 };
24         }
25     }
26     return c;
27 }

```

Observe that this functions always checks the pointer parameter `c` for validity. Also, it guarantees to initialize all other fields to 0 by assigning compound literals in both distinguished cases.

The library function `malloc` can fail for different reasons. *E.g.* the memory system might be exhausted from previous calls to it, or the reclaimed size for allocation might just be too large. On general purpose systems that you are probably using for your learning experience such a failure will be rare (unless voluntarily provoked) but still is a good habit to check for it:

**Rule 2.13.1.3** `malloc` indicates failure by returning a null pointer value.

Destruction of such an object is even simpler: we just have to check for the pointer and then we may `free` the `tab` field unconditionally.

circular.c

```

29 void circular_destroy(circular* c) {
30     if (c) {
31         free(c->tab);
32         circular_init(c, 0);
33     }
34 }

```

The library function `free` has the friendly property that it accepts a null parameter and just does nothing in that case.

The implementation of some of the other functions uses an internal function to compute the “circular” aspect of the buffer. It is declared `static` such that it is only visible for those functions and such that it doesn’t pollute the identifier name space, see Rule 2.9.2.3.

circular.c

```

50 static size_t circular_getpos(circular* c, size_t pos) {
51     pos += c->start;
52     pos %= c->max_len;
53     return pos;
54 }

```

Obtaining a pointer to an element of the buffer is now quite simple.

circular.c

```

68 double* circular_element(circular* c, size_t pos) {
69     double* ret = 0;
70     if (c) {
71         if (pos < c->max_len) {
72             pos = circular_getpos(c, pos);
73             ret = &c->tab[pos];
74         }
75     }
76     return ret;
77 }

```

With all of that information, you should now be able to implement all but one of the function interfaces nicely.<sup>[Exs 26]</sup> The one that is more difficult is `circular_resize`. It starts with some length calculations, and then distinguishes the cases if the request would enlarge or shrink the table:

circular.c

```

92 circular* circular_resize(circular* c, size_t nlen) {
93     if (c) {
94         size_t len = c->len;
95         if (len > nlen) return 0;
96         size_t olen = c->max_len;
97         if (nlen != olen) {
98             size_t ostart = circular_getpos(c, 0);
99             size_t nstart = ostart;
100             double* otab = c->tab;
101             double* ntab;
102             if (nlen > olen) {

```

Here we have the naming convention of using `o` (old) as the first character of a variable name that refers to a feature before the change, and `n` (new) its value afterwards. The end of the function then just uses a compound literal to compose the new structure by using the values that have been found during the case analysis.

circular.c

```

138     }
139     *c = (circular){
140         .max_len = nlen,
141         .start = nstart,
142         .len = len,
143         .tab = ntab,
144     };
145 }
146 }
147 return c;
148 }

```

<sup>[Exs 26]</sup> Write implementations of the missing functions.

Let us now try to fill the gap in the code above and look into the first case of enlarging an object. The essential part of this is a call to **realloc**:

```

103     ntab = realloc(c->tab, sizeof(double[nlen]));
104     if (!ntab) return 0;
```

circular.c

For this call **realloc** receives the pointer to the existing object and the new size that the relocation should have. It either returns a pointer to the new object with the desired size or null. In the line immediately after we check the later case and terminate the function if it was not possible to relocate the object.

The function **realloc** has interesting properties:

- The returned pointer may or may not be the same as the argument. It is left to the discretion of the runtime system to realize if the resizing can be performed in place (if there is space available behind the object, *e.g.*) or if a new object must be provided.
- If the argument pointer and the returned one are distinct (that is the object has been copied) nothing has to be done (or even should) with the previous pointer, the old object is taken care of.
- As far as this is possible, existing content of the object is preserved:
  - If the object is enlarged, the initial part of the object that corresponds to the previous size is left intact.
  - If the object shrank, the relocated object has a content that corresponds to the initial part before the call.
- If 0 is returned, that is the relocation request could not be fulfilled by the runtime system, the old object is unchanged. So nothing is lost, then.

Now that we know that the newly received object has the size that we want, we have to ensure that `tab` still represents a circular buffer. If previously the situation has been as in the first table, above, that is the part that corresponds to the buffer elements is contiguous, we have nothing to do. All data is nicely preserved.

If our circular buffer wrapped around, we have to do some adjustments:

```

105     // two separate chunks
106     if (ostart+len > olen) {
107         size_t ulen = olen - ostart;
108         size_t llen = len - ulen;
109         if (llen <= (nlen - olen)) {
110             /* cpy the lower one up after the old end */
111             memcpy(ntab + olen, ntab,
112                 llen*sizeof(double));
113         } else {
114             /* mv the upper one up to the new end */
115             nstart = nlen - ulen;
116             memmove(ntab + nstart, ntab + ostart,
117                 ulen*sizeof(double));
118         }
119     }
```

circular.c



The following table illustrates the difference in contents between before and after the changes for the first subcase, namely that the lower part finds enough place inside the part that had been added.

table index	0	1	2	3	4	5	6	7	8	9			
old content	81.0	99.0	ᄀᄀᄀᄀ	ᄀᄀᄀᄀ	ᄀᄀᄀᄀ	ᄀᄀᄀᄀ	ᄀᄀᄀᄀ	ᄀᄀᄀᄀ	6.0	7.7			
old position	2	3							0	1			
new position	2	3							0	1	2	3	
new content	81.0	99.0	ᄀᄀᄀᄀ	ᄀᄀᄀᄀ	ᄀᄀᄀᄀ	ᄀᄀᄀᄀ	ᄀᄀᄀᄀ	ᄀᄀᄀᄀ	6.0	7.7	81.0	99.0	ᄀᄀᄀᄀ
table index	0	1	2	3	4	5	6	7	8	9	10	11	12

The other case, where the lower part doesn't fit into the newly allocated part is similar. This time the upper half of the buffer is shifted towards the end of the new table.

table index	0	1	2	3	4	5	6	7	8	9			
old content	81.0	99.0	ᄀᄀᄀᄀ	ᄀᄀᄀᄀ	ᄀᄀᄀᄀ	ᄀᄀᄀᄀ	ᄀᄀᄀᄀ	ᄀᄀᄀᄀ	6.0	7.7			
old position	2	3							0	1			
new position	2	3								0	1		
new content	81.0	99.0	ᄀᄀᄀᄀ	ᄀᄀᄀᄀ	ᄀᄀᄀᄀ	ᄀᄀᄀᄀ	ᄀᄀᄀᄀ	ᄀᄀᄀᄀ	6.0	7.7	81.0	99.0	ᄀᄀᄀᄀ
table index	0	1	2	3	4	5	6	7	8	9	10	11	12

The handling of both cases shows a subtle difference, though. The first is handled with **memcpy**, source and target elements of the copy operation can't overlap, so using it here is safe. For the other case, as we see in the example, the source and target elements may overlap and thus the use of the less restrictive **memmove** function is required.<sup>[Exs 27]</sup>

13.1.2. *Ensuring consistency of dynamic allocations.* As in both our code examples, **malloc** and **free** should always come in pairs. This mustn't necessarily be inside the same function, but in most case simple counting of the occurrence of both should give the same number:

**Rule 2.13.1.4** For every **malloc** there must be a **free**.

If not this could indicate a **memory leak**<sup>C</sup>, a loss of allocated objects. This could lead to resource exhaustion of your platform, showing itself in low performance or random crashes.

**Rule 2.13.1.5** For every **free** there must be a **malloc**.

The memory allocation system is meant to be simple, thus **free** is only allowed for pointers that have been allocated with **malloc** or that are null.

**Rule 2.13.1.6** Only call **free** with pointers as they are returned by **malloc**.

They must not

- point to an object that has been allocated by other means, that is a variable or a compound literal,
- yet have been freed,
- only point to a smaller part of the allocated object.

Otherwise your program will crash. Seriously, this will completely corrupt the memory of your program execution, one of the worst types of crashes that you can have. Be careful.

LISTING 2.2. An example for shadowing by local variables

```

1 void squareIt(double* p) {
2     *p *= *p;
3 }
4 int main(void) {
5     double x = 35.0;
6     double* xp = &x;
7     {
8         squareIt(&x); /* referring to double x */
9         ...
10    int x = 0;      /* shadow double x */
11    ...
12    squareIt(xp);  /* valid use of double x */
13    ...
14    }
15    ...
16    squareIt(&x);  /* referring to double x */
17    ...
18 }

```

**13.2. Storage duration, lifetime and visibility.** We have already seen in different places that visibility of an identifier and accessibility of the object to which it refers are not the same thing. As a simple example take the variable(s) *x* in Listing 2.2:

Here the visibility scope of the identifier *x* that is declared in Line 5 starts from that line and goes to the end of the function **main**, but with a noticeable interruption: from Line 10 to 14 this visibility is *shadowed*<sup>C</sup> by another variable, also named *x*.

**Rule 2.13.2.1** *Identifiers only have visibility inside their scope, starting at their declaration.*

**Rule 2.13.2.2** *The visibility of an identifier can be shadowed by an identifier of the same name in a subordinate scope.*

We also see that visibility of an identifier and usability of the object that it represents, are not the same thing. First, the **double** *x* object is used by all calls to `squareIt`, although the identifier *x* is not visible at the point of definition of the function. Then, in Line 12 we pass the address of the **double** *x* variable to the function `squareIt` although the identifier is shadowed, there.

Another example concerns declarations that are tagged with the storage class **extern**. These always designate an object of static storage duration that is expected to be defined at file scope<sup>28</sup>, see Listing 2.3.

This program has three declarations for variables named *i*, but only two definitions: the declaration and definition in Line 6 shadows the one in Line 3. In turn declaration Line 8 shadows Line 6, but it refers to the same object as the object defined in Line 3.<sup>[Exs 29]</sup>

**Rule 2.13.2.3** *Every definition of a variable creates a new distinct object.*

So in the following the **char** arrays *A* and *B* identify distinct objects, with distinct addresses; the expression `A == B` must always be false.

[Exs 27] Implement shrinking of the table: it is important to reorganize the table contents before calling `realloc`.

<sup>28</sup>In fact, such an object can be defined at file scope in another translation unit.

[Exs 29] Which value is printed by this program?

LISTING 2.3. An example for shadowing by an **extern** variable

```

1  #include <stdio.h>
2
3  unsigned i = 1;
4
5  int main(void) {
6      unsigned i = 2;          /* a new object */
7      if (i) {
8          extern unsigned i;    /* an existing object */
9          printf("%u\n", i);
10     } else {
11         printf("%u\n", i);
12     }
13 }

```

```

1  char const A[] = { 'e', 'n', 'd', '\0', };
2  char const B[] = { 'e', 'n', 'd', '\0', };
3  char const* c = "end";
4  char const* d = "end";
5  char const* e = "friend";
6  char const* f = (char const[]) { 'e', 'n', 'd', '\0', };
7  char const* g = (char const[]) { 'e', 'n', 'd', '\0', };

```

But how many distinct array objects are there in total? It depends, the compiler has a lot of choices:

**Rule 2.13.2.4** *Read-only object literals may overlap.*

In the above example we have three string literals and two compound literals. These are all object literals and they are read-only: strings literals are read-only by definition, the two compound literals are **const**-qualified. Four of them have exactly the same base type and content ('e', 'n', 'd', '\0'), and so the four pointers c, d, f and g may all be initialized to the same address of one **char** array. The compiler may even save more memory: this address may just be &e[3], by using the fact that “end” appears at the end of “friend”.

As we have seen from the examples above, the usability of an object is not only a lexical property of an identifier or of the position of definition (for literals) but also depends on the state of execution of the program. The *lifetime*<sup>C</sup> of an object has a starting point and an end point:

**Rule 2.13.2.5** *Objects have a lifetime outside of which they can't be accessed.*

**Rule 2.13.2.6** *Referring to an object outside of its lifetime has undefined behavior.*

How this start and end point of an object are defined depends on the tools that we use to create it. We distinguish four different *storage durations*<sup>C</sup> for objects in C, *static*<sup>C</sup> when it is determined at compile time, *automatic*<sup>C</sup> when it is automatically determined at run time, *allocated*<sup>C</sup> when it is explicitly determined by function calls **malloc** and friends, and *thread*<sup>C</sup> when it is bound to a certain thread of execution.

For the first three types we already have seen a lot of examples. Thread storage duration (**\_Thread\_local** or **thread\_local**) is related to C's thread API, which we only will see later in Section 19.

Allocated storage duration, is straightforward: the lifetime of such an object starts from the corresponding call to **malloc**, **calloc**, **realloc** or **aligned\_alloc** that creates it. It ends with a call to **free** or **realloc** that destroys it, or, if no such call is issued with the end of the program execution.

The two other cases of storage duration need more explanation, and so we will discuss them in more length below.

13.2.1. *Static storage duration.* Objects with static storage duration can be defined by two different means:

- Objects that are *defined* in file scope. Variables or compound literals can have that property.
- Variables that are declared inside a function block and that have the storage class specifier **static**.

Such objects have a lifetime that is the whole program execution. Because they are considered alive before any application code is executed, they can only be initialized with expressions that are known at compile time or can be resolved by the system's process startup procedure.

```

1  double A = 37;
2  double* p
3    = &(double){ 1.0, };
4  int main(void) {
5      static B;
6  }
```

This defines four objects of static storage duration, those identified with A, p and B and a compound literal defined in Line 3. Three of them have type **double**, one has type **double\***.

All four objects are properly initialized from the start; three of them are initialized explicitly, B is initialized implicitly with 0.

**Rule 2.13.2.7** *Objects with static storage duration are always initialized.*

The example of B shows that an object with a lifetime that is the whole program execution isn't necessarily visible in the whole program. The **extern** example above also shows that an object with static storage duration that is defined elsewhere can become visible inside a narrow scope.

13.2.2. *Automatic storage duration.* This is the most complicated case in nature: rules for automatic storage duration are implicit and therefore need the most explanation. There are several cases of objects that can be defined explicitly or implicitly that fall under this category:

- any block scope variables that are not declared **static**, that are declared as **auto** (the default), or **register**
- block scope compound literals
- some temporary objects that are returned by function calls.

The simplest and most current case for the lifetime of automatic objects

**Rule 2.13.2.8** *Unless they are VLA or temporary objects, automatic objects have a lifetime corresponding to the execution of their block of definition.*

So this rule covers automatic variables and compound literals that are declared inside functions. Such objects of automatic storage duration have a big advantage for optimization: the compiler usually sees the full usage of such a variable and may by that decide if it may alias. This is where the difference between the **auto** and **register** variables comes into play:

**Rule 2.13.2.9** *The `&` operator is not allowed for variables declared with **register**.*

With that, we can't inadvertently violate Rule 2.12.3.2 and take the address of a **register** variable. As a simple consequence we get:

**Rule 2.13.2.10** *Variables declared with **register** can't alias.*

So with **register** variable declarations the compiler can be forced to tell us where there would eventually be some optimization potential.

**Rule 2.13.2.11** *Declare local variables in performance critical code as **register**.*

Let's get back to rule 2.13.2.8. It is quite particular if you think of it: the lifetime of such an object does already start when its scope of definition is entered and not, as one would perhaps expect, later, when its definition is first met during execution.

To note the difference let us look at Listing 2.4, which is in fact a variant of an example that can be found in the C standard document.

LISTING 2.4. A contrived example for the use of a compound literal

```

3 void fgoto(unsigned n) {
4     unsigned j = 0;
5     unsigned* p = 0;
6     unsigned* q;
7     AGAIN:
8     if (p) printf("%u: p and q are %s, *p is %u\n",
9                 j,
10                (q == p) ? "equal" : "unequal",
11                *p);
12     q = p;
13     p = &((unsigned){ j, });
14     ++j;
15     if (j <= n) goto AGAIN;
16 }
```

We will be particularly interested in the lines printed if this function is called as `fgoto(2)`. On my computer the output looks as

	Terminal
0	1: p and q are unequal, *p is 0
1	2: p and q are equal, *p is 1

Admittedly, this code is a bit contrived. It uses a new construct that we haven't yet seen in action, **goto**. As the name indicates this is a *jump statement*<sup>C</sup>. In this case it instructs to continue execution at the place of *label*<sup>C</sup> AGAIN. Later we will see contexts where using **goto** makes a bit more sense. The demonstrative purpose here is just to jump over the definition of the compound literal.

So let us look what happens with the **printf** call during the execution. For `n == 2` execution meets the corresponding line three times, but because `p` is 0, initially, at the first passage the **printf** call itself is skipped. The values of our three variables in that line is:

Here we see that for `j==2` pointers `p` and `q` hold addresses that are obtained at different iterations. So why then in my printout above I have that both addresses are equal? Is this just a coincidence? Or is there even undefined behavior because I am using the compound literal lexically at a place before it is defined?

j	p	q	printf
0	0	indetermined	skipped
1	addr of literal of $j = 0$	0	printed
2	addr of literal of $j = 1$	addr of literal of $j = 0$	printed

In fact the C standard prescribes that the output that is shown above *must* be produced. In particular, for  $j==2$  the values of  $p$  and  $q$  are equal and valid, and the value of the object they are pointing to is 1. Or stated otherwise:

In the example above, the use of  $*p$  is well defined, although lexically the evaluation of  $*p$  precedes the definition of the object. Also, there is exactly one such compound literal, and therefore the addresses are equal for  $j==2$ .

**Rule 2.13.2.12** *For an object that is not a VLA, lifetime starts when the scope of the definition is entered, and it ends when that scope is left.*

**Rule 2.13.2.13** *Initializers of automatic variables and compound literals are evaluated each time the definition is met.*

In fact, in our example the compound literal is visited three times, and set to values 0, 1 and 2 in turn.

For VLA the lifetime is given by a different rule, namely

**Rule 2.13.2.14** *For a VLA, lifetime starts when the definition is encountered, ends when the visibility scope is left.*

So for a VLA, our strange trick from above using **goto** would not be valid: we are not allowed to use the pointer to a VLA in code that precedes the definition, even if we still are inside the same block. The reason for this special treatment of VLA is that their size is a runtime property and therefore the space for it simply can't be allocated when the block of the declaration is entered.

The other case that has a special treatment are chimeras: sort-of-objects that are the return values of functions. As you know now, functions normally return values and not objects. There is one exceptions if the return type *contains* an array type, such as here:

```

1 struct demo { unsigned ory[1]; };
2 struct demo mem(void);
3
4 printf("mem().ory[0]_is_%u\n", mem().ory[0]);

```

The only reason that objects with temporary lifetime exist in C is to be able to access fields of such a function return value. Don't use them for anything else:

**Rule 2.13.2.15** *Objects of temporary lifetime are read-only.*

**Rule 2.13.2.16** *Temporary lifetime ends at the end of the enclosing full expression.*

**13.3. Initialization.** In Section 5.3 we already have seen the importance of initialization. It is crucial to guarantee that a program starts in a well defined state and stays so during all execution. The storage duration of an object determines how it is initialized.

**Rule 2.13.3.1** *Objects of static or thread storage duration are initialized per default.*

As you probably recall, such a default initialization is the same as to initialize all fields of an object by 0. In particular, default initialization works well for base types that might have a non-trivial representation for their "0" value, namely pointers and floating point types.

For other objects, automatic or allocated, we must do something

**Rule 2.13.3.2** *Objects of automatic or allocated storage duration must be initialized explicitly.*

The simplest way to achieve initialization are initializers, they put variables and compound literals in a well defined state as soon as they become visible.

For arrays that we allocate as VLA or through dynamic allocation this is not possible, so we have to provide initialization through assignment. In principle we could do this manually each time we allocate such an object, but such code becomes difficult to read and to maintain, because such initialization parts may then visually separate definition and use. The easiest way to avoid this is to encapsulate initialization into functions:

**Rule 2.13.3.3** *Systematically provide an initialization function for each of your data types.*

Here, the emphasis lays on “systematically”: you should have a consistent convention how such initializing functions should work and how they should be named. To see that let us go back to `rat_init`, the initialization function for our `rat` data type. It implements a specific API for such functions:

- For a type `toto` the initialization function is named `toto_init`.
- The first argument to such a `_init` function is the pointer to the object that is to be initialized.
- If that pointer to object is null, the function does nothing.
- Other arguments can be provided to pass initial values for certain fields.
- The function returns the pointer to the object that it received or 0 if an error occurred.

With such properties such a function can be used easily in an initializer for a pointer:

```
1 rat const* myRat = rat_init(malloc(sizeof(rat)), 13, 7);
```

Observe that this has several advantages:

- If the call to `malloc` fails by returning 0, the only effect is that `myRat` is initialized to 0. Thus `myRat` is always in a well defined state.
- If we don't want the object to be changed afterwards we can qualify the pointer target as `const` from the start. All modification of the new object happens inside the initialization expression on the right side.

Since such initialization can then appear in many places we can also encapsulate this into another function:

```
1 rat* rat_new(long long numerator,
2             unsigned long long denominator) {
3     return rat_init(malloc(sizeof(rat)),
4                     numerator,
5                     denominator);
6 }
```

The initialization using that function becomes

```
1 rat const* myRat = rat_new(13, 7);
```

For macro addicts as myself we can even easily define a type generic macro that does such an encapsulation once and for all

```
1 #define P99_NEW(T, ...) T ## _init(malloc(sizeof(T)), __VA_ARGS__
    )
```

With this we could have written the initialization above as

```
1 rat const* myRat = P99_NEW(rat, 13, 7);
```

Which has the advantage to be at least as readable as the `rat_new` variant, but avoids the additional declaration of such a function for all types that we define.

Such macro definitions are frowned upon by many, so many projects would probably not accept this as a general strategy, but you should at least be aware that such a possibility exists. It uses two features of macros that we have not yet encountered:

- Concatenation of tokens is achieved with the `##` operator. Here `T ## _init` just melts the argument `T` and `_init` into one token: with `rat` this produces `rat_init`, with `toto` this produces `toto_init`.
- The construct `...` provides an argument list of variable length. The whole set of arguments that is passed after the first is accessible inside the macro expansion as `__VA_ARGS__`. By that we can pass any number of arguments as required by the corresponding `_init` function to `P99_NEW`.

If we have to initialize arrays by means of a `for`-loop things get even uglier. Here also it is easy to encapsulate with a function:

```
1 rat* rat_vinit(size_t n, rat p[n]) {
2     if (p)
3         for (size_t i = 0; i < n; ++i)
4             rat_init(p+i, 0, 1);
5     return p;
6 }
```

With such a function, again, initialization becomes straightforward:

```
1 rat* myRatVec = rat_vinit(44, malloc(sizeof(rat[44])));
```

Here an encapsulation into a function is really better, since repeating the size may easily introduce errors

```
1 rat* rat_vnew(size_t size) {
2     return rat_vinit(size, malloc(sizeof(rat[size])));
3 }
```

**13.4. Digression: a machine model.** Up to now, we mostly argued about C code from within, using the internal logic of the language to describe what was going on. This section here is an optional digression that deviates from that. If you really can't bear it, yet, you may skip over, otherwise remember Rule B and dive.

We will see how a typical compiler translates C to a language that is closer to a specific machine, why certain choices for such a translation are made, and how these choices reflect back into the definition of C. To do so we have to have an idea (a *model*) about how computers actually work.

Traditionally, computer architectures were described with the von Neumann model<sup>30</sup>. In this model a processing unit has a finite number of hardware *registers* that can hold integer values, a *main memory* that holds the program as well as data and that is linearly addressable, and a finite *instruction set* that describes the operations that can be done with these components.

<sup>30</sup>Invented around 1945 by J. Presper Eckert and John William Mauchly for the ENIAC project; first described by John von Neumann (1903 – 1957, also Neumann János Lajos and Johann Neumann von Margitta) one of the pioneers of modern science, in von Neumann [1945].



The intermediate programming language family that is usually used to describe machine instructions as they are understood by your CPU are called *assembler*<sup>C</sup>, and it still pretty much builds upon the von Neumann model. There is not one unique assembler language (as *e.g.* C that is valid for all platforms) but a whole set of *dialects* that take different particularities into account: of the CPU, the compiler or the operating system. The assembler that we use in the following is the one used by the `gcc` compiler for the `x86_64` processor architecture.<sup>[Exs 31]</sup> If you don't know what that means, don't worry, this is just an example of one such architecture.

Listing 2.5 shows an assembler print out for the function `fgoto` from Listing 2.4. Such assembler code operates with *instructions*<sup>C</sup> on hardware registers and memory locations. *E.g.* the line `movl $0, -16(%rbp)` stores (“moves”) the value 0 to the location in memory that is 16 bytes below the one indicated by register `%rbp`. The assembler program also contains *labels*<sup>C</sup> that identify certain points in the program. *E.g.* `fgoto` is *entry point*<sup>C</sup> of the function and `.L_AGAIN` is the counterpart in assembler of the `goto` label `AGAIN` in C.

As you probably have guessed, the text on the right after the `#` character are comments that try to link individual assembler instructions to their C counterparts.

This assembler function uses hardware registers `%eax`, `%ecx`, `%edi`, `%edx`, `%esi`, `%rax`, `%rbp`, `%rcx`, `%rdx`, and `%rsp`. This is much more than the original von Neumann machine had, but the main ideas are still present: we have some general purpose registers that are used to represent values from our data, such as `j` or `p`, of our C program. Two others have a very special role, `%rbp` (base pointer) and `%rsp` (stack pointer).

The function disposes of a reserved area in memory, often called *The Stack*<sup>C</sup>, that holds its local variables and compound literals. The “upper” end of that area is designated by the `%rbp` register, and the objects are accessed with negative offsets relative to that register. *E.g.* the variable `n` is found from position `-36` before `%rbp` encoded as `-36(%rbp)`. The following table represents the layout of this memory chunk that is reserved for function `fgoto`, and the values that are stored there at three different points of the execution of the function.

... <code>printf</code>		<code>fgoto</code>							caller ...
position		-48 ...	-36 ...	-28 ...	-24 ...	-16 ...	-8 ...	-4 ...	<code>rbp ...</code>
meaning			<code>n</code>	<code>cmp_lit</code>	<code>q</code>	<code>p</code>		<code>j</code>	
after init	<code>%d</code>	<code>%d</code>	2	<code>%d</code>	<code>%d</code>	0	<code>%d</code>	0	
after iter 0	<code>%d</code>	<code>%d</code>	2	0	0	<code>rbp-28</code>	<code>%d</code>	1	
after iter 1	<code>%d</code>	<code>%d</code>	2	1	<code>rbp-28</code>	<code>rbp-28</code>	<code>%d</code>	2	

This example is of particular interest to learn about automatic variables, and how these are set up, when execution enters the function. On this particular machine when entering `fgoto`, three registers hold information for this call: `%edi` holds the function argument, `n`, `%rbp` points to the base address of the calling function and `%rsp` points to the top address in memory where this call to `fgoto` may store its data.

Now let us consider how the above assembler code sets up things. Right at the start `fgoto` executes three instructions to set up its “world” correctly. It saves `%rbp` because it needs this register for its own purpose, it moves the value from `%rsp` to `%rbp` and then decrements `%rsp` by 48. Here, 48 is the number of bytes that the compiler has computed for all automatic objects that that `fgoto` needs. Because of this simple type of setup, the space that is reserved by that procedure is not initialized but filled with garbage. In the three following instructions three of the automatic objects are then initialized (`n`, `j` and `p`), but others remain uninitialized until later.

After having done such a setup, the function is ready to go. In particular it can easily call another function itself: `%rsp` now points to the top of a new memory area that such a

[Exs 31] Find out which compiler arguments produce assembler output for your platform.

LISTING 2.5. An assembler version of the `fgoto` function

```

10      .type    fgoto, @function
11  fgoto:
12      pushq    %rbp                # save base pointer
13      movq     %rsp, %rbp          # load stack pointer
14      subq     $48, %rsp           # adjust stack pointer
15      movl     %edi, -36(%rbp)      # fgoto#0 ==> n
16      movl     $0, -4(%rbp)        # init j
17      movq     $0, -16(%rbp)       # init p
18  .L_AGAIN:
19      cmpq     $0, -16(%rbp)       # if (p)
20      je       .L_ELSE
21      movq     -16(%rbp), %rax      # p ==> rax
22      movl     (%rax), %edx        # *p ==> edx
23      movq     -24(%rbp), %rax     # ( == q)?
24      cmpq     -16(%rbp), %rax     # (p == )?
25      jne      .L_YES
26      movl     $.L_STR_EQ, %eax    # yes
27      jmp      .L_NO
28  .L_YES:
29      movl     $.L_STR_NE, %eax    # no
30  .L_NO:
31      movl     -4(%rbp), %esi      # j ==> printf#1
32      movl     %edx, %ecx         # *p ==> printf#3
33      movq     %rax, %rdx        # eq/ne ==> printf#2
34      movl     $.L_STR_FRMT, %edi  # frmt ==> printf#0
35      movl     $0, %eax          # clear eax
36      call     printf
37  .L_ELSE:
38      movq     -16(%rbp), %rax     # p ==|
39      movq     %rax, -24(%rbp)     # ==> q
40      movl     -4(%rbp), %eax     # j ==|
41      movl     %eax, -28(%rbp)    # ==> cmp_lit
42      leaq     -28(%rbp), %rax    # &cmp_lit ==|
43      movq     %rax, -16(%rbp)    # ==> p
44      addl     $1, -4(%rbp)       # ++j
45      movl     -4(%rbp), %eax     # if (j
46      cmpl     -36(%rbp), %eax    # <= n)
47      jbe      .L_AGAIN          # goto AGAIN
48      leave
49      ret                        # return

```

called function can use. This can be seen in the middle part, after the label `.L_NO`. This part implements the call to `printf`: it stores the four arguments that the function is supposed to receive in registers `%edi`, `%esi`, `%ecx`, `%rdx`, in that order, clears `%eax`, and then calls the function.

To summarize, the setup of a memory area for the automatic objects (without VLA) of a function only needs very few instructions, regardless how many such automatic objects are effectively used by the function. If the function had more, the magic number 48 from above would just have to be modified to the new size of the area.

As a consequence of the way this is done,

- automatic objects are usually available from the start of a function or scope
- initialization of automatic *variables* is not enforced.

LISTING 2.6. An optimized assembler version of the `fgoto` function

```

12      .type    fgoto, @function
13  fgoto:
14      pushq    %rbp                # save base pointer
15      pushq    %rbx                # save rbx register
16      subq     $8, %rsp            # adjust stack pointer
17      movl     %edi, %ebp          # fgoto#0 ==> n
18      movl     $1, %ebx            # init j, start with 1
19      xorl     %ecx, %ecx          # 0 ==> printf#3
20      movl     $.L_STR_NE, %edx    # "ne" ==> printf#2
21      testl    %edi, %edi          # if (n > 0)
22      jne      .L_N_GT_0
23      jmp      .L_END
24  .L_AGAIN:
25      movl     %eax, %ebx          # j+1 ==> j
26  .L_N_GT_0:
27      movl     %ebx, %esi          # j ==> printf#1
28      movl     $.L_STR_FRMT, %edi  # frmt ==> printf#0
29      xorl     %eax, %eax          # clear eax
30      call     printf
31      leal     1(%rbx), %eax       # j+1 ==> eax
32      movl     $.L_STR_EQ, %edx    # "eq" ==> printf#2
33      movl     %ebx, %ecx          # j ==> printf#3
34      cmpl     %ebp, %eax          # if (j <= n)
35      jbe      .L_AGAIN            # goto AGAIN
36  .L_END:
37      addq     $8, %rsp            # rewind stack
38      popq     %rbx                # restore rbx
39      popq     %rbp                # restore rbp
40      ret

```

This maps well the rules for the lifetime and initialization of automatic objects in C.

Now the assembler output from above is only half of the story, at most. It was produced without optimization, just to show the principle assumptions that can be made for such code generation. When using optimization the `as-if` Rule 1.5.0.7 allows to reorganize the code substantially. With full optimization my compiler produces something like Listing 2.6.

As you can see the compiler has completely restructured the code. This code just reproduces the *effects* that the original code had, namely its output is the same as before. But it doesn't use objects in memory, doesn't compare pointers for equality or even has any trace of the compound literal. *E.g.* it doesn't implement the iteration for `j=0` at all. This iteration has no effects, so it is simply omitted. Then, for the other iterations it distinguishes a version with `j=1`, where the pointers `p` and `q` of the C program are known to be different. Then, the general case just has to increment `j` and to set up the arguments for `printf` accordingly.<sup>[Exs 32][Exs 33]</sup>

All we have seen here has been code that didn't use VLA. These change the picture, because the trick that simply modified `%rsp` by a constant doesn't work if the needed memory is not of constant size. For a VLA the program has to compute the size with the knowledge at the very point of definition, to adjust `%rsp` accordingly, there, and then to undo that modification of `%rsp` once execution leaves the scope of definition.

[Exs 32] Using the fact that `p` is actually assigned the same value over and over, again, write a C program that gets closer to how the optimized assembler version looks like.

[Exs 33] Even the optimized version leaves room for improvement, the inner part of the loop can still be shortened. Write a C program that explores this potential when compiled with full optimization.

## 14. More involved use of the C library

**14.1. Text processing.** Now that we know about pointers and how they work, we will revisit some of the C library functions for text processing. As a first example consider the following program that reads a series of lines with numbers from **stdin** and writes these same numbers in a normalized way to **stdout**, namely as comma separated hexadecimal numbers.

numberline.c

```

197 int fprintrnumbers_opt(FILE*restrict stream,
198                       char const form[restrict static 1],
199                       char const sep[restrict static 1],
200                       size_t len, size_t nums[restrict len]) {
201     if (!stream) return -EFAULT;
202     if (len && !nums) return -EFAULT;
203     if (len > INT_MAX) return -EOverflow;
204
205     int err = errno;
206     size_t const seplen = strlen(sep);
207
208     size_t tot = 0;
209     size_t mtot = len*(seplen+10);
210     char* buf = malloc(mtot);
211
212     if (!buf) return error_cleanup(ENOMEM, err);
213
214     for (size_t i = 0; i < len; ++i) {
215         tot += sprintf(&buf[tot], form, nums[i]);
216         ++i;
217         if (i >= len) break;
218         if (tot > mtot-20) {
219             mtot *= 2;
220             char* nbuf = realloc(buf, mtot);
221             if (buf) {
222                 buf = nbuf;
223             } else {
224                 tot = error_cleanup(ENOMEM, err);
225                 goto CLEANUP;
226             }

```

This program splits the job in three different tasks: `fgetline` to read a line of text, `numberline` that splits such a line in a series of numbers of type `size_t`, and `fprintrnumbers` to print them.

At the heart is the function `numberline`. It splits the `lbuf` string that it receives in numbers, allocates an array to store them and also returns the count of these numbers through the pointer argument `np` if that is provided.

numberline.c

numberline: interpret string *lbuf* as a sequence of numbers represented with *base*

**Returns:** a newly allocated array of numbers as found in *lbuf*

**Parameters:**

<i>lbuf</i>	is supposed to be a string
<i>np</i>	if non-null, the count of numbers is stored in <i>*np</i>
<i>base</i>	value from 0 to 36, with the same interpretation as for <b>strtoull</b>

**Remarks:** The caller of this function is responsible to **free** the array that is returned.

```
size_t* numberline(size_t size, char const lbuf[restrict size],
                  size_t* restrict np, int base);
```

That function itself is split into two parts, that perform quite different tasks; one that performs the task of interpretation of the line, `numberline_inner`. The other other, `numberline` itself, is just a wrapper around the first that verifies or ensures the prerequisites for the first.

Function `numberline_inner` puts the C library function **strtoull** in a loop that collects the numbers and returns a count of them.

numberline.c

```
97 static
98 size_t numberline_inner(char const* restrict act,
99                        size_t numb[restrict], int base) {
100     size_t n = 0;
101     for (char* next = 0; act[0]; act = next) {
102         numb[n] = strtoull(act, &next, base);
103         if (act == next) break;
104         ++n;
105     }
106     return n;
107 }
```

Now we see the use of the second parameter of **strtoull**. Here, it is the address of the variable `next`, and `next` is used to keep track of the position in the string that ends the number. Since `next` is a pointer to **char**, the argument to **strtoull** is a pointer to a pointer to **char**.

Suppose **strtoull** is called as **strtoull**("0789a", &next, base). According to the value of the parameter `base` that string is interpreted differently. If for example `base` has value 10, the first non-digit is the character 'a' at the end.

base	digits	number	*next
8	2	7	'8'
10	4	789	'a'
16	5	30874	'\0'
0	2	7	'8'

There are two conditions that may end the parsing of the line that `numberline_inner` receives.

- `act` points to a string termination, namely to a 0 character.

- Function `strtoull` doesn't find a number, in which case `next` is set to the value of `act`.

These two conditions are found as the controlling expression of the `for`-loop and as `if-break` condition inside.

Observe that the C library function `strtoull` has a historic weakness: the first argument has type `char const*` whereas the second has type `char**`, without `const`-qualification. This is why we had to type `next` as `char*` and couldn't use `char const*`. As a result of a call to `strtoull` we could inadvertently modify a read-only string and crash the program.

**Rule 2.14.1.1** *The string `strto...` conversion functions are not `const`-safe.*

Now, the function `numberline` itself provides the glue around `numberline_inner`:

- If `np` is null, it is set to point to an auxiliary.
- The input string is checked for validity.
- An array with enough elements to store the values is allocated and tailored to the appropriate size, once the correct length is known.

```

109 size_t* numberline(size_t size, char const lbuf[restrict size],
110                   size_t*restrict np, int base){
111     size_t* ret = 0;
112     size_t n = 0;
113     /* Check for validity of the string, first. */
114     if (memchr(lbuf, 0, size)) {
115         /* The maximum number of integers encoded. To see that this
116            may be as much look at the sequence 08 08 08 08 ... */
117         ret = malloc(sizeof(size_t)[1+(2*size)/3]);
118
119         n = numberline_inner(lbuf, ret, base);
120
121         /* Supposes that shrinking realloc will always succeed. */
122         size_t len = n ? n : 1;
123         ret = realloc(ret, sizeof(size_t)[len]);
124     }
125     if (np) *np = n;
126     return ret;
127 }

```

We use three functions from the C library: `memchr`, `malloc` and `realloc`. As in previous examples a combination of `malloc` and `realloc` ensures that we have an array of the necessary length.

The call to `memchr` returns the address of the first byte that has value 0, if there is any, or `(void*) 0` if there is none. Here, this is just used to check that within the first `size` bytes there effectively is a 0-character. By that it guarantees that all the string functions that are used underneath (in particular `strtoull`) operate on a 0-terminated string.

With `memchr` we encounter another problematic interface. It returns a `void*` that potentially points into a read-only object.

**Rule 2.14.1.2** *The `memchr` and `strchr` search functions are not `const`-safe.*

In contrast to that, functions that return an index position within the string would be safe.

**Rule 2.14.1.3** The **strspn** and **strcspn** search functions are **const-safe**.

Unfortunately, they have the disadvantage that they can't be used to check if a **char**-array is in fact a string or not. So they can't be used, here.

Now, let us look into the second function in our example:

numberline.c

fgetline: read one text line of at most size-1 bytes.

The `'\n'` character is replaced by 0.

**Returns:** `s` if an entire line was read successfully. Otherwise, 0 is returned and `s` contains a maximal partial line that could be read. `s` is null terminated.

---

```
char* fgetline(size_t size, char s[restrict size],
              FILE* restrict stream);
```

This is quite similar to the C library function **fgets**. The first difference is the interface: the parameter order is different and the `size` parameter is a **size\_t** instead of an **int**. As **fgets**, it returns a null pointer if the read from the stream failed. Thereby the end-of-file condition is easily detected on `stream`.

More important is that `fgetline` handles another critical case more gracefully. It detects if the next input line is too long or if the last line of the stream ends without a `'\n'` character.

numberline.c

```
129 char* fgetline(size_t size, char s[restrict size],
130               FILE* restrict stream) {
131     s[0] = 0;
132     char* ret = fgets(s, size, stream);
133     if (ret) {
134         /* s is writable so can be pos. */
135         char* pos = strchr(s, '\n');
136         if (pos) *pos = 0;
137         else ret = 0;
138     }
139     return ret;
140 }
```

The first two lines of the function guarantee that `s` is always null terminated: either by the call to **fgets**, if successful, or by enforcing it to be an empty string. Then, if something was read, the first `'\n'` character that can be found in `s` is replaced by 0. If none is found, a partial line has been read. In that case the caller can detect this situation and call `fgetline` again to attempt to read the rest of the line or to detect an end-of-file condition.<sup>[Exs 34]</sup>

Besides **fgets** this uses **strchr** from the C library. The lack of **const-safeness** of this function is not an issue, here, since `s` is supposed to be modifiable, anyhow. Unfortunately, with the interfaces as they exist now, we always have to do this assessment ourselves. Later, in Section 23.3 we will see a proposal to improve on that situation.

Since it involves a lot of detailed error handling, We will only go into all details of the function `fprintrnumbers` in Section 15, below. For our purpose here we restrict ourselves to the discussion of function `sprintrnumbers`, that is a bit simpler because it

[Exs 34] Improve the **main** of the example such that it is able to cope with arbitrarily long input lines.

only writes to a string, instead of a stream, and because it just assumes that the buffer `buf` that it receives provides enough space.

numberline.c

`sprintrnumbers`: print a series of numbers *nums* in *buf*, using `printf` format *form*, separated by *sep* characters and terminated with a newline character.

**Returns:** the number of characters printed to *buf*.

This supposes that *tot* and *buf* are big enough and that *form* is a format suitable to print `size_t`.

```
int sprintrnumbers(size_t tot, char buf[restrict tot],
                  char const form[restrict static 1],
                  char const sep[restrict static 1],
                  size_t len, size_t nums[restrict len]);
```

Function `sprintrnumbers` uses a function of the C library that we haven't met yet, `sprintf`. Its formatting capacities are the same as those of `printf` or `fprintf`, only that it doesn't print to a stream, but to a `char` array, instead.

numberline.c

```
147 int sprintrnumbers(size_t tot, char buf[restrict tot],
148                  char const form[restrict static 1],
149                  char const sep[restrict static 1],
150                  size_t len, size_t nums[restrict len]) {
151     char* p = buf; /* next position in buf */
152     size_t const seplen = strlen(sep);
153     if (len) {
154         size_t i = 0;
155         for (;;) {
156             p += sprintf(p, form, nums[i]);
157             ++i;
158             if (i >= len) break;
159             memcpy(p, sep, seplen);
160             p += seplen;
161         }
162     }
163     memcpy(p, "\n", 2);
164     return (p-buf)+1;
165 }
```

Function `sprintf` always ensures that a 0 character is placed at the end of the string. It also returns the length of that string, that is the number of characters before the 0 character that have been written. This is used above to update the pointer to the current position in the buffer. `sprintf` still has an important vulnerability:

**Rule 2.14.1.4** `sprintf` makes no provision against buffer overflow.

That is, if we pass an insufficient buffer as a first argument, bad things will happen. Here inside `sprintrnumbers`, much the same as `sprintf` itself, we *suppose* that the buffer is large enough to hold the result. If we aren't sure that the buffer can hold the result, we can use the C library function `snprintf`, instead.



```
1 int snprintf(char*restrict s, size_t n, char const*restrict form,
   ...);
```

This function ensures in addition that never more than `n` bytes are written to `s`. If the return value is larger or equal to `n` the string is been truncated to fit. In particular, if `n` is 0 nothing is written into `s`.

**Rule 2.14.1.5** Use `snprintf` when formatting output of unknown length.

In summary `snprintf` has a lot of nice properties:

- The buffer `s` will not overflow.
- After a successful call `s` is a string.
- When called with `n` and `s` set to 0, `snprintf` just returns the length of the string that would have been written.

By using that, a simple `for`-loop to compute the length of all the numbers printed on one line looks as follows:

```
180 /* Count the chars for the numbers. */
181 for (size_t i = 0; i < len; ++i)
182     tot += snprintf(0, 0, form, nums[i]);
```

numberline.c

We will later see how this is used in the context of `fprintrnumbers`.

**14.2. Formatted input.** Similar as the `printf` family of functions for formatted output, the C library has a series of functions for formatted input: `fscanf`, for input from an arbitrary stream, `scanf` for `stdin`, and `sscanf` from a string. For example the following would read a line of three `double` values from `stdin`:

```
1 double a[3];
2 /* Read and process an entire line with three double values */
3 if (scanf("%lg%lg%lg", &a[0], &a[1], &a[2]) < 3) {
4     printf("not_enough_input_values!\n");
5 }
```

Tables 1 to 3 give an overview of the format for specifiers. Unfortunately, these functions are more difficult to use than `printf` and also they have conventions that diverge from `printf` in subtle points:

- To be able to return values for all formats the arguments are pointers to the type that is scanned.
- White space handling is subtle and sometimes unexpected. A space character, ' ', in the format matches any sequence of white-space, namely spaces, tabs and newline characters. Such a sequence may in particular be empty or contain several newline characters.

TABLE 1. Format specifications for `scanf` and similar functions, with the general syntax `[XX] [WW] [LL] SS`

XX	*	assignment suppression
WW	field width	maximum number of input characters
LL	modifier	select width of target type
SS	specifier	select conversion

TABLE 2. Format specifiers for **scanf** and similar functions. With an **'l'** modifier, specifiers for characters or sets of characters (**'c'**, **'s'**, **'['**) transform multibyte character sequences on input to wide character **wchar\_t** arguments, see Section 14.3.

SS	conversion	pointer to	skip space	analogous to function
<b>'d'</b>	decimal	signed integer	yes	<b>strtoul</b> , base 10
<b>'i'</b>	decimal, octal or hex	signed integer	yes	<b>strtoul</b> , base 0
<b>'u'</b>	decimal	unsigned integer	yes	<b>strtoul</b> , base 10
<b>'o'</b>	octal	unsigned integer	yes	<b>strtoul</b> , base 8
<b>'x'</b>	hexadecimal	unsigned integer	yes	<b>strtoul</b> , base 16
<b>'aefg'</b>	floating point	floating point	yes	<b>strtod</b>
<b>'%'</b>	<b>'%'</b> character	no assignment	no	
<b>'c'</b>	character (sequence)	character	no	<b>memcpy</b>
<b>'s'</b>	non-white-space	string	yes	<b>strcspn</b> with <b>"_\\f\\n\\r\\t\\v"</b>
<b>'['</b>	scan set	string	no	<b>strspn</b> or <b>strcspn</b>
<b>'p'</b>	address	<b>void</b>	yes	
<b>'n'</b>	character count	signed integer	no	

TABLE 3. Format modifiers for **scanf** and similar functions. Note that the significance for **float\*** and **double\*** arguments is different than for **printf** formats.

character	type
<b>"hh"</b>	<b>char</b> types
<b>"h"</b>	<b>short</b> types
<b>" "</b>	<b>signed</b> , <b>unsigned</b> , <b>float</b> , <b>char</b> arrays and strings
<b>"l"</b>	<b>long</b> integer types, <b>double</b> , <b>wchar_t</b> characters and strings
<b>"ll"</b>	<b>long long</b> integer types
<b>"j"</b>	<b>intmax_t</b> , <b>uintmax_t</b>
<b>"z"</b>	<b>size_t</b>
<b>"t"</b>	<b>ptrdiff_t</b>
<b>"L"</b>	<b>long double</b>

- String handling is different. As the arguments to the **scanf** functions are pointers anyhow, formats **"%c"** and **"%s"** both refer to an argument of type **char\***. Where **"%c"** reads a character array of fixed size (of default 1), **"%s"** matches any sequence of non-white-space characters and adds a terminating 0 character.
- The specification of types in the format have subtle differences compared to **printf**, in particular for floating point types. To be consistent between the two, best is to use **"%lg"** or similar for **double** and **"%Lg"** for **long double**, both for **printf** and **scanf**.
- There is a rudimentary utility to recognize character classes. For example, a format of **"%[aeiouAEIOU]"** can be use to scan for the vowels of the Latin alphabet. In such a character class specification the caret **'^'** negates the class if it is found at the beginning. Thus **"%[^\\n] %\* [\\n]"** scans a whole line (but which must be non-empty) and then discards the newline character at the end of the line.

These particularities make the **scanf** family of functions difficult to use. E.g. our seemingly simple example above has the flaw (or feature) that it is not restricted to read

a single input line, but it would happily accept three **double** values that are spread over several lines.<sup>[Exs 35]</sup> in most cases where you have a regular input pattern such as a series of numbers they are best avoided.

**14.3. Extended character sets.** Up to now we have used only a limited set of character to specify our programs or the contents of string literals that we printed on the console: namely a set consisting of the Latin alphabet, Arabic numerals and some punctuation characters. This limitation is a historic accident that originates in the early market domination by the American computer industry, on one hand, and the initial need to encode characters with a very limited number of bits on the other.<sup>36</sup> As we already have seen by the use of the type name **char** for the basic data cell, the concepts of a text character and of an atomic data component was not very well separated at the start.

Latin from which we inherited our character set, is a language that, as a spoken language, is long dead. Its character set is not sufficient to encode the particularities of the phonetics of other language. Among the European languages English has the particularity that it encodes the missing sounds with combinations of letters such as “ai”, “ou” or “gh” (*fair enough*), not with diacritic marks, special characters or ligatures (*fär ínóff*), as do most of its cousins. So for other languages that use the Latin alphabet the possibilities were already quite restricted, but for languages and cultures that use completely different scripts (Greek, Russian) or even completely different concepts (Japanese, Chinese) this restricted American character set was clearly not sufficient.

During the first years of market expansion out to the world different computer manufacturers, countries and organizations provided native language support for their respective communities more or less randomly, added specialized support for graphical characters, mathematical typesetting, music scores ... without coordination, an utter chaos. As a result interchanging textual information between different systems, countries, cultures was difficult if not impossible in many cases; writing portable code that could be used in the context of different language *and* different computing platforms had much resemblance to black arts.

Luckily, these yearlong difficulties are now mainly mastered and on modern systems we will be able to write portable code that uses “extended” characters in a unified way. The following code snippet shows how this is supposed to work:

```

87  setlocale(LC_ALL, "");
88  /* Multibyte character printing only works after the locale
89     has been switched. */
90  draw_sep(TOPLEFT "_@_2014_ jenz 'gvz,t&t_", TOPRIGHT);

```

mbstrings-main.c

That is, near the beginning of our program we switch to the “native” locale, and then we can use and output text containing extended characters, here phonetics (so-called IPA). The output of this looks similar to

```

|      © 2014 jenz 'gvz,t&t      |
|_____|

```

The means to achieve this are quite simple. We have some macros with magic string literals for vertical and horizontal bars, topleft and topright corners:

<sup>[Exs 35]</sup> Modify the format string in the example such that it only accepts three numbers on a single line, separated by blanks, and such that the terminating newline character (eventually preceded by blanks) is skipped.

<sup>36</sup>The character encoding that is dominantly used for the basic character set referred to as ASCII, American standard code for information interchange.

```

43 #define VBAR "\u2502"      /**< a vertical bar character */
44 #define HBAR "\u2500"      /**< a horizontal bar character */
45 #define TOPLEFT "\u250c"   /**< topleft corner character */
46 #define TOPRIGHT "\u2510"  /**< topright corner character */

```

And an adhoc function that nicely formats an output line:

```

draw_sep: Draw multibyte strings start and end separated by a horizontal line.

void draw_sep(char const start[static 1],
              char const end[static 1]) {
    fputs(start, stdout);
    size_t slen = mbsrlen(start, 0);
    size_t elen = 90 - mbsrlen(end, 0);
    for (size_t i = slen; i < elen; ++i) fputs(HBAR, stdout);
    fputs(end, stdout);
    fputc('\n', stdout);
}

```

This uses a function to count the number of print characters in a “multibyte string” (`mbsrlen`) and our old friends `fputs` and `fputc` for textual output.

The start of all of this by the call to `setlocale` is important. Chances are that otherwise you’d see just garbage if you output characters from the extended set to your terminal. But once you do that and your system is well installed, such characters placed inside multibyte strings “fär\_inóff” should work out not to badly.

Here, a *multibyte character* is a sequence of bytes that are interpreted as representing a single character of the extended character set, and a *multibyte string* is a string that contains such multibyte characters. Luckily, these beasts are compatible with ordinary strings as we have handled them so far:

**Rule 2.14.3.1** *Multibyte characters don’t contain null bytes.*

**Rule 2.14.3.2** *Multibyte strings are null terminated.*

Thus, many of the standard string function such as `strcpy` work out of the box for multibyte strings. They introduce one major difficulty, though, namely the fact that the number of printed characters can no longer be directly deduced from the number of elements of a `char` array or by the function `strlen`. This is why in the code above we use the (non-standard) function `mbsrlen`.

**mbsrlen:** Interpret a mb string in *mbs* and return its length when interpreted as a wide character string.

**Returns:** the length of the mb string or `-1` if an encoding error occurred.

This function can be integrated into a sequence of searches through a string, as long as a *state* argument is passed to this function that is consistent with the mb character starting in *mbs*. The state itself is not modified by this function.

**Remarks:** *state* of `0` indicates that *mbs* can be scanned without considering any context.

```

size_t mbsrlen(char const*restrict mbs,
              mbstate_t const*restrict state);

```

As you can already see from that description, parsing multibyte strings for the individual multibyte characters can be a bit more complicated. In particular, we generally we may need to keep a parsing state by means of the type `mbstate_t` that is provided by the C standard in the header files `wchar.h`<sup>37</sup>. This header provides utilities for multibyte strings and characters, but also for a so-called *wide character* type `wchar_t`. We will see all of that below.

```
#include <wchar.h>
```

But before we can do that we have to introduce another international standard, ISO 10646, so-called *Unicode*. As the naming indicates Unicode<sup>38</sup> attempts to provide a unified framework for character codes. It provides a huge table<sup>39</sup> of basically all character *concepts* that have been conceived by mankind so far. “Concept” here is really important; we have to distinguish that from the print form or *glyph* that a particular character may have in a certain typesetting, such as a “Latin capital letter A” can appear as A, A, A or A in the present text. Other such conceptual characters like the character “Greek capital letter Alpha” may even be printed with the same or similar glyph A.

Unicode places each character concept, *code point* in its own jargon, into a linguistic or technical context. In addition to the definition of the character itself it classifies it, *e.g.* as being a capital letter, and relates it to other code points, *e.g.* by stating that “A” is the capitalization of “a”.

If you need special characters for your particular language, there are good chances that you have them on your keyboard and that you can enter them into multibyte string for coding in C just “as-is”. That is that your system is configured to insert the whole byte sequence for an “ä”, say, directly into the text and doing all the magic that is needed for you. If you don’t have or want that, you can use the technique that we used for the macros `HBAR` etc above. There we have used an escape sequence that is new in C11<sup>40</sup>: a backslash and a “u” followed by four hexadecimal digits encodes a Unicode codepoint. *E.g.* the codepoint for “latin small letter a with diaeresis” is 228 or `0xE4`. Inside a multibyte string this then reads as `"\\u00E4"`. Since four hexadecimal digits can only address 65536 codepoints, there is another possibility to specify 8 hexadecimal digits, introduced by a backslash and a capital “U”, but you will only encounter these in very much specialized contexts.

In the example above we encoded 4 graphical characters with such Unicode specifications, characters that most likely are not placed on any keyboard. There are several online sites that allow you to find the code point for any character that you are searching.

If we want to do more than simple input/output with multibyte characters and strings, things become a bit more complicated. Already simple counting of the characters is not trivial, `strlen` does give the right answer, and other string functions such as `strchr`, `strspn` or `strstr` don’t work as expected. Fortunately, the C standard gives us a whole set of replacement functions, usually prefixed with `wcs` instead of `str`, that will work on *wide character strings*, instead. The `mbsrlen` function that we introduced above can be coded as

The core of this function is the use of the library function `mbsrtowcs`, “*multibyte string (mbs), restartable, to wide character string (wcs)*”, which constitutes one of the primitives that the C standard provides to handle multibyte strings:

```
1 size_t mbsrtowcs(wchar_t*restrict dst, char const**restrict src,
2                 size_t len, mbstate_t*restrict ps);
```

So once we decrypted the abbreviation of the name we know that this function is supposed to convert an mbs, `src`, to a wcs, `dst`. Here, *wide characters* (wc) of type `wchar_t` are used to encode exactly one character of the extended character set and these wide characters

<sup>37</sup>The header `uchar.h` also provides this type.

<sup>38</sup><http://www.joelonsoftware.com/articles/Unicode.html>

<sup>39</sup>Today Unicode has about 110000 code points.

<sup>40</sup><http://donslashzero.net/2014/05/21/the-interesting-state-of-unicode-in-c/>

mbstrings.c

```

30 size_t mbsrlen(char const*s, mbstate_t const*restrict state) {
31     if (!state) state = MBSTATE;
32     mbstate_t st = *state;
33     size_t mblen = mbsrtowcs(0, &s, 0, &st);
34     if (mblen == -1) errno = 0;
35     return mblen;
36 }

```

are used to form wcs pretty much in the same way as **char** compose ordinary strings: they are null terminated arrays of such wide characters.

The C standard doesn't restrict the encoding that is used for **wchar\_t** much, but any sane environment should nowadays use Unicode for its internal representations. You can check this with two macros as follows:

mbstrings.h

```

23 #ifndef __STDC_ISO_10646__
24 # error "wchar_t wide characters have to be Unicode code points"
25 #endif
26 #ifdef __STDC_MB_MIGHT_NEQ_WC__
27 # error "basic character codes must agree on char and wchar_t"
28 #endif

```

Modern platforms typically implement **wchar\_t** by either 16 bit or 32 bit integer types. Which of these should usually not be of much concern to you, if you only use the code points that are representable with 4 hexadecimal digits in the `\uXXXX` notation. Those platforms that use 16 bit effectively can't use the other code points in `\UXXXXXXXX` notation, but this shouldn't bother you much.

Wide characters and wide character string literals follow analogous rules as we have seen them for **char** and strings. For both a prefix of `L` indicates a wide character or string, e.g. `L'ä'` and `L'\u00E4'` are the same character, both of type **wchar\_t**, and `L"b\u00E4"` is an array of 3 elements of type **wchar\_t** that contains the wide characters `L'b'`, `L'ä'` and `0`.

Classification of wide characters is also done in a similar way as for simple **char**.

```
#include <wctype.h>
```

The header `wctype.h` provides the necessary functions and macros.

To come back to **mbsrtowcs**, this function *parses* the multibyte string `src` into snippets that correspond to multibyte character, and assigns the corresponding code point to the wide characters in `dst`. The parameter `len` describes the maximal length that the resulting wcs may have. The parameter `state` points to a variable that stores an eventual *parsing state* of the mbs, we will discuss this concept briefly a bit later.

As you can see now, the function **mbsrtowcs** has two particularities. First when called with a null pointer for `dst` it simply doesn't store the wcs but only returns the size that such a wcs would have. Second, it can produce a *coding error* if the mbs is not encoded correctly. In that case the function returns `(size_t)-1` and sets **errno** to the value **EILSEQ**, see `errno.h`. Part of the code of **mbsrlen** is actually a "repair" of that error strategy by setting **errno** to 0, again.

```
#include <errno.h>
```

Let's now see into a second function that will help us to handle mbs:

mbstrings.h

**mbstrdup:** Interpret a sequence of bytes in *s* as mb string and convert it to a wide character string.

**Returns:** a newly malloc'ed wide character string of the appropriate length, 0 if an encoding error occurred.

**Remarks:** This function can be integrated into a sequence of such searches through a string, as long as a *state* argument is passed to this function that is consistent with the mb character starting in *c*. The state itself is not modified by this function.

*state* of 0 indicates that *s* can be scanned without considering any context.

```
wchar_t* mbstrdup(char const*s, mbstate_t const*restrict state);
```

So this function returns a freshly allocated wcs with the same contents as the mbs *s* that it receives on input. Besides the fuzz with the *state* parameter, its implementation is straightforward:

mbstrings.c

```
38 wchar_t* mbstrdup(char const*s, mbstate_t const*restrict state) {
39     size_t mblen = mbsrlen(s, state);
40     if (mblen == -1) return 0;
41     mbstate_t st = state ? *state : *MBSTATE;
42     wchar_t* S = malloc(sizeof(wchar_t[mblen+1]));
43     /* We know that s converts well, so no error check */
44     if (S) mbsrtowcs(S, &s, mblen+1, &st);
45     return S;
46 }
```

After determining the length of the target string, we use **malloc** to allocate space and **mbsrtowcs** to copy over the data.

To have a more fine grained control over the parsing of an mbs, the standard provides the function **mbrtowc**:

```
1 size_t mbrtowc(wchar_t*restrict pwc,
2               const char*restrict s, size_t len,
3               mbstate_t* restrict ps);
```

In this interface, parameter *len* denotes the maximal position in *s* that is scanned for a single multibyte character. Since in general we don't know how such a multibyte encoding works on the target machine, we have to do some guess work that helps us determine *len*. To encapsulate such a heuristic, we cook up the following interface. It has the similar semantic as **mbrtowc** but avoids the specification of *len*:

mbstrings.h

**mbrtow:** Interpret a sequence of bytes in *c* as mb character and return that as wide character through *C*.

**Returns:** the length of the mb character or  $-1$  if an encoding error occurred.

This function can be integrated into a sequence of such searches through a string, as long as the same *state* argument passed to all calls to this or similar functions.

**Remarks:** *state* of 0 indicates that *c* can be scanned without considering any context.

---

```
size_t mbrtow(wchar_t*restrict C, char const c[restrict static
              1],
              mbstate_t*restrict state);
```

So this function returns the number of bytes that were identified for the first multibyte character in the string, or  $-1$  when on error. **mbrtowc** as another possible return value,  $-2$ , for the case that *len* wasn't big enough. The implementation uses that return value, to detect such a situation and to adjust *len* until it fits:

mbstrings.c

```
14 size_t mbrtow(wchar_t*restrict C, char const c[restrict static
15             1],
16             mbstate_t*restrict state) {
17     if (!state) state = MBSTATE;
18     size_t len = -2;
19     for (size_t maxlen = MB_LEN_MAX; len == -2; maxlen *= 2)
20         len = mbrtowc(C, c, maxlen, state);
21     if (len == -1) errno = 0;
22     return len;
```

Here, **MB\_LEN\_MAX** is a standard value that will be a good upper bound for *len* in most situations.

Let us now go to a function that uses the capacity of **mbrtow** to identify mbc and to use that to search inside a mbs.



mbstrings.h

**mbsrwc**: Interpret a sequence of bytes in *s* as mb string and search for wide character *C*.

**Returns:** the *occurrence*'th position in *s* that starts a mb sequence corresponding to *C* or 0 if an encoding error occurred.

If the number of occurrences is less than *occurrence* the last such position is returned. So in particular using **SIZE\_MAX** (or -1) will always return the last occurrence.

**Remarks:** This function can be integrated into a sequence of such searches through a string, as long as the same *state* argument passed to all calls to this or similar functions and as long as the continuation of the search starts at the position that is returned by this function.

*state* of 0 indicates that *s* can be scanned without considering any context.

```
char const* mbsrwc(char const s[restrict static 1],
                  mbstate_t* restrict state,
                  wchar_t C, size_t occurrence);
```

mbstrings.c

```
68 char const* mbsrwc(char const s[restrict static 1], mbstate_t*
69   restrict state,
70   wchar_t C, size_t occurrence) {
71   if (!C || C == WEOF) return 0;
72   if (!state) state = MBSTATE;
73   char const* ret = 0;
74
75   mbstate_t st = *state;
76   for (size_t len = 0; s[0]; s += len) {
77     mbstate_t backup = st;
78     wchar_t S = 0;
79     len = mbrtow(&S, s, &st);
80     if (!S) break;
81     if (C == S) {
82       *state = backup;
83       ret = s;
84       if (!occurrence) break;
85       --occurrence;
86     }
87   }
88   return ret;
}
```

As said, all of this encoding with multibyte strings and simple IO works perfectly fine if we have an environment that is consistent. That is if it uses the same multibyte encoding within your source code as for other text files and on your terminal. Unfortunately here not all environments use the same encoding, yet, so you may encounter difficulties when transferring text files (including sources) or executables from one environment to the other. Besides the definition of the big character table, Unicode also defines 3 encodings that are now widely used and that hopefully will replace all others, eventually. These are called *UTF-8*, *UTF-16* and *UTF-32* for *Unicode Transformation Format* with 8 bit, 16 bit and 32 bit words, respectively. With C11, the C language now includes rudimentary direct

support for these encodings without having to rely on the `locale`. String literals with these encodings can be coded as `u8"text"`, `u"text"` and `U"text"` which have types `char[]`, `char16_t[]` and `char32_t[]`, respectively.

Good chances are that the multibyte encoding on a modern platform is UTF-8, anyhow, and then you wouldn't need these special literals and types. They would be mostly useful in a context where you'd have to ensure one of these encodings, *e.g.* in a network communication. Life on legacy platforms might be more difficult, see here for an overview for the Windows<sup>41</sup> platform.

**14.4. Binary files.** In Section 8.2 we have already seen that input and output to streams can also be performed in *binary* mode in contrast to the usual *text mode* as we have used it up to now. To see the difference, remember that text mode IO doesn't write the bytes that we pass to `printf` or `fputs` one-to-one to the target file or device.

- Depending on the target platform, a `'\n'` character can be encoded as one or several characters.
- Spaces that precede a new line can be suppressed.
- Multibyte characters can be transcribed from the execution character set (the program's internal representation) to the character set of the file system underlying the file.

And similar observations hold for reading data from text files.

If the data that we manipulate is effectively human readable text, all of this is fine, we can consider ourselves happy that the IO functions together with `setlocale` make this mechanism as transparent as possible. But if we are interested in reading or writing binary data just as it is present in some C objects, this can be quite a burden and lead to serious difficulties. In particular, binary data could implicitly map to the end-of-line convention of the file, and thus a write of such data could change the file's internal structure.

So to read and right binary data easier, we need some more interfaces.

```

1 size_t fread(void* restrict ptr, size_t size, size_t nmemb,
2             FILE* restrict stream);
3 size_t fwrite(void const* restrict ptr, size_t size, size_t nmemb,
4             FILE* restrict stream);
5 int fseek(FILE* stream, long int offset, int whence);
6 long int ftell(FILE* stream);

```

The use of `fread` and `fwrite` is relatively straightforward. Each stream has a current *file position* for reading and writing. If successful, these two functions read or write `size*nmemb` bytes from that position onward and then update the file position to the new value. The return value of both functions is the number of bytes that have been read or written, so usually `size*nmemb`, and thus an error occurred if the return value is less than that.

The functions `ftell` and `fseek` can be used to operate on that file position: `ftell` returns the position in terms of bytes from the start of the file, `fseek` positions the file according to the arguments `offset` and `whence`. Here `whence` can have one of the values, `SEEK_SET` refers to the start of the file and `SEEK_CUR` to current file position before the call.<sup>42</sup>

By means of these four functions, we may effectively move forward and backward in a stream that represents a file and read or write any byte of it as it pleases. This can *e.g.* be used to write out a large object in its internal representation to a file and read it in later by a different program, without performing any modifications.

This interface has some restrictions, though:

<sup>41</sup><http://www.nubaria.com/en/blog/?p=289>

<sup>42</sup>There is also `SEEK_END` for the end-of-file position, but this may have platform defined glitches.

- Since this works with internal representations of objects this is only portable between platforms and program executions that use that same representation, *e.g.* the same endianness. Different platforms, operating systems and even program executions can have different representations.
- The use of the type **long** for file positions limits the size of files that can easily be handled with **ftell** and **fseek** to **LONG\_MAX** bytes. On most modern platforms this corresponds to 2GiB.<sup>[Exs 43]</sup>

## 15. Error checking and cleanup

C programs can encounter a lot of error conditions. Errors can be programming errors, bugs in the compiler or OS software, hardware errors or in some resource exhaustion (*e.g.* out of memory), or — any malicious combination of these. For our program to be reliable, we have to detect such error conditions and to deal with them gracefully.

As a first example take the following description of a function `fprintrnumbers`, that continues the series of functions that we discussed in Section 14.1.

numberline.c

`fprintrnumbers`: print a series of numbers *nums* on *stream*, using **printf** format *form*, separated by *sep* characters and terminated with a newline character.

**Returns**: the number of characters printed to *stream*, or a negative error value on error.

If *len* is 0, an empty line is printed and 1 is returned.

Possible error returns are:

- **EOF** (which is negative) if *stream* was not ready to be written to
- **-EOverflow** if more than **INT\_MAX** characters would have to be written, including the case that *len* is greater than **INT\_MAX**.
- **-EFAULT** if *stream* or *numb* are 0
- **-ENOMEM** if a memory error occurred

This function leaves **errno** to the same value as occurred on entry.

```
int fprintrnumbers(FILE* restrict stream,
                  char const form[restrict static 1],
                  char const sep[restrict static 1],
                  size_t len, size_t numb[restrict len]);
```

As you can see, this function distinguishes four different error conditions, that are indicated by the return of negative constant values. The macros for these values are generally provided by the platform in `errno.h` and all start with the capital letter E. Unfortunately the C standard itself imposes only **EOF** (which is negative), and **EDOM**, **EILSEQ** and **ERANGE** which are positive. Other values may or may not be provided. Therefore, in the initial part of our code we have a sequence of preprocessor statements that give default values for those that are missing.

```
#include <errno.h>
```

<sup>[Exs 43]</sup> Write a function `fseekmax` that uses `intmax_t` instead of `long` and that achieves large seek values by combining calls to `fseek`.

numberline.c

```

36 #include <limits.h>
37 #include <errno.h>
38 #ifndef EFAULT
39 # define EFAULT EDOM
40 #endif
41 #ifndef EOVERFLOW
42 # define EOVERFLOW (EFAULT-EOF)
43 # if EOVERFLOW > INT_MAX
44 # error EOVERFLOW constant is too large
45 # endif
46 #endif
47 #ifndef ENOMEM
48 # define ENOMEM (EOVERFLOW+EFAULT-EOF)
49 # if ENOMEM > INT_MAX
50 # error ENOMEM constant is too large
51 # endif
52 #endif

```

The idea of all of this is that we want to be sure to have distinct values for all of these macros. Now the implementation of the function itself looks as follows:

numberline.c

```

167 int fprintfnumbers(FILE*restrict stream,
168                  char const form[restrict static 1],
169                  char const sep[restrict static 1],
170                  size_t len, size_t nums[restrict len]) {
171     if (!stream) return -EFAULT;
172     if (len && !nums) return -EFAULT;
173     if (len > INT_MAX) return -EOVERFLOW;
174
175     size_t tot = (len ? len : 1)*strlen(sep);
176     int err = errno;
177     char* buf = 0;
178
179     if (len) {
180         /* Count the chars for the numbers. */
181         for (size_t i = 0; i < len; ++i)
182             tot += snprintf(0, 0, form, nums[i]);
183         /* We return int so we have to constrain the max size. */
184         if (tot > INT_MAX) return error_cleanup(EOVERFLOW, err);
185     }
186
187     buf = malloc(tot+1);
188     if (!buf) return error_cleanup(ENOMEM, err);
189
190     sprintnumbers(tot, buf, form, sep, len, nums);
191     /* print whole line in one go */
192     if (fputs(buf, stream) == EOF) tot = EOF;
193     free(buf);
194     return tot;
195 }

```

In fact, error handling dominates pretty much the coding effort for the whole function. The first three lines handle errors that occur on entry to the functions and reflect missed pre-conditions, or to speak in the language of Annex K, see Section 8, *runtime constraint violations*<sup>C</sup>.

Dynamic runtime errors are a bit more difficult to handle. In particular, some functions of the C library may use the pseudo variable `errno` to communicate an error condition. If we want to capture and repair all errors, we have to avoid any change to the global state of the execution, including to `errno`. This is done by saving the current value on entry to the function and restoring it in case of an error by a call to the small function `error_cleanup`.

numberline.c

```

142 static inline int error_cleanup(int err, int prev) {
143     errno = prev;
144     return -err;
145 }
```

Now the core of the function itself computes the total number of bytes that should be printed in a `for`-loop over the input array. In the body of the loop, `snprintf` with two 0 arguments is used to compute the size for each number. Then our function `sprintnumbers` from Section 14.1 is used to produce a big string that then is printed through `fputs`.

Observe that there is no error exit after a successful call to `malloc`. If an error is detected on return from the call to `fputs` the information is stored in the variable `tot`, but the call to `free` is not skipped. So even if such an output error occurs, no allocated memory is left leaking. Here, taking care of a possible IO error was relatively simple because the call to `fputs` occurred close to the call to `free`.

The function `fprintnumbers_opt` needs more care. It tries to optimize the procedure even further by printing the numbers immediately instead of counting the required bytes first. This may encounter more error conditions as we go, and we have to take care of them by still guaranteeing to issue a call to `free` at the end. The first such condition is that the buffer that we allocated initially is too small. If the call to `realloc` to enlarge it fails, we have to retreat carefully. The same holds if we encounter the unlikely condition that the total length of your string exceeds `INT_MAX`.

For both cases the function uses `goto`, to jump to the cleanup code that then calls `free`. With C, this is a well established technique that ensures that the cleanup effectively takes place and that, on the other hand, avoids hard to read nested `if-else` conditions. The rules for `goto` are relatively simple

**Rule 2.15.0.1** *Labels for `goto` are visible in the whole function that contains them.*

**Rule 2.15.0.2** *`goto` can only jump to a label inside the same function.*

**Rule 2.15.0.3** *`goto` should not jump over variable initializations.*

The use of `goto` and similar jumps in programming languages has been subject to intensive debate, starting from an article by Dijkstra [1968]. Still today you will find people that seriously object code as it is given here, but let us try to be pragmatic about that: code with or without `goto` can be ugly and hard to follow. The main idea here is to have the “normal” control flow of the function mainly undisturbed and to clearly mark exceptional changes to the control flow with a `goto` or `return`. Later in Section 18.4 we will see another tool in C that allows even more drastic changes to the control flow, namely `setjmp/longjmp`, that will enable us to jump to other positions on the stack of calling functions.

numberline.c

```

197 int fprintfnumbers_opt(FILE*restrict stream,
198                        char const form[restrict static 1],
199                        char const sep[restrict static 1],
200                        size_t len, size_t nums[restrict len]) {
201     if (!stream) return -EFAULT;
202     if (len && !nums) return -EFAULT;
203     if (len > INT_MAX) return -Eoverflow;
204
205     int err = errno;
206     size_t const seplen = strlen(sep);
207
208     size_t tot = 0;
209     size_t mtot = len*(seplen+10);
210     char* buf = malloc(mtot);
211
212     if (!buf) return error_cleanup(ENOMEM, err);
213
214     for (size_t i = 0; i < len; ++i) {
215         tot += sprintf(&buf[tot], form, nums[i]);
216         ++i;
217         if (i >= len) break;
218         if (tot > mtot-20) {
219             mtot *= 2;
220             char* nbuf = realloc(buf, mtot);
221             if (nbuf) {
222                 buf = nbuf;
223             } else {
224                 tot = error_cleanup(ENOMEM, err);
225                 goto CLEANUP;
226             }
227         }
228         memcpy(&buf[tot], sep, seplen);
229         tot += seplen;
230         if (tot > INT_MAX) {
231             tot = error_cleanup(Eoverflow, err);
232             goto CLEANUP;
233         }
234     }
235     buf[tot] = 0;
236
237     /* print whole line in one go */
238     if (fputs(buf, stream) == EOF) tot = EOF;
239 CLEANUP:
240     free(buf);
241     return tot;
242 }

```

## LEVEL 3



## Experience

Once you feel more comfortable when coding in C, you will perhaps be tempted to do complicated things to “optimize” your code. Whatever you think you are optimizing, there is a good chance that you get it wrong: premature optimization can do much harm in terms of readability, soundness, maintainability, ...

Knuth [1974] coined the following phrase that should be your motto for this level:

**Rule D** *Premature optimization is the root of all evil.*

### 16. Performance

**Safety first.** Its good performance is often cited as one of the main reasons why C is used as widely. While there is some truth to the idea that many C programs outperform code of similar complexity written in other programming languages, this aspect of C may come with a substantial cost, especially concerning safety. This is because C, in many places, doesn’t enforce rules, but places the burden of verifying them on the programmer. Important examples for such cases are

- out-of-bounds access of arrays,
- accessing uninitialized objects,
- accessing objects after their lifetime has ended, or
- integer overflow.

This can result in program crashes, loss of data, incorrect results, exposure of sensitive information, loss of money or lives.

C compilers have become much better in recent years, basically they complain about all problems that are detectable at compile time. But still severe problems in code can remain undetected in code that tries to be clever. Many of these problems are avoidable or at least detectable by very simple means:

- All block scope variables should be initialized, thereby eliminating half the problems with uninitialized objects.
- Dynamical allocation should be done with `calloc` instead of `malloc` where ever that is suitable. This avoids another quarter of the problems with uninitialized objects.
- A specific initialization function should be implemented for more complicated data structures that are allocated dynamically. The eliminates the rest of the problems with uninitialized objects.
- Functions that receive pointers should use array syntax and distinguish different cases:
  - A pointer to a single object of the type. These functions should use the `static 1` notation, and by that indicate that they expect a pointer that is non-null:

```
1 void func(double a[static 1]);
```

- A pointer to a collection of objects of known number. These functions should use the **static** N notation, and by that indicate that they expect a pointer that points to at least that number of elements:

```
1 void func(double a[static 7]);
```

- A pointer to a collection of objects of unknown number. These functions should use the VLA notation:

```
1 void func(size_t n, double a[n]);
```

- A pointer to a single object of the type or a null pointer. Such a function must guarantee that even when it receives a null pointer, the execution remains in a defined state:

```
1 void func(double* a);
```

Even though compilers only start to implement checks for these cases, already writing these down as such will help you to avoid out-of-bounds errors.

- Taking addresses of block scope (local) variables should be avoided, if possible. Therefore it is good practice to mark all variables in complex code with **register**.
- Use unsigned integer types for loop indices and treat wrap around explicitly.

**Optimizers are good enough.** Other than some urban myth suggests, applying these rules will usually not negatively impact performance of your code.

**Rule 3.16.0.1** *Optimizers are clever enough to eliminate unused initializations.*

**Rule 3.16.0.2** *The different notations of pointer arguments to functions result in the same binary code.*

**Rule 3.16.0.3** *Not taking addresses of local variables helps the optimizer because it inhibits aliasing.*

Once we have applied these rules and have ensured that our implementation is safe, we may have a look at the performance of the program. What constitutes good performance and how we would measure it is a difficult subject by itself. A first question concerning performance should always be the question of relevance: *e.g.* improving the runtime of an interactive program from 1 *ms* to 0.9 *ms* usually makes no sense at all, and any effort spend for such an improvement is probably better invested elsewhere.

Now if we have a program that has real, observable issues with performance we still should be careful what means we apply to improve performance and in which places.

**Help the compiler.** Nevertheless, there are situations in which we can help our compiler (and future versions of it) to optimize code better, because we can specify specific properties of our code that it can't deduce automatically. C introduces keywords for this purpose that are quite special in the sense that they do not constrain the compiler but the programmer. They all have the property that *removing them* from valid code where they are present should not change the semantics. Because of that property, they are sometimes presented as “useless” or even obsolete features. Be careful when you encounter such statements, people that make such claims tend not to have a deep understanding of C, its memory model or its optimization possibilities. And, in particular they don't seem to have a deep understanding of cause and effect, either.

The keywords that introduce these optimization opportunities are **register** (C90), **inline**, **restrict** (both from C99) and **alignas** (respectively **\_Alignas**, C11). As



indicated all four have the property that they could be omitted from any program without changing its semantic.

In Section 13.2 we have already spoken to some extent about **register**, so we will not go into more detail than that. Just remember that it can help to avoid aliasing between objects that are defined locally in a function. As already stated there, I think that this is a feature that is much underestimated in the C community. Later in Section 21, I will even propose ideas how this feature could be at the heart of a future improvement of C that would be bringing global constants of any object type and even more optimization opportunities for small pure functions.

From the three others, **restrict** also helps controlling aliasing. C11's new **alignas** and the related **alignof** help to position objects on cache boundaries and thus improve memory access. C99's **inline** helps to ensure that code of short functions can be directly replaced at the caller side of the function.

**16.1. Inline functions.** For C programs, the standard tool to write modular code are functions. As we have seen above, they have several advantages:

- They clearly separate interface and implementation. Thereby they allow to improve code incrementally, from revision to revision, or to rewrite a functionality from scratch if deemed necessary.
- If we avoid to communicate to the rest of the code via global variables, they ensure that the state that a function needs is local. Thereby the state is present in the parameters of the call and local variables only. Optimization opportunities may thus be detected much easier.

Unfortunately, they also have some downsides from a performance point-of-view:

- Even on modern platforms, a function call has a certain overhead. Usually, when calling a function some stack space is put aside, local variables are initialized or copied. Control flow jumps to a different point in the executable, that might or might not be in the execution cache.
- Depending on the calling convention of the platform, if the return value of a function is a **struct** the whole return value may have to be copied where the caller of the function expects the result.

If, by coincidence, the code of the caller (say `fcaller`) and the callee (say `fsmall`) are present inside the same translation unit, a good compiler may avoid these downsides by so-called *inlining*. Here, the compiler does something equivalent to replace the call to `fsmall` by the code of `fsmall` itself. Then, there is no call, so no call overhead.

Even better, since the code of `fsmall` is now inlined, all instructions of `fsmall` are seen in that new context. The compiler can detect *e.g.*:

- dead branches that are never executed,
- repeated computation of an expression where the result is already known,
- that the function (as called) may only return a certain type of value.

**Rule 3.16.1.1** *Inlining can open a lot of optimization opportunities.*

A traditional C compiler can only inline functions for which it also knows the definition, only knowing the declaration is not enough. Therefore, programmers and compiler builders have studied the possibilities to increase inlining by making function definitions visible. Without additional support from the language there are two strategies to do so:

- Concatenate all code of a project into a single large file and then to compile all that code in one giant translation unit. Doing such a thing systematically is not as easy as it sounds: we have to ensure that concatenation order of source files doesn't produce definition cycles and that we don't have naming conflicts (*e.g.* two TU with a **static** function `init`, each).

- Functions that should be inlined are placed in header files and then included by all TU that need them. To avoid the multiple definition of the function symbol in each of the TU, such functions must be declared **static**.

Where the first approach is unfeasible for large projects, the second approach is relatively easy to put in place. Nevertheless it has drawbacks:

- If the function is too big to be inlined by the compiler, it is instantiated separately in every TU. That is a function that falls in that category (is big!) will potentially have a lot of copies and increase the size of the final executable.
- Taking a pointer of such function will give the address of the particular instance in the current TU. Comparison of two such pointers that have been obtained in different TU will not compare equal.
- If such a **static** function that is declared in a header file is not used in a TU, the compiler will usually warn about that non-use. So if we have a lot of such small functions in header files, we will see a lot of warnings producing a lot of false alarms.

To avoid these drawbacks, C99 has introduced the **inline** keyword. Other than the naming might suggest this does not force a function to be inlined, but only provides the means such that it *may*.

- A function definition that is declared with **inline** can be used in several TU without causing a multiple symbol definition error.
- All pointers to the same **inline** function will compare equal, even if it is obtained in different TU.
- An **inline** function that is not used in a specific TU will be completely absent from the binary of that TU and in particular not contribute to its size.

The latter point is generally an advantage, but has one simple problem: no symbol for the function would ever be emitted, even for programs that might need such a symbol. Therefore, C99 has introduced a special rule for **inline** functions:

**Rule 3.16.1.2** *Adding a compatible declaration without **inline** keyword ensures the emission of the function symbol in the current TU.*

As an example, suppose we have an **inline** function like this in a header file, say `toto.h`:

```

1 // Inline definition in a header file.
2 // Function argument names and local variables are visible
3 // to the preprocessor and must be handled with care.
4 inline
5 toto* toto_init(toto*toto_x){
6     if (toto_x) {
7         *toto_x = (toto){ 0 };
8     }
9     return toto_x;
10 }
```

Such a function is a perfect candidate for inlining. It is really small and the initialization of any variable of type `toto` is probably best made in place. The call overhead is of the same order as the inner part of the function, and in many cases the caller of the function may even omit the test for the **if**.

**Rule 3.16.1.3** *An **inline** function definition is visible in all TU.*

This function *may* be inlined by the compiler in all TU that see this code, but none of them would effectively emit the symbol `toto_init`. But we can (and should) enforce the emission in one TU, `toto.c`, say, by adding a line as in the following:

```

1  #include "toto.h"
2
3  // Instantiate in exactly one TU.
4  // The parameter name is omitted to avoid macro replacement.
5  toto* toto_init(toto*);

```

**Rule 3.16.1.4** *An **inline** definition goes in a header file.*

**Rule 3.16.1.5** *An additional declaration without **inline** goes in exactly one TU.*

As said, that mechanism of **inline** functions is there to help the compiler take the decision to effectively inline a function or not. In most cases, the heuristics that compiler builders have implemented to take that decision are completely appropriate and you can't do better. They know the particular platform for which the compilation is done much better than you: maybe this platform didn't even exist when you wrote your code. So they are in a much better position to compare the tradeoff between the different possibilities.

An important family of functions that may benefit from **inline** definitions are *pure functions* as we have met them in Section 10.4. If we look at the example of the `rat` structure (Listing 2.1) we see that all the functions implicitly copy the function arguments and the return value. If we rewrite all these functions as **inline** in the header file, all these copies can be avoided by an optimizing compiler.<sup>[Exs 1] [Exs 2]</sup>

So **inline** functions can be a precious tool to build portable code that shows good performance, we just help the compiler(s) to take the appropriate decision. Unfortunately using **inline** functions also has drawbacks that should also be taken into account for our design.

First, 3.16.1.3 implies that any change that you make to an **inline** function will trigger a complete rebuild of your project and of all the users of it.

**Rule 3.16.1.6** *Only expose functions as **inline** if you consider them to be stable.*

Second, the global visibility of the function definition also has the effect that local identifiers of the function (parameters or local variables) may be subject to macro expansion for macros that we don't even know of. In the example we used the `toto_` prefix to protect the function parameter from expansion by macros from other include files.

**Rule 3.16.1.7** *All identifiers that are local to an **inline** function should be protected by a convenient naming convention.*

Third, other than conventional function definitions, **inline** functions have no particular TU to which they are associated. Whereas a conventional function can access to state and functions that are local to the TU (**static** variables and functions), for an **inline** function it would not be clear which copy of which TU these refer to.

**Rule 3.16.1.8** ***inline** functions can't access identifiers of **static** functions.*

**Rule 3.16.1.9** ***inline** functions can't define or access identifiers of **static** objects.*

Here, emphasis is on the fact that the access is restricted to the *identifiers* and not the objects or functions themselves. There is no problem to pass a pointer to a **static** object or function to an **inline** function.

[Exs 1] Rewrite the examples of Section 10.4 with **inline**.

[Exs 2] Revisit the function examples of Section 7 and argue for each of them if they should be defined **inline**.

**16.2. Avoid aliasing: restrict qualifiers.** We have seen many examples of C library functions that use the keyword **restrict** to qualify pointers, and we also have used this qualification for our own functions. The basic idea of **restrict** is relatively simple: it tells the compiler that the pointer in question is the only access to the object it points to. Thus the compiler can make the assumption that changes to the object can only occur through that same pointer, and the object cannot change inadvertently. In other words, with **restrict** we are telling the compiler that the object does not alias with any other object that the compiler handles in this part of code.

**Rule 3.16.2.1** A **restrict**-qualified pointer has to provide exclusive access.

As often in C, such a declaration places the burden of verifying this property to the caller.

**Rule 3.16.2.2** A **restrict**-qualification constrains the caller of a function.

Consider *e.g.* the differences between **memcpy** and **memmove**:

```
1 void* memcpy(void* restrict s1, void const* restrict s2, size_t n);
2 void* memmove(void* s1, const void* s2, size_t n);
```

For **memcpy** both pointers are **restrict**-qualified. So for the execution of this function, the access through both pointers has to be exclusive. Not only that *s1* and *s2* must have different values, none of them must provide access to parts of the object of the other. In other words, the two objects that **memcpy** “sees” through the two pointers must not overlap. Assuming this can help to optimize the function.

In contrast to that, **memmove** does not make such an assumption. So *s1* and *s2* may be equal or the objects may overlap. The function must be able to cope with that situation. Therefore it might be less efficient, but it is more general.

We already have seen in Section 12.3 that it might be important for the compiler to decide if two pointers may in fact point to the same object, so-called aliasing. Pointers to different base types are not supposed to alias, unless one of them is a character type. So both parameters of **fputs** are declared with **restrict**.

```
1 int fputs(const char * restrict s, FILE * restrict stream);
```

Although it might seem very unlikely that any one might call **fputs** with the same pointer value for both parameters.

This specification is more important for functions as **printf** and friends.

```
1 int printf(const char * restrict format, ...);
2 int fprintf(FILE * restrict stream, const char * restrict format,
   ...);
```

The *format* parameter shouldn’t alias with *any* of the arguments that might be passed to the *...* part, *e.g.* the following code has undefined behavior:

```
1 char const* format = "format_printing_itself:_%s\n";
2 printf(format, format); // restrict violation
```

Most probably this example will still do what you think it does. If you abuse the *stream* parameter of your program might explode.

```
1 char const* format = "First_two_bytes_in_stdin_object:_.2s\n";
2 char const* bytes = (char*)stdin; // legal cast to char
3 fprintf(stdin, format, bytes); // restrict violation
```

Sure, code a this one is not very likely to occur in real life. But have in mind that character types have special rules concerning aliasing, and therefore all string processing function may be subject to missed optimization. You could add **restrict**-qualifications in many places where string parameters are involved, and for which you know that they are accessed exclusively through the pointer in question.

**16.3. Measurement and inspection.** We have several times spoken about performance of programs without talking, yet, about methods to asses it. And indeed, we humans are notoriously bad in predicting the performance of code. So our prime directive for questions concerning performance should be:

**Rule E** *Don't speculate about performance of code, verify it rigorously.*

The first step when we dive into a code project that may be performance critical will always be to chose the best algorithms that solve a the problem(s) at hand. This should be done even before coding starts, so we have to do a first complexity assessment by arguing (but not speculating!) about the behavior of such an algorithm.

**Rule 3.16.3.1** *Complexity assessment of algorithms needs proofs.*

Unfortunately, the discussion about complexity proofs is far beyond the scope of this book, so we will not be able to go into that. But then, fortunately, many other books have already been written about it. The interested reader may refer to the text book of Cormen et al. [2001] or to Knuth's treasure trove.

**Rule 3.16.3.2** *Performance assessment of code needs measurement.*

Measurement in experimental sciences is a difficult subject, obviously we can't tackle it here in full detail. But we should first be aware that the act of measuring modifies the observed. This holds in physics, where measuring the mass of an object necessarily displaces it, in biology, where collecting samples of species actually kills animals or plants, or in sociology, where asking for gender or immigration background upfront of a test changes the behavior of the test subjects. Not surprisingly it also holds in computer science and in particular for time measurement, since all such time measurements need time themselves to be accomplished.

**Rule 3.16.3.3** *All measurements introduce bias.*

At the worst, the impact of time measurements can go beyond a simple additional slack of time that is spent by doing the measurement itself. In the first place, a call to `timespec_get`, e.g, is a call to a function that wouldn't be there. The compiler has to do some precautions before any such call, in particular save hardware registers, and has to drop some assumptions about the state of the execution. So time measurement can suppress optimization opportunities. Also, such a function call usually translates into a *system call*, that is a call into the operating system, and this can have effects on many properties of the program execution, e.g on the process or task scheduling, or can invalidate data caches.

**Rule 3.16.3.4** *Instrumentation changes compile time and runtime properties.*

The art of experimental sciences is to address these issues and to ensure that the bias that is introduce by the measurement itself is small and such that the result of an experiment can be assessed qualitatively. Concretely, before we can do any time measurements on code that interests us, we have to asses the bias that time measurements themselves introduce. A general strategy to reduce the bias of measurement is to repeat an experiment

LISTING 3.1. Measuring four code snippets repeatedly.

```

68  timespec_get(&t[3], TIME_UTC);
69  /* A function call can usually not be optimized out. */
70  for (uint64_t i = 0; i < iterations; ++i) {
71      timespec_get(&tdummy, TIME_UTC);
72      accu1 += tdummy.tv_nsec;
73  }
74  timespec_get(&t[4], TIME_UTC);
75  /* A function call can usually not be optimized out, but
76     an inline function can. */
77  for (uint64_t i = 0; i < iterations; ++i) {
78      timespec_get(&tdummy, TIME_UTC);
79      stats_collect1(&sdummy[1], tdummy.tv_nsec);
80  }
81  timespec_get(&t[5], TIME_UTC);
82  for (uint64_t i = 0; i < iterations; ++i) {
83      timespec_get(&tdummy, TIME_UTC);
84      stats_collect2(&sdummy[2], tdummy.tv_nsec);
85  }
86  timespec_get(&t[6], TIME_UTC);
87  for (uint64_t i = 0; i < iterations; ++i) {
88      timespec_get(&tdummy, TIME_UTC);
89      stats_collect3(&sdummy[3], tdummy.tv_nsec);
90  }
91  timespec_get(&t[7], TIME_UTC);

```

several times and to collect statistics about their outcome. Most commonly used statistics are simple, they concern the number of experiments, their mean value, their variance or standard deviation, and sometimes their skew.

For computer science the repetition of an experiment can easily be automated by putting the code that is to be sampled inside a **for** loop and to place the measurements before and after this loop. Thereby, we can execute the sample code thousands or millions of times and compute the average time spent for a loop iteration. The hope then is that the time measurement itself can be neglected because the overall time spent in the experiment is, maybe, several seconds, whereas the time measurement itself may take just several milliseconds.

In the example code to this section, we will try to assess the performance of calls to `timespec_get` and a small utility that collects statistics of measurements. Partial Listing 3.1 contains four **for** loops around code that we want to investigate. All use a `tv_nsec` value obtained from `timespec_get` to collect them in a statistic. In this approach the experimental bias that we introduce is obvious, we use a call to `timespec_get` to measure its own performance. But this bias is easily mastered by augmenting the number of iterations, which reduces the bias. The experiments that we report below were performed with a value of iterations of  $2^{24} - 1$ .

For such a simple setting we can easily come up with a hypothesis how these `tv_nsec` values should be distributed. Namely, we may expect that all possible values are equally likely and thus that the expected value of our measurements is 499999999.5. The values of our measurements should not be concentrated around the average but equally distributed in the whole range, so the variance should be quite large. On the other hand, their distribution should be symmetric and the skew should be close to 0. And indeed, as a first outcome our program confirms that hypothesis:

Terminal

```
0 average value of nsec 4.953e+08 ± 58.3641% (-0.00606862 skew)
```

But this mostly trivial observation is not the goal of this, it only serves as an example of some code that we want to measure. The **for** loops in Listing 3.1 contain code that does this statistics collection with more and more sophistication.

timespec.c

```
struct timespec tdummy;
stats_sdummy[4] = { 0 };
```

The loop starting in Line 70 just accumulates the values, such that we may determine their average. The next loop (Line 77) uses a function `stats_collect1` that maintains a “running mean”, that is it implements a formula that computes a new average  $\mu_n$  by modifying the previous one by some  $\delta(x_n, \mu_{n-1})$ , where  $x_n$  is the new measurement and  $\mu_{n-1}$  is the previous average. The other two loops (Lines 82 and 87) then use functions `stats_collect2` and `stats_collect3`, respectively, that also use similar formulas for the *second* and *third moment*, respectively, namely to compute variance and skew. We will discuss these functions below.

But first, let us have a look at the tools that we use for the instrumentation of the code.

LISTING 3.2. Collecting time statistics with `timespec_diff` and `stats_collect2`.

```
102 for (unsigned i = 0; i < loops; i++) {
103     double diff = timespec_diff(&t[i+1], &t[i]);
104     stats_collect2(&statistic[i], diff);
105 }
```

We use `timespec_diff` from Section 11.3 to compute the time difference between two measurements and `stats_collect2` to sum up the statistics. The whole is then wrapped in another loop (not shown) that repeats that experiment 10 times. After finishing that loop we use functions for the `stats` type to print out the result:

LISTING 3.3. Print time statistics with `stats_mean` and `stats_rsdev_unbiased`.

```
109 for (unsigned i = 0; i < loops; i++) {
110     double mean = stats_mean(&statistic[i]);
111     double rsdev = stats_rsdev_unbiased(&statistic[i]);
112     printf("loop_%u: _E(t)_(sec):\t%5.2e_±_%4.02f%%,\tloop_body_
113           %5.2e\n",
114           i, mean, 100.0*rsdev, mean/iterations);
114 }
```

Here, obviously, `stats_mean` gives access to the mean value of the measurements. The function `stats_rsdev_unbiased` returns the *unbiased relative standard deviation*, that is a standard deviation that is unbiased<sup>3</sup> and that is normalized with the mean value.

A typical output of that on my laptop looks like follows:

<sup>3</sup>Such that it is a true estimation of the standard deviation of the expected time, not only of our arbitrary sample.

	Terminal
0	loop 0: E(t) (sec): 3.31e-02 ± 7.30%, loop body 1.97e-09
1	loop 1: E(t) (sec): 6.15e-03 ± 12.42%, loop body 3.66e-10
2	loop 2: E(t) (sec): 5.78e-03 ± 10.71%, loop body 3.45e-10
3	loop 3: E(t) (sec): 2.98e-01 ± 0.85%, loop body 1.77e-08
4	loop 4: E(t) (sec): 4.40e-01 ± 0.15%, loop body 2.62e-08
5	loop 5: E(t) (sec): 4.86e-01 ± 0.17%, loop body 2.90e-08
6	loop 6: E(t) (sec): 5.32e-01 ± 0.13%, loop body 3.17e-08

Here Lines 0, 1 and 2 correspond to loops that we have not shown, yet. Lines 3 to 6 correspond to the loops that we have seen above. Their relative standard deviations are less than 1%, so we can assert that we have a good statistic and that the times on the right are good estimates of the cost per iteration. *E.g* on my 2.1GHz laptop this means that the execution of one loop iteration of loops 3, 4, 5 or 6 takes about 36, 55, 61 and 67 clock cycles, respectively. So the extra cost of when replacing the simple sum by `stats_collect1` is 19 cycles, from there to `stats_collect2` is 6, and yet another 6 cycles are needed if we use `stats_collect3`, instead.

To see that this is plausible let us look at the `stats` type

```
1 typedef struct stats stats;
2 struct stats {
3     double moment[4];
4 };
```

where we reserve one **double** for all statistical “moments”. Function `stats_collect` in Listing 3.4 then shows how these are updated when we collect a new value that we insert

LISTING 3.4. Collect statistics up to the 3<sup>rd</sup> moment.

```
120 /**
121  ** @brief Add value @a val to the statistic @a c.
122  **/
123 inline
124 void stats_collect(stats* c, double val, unsigned moments) {
125     double n = stats_samples(c);
126     double n0 = n-1;
127     double n1 = n+1;
128     double delta0 = 1;
129     double delta = val - stats_mean(c);
130     double delta1 = delta/n1;
131     double delta2 = delta1*delta*n;
132     switch (moments) {
133     default:
134         c->moment[3] += (delta2*n0 - 3*c->moment[2])*delta1;
135     case 2:
136         c->moment[2] += delta2;
137     case 1:
138         c->moment[1] += delta1;
139     case 0:
140         c->moment[0] += delta0;
141     }
142 }
```



As previously mentioned, we see that this is a relatively simple algorithm to update the moments incrementally. Important features compared to a naive approach are that we avoid numerical imprecision by using the difference from the current estimation of the mean value, and that this can be done without storing all the samples. This approach has been first described for mean and variance (1<sup>st</sup> and 2<sup>nd</sup> moment) by Welford [1962] and was then generalized to higher moments, see Pébay [2008]. In fact, our functions `stats_collect1` etc are just instantiations of that for the chosen number of moments.

```

154 inline
155 void stats_collect2(stats* c, double val) {
156     stats_collect(c, val, 2);
157 }
```

The assembler listing of `stats_collect2` shows that our finding of using 25 cycles for this functions seems plausible: it corresponds to a handful of arithmetic instructions, loads and stores.<sup>4</sup>

LISTING 3.5. gcc's assembler for `stats_collect2(c)`.

```

vmovsd 8(%rdi), %xmm1
vmovsd (%rdi), %xmm2
vaddsd .LC2(%rip), %xmm2, %xmm3
vsubsd %xmm1, %xmm0, %xmm0
vmovsd %xmm3, (%rdi)
vdivsd %xmm3, %xmm0, %xmm4
vmulsd %xmm4, %xmm0, %xmm0
vaddsd %xmm4, %xmm1, %xmm1
vfmadd213sd 16(%rdi), %xmm2, %xmm0
vmovsd %xmm1, 8(%rdi)
vmovsd %xmm0, 16(%rdi)
```

Now by using the measurements above, we have still made one systematic error. We have taken the points of measure *outside* the **for** loops. By doing so, our measurements also comprise the instructions that correspond to the loops themselves. Listing 3.6 shows the three loops that we skipped above. These are basically “empty” in an attempt to measure the contribution of such a loop itself.

In fact, when trying to measure **for** loops with no inner statement, we face a severe problem: an empty loop with no effect can and will be eliminated at compile time by the optimizer. Under normal production conditions this is a good thing, but here when we want to measure this is annoying. Therefore we show three variants of loops, that should not be optimized out. The first declares the loop variable as **volatile** such that all operations on the variable must be emitted by the compiler. Listings 3.7 and 3.8 show gcc's and clang's version of this loop. We see that to comply to the **volatile** qualification of the loop variable, both have to issue several load and store instructions.

LISTING 3.7. gcc's version of the first loop of Listing 3.6.

```

.L510:
movq 24(%rsp), %rax
addq $1, %rax
movq %rax, 24(%rsp)
movq 24(%rsp), %rax
cmpq %rax, %r12
```

<sup>4</sup>This assembler shows x86\_64 assembler features that we have not yet seen: floating point hardware registers and instructions, SSE registers and instructions. Here, memory locations `(%rdi)`, `8(%rdi)` and `16(%rdi)` correspond to `c->moment[i]`, for  $i = 0, 1, 2$ , the name of the instruction minus the `v`-prefix and `sd`-postfix shows the operation that is performed, with `vfmadd213sd` a floating point multiply add instruction.

LISTING 3.6. Instrumenting three **for** loops with **struct timespec**.

```

53  timespec_get(&t[0], TIME_UTC);
54  /* volatile for i ensures that the loop is really done */
55  for (uint64_t volatile i = 0; i < iterations; ++i) {
56      /* do nothing */
57  }
58  timespec_get(&t[1], TIME_UTC);
59  /* volatile for s ensures that the loop is really done */
60  for (uint64_t i = 0; i < iterations; ++i) {
61      s = i;
62  }
63  timespec_get(&t[2], TIME_UTC);
64  /* opaque computation ensures that the loop is really done */
65  for (uint64_t i = 1; accu0 < upper; i += 2) {
66      accu0 += i;
67  }
68  timespec_get(&t[3], TIME_UTC);

```

```
ja .L510
```

LISTING 3.8. clang's version of the first loop of Listing 3.6.

```

.LBB9_17:
    incq    24(%rsp)
    movq    24(%rsp), %rax
    cmpq    %r14, %rax
    jnb     .LBB9_17

```

For the next loop we try to be a bit more economic by only forcing one **volatile** store to an auxiliary variable *s*. As we can see in Listings 3.9 the result is assembler code that looks quite efficient. Namely it consists of 4 instructions, an addition, a comparison, a jump and a store.

LISTING 3.9. gcc's version of the second loop of Listing 3.6.

```

.L509:
    movq    %rax, s(%rip)
    addq    $1, %rax
    cmpq    %rax, %r12
    jne     .L509

```

To come even closer to the loop of the “real” measurements, in the next loop we use a trick. We perform index computations and comparisons for which the result is meant to be opaque to the compiler. Listings 3.10 shows that this results in similar assembler as the previous, only that now we have a second addition instead of the store operation.

LISTING 3.10. gcc's version of the third loop of Listing 3.6.

```

.L500:
    addq    %rax, %rbx
    addq    $2, %rax
    cmpq    %rbx, %r13
    ja      .L500

```

Table 1 summarizes the results that we had collected above and relates the difference between the different measurements. As we might expect, we see that loop 1 with the **volatile** store is 80% faster than the one with a **volatile** loop counter. So in fact using a **volatile** loop counter is not a good idea it can deteriorate the measurement.

loop		sec per iteration	difference	gain/loss	conclusive
0	<b>volatile</b> loop	$1.97 \cdot 10^{-09}$			
1	<b>volatile</b> store	$3.66 \cdot 10^{-10}$	$-1.60 \cdot 10^{-09}$	-81%	yes
2	opaque addition	$3.45 \cdot 10^{-10}$	$-2.10 \cdot 10^{-11}$	-6%	no
3	plus <b>timespec_get</b>	$1.77 \cdot 10^{-08}$	$1.74 \cdot 10^{-08}$	+5043%	yes
4	plus mean	$2.62 \cdot 10^{-08}$	$8.5 \cdot 10^{-09}$	+48%	yes
5	plus variance	$2.90 \cdot 10^{-08}$	$2.8 \cdot 10^{-09}$	+11%	yes
6	plus skew	$3.17 \cdot 10^{-08}$	$2.7 \cdot 10^{-09}$	+9%	yes

TABLE 1. Comparison of measurements

On the other hand, moving from loop 1 to loop 2 has not a very pronounced impact. The 6% gain that we see is smaller than the standard deviation of the test, so we can't even be sure that there is a gain at all. If we would really like to know if there is a difference, we would have to do more tests and hope that the standard deviation would be narrowed down.

But for our goal to assess the time implications that our observation itself can have, the measurements are quite conclusive. The versions 1 and 2 of the **for** loop have an impact that is about one to two orders of magnitude below the impact of calls to **timespec\_get** or **stats\_collect**. So we can assume that the values that we see for loops 3 to 6 are good estimators for the expected time of the measured functions.

There is a strong platform dependent component in these measurements, namely time measurement with **timespec\_get**. In fact, we learned from this experience that on my machine<sup>5</sup> time measurement and statistics collection have a cost that is of the same order of magnitude. For me, personally, this was a surprising discovery when I wrote up this section, I thought that time measurement would be much more expensive.

We also learned that simple statistics such as the standard deviation are easy to obtain and can help to assert claims about performance differences.

**Rule 3.16.3.5** *Collecting higher order moments of measurements to compute variance and skew is simple and cheap.*

So, whenever you make performance claims in the future or see such claims made by others, make sure that the variability of the results has at least been addressed.

**Rule 3.16.3.6** *Run time measurements must be hardened with statistics.*

## 17. Functionlike macros

We already have encountered functionlike macros in several places. In particular, some interfaces in the C standard library are typically implemented by using these, e.g. the type generic interfaces in `tgmath.h`. We also have already seen in Section 10.3 that functionlike macros can easily obfuscate our code and need a certain, restrictive, set of rules. The easiest strategy to avoid many of the problems that come with function-like macros is to only use them where they are irreplaceable, and to use appropriate means where they are replaceable:

```
#include <tgmath.h>
```

**Rule 3.17.0.1** *Whenever possible, prefer an **inline** function to a functional macro.*

That is in situations, where we have a fixed number of arguments with a known type we should provide a proper type-safe interface in form of a function prototype. To achieve the same performance as with a functionlike macro it is then completely sufficient to provide an **inline** definition in a header file.

<sup>5</sup>A commodity Linux laptop with a recent system and modern compilers as of 2016

But there are many situations, where functionlike macros can do more than that. They can

- force certain type mapping and argument checking,
- trace execution,
- provide interfaces with a variable number of arguments,
- provide type generic interfaces,
- provide default arguments to functions.

In the following, I will try to explain how such features can be implemented. We will also discuss two other features of C that are clearly to be distinguished. One, **\_Generic**, because it is much useful in macros and would be very tedious to use without them, and the other, *variadic functions*, because it is now mostly obsolete and should *not* be used in new code.

There also is some warning about this section in order. Macro *programming* quickly becomes ugly and barely readable, so you will need some patience and good will to understand some of the code, here. But the *usage* of these macros shouldn't be difficult. In fact, you can see this even as the main requirement of functional macros:

**Rule 3.17.0.2** *A functional macro shall provide a simple interface to a complex task.*

**17.1. how does it work.** To provide the features that we listed, C has chosen a path that is quite different from other popular programming languages, namely textual replacement. As we have already seen, macros are replaced in a very early stage of compilation, called *preprocessing*. This replacement follows a strict set of rules that are specified in the C standard, and all compilers (on the same platform) should preprocess any source code to exactly the same intermediate code.

Let us look at the following example:

```
1 #define MINSIZE(X, Y) (sizeof(X)<sizeof(Y)?sizeof(X):sizeof(Y))
2 #define BYTECOPY(T, S) memcpy(&(T), &(S), MINSIZE(T, S))
```

These macros fulfill our requirements about functionlike macros, they only evaluate each arguments once<sup>[Exs 6]</sup>, parenthesise all arguments with `()`, and have no hidden effects such as an unexpected control flow.

We see two macro definitions for macros `MINSIZE` and `BYTECOPY`. The first has an *parameter list* `(X, Y)` that defines two parameters `X` and `Y`, and a *replacement text* `(sizeof(X)<sizeof(Y)?sizeof(X):sizeof(X))` that refers to `X` and `Y`. Similarly, `BYTECOPY` also has two parameters `T` and `S` and a replacement text starting with `memcpy`.

The parameters of a macro must be identifiers. A special scope rule restricts the validity of these identifiers to the use inside the replacement text.

When the compiler encounters the name of a functional macro, followed by a closing pair of `()`, such as in `BYTECOPY(A, B)`, it considers this as a *macro call* and replaces it textually according to the following rules:

- (1) The definition of the macro is temporarily disabled to avoid infinite recursion.
- (2) The text inside the `()`, the *argument list*, is scanned for parenthesis and commas. Each opening parenthesis `(` must match a `)`. A comma that is not inside such additional `()` is used to separate the argument list into the arguments. For the case that we handle here, the number of arguments must match the number of parameters of the definition of the macro.
- (3) Each argument is recursively expanded for macros that might appear in them. In our example, `A` could be yet-another macro and expand to some variable name such as `redA`.

<sup>[Exs 6]</sup> Why is this so?

- (4) The resulting text fragments from the expansion of the arguments are assigned to the parameters.
- (5) A copy of the replacement text is made and all occurrences of the parameters is replaced by their respective definitions.
- (6) The resulting replacement text is subject to macro replacement, again.
- (7) This final replacement text is inserted in the source instead of the macro call.
- (8) The definition of the macro is re-enabled.

This procedure looks a bit complicated at a first glance but is effectively quite easily to implement and provides a reliable sequence of replacements. It is guaranteed to avoid infinite recursion and complicated local “variable” assignments. In our case the result of the expansion of `BYTECOPY(A, B)` would be

```
1 memcpy(&(redA), &(B), (sizeof((redA))<sizeof((B))?sizeof((redA)):
    sizeof((B))))
```

We already know that identifiers of macros (function-like or not) live in a name space of their own. This has a very simple reason:

**Rule 3.17.1.1** *Macro replacement is done in an early translation phase, before any other interpretation is given to the tokens that compose the program.*

So the preprocessing phase knows nothing about keywords, types, variables or other constructs of later translation phases.

Since recursion is explicitly disabled for macro expansion, there can even be functions that use the same identifier as a function-like macro. E.g the following is valid C:

```
1 inline
2 char const* string_literal(char const str[static 1]){
3     return str;
4 }
5 #define string_literal(S) string_literal(" S ")
```

It defines a function `string_literal` that receives a character array as argument, and a macro of the same name that calls the function with some weird arrangement of the argument, the reason for which we will see below. There is a more specialized rule that helps to deal with situations where there is a macro and a function of the same name. It is analogous to Rule 2.11.8.1 for function-to-pointer conversion.

**Rule 3.17.1.2** *If the name of functional macro is not followed by `()` it is not expanded.*

In the above example the definition of the function and of the macro is depending on the order of appearance. If the macro definition would be given first, this would immediately expand to something like

```
1 inline
2 char const* string_literal(" char const str[static 1] ") { //
3     error
4     return str;
5 }
```

which is erroneous. But if we surround the name `string_literal` by parenthesis, it is not expanded and remains a valid definition. A complete example could look like:

```
1 // header file
2 #define string_literal(S) string_literal(" S ")
3 inline char const* (string_literal)(char const str[static 1]){
```

```

4   return str;
5 }
6 extern char const* (*func)(char const str[static 1]);
7 // one translation unit
8 char const* (string_literal)(char const str[static 1]);
9 // another translation unit
10 char const* (*func)(char const str[static 1]) = string_literal;

```

That is, both the inline definition and the instantiating declaration of the function are protected by surrounding `()`, and don't expand the functional macro. The last line shows another common usage of this feature. Here `string_literal` is not followed by `()` and so both rules are applied. First Rule 3.17.1.2 inhibits the expansion of the macro and then Rule 2.11.8.1 evaluates the use of the function to a pointer to that function.

### 17.2. Argument checking.

As said above, in cases where we have a fixed number of arguments with types that are well modeled by C's type system, we should use functions and not function-like macros. Unfortunately, C's type system doesn't cover all special cases that we might want to distinguish.

An important such case are string literals. As we have seen in Section 5.4.1, string literals are read-only but are not even `const` qualified. Also an interface with `[static 1]` as for the *function* `string_literal` above is not enforced by the language, because prototypes without `[static 1]` are equivalent. In C there is no way to prescribe for a parameter `str` of a function interface, that it should fulfill the following constraints:

- is a character pointer,
- must be non-null,
- must be unmutable<sup>7</sup>,
- must be 0-terminated.

All these properties could be particular useful to check at compile time, but we have simply no way to specify them in a function interface.

The *macro* `string_literal` fills that gap in the language specification. The weird empty string literals in its expansion `"X "` ensure that `string_literal` can only be called with a string literal.

```

1  string_literal("hello"); // " " "hello" "
2  char word[25] = "hello";
3  ...
4  string_literal(word);    // " " word " " // error

```

The macro and function `string_literal` are just a simple example for this strategy. A more useful example would be

```

.
macro_trace.h
12 /**
13  ** @brief A simple version of the macro that just does
14  ** a @c fprintf or nothing.
15  **/
16 #if NDEBUG
17 # define TRACE_PRINT0(F, X) do { /* nothing */ } while (false)
18 #else
19 # define TRACE_PRINT0(F, X) fprintf(stderr, F, X)
20 #endif

```

<sup>7</sup>`const` only constrains the called function, not the caller

a macro that could be used in the context of a debug build of a program to insert debugging output.

```
17  TRACE_PRINT0("my_favorite_variable:_%g\n", sum);
```

This looks harmless and efficient, but has a pitfall: the argument `F` can be any pointer to `char`, in particular it could be a format string that sits in a modifiable memory region. This may have the effect that an erroneous or malicious modification of that string leads to an invalid format, and thus to a crash of the program or could divulge secrets. In Section 17.4 we will see more in detail why this is particularly dangerous for functions like `fprintf`.

In simple code as in the example, where we pass simple string literals to `fprintf`, these problems should not occur. Modern compiler implementations are able to trace arguments to `fprintf` (and similar) to check if format specifiers and other arguments match.

This check doesn't work, if the format that is passed to `fprintf` is not a string literal but just any pointer to `char`. To inhibit that we can enforce the use of a string literal here:

```
22  /**
23   ** @brief A simple version of the macro that ensures that the @c
24   ** fprintf format is a string literal.
25   **
26   ** As an extra, it also adds a newline to the print out, so
27   ** the user doesn't has to specify it each time.
28   **/
29  #if NDEBUG
30  # define TRACE_PRINT1(F, X) do { /* nothing */ } while (false)
31  #else
32  # define TRACE_PRINT1(F, X) fprintf(stderr, " F "\n", X)
33  #endif
```

Now, `F` must receive a string literal and the compiler then can do the work and warn us about a mismatch.

The macro `TRACE_PRINT1` still has a weak point. If it is used with `DEBUG` unset, the arguments are ignored and thus not checked for consistency. This can have the long term effect that a mismatch remains undetected for a long time and all of a sudden appears when debugging.

So the next version of our macro is defined in two steps. The first uses a similar `#if/#else` idea to define a new macro `TRACE_ON`.

```
35  /**
36   ** @brief A macro that resolves to @c 0 or @c 1 according to @c
37   ** NDEBUG being set.
38   **/
39  #ifndef NDEBUG
40  # define TRACE_ON 0
41  #else
42  # define TRACE_ON 1
43  #endif
```

In contrast to the `NDEBUG` macro, which could be set to any value by the programmer, this new macro is then guaranteed to hold either 1 or 0. Secondly, `TRACE_PRINT2` is defined with a regular `if` conditional.

```
45  /**
```

```

46  ** @brief A simple version of the macro that ensures that the @c
47  ** fprintf call is always evaluated.
48  **/
49  #define TRACE_PRINT2(F, X) \
50  do { if (TRACE_ON) fprintf(stderr, " F "\n", X); } while (false)

```

Whenever its argument is 0, any modern compiler should be able to optimize the call to `fprintf` out. What it shouldn't omit is the argument check for parameters `F` and `X`. So regardless whether or not we are debugging, the arguments to the macro must always be matching as `fprintf` expects it.

Similar to the use of the empty string literal `"` above, there are other tricks to force a macro argument to be of a particular type. One of these tricks consists in adding an appropriate 0: `+0` would force the argument to be any arithmetic type (integer, float or pointer), something like `+0.0F` promotes to a floating type. *E.g* if we want to have a simpler variant to just print a value for debugging, without keeping track of the type of the value, this could be sufficient to our needs:

```

macro_trace.h
52  /**
53  ** @brief Trace a value without having to specify a format.
54  **/
55  #define TRACE_VALUE0(HEAD, X) TRACE_PRINT2(HEAD "%Lg", (X)+0.0L)

```

It works for any value `X` that is either an integer or a floating point. The format `"%Lg"` for a **long double** ensures that any value is presented in a suitable way. Evidently, the `HEAD` argument now must not contain any `fprintf` format, but the compiler will tell us if there is a mismatch, anyhow.

Then, compound literals can be a convenient way to check if the value of a parameter `X` is assignment compatible to a type `T`. Consider the following first attempt to print a pointer value

```

macro_trace.h
57  /**
58  ** @brief Trace a pointer without having to specify a format.
59  **
60  ** @warning Uses a cast of @a X to @c void*.
61  **/
62  #define TRACE_PTR0(HEAD, X) TRACE_PRINT2(HEAD "%p", (void*) (X))

```

It tries to print a pointer value with a `"%p"` format, which expects a generic pointer of type `void*`. Therefore the macro uses a *cast* to convert the value and type of `X` to `void*`. As most casts, a cast here can go wrong if `X` isn't a pointer: because the cast tells the compiler that we know what we are doing all type checks are actually switched off.

This can be avoided by assigning `X` first to an object of type `void*`. Assignment only allows a restricted set of *implicit conversions*, namely here the conversion of any pointer to an object type to `void*`.

```

macro_trace.h
64  /**
65  ** @brief Trace a pointer without specifying a format.
66  **/
67  #define TRACE_PTR1(HEAD, X) \
68  TRACE_PRINT2(HEAD "%p", ((void*){ 0 } = (X)))

```

The trick here is to use something like `((T){ 0 } = (X))` to check if `X` is assignment compatible to type `T`. Here, the compound literal `((T){ 0 })` first creates a temporary object of type `T` to which we then assign `X`. Again, a modern optimizing compiler



should optimize the use of the temporary object away and only do the type checking for us.

### 17.3. Accessing the calling context.

Since macros are just textual replacements, they can interact much more closely with the context of their caller. In general, for usual functionality this isn't desirable and we are better off with the clear separation between the context of the caller (evaluation of function arguments) and the one of the callee (use of function parameters).

In the context of debugging, though, we usually want to break that strict separation to observe part of the state at a specific point in our code. In principle we could access any variable inside a macro, but generally we want some more specific information of the calling environment, namely a trace of the position a particular debugging output originates from.

C offers several constructs for that purpose. It has a special macro `__LINE__` that always expands to an integer constant of the actual line in the source.

```

macro_trace.h
70 /**
71  ** @brief Add the current line number to the trace.
72  **/
73 #define TRACE_PRINT3(F, X) \
74 do { \
75     if (TRACE_ON) \
76         fprintf(stderr, "%lu:_" F "\n", __LINE__ + 0UL, X); \
77 } while (false)

```

Likewise, macros `__DATE__`, `__TIME__` and `__FILE__` contain string literals with the date and time of compilation and with the name of the current translation unit. Another construct, `__func__`, is a local `static` variable that holds the name of the current function.

```

macro_trace.h
79 /**
80  ** @brief Add the name of the current function to the trace.
81  **/
82 #define TRACE_PRINT4(F, X) \
83 do { \
84     if (TRACE_ON) \
85         fprintf(stderr, "%s:%lu:_" F "\n", \
86             __func__, __LINE__ + 0UL, X); \
87 } while (false)

```

If the following invocation

```

macro_trace.c
24 TRACE_PRINT4("my_favorite_variable:_%g", sum);

```

is at line number 24 of the source file and `main` is its surrounding function, the corresponding output looks similar to

```

Terminal
0  main:24: my favorite variable: 889

```

Another pitfall that we should have in mind if we are using `fprintf` automatically as above is that *all* arguments in its list must have the correct type as given in the specifier. For `__func__` this is no problem, by its definition we know that this is a `char` array, so

the `"%s"` specifier is fine. `__LINE__` is different. We know that it is a decimal constant representing the line number. So if we revisit the rules for the types of decimal constants in Section 5.2 we see that the type depends on the value. On embedded platforms `INT_MAX` might be as small as 32767 and very large sources (perhaps automatically produced) may have more lines than that. A good compiler should warn us when such a situation arises:

**Rule 3.17.3.1** *The line number in `__LINE__` may not fit into an `int`.*

In our macros we avoid the problem by fixing the type to `unsigned long`.<sup>8</sup>

There is another type of information from the calling context that is often quite helpful for traces, namely the actual expressions that we passed to the macro as arguments. As this is often used for debugging purposes, C has a special operator for this: `#`. If such a `#` appears before a macro parameter in the expansion, the actual argument to this parameter is *stringified* that is all its textual content is placed into a string literal. The following variant of our trace macro has a `#X`

```

89  /**
90   ** @brief Also add a textual version of the expression that is
91   ** evaluated.
92   **/
93  #define TRACE_PRINT5(F, X) \
94  do { \
95      if (TRACE_ON) \
96          fprintf(stderr, \
97              "%s:%lu: (" #X ") :_ F "\n", \
98              __func__, __LINE__+0UL, X); \
99  } while (false)

```

that is replaced by the text of the second argument at each call of the macro. For the following invocations

```

25  TRACE_PRINT5("my_favorite_variable:%g", sum);
26  TRACE_PRINT5("a_good_expression:%g", sum*argc);

```

the corresponding output looks similar to

```

0  main:25:(sum): my favorite variable: 889
1  main:26:(sum*argc): a good expression: 1778

```

As the preprocessing phase knows nothing about the interpretation of these arguments, this replacement is purely textual and should appear as in the source, with some possible adjustments for white space.

For completeness, we also mention another operator that is only valid in the preprocessing phase, the `##` operator. It is of even more specialized use, namely it is a *token concatenation operator*. It can be useful when writing entire macro libraries where we have to generate names for types or functions automatically.

**17.4. Variable length argument lists.** We have already seen functions that accept argument lists of variable length, namely `printf`, `scanf` and friends. Their declarations have the token `...` at the end of the parameter list to indicate that feature, namely that after an initial number of known arguments (such as the format for `printf`) a list of arbitrary

<sup>8</sup>Hoping that no source will have more than 4 billion lines.

length of additional arguments can be provided. Below, we will briefly mention how such functions can be defined, but this feature is dangerous and merely obsolete, so we will not much insist on it. Alternatively, we present a similar feature for *macros*, that can mostly be used to replace the feature for functions.

17.4.1. *Variadic macros*. Variable length argument macros, *variadic macros* for short, use the same token “...” to indicate that feature. As for functions, this token must appear at the end of the parameter list.

```

macro_trace.h
101 /**
102  ** @brief Allow multiple arguments to be printed in the
103  ** same trace.
104  **/
105 #define TRACE_PRINT6(F, ...) \
106 do { \
107     if (TRACE_ON) \
108         fprintf(stderr, "%s:%lu:_" F "\n", \
109                 __func__, __LINE__+0UL, __VA_ARGS__); \
110 } while (false)

```

Here, in `TRACE_PRINT6`, this indicates that after the format argument `F` any non-empty list of additional arguments may be provided in a call. This list of expanded arguments is accessible in the expansion through the identifier `__VA_ARGS__`. Thus a call such as

```

macro_trace.c
27 TRACE_PRINT6("a_collection:_%g,_%i", sum, argc);

```

just passes the arguments “through” to `fprintf` and results in the output

```

Terminal
0 main:27: a collection: 889, 2

```

Unfortunately, as it is written, the list in `__VA_ARGS__` cannot be empty or absent. So for what we have seen so far, we’d have to write a separate macro for the case that the list is absent.

```

macro_trace.h
112 /**
113  ** @brief Only trace with a text message, no values printed.
114  **/
115 #define TRACE_PRINT7(...) \
116 do { \
117     if (TRACE_ON) \
118         fprintf(stderr, "%s:%lu:_" __VA_ARGS__ "\n", \
119                 __func__, __LINE__+0UL); \
120 } while (false)

```

But with some more effort, these two functionalities can be united into a single macro

```

macro_trace.h
138 ** @brief Trace with or without values.
139 **
140 ** This implementation has the particularity of adding a format
141 ** @c "%.0d" to skip the last element of the list which was
142 ** artificially added.
143 **/
144 #define TRACE_PRINT8(...) \

```

```

145 TRACE_PRINT6 (TRACE_FIRST ( __VA_ARGS__ ) "%.0d", \
146             TRACE_LAST ( __VA_ARGS__ ) )

```

Here, `TRACE_FIRST` and `TRACE_LAST` are macros that give access to the first and remaining arguments in the list, respectively. Both are relatively simple. They use auxiliary macros that enable us to distinguish a first parameter `_0` from the remainder `__VA_ARGS__`. Since we want to be able to call both with one or more arguments, they add a new argument “0” to the list. For `TRACE_FIRST` this goes well, this additional 0 is just ignored as are the rest of the arguments.

```

macro_trace.h
122 /**
123  ** @brief Extract the first argument from a list of arguments.
124  **/
125 #define TRACE_FIRST(...) TRACE_FIRST0 ( __VA_ARGS__ , 0)
126 #define TRACE_FIRST0(_0, ...) _0

```

For `TRACE_LAST` this is a bit more problematic, since this extends the list in which we are interested in by an additional value.

```

macro_trace.h
128 /**
129  ** @brief Remove the first argument from a list of arguments.
130  **
131  ** @remark This is only suitable in our context,
132  ** since this adds an artificial last argument.
133  **/
134 #define TRACE_LAST(...) TRACE_LAST0 ( __VA_ARGS__ , 0)
135 #define TRACE_LAST0(_0, ...) __VA_ARGS__

```

Therefore `TRACE_PRINT6` compensates this by an additional format specifier, `"%.0d"`, that prints an `int` of width 0, that is, nothing. Testing it for the two different use cases

```

macro_trace.c
29 TRACE_PRINT8 ("a_collection:_%g,_%i", sum, argc);
30 TRACE_PRINT8 ("another_string");

```

gives us exactly what we want:

```

Terminal
0 main:29: a collection: 889, 2
1 main:30: another string

```

The `__VA_ARGS__` part of the argument list also can be stringified just as any other macro parameter.

```

macro_trace.h
148 /**
149  ** @brief Trace by first giving a textual representation of the
150  ** arguments.
151  **/
152 #define TRACE_PRINT9(F, ...) \
153 TRACE_PRINT6 (" (" #__VA_ARGS__ ")_" F, __VA_ARGS__ )

```

The textual representation of the arguments

```

macro_trace.c
31 TRACE_PRINT9 ("a_collection:_%g,_%i", sum*acos(0), argc);

```

is inserted including the commas that separate them.

Terminal

```
0 main:31: (sum*acos(0), argc) a collection: 1396.44, 2
```

So far, our variants of the trace macro that have a variable number of arguments must also receive the correct format specifiers in the format argument `F`. This can be a tedious exercise, since it forces us to always keep track of the type of each argument in the list that is to be printed. A combination of an **inline** function and a macro can help us, here. First let us look at the function.

macro\_trace.h

```
166 /**
167  ** @brief A function to print a list of values.
168  **
169  ** @remark Only call this through the macro ::TRACE_VALUES
170  ** that will provide the necessary contextual information.
171  **/
172 inline
173 void trace_values(FILE* s,
174                  char const func[static 1],
175                  size_t line,
176                  char const expr[static 1],
177                  char const head[static 1],
178                  size_t len, long double const arr[len]) {
179     fprintf(s, "%s:%zu: (%s)_%s_%Lg", func, line,
180            trace_skip(expr), head, arr[0]);
181     for (size_t i = 1; i < len-1; ++i)
182         fprintf(s, ",_%Lg", arr[i]);
183     fputc('\n', s);
184 }
```

It prints a list of **long double** values after preceding them with the same header information as we have done this before. Only this time the function receives the list of values through an array of **long double** of known length `len`. For reasons that we will see below, the function actually always skips the last element of the array. Using a function `trace_skip`, it also skips an initial part of the parameter `expr`.

The macro that passes the contextual information to the function comes in two levels. The first is just massaging the argument list in different ways:

macro\_trace.h

```
204 /**
205  ** @brief Trace a list of arguments without having to specify
206  ** the type of each argument.
207  **
208  ** @remark This constructs a temporary array with the arguments
209  ** all converted to @c long double. Thereby implicit conversion
210  ** to that type is always guaranteed.
211  **/
212 #define TRACE_VALUES(...) \
213 TRACE_VALUES0(ALEN(__VA_ARGS__), \
214               #__VA_ARGS__, \
215               __VA_ARGS__, \
216               0 \
```

217 )

First with the help of `ALen`, which we will see later, it evaluates the number of elements in the list. Then it stringifies the list and finally appends the list itself plus an additional 0. All this is fed into `TRACE_VALUES0`:

```

macro trace.h
219 #define TRACE_VALUES0 (NARGS, EXPR, HEAD, ...) \
220 do { \
221     if (TRACE_ON) { \
222         if (NARGS > 1) \
223             trace_values(stderr, __func__, __LINE__ + 0UL, \
224                          " " EXPR " ", " " HEAD " ", NARGS, \
225                          (long double const[NARGS]) { __VA_ARGS__ }); \
226         else \
227             fprintf(stderr, "%s:%lu:_%s\n", \
228                    __func__, __LINE__ + 0UL, HEAD); \
229     } \

```

Here, the list without `HEAD` is used as an initializer of a compound literal of type `long double const[NARG]`. The 0 that we added above ensures that the initializer is never empty. With the information on the length of the argument list, we are also able to make a case distinction, here, if the only argument is just the format string.

17.4.2. *A detour: variadic functions.* Let us now have a brief look into *variadic functions*, i.e. functions with variable length argument lists. As already mentioned these are specified by using the `...` operator in the function declaration, such as in

```
1 int printf(char const format[static 1], ...);
```

Such functions have a fundamental problem in their interface definition. Other than for “normal” functions, at the call side it is not clear to which parameter type an argument should be converted. *E.g.* if we call `printf("%d", 0)` it is not directly clear for the compiler what “kind” of 0 the called function is expecting. For such cases C has a set of rules to determine the type to which an argument is converted. These are almost identical to the rules for arithmetic:

**Rule 3.17.4.1** *When passed to a variadic parameter, all arithmetic types are converted as for arithmetic operations, with the exception of **float** arguments which are converted to **double**.*

So in particular, when they are passed to a variadic parameter, types such as `char` and `short` are converted to a wider type, usually `int`.

So far, so good, now we know how such functions get called. But unfortunately, these rules tell us nothing about the type that the called function should expect to receive.

**Rule 3.17.4.2** *A variadic function has to receive valid information about the type of each argument in the variadic list.*

The `printf` functions get away with this difficulty by imposing a specification of the types inside the `format` argument. Let us look at the following short code snippet:

```

1 unsigned char zChar = 0;
2 printf("%hhu", zChar);

```

This has the effect that `zChar` is evaluated, promoted to `int`, passed as argument to `printf`, which then reads this `int` and re-interprets this value as `unsigned char`. This mechanism is

**complicated:** because the implementation of the function must provide specialized code for all the basic types,

**errorprone:** because each call depends on the fact that the argument types are correctly transmitted to the function, and

**exigent:** because the programmer has to check the type of each argument.

In particular the latter can cause serious portability bugs, because constants can have different types from platform to platform. *E.g* the innocent call

```
1 printf("%d:_%s\n", 65536, "a_small_number"); // not portable
```

will work well on most platforms, namely those that have an **int** type with more than 16 bit. But on some platforms it may fail at runtime because 65536 is **long**. The worst example for such a potential failure is the macro **NULL**.

```
1 printf("%p:_%s\n", NULL, "print_of_NULL"); // not portable
```

As we have seen in Section 11.7, **NULL** is only guaranteed to be a null pointer constant. Compiler implementors are free to choose which variant they provide, some chose **(void\*) 0**, thus with a type of **void\***, most chose 0, thus with a type of **int**. On platforms that have different width for pointers and **int**, *e.g* all modern 64 bit platforms, the result is a crash of the program.<sup>9</sup>

**Rule 3.17.4.3** *Using variadic functions is not portable, unless each argument is forced to a specific type.*

This is quite different from the use of variadic macros as we have seen in the example of **TRACE\_VALUES**. There we used the variadic list as an initializer to an array, and so all elements were automatically converted to the correct target type.

**Rule 3.17.4.4** *Avoid variadic functions for new interfaces.*

They are just not worth the pain. But, if you have to implement a variadic function, you need the C library header **stdarg.h**. It defines one type, **va\_list**, and four function-like macros that can be used the different arguments behind a **va\_list**. Their pseudo interfaces look as:

```
#include <stdarg.h>
```

```
1 void va_start(va_list ap, parmN);
2 void va_end(va_list ap);
3 type va_arg(va_list ap, type);
4 void va_copy(va_list dest, va_list src);
```

The first example, shows how to actually avoid programming the core part of a variadic function. For all that concerns formatted printing there are already functions, that we should use.

```

.
20 FILE* iodebug = 0;
21
22 /**
23  ** @brief Print to the debug stream @c iodebug
24  **/
25 #ifdef __GNUC__
26 __attribute__((format(printf, 1, 2)))
27 #endif
28 int printf_debug(const char *format, ...) {

```

va\_arg.c

<sup>9</sup>Remember Rule 2.11.7.1.

```

29  int ret = 0;
30  if (iodebug) {
31      va_list va;
32      va_start(va, format);
33      ret = vfprintf(iodebug, format, va);
34      va_end(va);
35  }
36  return ret;
37  }

```

The only thing that we do with the `va_start` and `va_end` is to create a `va_list` argument list and pass this information on to the C library function `vfprintf`. This completely spares us to do the case analysis and the tracking of the arguments. Here the conditional `__attribute__` is compiler specific (here for `gcc` and friends). Such an add-on may be very helpful in situations where a known parameter convention is applied and where the compiler can do some good diagnostic to ensure the validity of the arguments.

Now we look into a variadic function that receives `n` `double` values and that sums them up:<sup>[Exs 10]</sup>

```

. va_arg.c
6  /**
7   ** @brief A small, useless function to show how variadic
      functions
8   ** work.
9   **/
10 double sumIt(size_t n, ...) {
11     double ret = 0.0;
12     va_list va;
13     va_start(va, n);
14     for (size_t i = 0; i < n; ++i)
15         ret += va_arg(va, double);
16     va_end(va);
17     return ret;
18 }

```

The `va_list` is initialized by using the last argument before the list. Observe that by some magic `va_start` receives `va` as such and not with an address operator `&`. Then, inside the loop every value in the list is received through a use of the `va_arg` macro, which needs an explicit specification (here `double`) of its *type* argument. Also we have to maintain the length of the list ourselves, here by passing the length as an argument to the function. The encoding of the argument type (here implicit) and the detection of the end of the list is left up to the programmer of the function.

**Rule 3.17.4.5** The `va_arg` mechanism doesn't give access to the length of the `va_list`.

**Rule 3.17.4.6** A variadic function needs a specific convention for the length of the list.

**17.5. Type generic programming.** One of the genuine additions of C11 to the C language has been direct language support for type generic programming. C99 already had `tgmath.h`, see Section 8.1, for type generic mathematical functions, but didn't offer much to program such interfaces yourself. The specific add on is the keyword `_Generic` which introduces a primary expression of the following form

```
#include <tgmath.h>
```

<sup>[Exs 10]</sup> Variadic functions that only receive arguments of all the same type, can be replaced by a variadic macro and an `inline` function that takes an array. Do it.



```

1  _Generic(controlling expression,
2      type1: expression1,
3      ... ,
4      typeN: expressionN)

```

This is much similar to a **switch** statement, only that the *controlling expression* is only taken for its type (but see below), and the result is one of the expressions *expression1* ... *expressionN* chosen by the corresponding type specific *type1* ... *typeN*, of which one may be simply the keyword **default**.

One of the simplest use cases, and primarily what the C committee had in mind, is to use **\_Generic** for a type generic macro interface by providing a choice between function pointers. A basic example for this can be the `tgmath.h` interfaces, e.g. **fabs**. **\_Generic** is not a macro feature itself, but can only conveniently be used in a macro expansion. By neglecting complex floating point types, such a macro for **fabs** could look like:

```

1  #define fabs(X) \
2  _Generic( (X), \
3      float: fabsf, \
4      long double: fabsl, \
5      default: fabs)(X)

```

So this macro distinguishes two specific types, **float** and **long double**, which chose the corresponding functions **fabsf** or **fabsl**, respectively. If the argument *X* is of any other type, it is mapped to the **default** case of **fabs**. That is, other arithmetic types such as **double** and integer types are mapped to **fabs**.<sup>[Exs 11][Exs 12]</sup>

Now, once the resulting function pointer is determined, it is applied to the argument list (*X*) that follows the **\_Generic** primary expression.

Here comes a more complete example.

```

. generic.h
6  inline
7  double min(double a, double b) {
8      return a < b ? a : b;
9  }
10
11  inline
12  long double minl(long double a, long double b) {
13      return a < b ? a : b;
14  }
15
16  inline
17  float minf(float a, float b) {
18      return a < b ? a : b;
19  }
20
21  /**
22   * @brief Type generic minimum for floating point values
23   */
24  #define min(A, B) \
25  _Generic( (A)+(B), \
26      float: minf, \
27      long double: minl,

```

[Exs 11] Find the two reasons why this occurrence of **fabs** in the macro expansion is not itself expanded.

[Exs 12] Extend the **fabs** macro to cover complex floating point types.

28 `default: min) ((A), (B))`

It implements a type generic interface for the minimum of two real values. Three different **inline** functions for the three floating point types are defined, and then used in a similar way as for **fabs**. The difference to there is that these functions need two arguments and not only one, and so the **\_Generic** expression must decide on a combination of the two types. This is done, by using the sum of the two arguments as *controlling expression*. As an effect, argument promotions and conversion are effected to the arguments of that sum, and so the **\_Generic** expression chooses the function for the wider of the two types, or **double** if both arguments are integers.

The difference to just having one function for **long double**, say, is

**Rule 3.17.5.1** *The result type of a **\_Generic** expression depends on the type of chosen expression.*

This is in contrast of what is happening *e.g.* for the ternary operator `a ? b : c`. Here, the return type is computed by combining the two types of `b` and `c`. For the ternary operator, this must be done like that because `a` may be different from run to run, and so either of `b` or `c` may be selected. Since **\_Generic** makes its choice upon the type, this choice is fixed at compile time, and so the compiler can know the resulting type of the choice in advance.

In our example, we can be sure that all generated code that uses our interface will never uses wider types than the programmer has foreseen. In particular, our `min` macro here should always result in the compiler inlining the appropriate code for the types in question.<sup>[Exs 13][Exs 14]</sup>

**Rule 3.17.5.2** *Using **\_Generic** with **inline** functions adds optimization opportunities.*

The interpretation of what it means to talk about the *type of the controlling expression* is a bit ambiguous, and so up to a recent clarification by the C standard's committee there had been diverging interpretations. In fact, what the above examples imply, is that this type is the type of the expression *as-if* it would be passed to a function. This means in particular:

- If there are any, type qualifiers are dropped from the type of the controlling expression.
- An array type is converted to a pointer type to the base type.
- A function type is converted to a pointer to function.

As a consequence,

**Rule 3.17.5.3** *The type expressions in a **\_Generic** expression should only be unqualified types, no array types and no function types.*

That doesn't mean that the type expressions can't be pointers to one of the above, that is a pointer to a qualified type, a pointer to an array or a pointer to a function. But generally, this rule makes the task of writing a type generic macro easier, since we do not have to take all combinations of qualifiers into account. There are 3 qualifiers (or 4 for pointer types), so otherwise all different combinations would lead to 8 (or even 16) different type expressions per base type. The following example `MAXVAL` is already relatively long, it has as special case for all 17 orderable types. If we would have also to track qualifications, we would have to specialize 136 cases.

<sup>[Exs 13]</sup> Extend the `min` macro to cover all wide integer types.

<sup>[Exs 14]</sup> Extend `min` to cover pointer types, as well.

```

30  /**
31   ** @brief The maximum value for the type of @a X
32   **/
33  #define MAXVAL(X) \
34  _Generic( (X), \
35      _Bool: (_Bool)+1, \
36      char: (char)+CHAR_MAX, \
37      signed char: (signed char)+SCHAR_MAX, \
38      unsigned char: (unsigned char)+UCHAR_MAX, \
39      signed short: (signed short)+SHRT_MAX, \
40      unsigned short: (unsigned short)+USHRT_MAX, \
41      signed: INT_MAX, \
42      unsigned: UINT_MAX, \
43      signed long: LONG_MAX, \
44      unsigned long: ULONG_MAX, \
45      signed long long: LLONG_MAX, \
46      unsigned long long: ULLONG_MAX, \
47      float: FLT_MAX, \
48      double: DBL_MAX, \
49      long double: LDBL_MAX)

```

This is an example where a `_Generic` expression is used differently than above, where we “just” chose a function pointer and then called the function. Here the resulting value is an integer constant expressions. This never could be realized by function calls, and it would be very tedious to implement just with macros.<sup>[Exs 15]</sup> Again, with a conversion trick we can get rid of some cases in which we might not be interested in:

```

51  /**
52   ** @brief The maximum promoted value for @a XT, where XT
53   ** can be an expression or a type name
54   **
55   ** So this is the maximum value of when fed to an arithmetic
56   ** operation such as @c +.
57   **
58   ** @remark Narrow types are promoted, usually to @c signed,
59   ** or maybe to @c unsigned on rare architectures.
60   **/
61  #define maxof(XT) \
62  _Generic(0+(XT)+0, \
63      signed: INT_MAX, \
64      unsigned: UINT_MAX, \
65      signed long: LONG_MAX, \
66      unsigned long: ULONG_MAX, \
67      signed long long: LLONG_MAX, \
68      unsigned long long: ULLONG_MAX, \
69      float: FLT_MAX, \
70      double: DBL_MAX, \
71      long double: LDBL_MAX)

```

Here, the special form of the controlling expression adds an additional feature. The expression `0+(identifier)+0` is valid if `identifier` is a variable or if it is a type. If it is a variable, the type of the variable is used and it is evaluated just as any other expression. Then integer promotion is performed to it and the resulting type is deduced.

[Exs 15] Write an analogous macro for the minimum value.

If it is a type, (identifier)+0 is read as cast of +0 to type identifier. Adding 0+ from the left then still ensures that integer promotion is performed if necessary, and so the result is the same if X is a type T or an expression of the same type T.

Another requirement for the type expressions in a **\_Generic** expression is that the choice must be unambiguous at compile time.

**Rule 3.17.5.4** *The type expressions in a **\_Generic** expression must refer to mutually incompatible types.*

**Rule 3.17.5.5** *The type expressions in a **\_Generic** expression cannot be a pointer to VLA.*

A different model than the *function-pointer-call* variant can be convenient, but it also has some pitfalls. Let us try to use **\_Generic** to implement the two macros **TRACE\_FORMAT** and **TRACE\_CONVERT** that are used in the following:

```
macro_trace.h
278 /**
279  ** @brief Trace a value without having to specify a format.
280  **
281  ** This variant works correctly with pointers.
282  **
283  ** The formats are tuneable by changing the specifiers in
284  ** ::TRACE_FORMAT.
285  **/
286 #define TRACE_VALUE1(F, X) \
287     do { \
288         if (TRACE_ON) \
289             fprintf(stderr, TRACE_FORMAT("%s:%lu:_" F, X), \
290                 __func__, __LINE__+0UL, TRACE_CONVERT(X)); \
291     } while (false)
```

**TRACE\_FORMAT** is straightforward. We distinguish 6 different cases:

```
macro_trace.h
232 /**
233  ** @brief Return a format that is suitable for @c fprintf.
234  **
235  ** @return The argument @a F must be a string literal,
236  ** so will be return value.
237  **
238  **/
239 #define TRACE_FORMAT(F, X) \
240 _Generic((X)+0LL, \
241     unsigned long long: " F "%llu\n", \
242     long long: " F "%lld\n", \
243     float: " F "%f\n", \
244     double: " F "%lf\n", \
245     long double: " F "%Lf\n", \
246     default: " F "%p\n")
```

The **default** case, when no arithmetic type is matched, supposes that the argument has a pointer type. In that case, to be a correct parameter for **fprintf**, the pointer must be converted to **void\***. Our goal is to implement such a conversion through the **TRACE\_CONVERT**.

A first try could look like the following:

```

1 #define TRACE_CONVERT_WRONG(X)          \
2 _Generic( (X)+0LL,                       \
3           unsigned long long: (X)+0LL,   \
4           ...                          \
5           default: ((void*){ 0 } = (X)))

```

This uses the same trick as for `TRACE_PTR1` to convert the pointer to `void*`. Unfortunately this implementation is wrong:

**Rule 3.17.5.6** *All choices expression1 ... expressionN in a `_Generic` must be valid.*

Here, if e.g X is an `unsigned long long`, say `1LL`, the `default` case would read

```

1 ((void*){ 0 } = (1LL))

```

which would be assigning a non-zero integer to a pointer, which is erroneous.<sup>16</sup>

We tackle this by two steps. First we have a macro that returns either its argument, the `default`, or a literal zero.

```

macro_trace.h
248 /**
249  ** @brief Return a value that forcibly can be interpreted as
250  ** pointer value.
251  **
252  ** That is any pointer will be returned as such, but other
253  ** arithmetic values will result in a @c 0.
254  **/
255 #define TRACE_POINTER(X)          \
256 _Generic( (X)+0LL,               \
257           unsigned long long: 0, \
258           long long: 0,          \
259           float: 0,              \
260           double: 0,             \
261           long double: 0,        \
262           default: (X) )

```

This has the advantage, that a call to `TRACE_POINTER(X)` can always be assigned to a `void*`. Either X itself is a pointer, so assignable to `void*`, or it is of another arithmetic type and the result of the macro invocation is 0. Put all together `TRACE_CONVERT` looks as follows:

```

macro_trace.h
264 /**
265  ** @brief Return a value that is either a promoted to a wide
266  ** integer, a floating point or a @c void* if @a X is a
267  ** pointer.
268  **/
269 #define TRACE_CONVERT(X)          \
270 _Generic( (X)+0LL,               \
271           unsigned long long: (X)+0LL, \
272           long long: (X)+0LL,          \
273           float: (X)+0LL,              \
274           double: (X)+0LL,             \
275           long double: (X)+0LL,        \
276           default: ((void*){ 0 } = TRACE_POINTER(X)) )

```

<sup>16</sup>Remember that conversion from non-zero integers to pointers must be made explicit through a cast.

17.5.1. *Default arguments.* Some functions of the C library have parameters that receive the same boring arguments most of the time. This is *e.g* the case for `strtoul` and relatives. Remember that these receive three arguments:

```
1 unsigned long int strtoul(char const nptr[restrict],
2                          char** restrict endptr,
3                          int base);
```

The first is the string that we want to convert into an **unsigned long**. `endptr` will point to the end of the number in the string, and `base` is the integer base for which the string is interpreted. Two special conventions apply: if `endptr` may be a null pointer and if `base` is 0, the string is interpreted as hexadecimal (leading "0x"), octal (leading "0") or decimal otherwise.

Most of the time, `strtoul` is used without the `endptr` feature and with the symbolic base set to 0, *e.g* in something like

```
1 int main(int argc, char* argv[argc+1]) {
2     if (argc < 2) return EXIT_FAILURE;
3     size_t len = strtoul(argv[1], 0, 0);
4     ...
5 }
```

to convert the first command line argument of a program to a length value. To avoid this repetition and to have the reader of the code concentrate on the important things, we can introduce an intermediate level of macros that provide these 0 arguments if they are omitted:

```
generic.h
114 /**
115  ** @brief Call a three-parameter function with default arguments
116  ** set to 0
117  **/
118 #define ZERO_DEFAULT3(...) ZERO_DEFAULT3_0(__VA_ARGS__, 0, 0, )
119 #define ZERO_DEFAULT3_0(FUNC, _0, _1, _2, ...) FUNC(_0, _1, _2)
120
121 #define strtoul(...) ZERO_DEFAULT3(strtoul, __VA_ARGS__)
122 #define strtoull(...) ZERO_DEFAULT3(strtoull, __VA_ARGS__)
123 #define strtol(...) ZERO_DEFAULT3(strtol, __VA_ARGS__)
124 #define strtoll(...) ZERO_DEFAULT3(strtoll, __VA_ARGS__)
```

Here, the macro `ZERO_DEFAULT3` works by subsequent addition and removal of arguments. It is supposed to receive a function name and at least one first argument that is to be passed to that function. First, a set of 0 is appended to the argument list and then, if this results in more than three combined arguments, the excess is omitted. So for a call with just one argument the sequence of replacements looks as follows:

```
strtoul(argv[1])
//      ...
ZERO_DEFAULT3(strtoul, argv[1])
//      ...
ZERO_DEFAULT3_0(strtoul, argv[1], 0, 0, )
//      FUNC    , _0    , _1, _2, ...
strtoul(argv[1], 0, 0)
```

Because of the special rule that inhibits recursion in macro expansion, the final function call to `strtoul` will not be expanded further and passed on to the next compilation phases.

If instead we call `strtoul` with three arguments

```

strtoul(argv[1], ptr, 10)
// ...
ZERO_DEFAULT3(strtoul, argv[1], ptr, 10)
// ...
ZERO_DEFAULT3_0(strtoul, argv[1], ptr, 10, 0, 0, )
//          FUNC , _0 , _1 , _2, ...
strtoul(argv[1], ptr, 10)

```

the sequence of replacements effectively results in exactly the same tokens with which we started.

## 18. Variations in Control Flow

The *control flow*, see Figure 1, of a program execution describes how the individual statements of the program code are *sequenced*, that is, which statement is executed after another. Up to now we have mostly looked at code that let us deduce this control flow from syntax and some controlling expression. By that, each function can be described by a hierarchical composition of *basic blocks*. A basic block is a maximum sequence of statements such that once execution starts at the first of these statements it continues unconditionally until the last, and, such that all execution of any statement in the sequence starts with the first.

If we are supposing that all conditionals and loop statements use `{ }` blocks, in a simplified view such a basic block

- starts either at the beginning of a `{ }`-block or a **case** or jump label, and
- ends either at the end of the corresponding `{ }` block or at the next
  - statement that is target of a **case** or jump label,
  - body of a conditional or loop statement,
  - **return** statement,
  - **goto** statement, or
  - call to a function with special control flow.

Observe, in that definition no exception is made for general function calls: these are seen to temporarily suspend execution of a basic block but not to end it. Among the functions with special control flow that end a basic block are some that we know already, namely those that are marked with the keyword `_Noreturn` such as **exit** or **abort**. Another such function is **setjmp** which may return more than once, see below.

Code that is just composed of basic blocks that are stitched together by **if/else**<sup>17</sup> or loop statements has the double advantage of being easily readable for us, humans, and to lead to better optimization opportunities for the compiler. Both can directly deduce the lifetime and access pattern of variables and compound literals in basic blocks, and then capture how these are mended by the hierarchical composition of the basic blocks into their function.

A theoretical foundation of this structured approach has been given quite early for Pascal programs by Nishizeki et al. [1977] and extended to C and other imperative languages by Thorup [1995]. They prove that structured programs, that is programs without **goto** or other arbitrary jump constructs, have a control flow that matches nicely onto a “tree-like” decomposition that can be deduced from the syntactical nesting of the program. Unless you don’t have to, you should stick to that programming model.

Nevertheless, some exceptional situations require exceptional measures. Generally, changes of the control flow of a program can originate from

**conditional statements:** **if/else**, **switch/case**,  
**loop statements:** **do { } while ( )**, **while ( )**, **for ( )**,

<sup>17</sup>**switch/case** statements complicate the view a bit.



**functions:** function calls, **return** statement or **\_\_Noreturn** specification

**short jumps:** **goto** and labels,

**long jumps:** **setjmp/longjmp**, `getcontext/setcontext`<sup>18</sup>

**interrupts:** signals and **signal** handlers,

**threads:** **thrd\_create, thrd\_exit**

These changes in flow control can mix up the knowledge that the compiler has of the abstract state of the execution. Roughly, the complexity of the knowledge that a human or mechanical reader has to track increases from top to bottom in that list. Up to now we only have seen the first four constructs. These correspond to *language* features, features that are determined by syntax, such as keywords, or by operators, such as the `()` of a function call. The latter three are introduced by *C library* interfaces. They provide changes in the control flow of a program that can jump across function boundaries (**longjmp**), can be triggered by events that are external to the program (interrupts), or can even establish a concurrent control flow, another *thread of execution*.

Various difficulties may arise when objects are under the effect of unexpected control flow:

- Objects could be used outside their lifetime.
- Objects could be used uninitialized.
- Values of objects could be misinterpreted by optimizing. (**volatile**)
- Objects could be partially modified. (**sig\_atomic\_t, atomic\_flag, \_Atomic** with *lock free* property and *relaxed* consistency)
- Updates to objects could be sequenced unexpectedly. (all **\_Atomic**)
- Execution must be guaranteed to be exclusive inside a *critical section*. (**mtx\_t**)

Because access to the objects that constitute the state of a program becomes complicated, C provides features that help to cope with the difficulties. Above, they are noted in parenthesis, and we will discuss them in detail in the following sections.

**18.1. Sequencing.** Before we can look into the details how the control flow of a program can change in unexpected ways, we must better understand what the “normal” way of a sequence of C statements guarantees, and what it does not. We have already seen in Section 4.5 that the evaluation of C *expressions* does not necessarily follow the lexicographical order as they are written. *E.g.* the evaluation of function arguments can occur in any order. The different expressions that constitute the arguments can even be interleaved to the discretion of the compiler, or depending on the availability of resources at execution time. We say that function argument expressions are *unsequenced*.

There are several reasons for establishing only relaxed rules for evaluation. One is to allow for the easy implementation of optimizing compilers. Efficiency of the compiled code has always been a strong point of C compared to other programming languages.

But another one is probably also that C does not add arbitrary restrictions when they don't have a convincing mathematical or technical foundation. Mathematically, the two operands *a* and *b* in *a+b* are freely exchangeable. Imposing an evaluation order would break this rule, and arguing about a C program would become more complicated. *E.g.* the set of values for an unsigned integer type then would not have the property of forming a finite ring.

In the absence of threads, most of C's formalization of this is done with so-called *sequence points*. These are points in the syntactical specification of the program that impose a serialization of the execution. But we will also later see additional rules that force sequencing between the evaluation of certain expressions that don't imply sequence points.

On a high level, a C program can be seen as a series of sequence points that are reached one after the other, and the code between such sequence points may be executed in any order, be interleaved or obey certain other sequencing constraints. In the simplest

<sup>18</sup>Defined in POSIX systems.



case, *e.g.* two statements that are separated by a `;`, a statement before a sequence point is *sequenced* before the statement after the sequence point.

But even the existence of sequence points may not impose a particular order between two expressions, it only imposes that there is *some* order. To see that, consider the following code that is “well defined”:

```

sequence_point.c
3 unsigned add(unsigned* x, unsigned const* y) {
4     return *x += *y;
5 }
6 int main(void) {
7     unsigned a = 3;
8     unsigned b = 5;
9     printf("a=%u, b=%u\n", add(&a, &b), add(&b, &a));
10 }

```

From Section 4.5 we remember that the two arguments to `printf` can be evaluated in any order, and the rules for sequence points that we will see below will tell us that the function calls to `add` impose sequence points. As a result, we have two possible outcomes of this code. Either, the first `add` is executed first, entirely, and then the second, or the other way around. For the first possibility we have:

- `a` is changed to 8 and that value is returned.
- `b` is changed to 13 and that value is returned.

The output of such an execution is

```

Terminal
0 a = 8, b = 13

```

For the second we get:

- `b` is changed to 8 and that value is returned.
- `a` is changed to 11 and that value is returned.

with the output

```

Terminal
0 a = 11, b = 8

```

That is, although the behavior of this program is defined, its outcome is not completely determined by the C standard. The specific terminology that the C standard applies to such a situation is that the two calls are *indeterminately sequenced*. This is not just a theoretical discussion, the two commonly used open source C compilers `gcc` and `clang` differ on this simple code. Let me stress on this again: all of this is defined behavior. Don’t expect a compiler to warn you on such problems.

### Rule 3.18.1.1 Side effects in functions can lead to indeterminate results.

Here is a list of all sequence points that are defined in terms of C’s grammar:

- The end of a statement, either by semicolon (`;`) or closing brace (`}`).
- The end of an expression before the comma operator (`,`).<sup>19</sup>
- The end of a declaration, either by semicolon (`;`) or comma (`,`).<sup>20</sup>

<sup>19</sup>Be careful, commas that separate function arguments are not in this category.

<sup>20</sup>This also holds for a comma that ends the declaration of an enumeration constant.

- The end of the controlling expressions of **if**, **switch**, **for**, **while**, conditional evaluation (`?:`) or short circuit evaluation (`||` and `&&`).
- After the evaluations of the function designator (usually a function name) and the function arguments of a function call<sup>21</sup> but before the actual call.
- The end of a **return** statement.

There are other sequencing restrictions besides those implied by sequence points. The first two are more or less obvious but should be stated nevertheless:

**Rule 3.18.1.2** *The specific operation of any operator is sequenced after the evaluation of all its operands.*

**Rule 3.18.1.3** *The effect of updating an object by any of the assignment, increment or decrement operators is sequenced after the evaluation of its operands.*

For function calls there also is an additional rule, that says that execution of a function is always completed before any other expression:

**Rule 3.18.1.4** *A function call is sequenced with respect to all evaluations of the caller.*

As we have seen above, this might be indeterminately sequenced, but sequenced nevertheless.

Another source of indeterminately sequenced expressions originate from initializers:

**Rule 3.18.1.5** *Initialization list expressions for array or structure types are indeterminately sequenced.*

Last but not least, some sequence points are also defined for the C library:

- After the actions of format specifiers of the IO functions.
- Before any C library function returns.<sup>22</sup>
- Before and after calls to the comparison functions used for searching and sorting.

The later two impose similar rules to C library functions as to ordinary functions. This is needed because the C library itself might not necessarily be implemented in C.

**18.2. Short jumps.** We already have seen a feature that interrupts the common control flow of a C program: **goto**. As you hopefully remember from Section 15, this is implemented with two constructs: *labels* mark positions in the code and **goto** statements *jump* to these marked positions *inside the same function*.

We also have seen that such jumps already have complicated implications on the lifetime and visibility of local objects. In particular, there is a difference in the lifetime of objects that are defined inside loops and inside a set of statements that is repeated by **goto**<sup>23</sup>. Consider the following two snippets:

```

1  size_t* ip = 0
2  while(something)
3      ip = &(size_t){ fun() };          /* life ends with while */
4                                          /* good: resource is freed */
5  printf("i_is_%d", *ip)                /* bad: object is dead */

```

versus

<sup>21</sup>This sees the function designator on the same level as the function arguments.

<sup>22</sup>Beware that library functions that are implemented as macros, may not define a sequence point.

<sup>23</sup>see ISO 9899:2011 6.5.2.5 p16

```

1  size_t* ip = 0
2  RETRY:
3      ip = &(size_t){ fun() };           /* life continues */
4      if (condition) goto RETRY;
5                                          /* bad: resource is blocked */
6  printf("i_is_%d", *ip)                 /* good: object is alive */

```

Both define a local object in a loop by using a compound literal. The address of that compound literal is assigned to a pointer and so the object remains accessible outside the loop, and can *e.g.* be used in a `printf` statement.

It looks as if they both are semantically equivalent, but they are not. For the first, the object that corresponds to the compound literal only lives in the scope of the `while` statement.

**Rule 3.18.2.1** *Each iteration defines a new instance of a local object.*

Therefore the access to the object in the `*ip` expression is invalid. When omitting the `printf` in the example, the `while` loop has the advantage that the resources that are occupied by the compound literal can be reused.

For the second there is no such restriction: the scope of the definition of the compound literal is the whole surrounding block. So by Rule 2.13.2.12 the object is alive until that block is left. This is not necessarily good: the object occupies resources that could otherwise be reassigned.

In cases where there is no need for the `printf` statement (or similar access), the first snippet is clearer and has better optimization opportunities. Therefore under most circumstances it is preferable.

**Rule 3.18.2.2** *`goto` should only be used for exceptional changes in control flow.*

Here exceptional usually means that we encounter an transitional error condition that requires local cleanup, such as we have shown in Section 15. But it could also mean specific algorithmic conditions as we can see in Listing 3.11.

The function `descend` implements a simple *recursive descent parser* that recognizes `{ }` constructs in a text given on `stdin` and indents this text on output, according to the nesting of the `{ }`. More formally this function detects text as of the following recursive definition.

**program** := some-text<sub>\*</sub> [ ' { ' program' ' } ' some-text<sub>\*</sub> ]<sub>\*</sub>

and prints such a program conveniently by changing the line structure and indentation.

Here two labels, `NEW_LINE` and `ASCEND`, and two macros, `LEFT` and `RIGHT`, reflect the actual state of the parsing. `NEW_LINE` is jump target when a new line is to be printed, `ASCEND` is used if a `}` is encountered or if the stream ended. `LEFT` and `RIGHT` are used as `case` labels if left or right curly braces are detected.

The reason to have `goto` and labels, here, is that both states are detected in two different places of the function, and different levels of nesting. In addition, the names of the labels reflect their purpose and thereby provide additional information about the structure.

**18.3. Functions.** Function `descend` has more complications than the twisted local jump structure, it also is recursive. As we already have seen, C handles recursive functions quite simply.

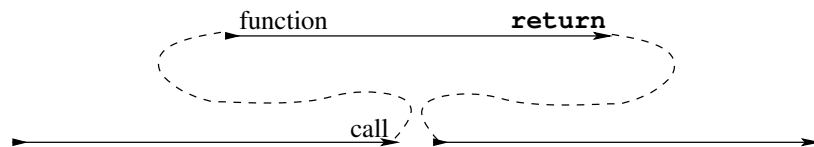
**Rule 3.18.3.1** *Each function call defines a new instance of a local object.*

LISTING 3.11. A simple recursive descent parser for code indentation.

```

60 static
61 char const* descend(char const* act,
62                     unsigned dp[restrict static 1], // bad TBR
63                     size_t len, char buffer[len],
64                     jmp_buf jmpTarget) {
65     if (dp[0]+3 > sizeof head) longjmp(jmpTarget, tooDeep);
66     ++dp[0];
67     NEW_LINE:                                     // loop on output
68     while (!act || !act[0]) {                     // loop for input
69         if (interrupt) longjmp(jmpTarget, interrupted);
70         act = skipSpace(fgets(buffer, len, stdin));
71         if (!act) {                               // end of stream
72             if (dp[0] != 1) longjmp(jmpTarget, plusL);
73             else goto ASCEND;
74         }
75     }
76     fputs(&head[sizeof head - (dp[0] + 2)], stdout); // header
77
78     for (; act && act[0]; ++act) { // remainder of the line
79         switch (act[0]) {
80             case LEFT:                          // descend on left brace
81                 act = end_line(act+1, jmpTarget);
82                 act = descend(act, dp, len, buffer, jmpTarget);
83                 act = end_line(act+1, jmpTarget);
84                 goto NEW_LINE;
85             case RIGHT:                         // return on right brace
86                 if (dp[0] == 1) longjmp(jmpTarget, plusR);
87                 else goto ASCEND;
88             default:                            // print char and go on
89                 putchar(act[0]);
90         }
91     }
92     goto NEW_LINE;
93     ASCEND:
94     --dp[0];
95     return act;
96 }

```

FIGURE 1. Control flow of function calls, **return** jumps to the next instruction after the call.

So usually different recursive calls to the same function that are active simultaneously don't interact, everybody has their own copy of the program state.

But here, because of the pointers, this principle is weakened. The data to which `buffer` and `dp` point is modified. For `buffer` this is probably unavoidable, it will contain the data that we are reading. But `dp` could (and should) be replaced by a simple **unsigned**

argument.<sup>[Exs 24]</sup> Our implementation only has `dp` as pointer because we want to be able to track the depth of the nesting in case an error occurs. So if we abstract from the calls to `longjmp` that we did not yet explain, using such a pointer is bad. The state of the program is more difficult to follow, and we miss optimization opportunities.<sup>[Exs 25]</sup>

In our particular example, because `dp` is **restrict** qualified and not passed to the calls to `longjmp` (see below) and it is only incremented at the beginning and decremented at the end, `dp[0]` is restored to its original value just before the return from the function. So, seen from the outside, it appears that `descend` doesn't change that value at all.

If the function code of `descend` is visible at the call side, a good optimizing compiler can deduce that `dp[0]` did not change through the call. If `longjmp` weren't special, this would be a nice optimization opportunity. Below we will see, how the presence `longjmp` invalidates this optimization and in fact leads to buggy code.

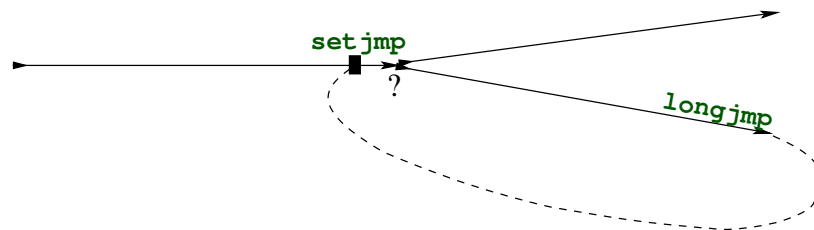


FIGURE 2. Control flow with `set jmp` and `longjmp`, `longjmp` jumps to the position marked by `set jmp`.

**18.4. Long jumps.** Our function `descend` may also encounter exceptional conditions that can not be repaired. We use an enumeration type to name them. Here, `eofOut` is reached if `stdout` can't be written, and `interrupted` refers to an *asynchronous signal* that our running program received. We will discuss this concept later.

```

32  /**
33  ** @brief exceptional states of the parse algorithm
34  **/
35  enum state {
36      execution = 0,      /*< normal execution
37      plusL,              /*< too many left parenthesis
38      plusR,              /*< too many right parenthesis
39      tooDeep,           /*< nesting too deep to handle
40      eofOut,            /*< end of output
41      interrupted,      /*< interrupted by signal
42  };

```

We use the function `longjmp` to deal with these situations, and we place the corresponding calls directly at the place in the code where we recognize that such condition is reached.

- `tooDeep` is easily recognized at the beginning of the function;
- `plusL` can be detected when we encounter the end of the input stream while we are not at the first recursion level;
- `plusR` occurs when we encounter a closing `}` while we are at the first recursion level;

[Exs 24] Change `descend` such that it receives an `unsigned` `depth` instead of a pointer.

[Exs 25] Compare the assembler output of the initial version against your version without `dp` pointer.

- `eofOut` is reached if a write to `stdout` returned an “end of file”, `EOF`, condition;
- and `interrupted` is checked before each new line that is read from `stdin`.

Since `stdout` is line buffered, we only check for `eofOut` when we write the `'\n'` character. This happens inside the short function `end_line`:

```

48 char const* end_line(char const* s, jmp_buf jmpTarget) {
49     if (putchar('\n') == EOF) longjmp(jmpTarget, eofOut);
50     return skipSpace(s);
51 }

```

basic\_blocks.c

```
#include <setjmp.h>
```

The function `longjmp` comes with a companion macro `setjmp` that is used to establish a jump target to which a call of `longjmp` may refer. The header `setjmp.h` provides the following prototypes:

```

_Noreturn void longjmp(jmp_buf target, int condition);
int setjmp(jmp_buf target);    // really a macro, not a function

```

The function `longjmp` also has the `_Noreturn` property and so we are assured that once we detect one of the exceptional conditions, execution of the current call to `descend` will never continue.

**Rule 3.18.4.1** *`longjmp` never returns to the caller.*

This is a valuable information to the optimizer. In `descend`, `longjmp` is called in 5 different places and the compiler can substantially simplify the analysis of the different branches. *E.g* after the tests `!act` it can be assumed that `act` is non-null on entry to the `for` loop.

Normal syntactical labels are only valid `goto` targets within the same function as they are declared. In contrast to that, a `jmp_buf` is an opaque object that can be declared anywhere and that can be used as long as it is alive and its contents is valid. In `descend` we use just one jump target of type `jmp_buf`, that we declare as a local variable. This jump target is setup in the base function `basic_blocks` that serves as interface to `descend`, see Listing 3.12. This function mainly consist of one big `switch` statement that handles all the different conditions.

The 0 branch of that `switch` is taken when we come here through the normal control flow. This is one of the basic principles for `setjmp`:

**Rule 3.18.4.2** *When reached through normal control flow, a call to `setjmp` marks the call location as a jump target and returns 0.*

As said, `jmpTarget` must be alive and valid when we call `longjmp`. So for an `auto` variable, the scope of declaration of the variable must not have been left, otherwise it would be dead. For the validity, all of the context of the `setjmp` must still be active when we call `longjmp`. Here, we avoid complications by having `jmpTarget` declared in the same scope as the call to `setjmp`.

**Rule 3.18.4.3** *Leaving the scope of a call to `setjmp` invalidates the jump target.*

Once we enter `case 0` and call `descend` we may end up in one of the exceptional conditions and call `longjmp` to terminate the parse algorithm. This passes back control to the call location that had been marked in `jmpTarget` just as if we just returned from the call to `setjmp`. The only visible difference is that now the return value is the condition that we passed as a second argument to `longjmp`. If for example we encountered the `tooDeep` condition at the beginning of a recursive call to `descend` and

LISTING 3.12. The user interface for the recursive descent parser.

```

100 void basic_blocks(void) {
101     char buffer[maxline];
102     unsigned depth = 0;
103     char const* format =
104         "All_%0.0d%c%c_blocks_have_been_closed_correctly\n";
105     jmp_buf jmpTarget;
106     switch (setjmp(jmpTarget)) {
107     case 0:
108         descend(0, &depth, maxline, buffer, jmpTarget);
109         break;
110     case plusL:
111         format =
112             "Warning:_%d_%c%c_blocks_have_not_been_closed_properly\n";
113         break;
114     case plusR:
115         format =
116             "Error:_closing_too_many_(%d)_%c%c_blocks\n";
117         break;
118     case tooDeep:
119         format =
120             "Error:_nesting_(%d)_of_%c%c_blocks_is_too_deep\n";
121         break;
122     case eofOut:
123         format =
124             "Error:_EOF_for_stdout_at_nesting_(%d)_of_%c%c_blocks\n";
125         break;
126     case interrupted:
127         format =
128             "Interrupted_at_level_%d_of_%c%c_block_nesting\n";
129         break;
130     default:;
131         format =
132             "Error:_unknown_error_within_(%d)_%c%c_blocks\n";
133     }
134     fflush(stdout);
135     fprintf(stderr, format, depth, LEFT, RIGHT);
136     if (interrupt) {
137         SH_PRINT(stderr, interrupt,
138                 "is_somebody_trying_to_kill_us?");
139         raise(interrupt);
140     }
141 }

```

called `longjmp(jmpTarget, tooDeep)`, we jump back to the controlling expression of the `switch` and receive the return value of `tooDeep`. Execution then continues at the corresponding `case` label.

**Rule 3.18.4.4** A call to `longjmp` transfers control directly to the position that was set by `setjmp` as if that had returned the condition argument.

Beware though, that precautions have been taken to make it impossible to cheat and to retake the “normal” path a second time.

**Rule 3.18.4.5** A 0 as condition parameter to `longjmp` is replaced by 1.

The `set jmp/long jmp` mechanism is very powerful and can avoid a whole cascade of returns from functions calls. In our example, if we allow the maximal depth of nesting of the input program of 30, say, the detection of the `tooDeep` condition will happen when there are 30 active recursive calls to `descend`. A regular error return strategy would `return` to each of these and do some work on each level. A call to `long jmp` allows us to shorten all these returns and proceed the execution directly in the `switch` of `basic_blocks`.

Because `set jmp/long jmp` is allowed to make some simplified assumptions, this mechanism is surprisingly efficient. Depending on the processor architecture it usually needs no more than 10 to 20 assembler instructions. The strategy followed by the library implementation is usually quite simple: `set jmp` saves the essential hardware registers, including stack and instruction pointers, in the `jmp_buf` object, `long jmp` restores them from there and passes control back to the stored instruction pointer.<sup>26</sup>

One of the simplifications `set jmp` makes is about its return. Its specification says it returns an `int` value, but this value cannot be used inside arbitrary expressions.

**Rule 3.18.4.6** *`set jmp` may only be used in simple comparisons inside controlling expression of conditionals.*

So it can be used directly in a switch statement as in our example, it can be tested for `==`, `<` etc, but the return value of `set jmp` may not be used in an assignment. This guarantees that the `set jmp` value is only compared to a known set of values, and the change in the environment when returning from `long jmp` may just be a special hardware register that controls the effect of conditionals.

As said, this saving and restoring of the execution environment by the `set jmp` call is minimal. Only a minimal necessary set of hardware registers are saved and restored. No precautions are taken get local optimizations in line or even to take the fact into account that the call location may be visited a second time.

**Rule 3.18.4.7** *Optimization interacts badly with calls to `set jmp`.*

If you execute and test the code in the example, you will see that there actually is a problem in our simple usage of `set jmp`. If we trigger the `plusL` condition, namely if we feed a partial program with missing closing `}` we would expect the diagnostic to read something like

Terminal	Warning: 3 { } blocks have not been closed properly
----------	---

Depending on the optimization level of your compilation, instead of the 3 you will most probably see a 0, independently of the input program. This is because the optimizer does an analysis based on the assumption that the `switch` cases are mutually exclusive. It only suspects the value of `depth` to change if execution goes through `case 0` and thus the call of `descend`. From inspection of `descend`, see Section 18.3, we know that the value of `depth` is always restored to its original value before return, and so the compiler may assume that the value doesn't change through this code path. Then, none of the other `case` s changes `depth`, so the compiler can assume that `depth` is always 0 for the `fprintf` call.

As a consequence, optimization can't make correct assumptions about objects that are changed in the normal code path of `set jmp` and referred to in one of the exceptional paths. There is only one recipe against that:

<sup>26</sup>For the vocabulary of this you might want to read or re-read Section 13.4.



**Rule 3.18.4.8** *Objects that are modified across **longjmp** must be **volatile**.*

Syntactically, the qualifier **volatile** applies similar to the other qualifiers **const** and **restrict** that we already encountered. If we declare `depth` with that qualifier

```
unsigned volatile depth = 0;
```

and amend the prototype of **descend** accordingly, all accesses to this object will use the value that is stored in memory. Optimizations that try to make assumptions about its value are blocked out.

**Rule 3.18.4.9** ***volatile** objects are reloaded from memory each time they are accessed.*

**Rule 3.18.4.10** ***volatile** objects are stored to memory each time they are modified.*

So **volatile** objects are protected from optimization, or, if we look at it negatively, they inhibit optimization. Therefore you should only make objects **volatile** if you really need them to be.<sup>[Exs 27]</sup>

Finally, note some subtleties of the **jmp\_buf** type. Remember that it is an opaque type, you should never make assumptions about its structure or its individual fields.

**Rule 3.18.4.11** *The **typedef** for **jmp\_buf** hides an array type.*

And because it is an opaque type, we don't know anything about the base type, `jmp_buf_base`, say, of the array. By that:

- An object of type **jmp\_buf** cannot be assigned to.
- A **jmp\_buf** function parameter is rewritten to a pointer to `jmp_buf_base`.
- Such a function always refers to the original object and not to a copy.

In a way, this emulates a pass by reference mechanism, for which other programming languages such as C++ have explicit syntax. Generally, using this trick is not a good idea, the semantics of a **jmp\_buf** depends on being a locally declared variable (**basic\_blocks**, unassignable) or a function parameter (**descend**, modifiable because pointer), and we cannot use the more specific declarations from Modern C for the function parameter that would be adequate, namely something like

```
jmp_buf_base jmpTarget[restrict const static 1]
```

to insist that the pointer shouldn't be changed inside the function, that it must not be 0 and that the access to it can be considered unique for the function. As of today, we would not design this type like this and you should not try to copy that trick for the definition of your own types.

**18.5. Signal handlers.** As we have seen, **setjmp/longjmp** can be used to handle exceptional conditions that we detect ourselves during the execution of our code. A *signal handler* is a tool that handles exceptional conditions that arise differently, namely that are triggered by some event that is external to the program. Technically, there are two types of such external events: *hardware interrupts*, also referred to as *traps* or *synchronous signals*, and *software interrupts* or *asynchronous signals*.

The first occurs when the processing device encounters a severe fault that it cannot deal with, e.g a division by zero, addressing an non-existing memory bank or using a

<sup>[Exs 27]</sup> Your version of **descend** that passes `depth` as value, might not propagate the depth correctly if it encounters the **plusL** condition. Ensure that it copies that value to an object that can be used in by the **fprintf** call in **basic\_blocks**.



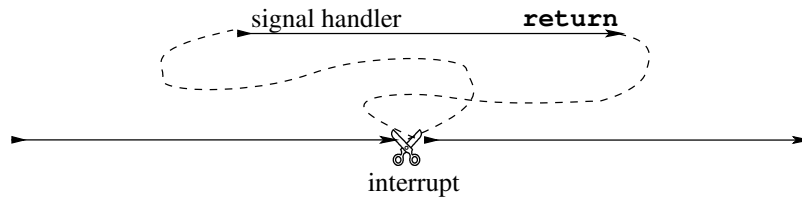


FIGURE 3. Control flow after an interrupt, **return** jumps to the position where the interrupt occurred.

misaligned address in an instruction that operates on a wider integer type. Such an event is *synchronous* with the program execution. It is directly caused by a faulting instruction, and so it can always be known at which particular instruction the interrupt was raised.

The second arises when the operating or runtime system decides that our program should terminate, either because some deadline is exceeded, a user has issued a termination request or the world as we know it is going to end. Such an event is *asynchronous*, because it can fall just in the middle of a multi-stage instruction, leaving the execution environment in an intermediate state.

Most modern processors have a builtin features to handle hardware interrupts, an *interrupt vector table*. This table is indexed by the different hardware faults that the platform knows about. Its entries are pointers to procedures, *interrupt handlers*, that are executed when the specific fault occurs. So if the processor detects such a fault, execution is automatically switched away from the user code, and such an interrupt handler is executed. Such a mechanism is not portable, because the names and locations of the faults are different from platform to platform. It is tedious to handle, because to program a simple application, we'd have to provide all handlers for all interrupts.

C's signal handlers provide us with an abstraction to deal with both types of interrupts, hardware and software, in a portable way. They work similar to what we describe for hardware interrupts, only that

- the names of (some of) the faults are standardized;
- all faults have a default handler (which is mostly implementation defined);
- and (most) handlers can be specialized.

In each item of that list there are parenthesized “reservations”, because upon a closer look it appears that C's interface for signal handlers is quite rudimentary and all platforms have their extensions and special rules.

**Rule 3.18.5.1** *C's signal handling interface is minimal and should only be used for elementary situations.*

`#include <signal.h>`

The interface is defined in the header `signal.h`. The C standard distinguishes six different values, called *signal numbers*. Three of these are typically caused by hardware interrupts<sup>28</sup>:

<b>SIGFPE</b>	an erroneous arithmetic operation, such as zero divide or an operation resulting in overflow
<b>SIGILL</b>	detection of an invalid function image, such as an invalid instruction
<b>SIGSEGV</b>	an invalid access to storage

The other three are usually triggered by software or users:

<sup>28</sup>Called computational exceptions by the standard.

**SIGABRT** abnormal termination, such as is initiated by the **abort** function  
**SIGINT** receipt of an interactive attention signal  
**SIGTERM** a termination request sent to the program

A specific platform will have other signal numbers, the standard reserves all identifiers starting with **SIG** for that purpose. Their use is undefined as of the C standard, but as such there is nothing bad about it. Undefined here really means what it says, if you use it, it has to be defined by some other authority than the C standard, *e.g.* your platform provider. Your code becomes less portable in consequence.

There are two standard dispositions for handling signals, both also represented by symbolic constants. **SIG\_DFL** refers to a default how a particular signal is normally handled on a given platform, and **SIG\_IGN** indicates that a particular signal is to be ignored. Then the programmer may write their own signal handlers. The handler for our parser looks quite simple:

```

143  /**
144  ** @brief a minimal signal handler
145  **
146  ** After updating the signal count, for most signals this
147  ** simply stores the signal value in "interrupt" and returns.
148  **/
149  static void signal_handler(int sig) {
150      sh_count(sig);
151      switch (sig) {
152          case SIGTERM: quick_exit(EXIT_FAILURE);
153          case SIGABRT: _Exit(EXIT_FAILURE);
154      #ifdef SIGCONT
155          // continue normal operation
156          case SIGCONT: return;
157      #endif
158      default:
159          /* reset the handling to its default */
160          signal(sig, SIG_DFL);
161          interrupt = sig;
162          return;
163      }
164  }
```

As you can see, such a signal handler receives the signal number `sig` as an argument and **switch** es according to that number. Here we have provisions for signal numbers **SIGTERM** and **SIGABRT**. All other signals are just “handled” by resetting the handler for that number to its default, storing the number in our global variable `interrupt`, and then to return to the point where the interrupt occurred.

The type of a signal handler has to be compatible with<sup>29</sup>

```

71  /**
72  ** @brief prototype of signal handlers
73  **/
74  typedef void sh_handler(int);
```

<sup>29</sup>There is no such *type* defined by the standard, though.

That is, it receives a signal number as argument and doesn't return anything. As such, this interface is quite limited and does not allow to pass enough information, in particular none about the location and circumstances for which the signal occurred.

Signal handlers are established by a call to `signal`, as we can already see in our function `signal_handler`. Here, it is just used to reset the signal disposition to the default. `signal` is one of the two function interfaces that are provided by `signal.h`:

```
#include <signal.h>
```

```
sh_handler* signal(int, sh_handler*);
int raise(int);
```

The return value of `signal` is the handler that had previously been active for the signal, or the special value `SIG_ERR` if an error occurred. Inside a signal handler, `signal` should only be used to change the disposition of the same signal number that was received by the call. The following function has the same interface as `signal` but provides a bit more information about the success of the call:

```

sighandler.c
92  /**
93   ** @ brief Enable a signal handler and catch the errors.
94   **/
95  sh_handler* sh_enable(int sig, sh_handler* hnd) {
96      sh_handler* ret = signal(sig, hnd);
97      if (ret == SIG_ERR) {
98          SH_PRINT(stderr, sig, "failed");
99          errno = 0;
100     } else if (ret == SIG_IGN) {
101         SH_PRINT(stderr, sig, "previously_ignored");
102     } else if (ret && ret != SIG_DFL) {
103         SH_PRINT(stderr, sig, "previously_set_otherwise");
104     } else {
105         SH_PRINT(stderr, sig, "ok");
106     }
107     return ret;
108 }
```

The `main` function for our parser uses this in a loop to establish signal handlers for all signal numbers that it may:

```

basic_blocks.c
187  // establish signal handlers
188  for (unsigned i = 1; i < sh_known; ++i)
189      sh_enable(i, signal_handler);
```

As an example a on my machine this provides the following information at the startup of the program:

	Terminal
0	sighandler.c:105: #1 (0 times), unknown signal number, ok
1	sighandler.c:105: SIGINT (0 times), interactive attention signal, ok
2	sighandler.c:105: SIGQUIT (0 times), keyboard quit, ok
3	sighandler.c:105: SIGILL (0 times), invalid instruction, ok
4	sighandler.c:105: #5 (0 times), unknown signal number, ok
5	sighandler.c:105: SIGABRT (0 times), abnormal termination, ok
6	sighandler.c:105: SIGBUS (0 times), bad address, ok
7	sighandler.c:105: SIGFPE (0 times), erroneous arithmetic operation, ok
8	sighandler.c:98: SIGKILL (0 times), kill signal, failed: Invalid argument
9	sighandler.c:105: #10 (0 times), unknown signal number, ok
10	sighandler.c:105: SIGSEGV (0 times), invalid access to storage, ok
11	sighandler.c:105: #12 (0 times), unknown signal number, ok
12	sighandler.c:105: #13 (0 times), unknown signal number, ok
13	sighandler.c:105: #14 (0 times), unknown signal number, ok
14	sighandler.c:105: SIGTERM (0 times), termination request, ok
15	sighandler.c:105: #16 (0 times), unknown signal number, ok
16	sighandler.c:105: #17 (0 times), unknown signal number, ok
17	sighandler.c:105: SIGCONT (0 times), continue if stopped, ok
18	sighandler.c:98: SIGSTOP (0 times), stop process, failed: Invalid argument

The second function **raise** can be used to deliver the specified signal to the current execution. We already used it at the end of **basic\_blocks** to deliver the signal that we had caught to the pre-installed handler.

The mechanism of signals is similar to **setjmp/longjmp**: the current state of execution is memorized, control flow is passed to the signal handler and a return from there restores the original execution environment and continues execution. The difference is that there is no special point of execution that is “marked” by a call to **setjmp**.

**Rule 3.18.5.2** *Signal handlers can kick in at any point of execution.*

Interesting signal numbers in our case are the software interrupts **SIGABRT**, **SIGTERM** and **SIGINT**, that usually can be sent to the application with some magic keystroke such as Ctrl-C. The first two will call **\_Exit** and **quick\_exit**, respectively. So if the program receives these signals, execution will be terminated. For the first, without calling any cleanup handlers, for the second by going through the list of cleanup handlers that were registered with **at\_quick\_exit**.

**SIGINT** will chose the **default** case of the signal handler and so it will eventually return to the point where the interrupt occurred.

**Rule 3.18.5.3** *After return from a signal handler, execution resumes exactly where it was interrupted.*

If that interrupt had occurred in function **descend**, it would first continue execution as if nothing had happened. Only when the current input line is processed and new one is needed, the variable **interrupt** will be checked and the execution will be wound down by calling **longjmp**. And effectively, the only difference between the situation before the interrupt and after, is that the variable **interrupt** has changed its value.

We also have a special treatment of a signal number that is not described by the C standard, **SIGCONT**, but come from my operating system, **POSIX**. To remain portable, the use of this signal number is protected by guards. This signal is meant to continue execution of a program that had previously been “stopped” that is for which execution had been suspended. In that case, the only thing to do is to return. By definition we don’t want any modification of the program state whatsoever.

So another difference to the `setjmp/longjmp` mechanism is that for them the return value of `setjmp` changed the execution path. A signal handler on the other hand is not supposed to change the state of execution. We have to invent a suitable convention to transfer information from the signal handler to the normal program. As for `longjmp`, objects that are potentially changed by a signal handler must be **volatile** qualified: the compiler cannot know where interrupt handlers may kick in, and thus all its assumptions about variables that change through signal handling can be false.

But signal handlers face another difficulty.

**Rule 3.18.5.4** *A C statement may correspond to several processor instructions.*

E.g a **double** `x` could be stored in two usual machine words, and a write (assignment) of `x` to memory could need two separate assembler statements to write both halves.

When considering “normal” program execution as we have discussed so far, splitting a C statement into several machine statements has no problem. Such subtleties are not directly observable.<sup>30</sup> With signals, the picture changes. If a such an assignment is split in the middle by the occurrence of a signal, only half of `x` is written, and the signal handler would see an inconsistent version version of it. One half corresponds to the previous value, the other to the new one. Such a zombie representation (half here, half there) must not even be a valid value for **double**.

**Rule 3.18.5.5** *Signal handlers need types with uninterruptible operations.*

Here, the term *uninterruptible operation* refers to an operation that always appears to be indivisible in the context of signal handlers: either it appears not to have started or to be completed. This doesn’t generally mean that they are undivided, just that we will not be able to observe such a division. The runtime system might have to force that property when a signal handler kicks in.

C has three different classes of types that provide uninterruptible operations:

- (1) The type `sig_atomic_t`, an integer type with a minimal width of 8 bit.
- (2) The type `atomic_flag`.
- (3) All `_Atomic` qualified types that have the lock-free property.

The first is present on all historic C platforms. Its use to store a signal number as in our example for variable `interrupt` is fine, but otherwise its guaranties are quite restricted. Only memory load (evaluation) and store (assignment) operations are known to be uninterruptible, other operations aren’t, and the width may be quite limited. In particular,

**Rule 3.18.5.6** *Objects of type `sig_atomic_t` should not be used as counters.*

because a simple `++` operation might already be divided into three (load, increment and store), and because it might easily overflow. The latter could trigger a hardware interrupt, which is really bad if we already are inside a signal handler.

The latter two classes were only introduced by C11 in the prospect of threads, see Section 19, and are only present if the feature test macro `__STDC_NO_ATOMICS__` has not been defined by the platform and if the header `stdatomic.h` has been included. The function `sh_count` uses these features and we will see an example for this below.

```
#include <stdatomic.h>
```

Because signal handlers for asynchronous signals should not access or even change the program state in an uncontrolled way, they cannot call other functions that would do so. Functions that *can* be used in such a context are called *asynchronous signal safe*. Generally, it is difficult to know from an interface specification if a function has this property, and the C standard only guarantees it for only a handful of functions, namely

<sup>30</sup>They are only observable from outside the program because such a program may take more time than expected.

- the `_Noreturn` functions `abort`, `_Exit`, `quick_exit`, that terminate the program;
- `signal` for the same signal number as the signal handler was called;
- some functions that act on atomic objects.

**Rule 3.18.5.7** *Unless specified otherwise, C library functions are not asynchronous signal safe.*

So by the C standard itself a signal handler cannot call `exit` or do any form of IO, but it can use `quick_exit` and the `at_quick_exit` handlers to execute some cleanup code.

As already noted, C's specifications for signal handlers are minimal, often a specific platform will allow for more. Therefore, portable programming with signals is tedious and exceptional conditions should generally be dealt with in a cascade as we have seen it in our examples:

- (1) Exceptional conditions that can be detected and handled locally can be dealt with by using `goto` to a limited number of labels.
- (2) Exceptional conditions that need not or cannot be handled locally should be returned as a special value from functions whenever this is possible, such as returning a null pointer instead of a pointer to object.
- (3) Exceptional conditions that change the global program state can be handled with `setjmp/longjmp` if exceptional return would be expensive or complex.
- (4) Exceptional conditions that result in a signal to be raised, can be caught by a signal handler, but should be handled after the return of the handler in normal flow of execution.

Since even the list of signals that the C standard specifies is minimal, dealing with the different possible conditions becomes complicated. The following shows how we can handle a collection of signal numbers that goes beyond those that are specified in the C standard:

```

sighandler.c
7  #define SH_PAIR(X, D) [X] = { .name = #X, .desc = " D ", }
8
9  /**
10  ** @brief Array that holds names and descriptions of the
11  ** standard C signals.
12  **
13  ** Conditionally, we also add some commonly used signals.
14  **/
15  sh_pair const sh_pairs[] = {
16      /* Execution errors */
17      SH_PAIR(SIGFPE, "erroneous_arithmetic_operation"),
18      SH_PAIR(SIGILL, "invalid_instruction"),
19      SH_PAIR(SIGSEGV, "invalid_access_to_storage"),
20  #ifdef SIGBUS
21      SH_PAIR(SIGBUS, "bad_address"),
22  #endif
23      /* Job control */
24      SH_PAIR(SIGABRT, "abnormal_termination"),
25      SH_PAIR(SIGINT, "interactive_attention_signal"),
26      SH_PAIR(SIGTERM, "termination_request"),
27  #ifdef SIGKILL
28      SH_PAIR(SIGKILL, "kill_signal"),
29  #endif
30  #ifdef SIGQUIT

```

```

31     SH_PAIR(SIGQUIT, "keyboard_quit"),
32 #endif
33 #ifdef SIGSTOP
34     SH_PAIR(SIGSTOP, "stop_process"),
35 #endif
36 #ifdef SIGCONT
37     SH_PAIR(SIGCONT, "continue_if_stopped"),
38 #endif
39 #ifdef SIGINFO
40     SH_PAIR(SIGINFO, "status_information_request"),
41 #endif
42 };

```

where the macro just initializes a an object of type `sh_pair`:

```

sighandler.h
10 /**
11  ** @brief A pair of strings to hold signal information
12  **/
13 typedef struct sh_pair sh_pair;
14 struct sh_pair {
15     char const* name;
16     char const* desc;
17 };

```

The use of `#ifdef` conditionals ensures that signal names that are not standard can be used and the designated initializer within `SH_PAIR` allows us to specify them in any order. Then the size of the array can be used to compute the number of known signal numbers for `sh_known`:

```

sighandler.c
44 size_t const sh_known = (sizeof sh_pairs/sizeof sh_pairs[0]);

```

If the platform has sufficient support for atomics, this information can also be used to define an array of atomic counters such that we can keep track of the number of times a particular signal was raised:

```

sighandler.h
31 #if ATOMIC_LONG_LOCK_FREE > 1
32 /**
33  ** @brief Keep track of the number of calls into a
34  ** signal handler for each possible signal.
35  **
36  ** Don't use this array directly.
37  **
38  ** @see sh_count to update this information.
39  ** @see SH_PRINT to use that information.
40  **/
41 extern unsigned long _Atomic sh_counts[];
42
43 /**
44  ** @brief Use this in your signal handler to keep track of the
45  ** number of calls to the signal @a sig.
46  **
47  ** @see sh_counted to use that information.
48  **/
49 inline
50 void sh_count(int sig) {

```



```

51     if (sig < sh_known) ++sh_counts[sig];
52 }
53
54 inline
55 unsigned long sh_counted(int sig){
56     return (sig < sh_known) ? sh_counts[sig] : 0;
57 }

```

An object that is qualified with `_Atomic` can be used with the same operators as other objects with the same base type, here the `++` operator. In general, such objects are then guaranteed to avoid race conditions with other threads, see below, and they are uninterruptible if the type has the so-called *lock-free* property. The latter here is tested with the feature test macro `ATOMIC_LONG_LOCK_FREE`.

The user interfaces here are `sh_count` and `sh_counted`. These use the array of counters if available and are otherwise replaced by trivial functions:

```

sighandler.h
59 #else
60 inline
61 void sh_count(int sig) {
62     // empty
63 }
64
65 inline
66 unsigned long sh_counted(int sig){
67     return 0;
68 }
69 #endif

```

## 19. Threads

Threads are another variation of control flow that allows to pursue several *tasks* concurrently. Here, a task is a part of the job that is to be executed by a program such that different tasks can be done with no or little interaction between each other.

Our main example for this will be a primitive game that we call B9 that is a variant of Conway's game of life, see Gardner [1970]. It models an matrix of primitive "cells" that are born, live and die according to very simple rules. We divide the game into four different tasks, that each proceeds iteratively. The cells go through *life cycles*, that compute birth or death events for all cells. The graphical presentation in the terminal goes through drawing cycles, that are updated as fast as the terminal allows. Spread between these are key strokes of the user that occur irregularly and that allow the user to add cells at chosen position. Figure 5 shows a schematic view of these tasks for B9.

The four tasks are:

**draw:** This task draws a pictures of cell matrix to the terminal, see Fig. 4.

**input:** This task captures the key strokes, updates a cursor position and creates cells.

**update:** This updates the state of the game from one life cycle to the next.

**account:** This task is tightly coupled with the *update* task and counts the number of living neighboring cells of each cell.

Each such task is executed by a *thread* that follows its own control flow, much as a simple program of its own. If the platform has several processors or cores, these threads may be executed simultaneously. But even if the platform does not have this capacity, the system will interleave the execution of the threads. The execution as a whole will appear to the user *as if* the events that are handled by the tasks are concurrent. This is crucial for our

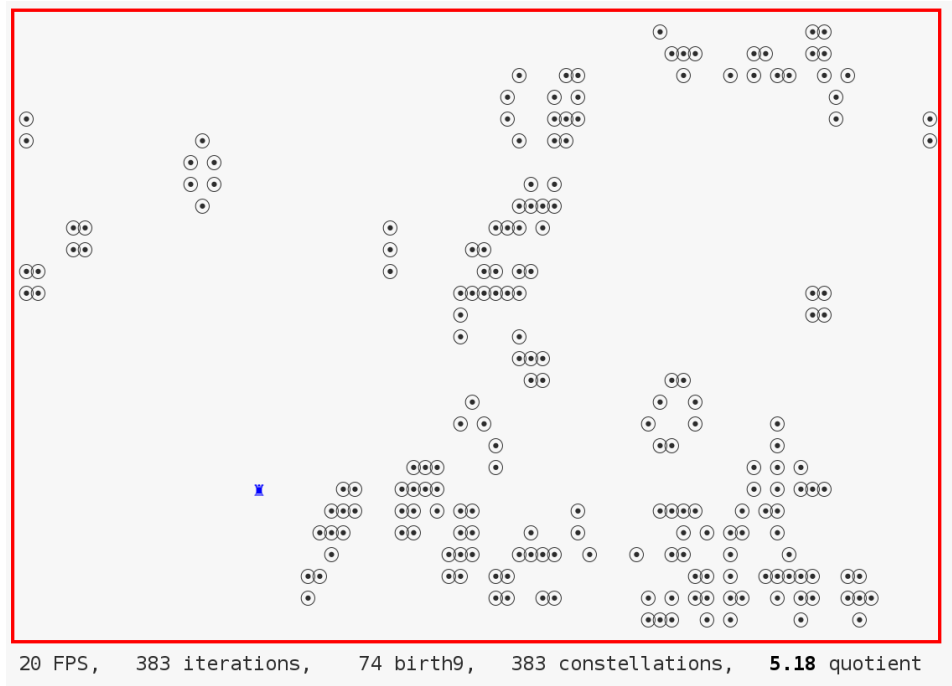


FIGURE 4. A screenshot of B9 showing several cells and the cursor position.

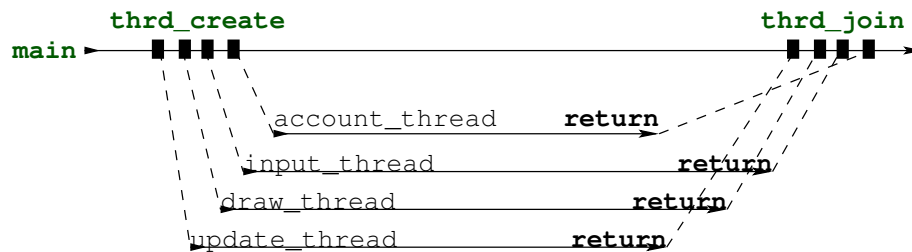


FIGURE 5. Control flow of the five threads of B9.

example, since we want that the game appears to continue constantly whether the player presses keys on the keyboard or not.

Threads in C are dealt through two principal function interfaces that can be used to start a new thread and then to wait for the termination of such a thread.

```
#include <threads.h>
typedef int (*thrd_start_t) (void*);
int thrd_create(thrd_t*, thrd_start_t, void*);
int thrd_join(thrd_t, int *);
```

Here the second argument of `thrd_create` is a function pointer of type `thrd_start_t`. This function is executed at the start of the new thread. As we can see from the `typedef` the function receives a `void*` pointer and returns an `int`. The type `thrd_t` is an opaque type, that will identify the newly created thread.

In our example, four calls in `main` to `thrd_create` create the four threads that correspond to the different tasks. These execute concurrently to the original thread of `main`. At the end, `main` waits for the four threads to terminate, it *joins* them. The four

```
static int update_thread(void*);
static int draw_thread(void*);
static int input_thread(void*);
static int account_thread(void*);
```

```

/* Create an object that holds the game's data. */
life L = LIFE_INITIALIZER;
life_init(&L, n0, n1, M);
/* Create four threads that operate all on that same object and
   collect their ids in "thrd". */
thrd_t thrd[4];
thrd_create(&thrd[0], update_thread, &L);
thrd_create(&thrd[1], draw_thread, &L);
thrd_create(&thrd[2], input_thread, &L);
thrd_create(&thrd[3], account_thread, &L);
/* Wait for the update thread to terminate. */
thrd_join(thrd[0], 0);
/* Tell everybody that the game is over. */
L.finished = true;
ungetc('q', stdin);
/* Wait for the other threads. */
thrd_join(thrd[1], 0);
thrd_join(thrd[2], 0);
thrd_join(thrd[3], 0);

```

```
static
int account_thread(void* Lv) {
    life*restrict L = Lv;
    while (!L->finished) {
```

```
117     }
118     return 0;
119 }
```

```

108 // ~~~~~
109 life_account(L);
110 if ((L->last + repetition) < L->accounted) {
111     L->finished = true;
112 }
113 // ^^^^^

```



In case that this is not possible (or maybe too complicated) C11 has added a storage class and a data type that allow us to handle thread local data. `__thread_local` is a storage class specifier that forces a thread specific copy of the variable that is declared as such. The header `threads.h` also provides a macro `thread_local`, that expands to

```
#include <threads.h>
```

**Rule 3.19.2.3** Use `thread_local` if initialization can be determined at compile time.

If a storage class specifier is not sufficient because you have to do dynamic initialization and destruction, we can use *thread specific storage*, `tss_t`. It abstracts the identification of thread specific data into an opaque id, referred to as `key`, and accessor function to set or get the data:

```
void* tss_get(tss_t key);           // returns pointer to object
int tss_set(tss_t key, void *val); // returns error indication
```

The function that is called at the end of a thread to destroy the thread specific data is specified as a function pointer of type `tss_dtor_t` when the `key` is created.

```
typedef void (*tss_dtor_t)(void*); // pointer to destructor
int tss_create(tss_t* key, tss_dtor_t dtor); // returns error indication
void tss_delete(tss_t key);
```

**19.3. Critical data and critical sections.** Other parts of the `life` structure cannot be protected as easily. They correspond to larger data, e.g. the board positions of the game. Perhaps you remember that arrays may not be qualified with `__Atomic` and even if we would be able to do so by some tricks, the result would not be very efficient. Therefore, we not only declare the members `Mv` (for the game matrix) and `visited` (to hash already visited constellations) but also a special member `mtx`:

```
life.h
10  mtx_t mtx;           //< mutex that protects Mv
11  cnd_t draw;          //< cnd that controls drawing
12  cnd_t acco;          //< cnd that controls accounting
13  cnd_t upda;          //< cnd that controls updating
14
15  void* restrict Mv;    //< bool M[n0][n1];
16  bool (*visited)[life_maxit]; //< hashing constellations
```

This member `mtx` has the special type `mtx_t` type for *mutual exclusion* that also comes with `threads.h`. It is meant to protect the *critical data*, namely `Mv`, while it is accessed in a well identified part of the code, a so-called *critical section*.

```
#include <threads.h>
```

The most simple use case for this mutex is in the center of the drawing thread, where two calls, `mtx_lock` and `mtx_unlock`, protect the access to the `life` data structure `L`.

```
B9.c
121 static
122 int input_thread(void* Lv) {
123     termin_unbuffered();
124     life* restrict L = Lv;
125     enum { len = 32, };
126     char command[len];
127     do {
128         int c = getchar();
129         command[0] = c;
130         switch(c) {
```

```

131 case GO_LEFT : life_advance(L, 0, -1); break;
132 case GO_RIGHT: life_advance(L, 0, +1); break;
133 case GO_UP   : life_advance(L, -1, 0); break;
134 case GO_DOWN : life_advance(L, +1, 0); break;
135 case GO_HOME : L->x0 = 1; L->x1 = 1;    break;
136 case ESCAPE  :
137     ungetc(termin_translate(termin_read_esc(len, command)),
            stdin);
138     continue;
139 case '+':      if (L->frames < 128) L->frames++; continue;
140 case '-':      if (L->frames > 1)   L->frames--; continue;
141 case '_':
142 case 'b':
143 case 'B':
144     mtx_lock(&L->mtx);
145     // VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
146     life_birth9(L);
147     // ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
148     mtx_unlock(&L->mtx);
149     break;
150 case 'q':
151 case 'Q':
152 case EOF:      goto FINISH;
153 }
154 cnd_signal(&L->draw);
155 } while (!(L->finished || feof(stdin)));
156 FINISH:
157     L->finished = true;
158     return 0;
159 }
```

This routine is mainly composed of the input loop which in turn contains a big switch to dispatch on different characters that the user typed into the keyboard. Only two of the **case** s need this kind of protection, namely **'b'** and **'B'** that trigger the forced “birth” of a  $3 \times 3$  cluster of cells around the current cursor position. In all other cases we only interact with atomic objects and so we can safely modify these.

The effect of locking and unlocking the mutex is simple. The call to `mtx_lock` blocks execution of the calling thread, until it can be guaranteed that no other thread is inside a critical section that is protected by the same mutex. We say that `mtx_lock` *acquires* the lock on the mutex, by that *holds* it and that then `mtx_unlock` *releases* it. C's mutex lock interfaces are defined as follows:

```
int mtx_lock(mtx_t*);
int mtx_unlock(mtx_t*);
int mtx_trylock(mtx_t*);
int mtx_timedlock(mtx_t*restrict, const struct timespec*restrict);
```

The two other calls enable us to test (`mtx_trylock`) if another thread already holds a lock (and thus we may avoid waiting) or to wait (`mtx_timedlock`) for a maximal period (and thus we may avoid blocking forever). The latter is only allowed if the mutex had been initialized as of being of the `mtx_timed` “type”, see below.

There are two other calls for dynamic initialization and destruction:

```
int mtx_init(mtx_t*, int);  
void mtx_destroy(mtx_t*);
```

Other than for more sophisticated thread interfaces, the use of `mtx_init` is mandatory. There is no static initialization defined for `mtx_t`.

**Rule 3.19.3.1** *Every mutex must be initialized with `mtx_init`.*

The second parameter of `mtx_init` specifies the “type” of the mutex. It must be one of four different values, `mtx_plain`, `mtx_timed`, `mtx_plain|mtx_recursive`, or `mtx_timed|mtx_recursive`.

As you probably have guessed, using `mtx_plain` versus `mtx_timed`, controls the possibility to use `mtx_timedlock`. The additional property `mtx_recursive` enables us to call `mtx_lock` and similar functions successively several times for the same thread, without unlocking it before.

**Rule 3.19.3.2** *A thread that holds a non-recursive mutex must not call any of the mutex lock function for it.*

The name `mtx_recursive` indicates that it is mostly used for recursive functions that call `mtx_lock` on entry of a critical section and `mtx_unlock` on exit.

**Rule 3.19.3.3** *A recursive mutex is only released after the holding thread issues as many calls to `mtx_unlock` as it has acquired locks.*

**Rule 3.19.3.4** *A locked mutex must be released before the termination of the thread.*

**Rule 3.19.3.5** *A thread must only call `mtx_unlock` on a mutex that it holds.*

From all of this we can deduce a simple rule of thumb:

**Rule 3.19.3.6** *Each successful mutex lock corresponds to exactly one call to `mtx_unlock`.*

Depending on the platform, a mutex may bind a system resource that is attributed each time `mtx_init` is called. Such a resource can be additional memory (e.g a call to `malloc`) or some special hardware. Therefore it is important to release such resources, once a mutex reaches the end of its lifetime:

**Rule 3.19.3.7** *A mutex must be destroyed at the end of its lifetime.*

So in particular, `mtx_destroy` must be called

- before the scope of a mutex with automatic storage duration ends,
- and before the memory of a dynamically allocated mutex is freed.

**19.4. Communicating through condition variables.** While we have seen that the input didn’t need much protection against races, the opposite holds for the account task. Its whole job is to scan through the entire position matrix and to account for the number of life neighbors every position has.

```

99  static
100  int account_thread(void* Lv) {
101      life*restrict L = Lv;
102      while (!L->finished) {
103          // block until there is work
104          mtx_lock(&L->mtx);
105          while (!L->finished && (L->accounted == L->iteration))
106              life_wait(&L->acco, &L->mtx);

```

B9.c

```

107 // VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
108 life_account(L);
109 if ((L->last + repetition) < L->accounted) {
110     L->finished = true;
111 }
112 // ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
113
114 cnd_signal(&L->upda);
115 mtx_unlock(&L->mtx);
116 }
117 return 0;
118 }

```

Here we see that the critical section covers the whole loop body. In addition to the proper accounting part that we already have seen above, there is first a phase in the critical section where the thread is actually paused until new accounting is necessary. More precisely there is a conditional loop that can only be left once either

- the game is finished, or
- another thread has advanced an iteration count.

The body of that loop is a call to `life_wait`, a function that suspends the calling thread for one second or until a specific event occurs:

```
19 int life_wait(cnd_t* cnd, mtx_t* mtx) {
20     struct timespec now;
21     timespec_get(&now, TIME_UTC);
22     now.tv_sec += 1;
23     return cnd_timedwait(cnd, mtx, &now);
24 }
```

Its main ingredient is a call to `cnd_timedwait` that takes a so-called *condition variable* of type `cnd_t`, a mutex and an absolute time bound.

Such condition variables are used to identify a condition for which a thread might want to wait. Above, in our example, you have seen declarations for three such condition variable members of `life`, namely `draw`, `acco` and `upda`. Each of these corresponds to test conditions that the drawing, the accounting and the update need such that they will proceed to perform their proper tasks. As we have seen, accounting has

```

105     while (!L->finished && (L->accounted == L->iteration))
106         life_wait(&L->acco, &L->mtx);

```

Similarly, update and draw have

```

43     while (!L->finished && (L->accounted < L->iteration))
44         life_wait(&L->upda, &L->mtx);

```

and

```
73     while (!L->finished
74           && (L->iteration <= L->drawn)
75           && (x0 == L->x0)
76           && (x1 == L->x1))
77         life_wait(&L->draw, &L->mtx);
```



The conditions in each of these loops reflects the cases when there is work to do for either of these tasks. Most importantly, we have to be sure not to confound the *condition variable*, which serves as some sort of identification of the condition, and the *condition expression*. A call to a wait function for `cnd_t` may return although nothing concerning the condition expression has changed:

**Rule 3.19.4.1** *On return from a `cnd_t` wait, the expression must be checked, again.*

This may be obvious in our example, since we are using `cnd_timedwait` under the hood, and the return might just be because the call timed out. But even if we use the untimed interface for condition wait, the call might return early. In our example code, the call might eventually return when the game is over, therefore our condition expression always contains a test for `L->finished`.

`cnd_t` comes with four principal control interfaces:

```
int cnd_wait(cnd_t*, mtx_t*);
int cnd_timedwait(cnd_t*restrict, mtx_t*restrict, const struct timespec *
    restrict);
int cnd_signal(cnd_t*);
int cnd_broadcast(cnd_t*);
```

The first works analogously to the second, only that there is no time out and a thread might never come back from the call, if the `cnd_t` parameter is never signaled.

`cnd_signal` and `cnd_broadcast` are on the other end of the control. We have already seen the first applied in `input_thread` and `account_thread`. They ensure that some thread (`cnd_signal`) or all threads (`cnd_broadcast`) that are waiting for the corresponding condition variable are woken up and return from the call to `cnd_wait` or `cnd_timedwait`. E.g, the input task *signals* the drawing task that something in the game constellation has changed, and that the board should be redrawn:

154

```
cnd_signal(&L->draw);
```

B9.c

The `mtx_t` parameter to the condition wait functions has an important role. The mutex must be held by the calling thread to the wait function. It is temporarily released during the wait, such that other threads can do their job to assert the condition expression. The lock is re-acquired just before returning from the wait call such that then the critical data can safely be accessed without races. Figure 6 shows a typical interaction between two threads, a mutex and a condition variable.

It shows that six function calls are involved in the interaction, four for the respective critical sections and the mutex, two for the condition variable.

The coupling between a condition variable and the mutex in the wait call should be handled with care.

**Rule 3.19.4.2** *A condition variable can only be used simultaneously with one mutex.*

But it is probably best practice to never change the mutex that is used with a condition variable at all.

Our example also shows that there can be many condition variables for the same mutex: we use our mutex with three different condition variables at the same time. This will be imperative in many applications, since the condition expressions under which threads will be accessing the same resource depends on their respective role.

In situations where several threads are waiting for the same condition variable and are woken up with a call to `cnd_broadcast`, they will not wake up all at once, but one after another as they re-acquire the mutex.

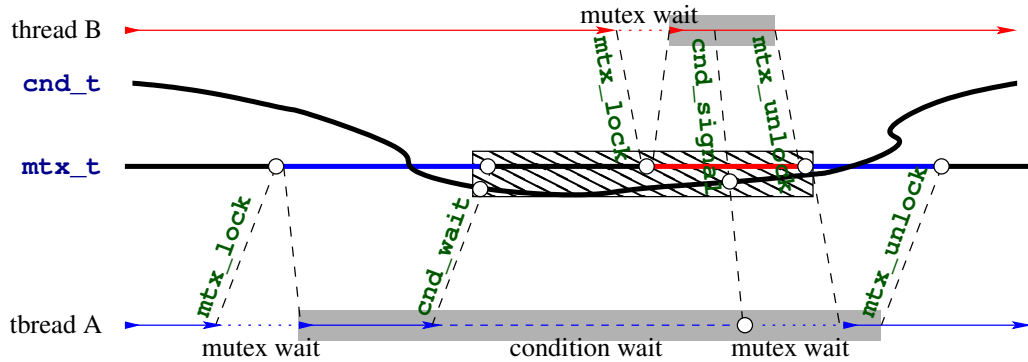


FIGURE 6. Control flow managed by `mtx_t` and `cnd_t`, critical sections underpinned in grey. The condition is associated to the mutex until the last waiter has re-acquired the mutex.

Similar to mutex', C11's condition variables may bind precious system resources. So they must be initialized dynamically, and they should be destroyed at the end of their lifetime.

**Rule 3.19.4.3** A `cnd_t` must be initialized dynamically.

**Rule 3.19.4.4** A `cnd_t` must be destroyed at the end of its lifetime.

The interfaces for these are straight forward:

```
int cnd_init(cnd_t *cond);
void cnd_destroy(cnd_t *cond);
```

**19.5. More sophisticated thread management.** Having just seen the thread creation and joining in our `main` above we may have the impression that threads are somehow hierarchically organized. But actually they are not, just knowing the id of a thread, its `thrd_t`, is sufficient to deal with it. There is only one thread with exactly one special property:

**Rule 3.19.5.1** Returning from `main` or calling `exit` terminates all threads.

If we want to terminate `main` after we have created some other threads, we have to take some precautions such that we do not terminate the other threads preliminarily. An example for such a strategy is given in the following modified version of B9's `main`:

```

210 life L = LIFE_INITIALIZER;
211
212 void B9_atexit(void) {
213     /* Put the board in a nice final picture. */
214     L.iteration = L.last;
215     life_draw(&L);
216     life_destroy(&L);
217 }
218
219 int main(int argc, char* argv[argc+1]) {
220     /* Use cmdline arguments for the size of the board. */
221     size_t n0 = 30;

```

B9-detach.c

```

222     size_t n1 = 80;
223     if (argc > 1) n0 = strtoull(argv[1], 0, 0);
224     if (argc > 2) n1 = strtoull(argv[2], 0, 0);
225     /* Create an object that holds the game's data. */
226     life_init(&L, n0, n1, M);
227     atexit(B9_atexit);
228     /* Create four threads that operate all on that same object and
229        discard their ids. */
230     thrd_create(&(thrd_t){0}, update_thread, &L);
231     thrd_create(&(thrd_t){0}, draw_thread, &L);
232     thrd_create(&(thrd_t){0}, input_thread, &L);
233     /* End this thread nicely and let the threads go on nicely. */
234     thrd_exit(0);
235 }

```

First, we have to use the function `thrd_exit` to terminate `main`. Other than a `return`, this ensures that the corresponding thread just terminates without impacting the other threads. Then, we have to make `L` a global variable, because we don't want its life to end, when `main` terminates. To arrange for the necessary cleanup we also install an `atexit` handler. The modified control flow is shown in Figure 7.

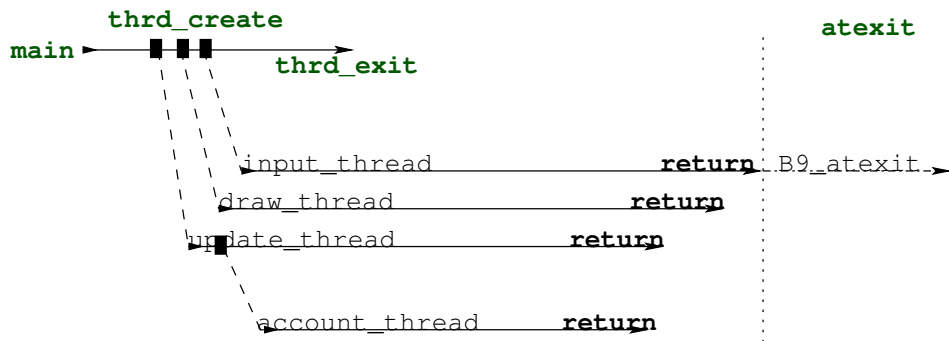


FIGURE 7. Control flow of the five threads of B9-detach. The thread that returns last executes the `atexit` handlers.

As a consequence of this different management, the four threads that are created are actually never joined. Each thread that is dead but is never joined eats up some resources that are kept until the end of the execution. Therefore it is good coding style to tell the system that a thread will never be joined, we say that we *detach* the corresponding thread. We do that by inserting a call to `thrd_detach` at the beginning of the thread functions. We also start the account thread from there, and not from `main` as we did previously.

```

38 int update_thread(void* Lv) {
39     /* Nobody should ever wait for this thread. */
40     thrd_detach(thrd_current());
41     /* Delegate part of our job to an auxiliary thread. */
42     thrd_create(&(thrd_t){0}, account_thread, Lv);

```

B9-detach.c

There are 6 more functions that can be used to manage threads, of which we already met `thrd_current`, `thrd_exit` and `thrd_detach`:

```

thrd_t thrd_current(void);
int thrd_equal(thrd_t, thrd_t);
_Noreturn void thrd_exit(int);

```

```
int thrd_detach(thrd_t);
int thrd_sleep(const struct timespec*, struct timespec*);
void thrd_yield(void);
```

A running C program may have much more threads than it has processing elements to its disposal. Nevertheless, a run time system should be able to *schedule* the threads smoothly by attributing time slices on some processor. If a thread actually has no work to do, it should not demand a time slice and leave the processing resources to other threads that might need it. This is one of the main features of the control data structures `mtx_t` and `cnd_t`:

**Rule 3.19.5.2** While blocking on `mtx_t` or `cnd_t` a thread frees processing resources.

If this is not enough, there are two other functions that can suspend execution:

- `thrd_sleep` allows a thread to suspend its execution for a certain time, such that hardware resources of the platform can be used by other threads in the mean time.
- `thrd_yield` just terminates the current time slice and waits for the next processing opportunity.

## 20. Atomic access and memory consistency

We will complete this level with a description of concepts that you will much likely not use even once in your programming life, at least not explicitly. Nevertheless, this is an important part of the C architecture model and therefore a must for experienced programmers. So try to apprehend this last section to increase your understanding of how things work, not to improve your operational skills. And, BTW, even though we will not even go into all the glorious details<sup>31</sup>, things can get a bit bumpy, please remain seated and buckle up.

If you review the pictures of control flow that we have seen throughout the last sections, you see that the interaction of different parts of a program execution already can get quite complicated. We have different levels of concurrent access to data:

- Plain old straightforward C code is only apparently sequential. Visibility of changes is only guaranteed between very specific points of the execution, sequence points, for direct data dependencies, and for the completion of function calls. Modern platforms take more and more advantage of the provided slack and perform unsequenced operations intermixed or in parallel in multiple execution pipelines.
- Long jumps and signal handlers are executed sequentially, but effects of stores may get lost on the way.
- Those accesses to atomic objects that we have seen so far warrant visibility of their changes everywhere and consistently.
- Threads run concurrently side by side and jeopardize data consistency if they don't regulate their shared access to data. Besides access to atomic objects, they can also be synchronized through calls to functions, such as `thrd_join` or `mtx_lock`.

But access to memory is not the only thing a program does. In fact, the abstract state of a program execution consists of the *points of execution* (one per thread), *intermediate values* (of computed expressions or evaluated objects), *stored values* or *hidden state*. Changes

<sup>31</sup>We will put aside `memory_order_consume` consistency and thus the dependency-ordered before relation.

to this state are usually described as *jumps* (points of execution), *value computation* (intermediate values) and *side effects* (stores or IO), or they can affect hidden state such as the lock state of a `mtx_t`, the initialization state of a `once_flag` or set or clear operations on an `atomic_flag`. We summarize all these possible changes of the abstract state with the term of *effect*.

**Rule 3.20.0.1** *Every evaluation has an effect.*

This is because any evaluation has the concept of a next such evaluation that will be performed after it. Even an expression like

```
(void) 0;
```

that drops the intermediate value sets the point of execution to the next statement, and thus the abstract state has changed.

In a complicated context it will be difficult to argue about the actual abstract state in a given moment in time. Generally, the whole abstract state of a program execution is not even observable and, in many cases, the concept of an overall abstract state is not even well defined. This is because we actually don't know what "time" means in this context. In an multi-threaded execution that is performed on several physical compute cores, there is no real notion of a reference time between them. So generally, C does not even make the assumption that an overall fine-grained notion of time exists between different threads.

**20.1. The “happend before” relation.** All we can hope for is to have enough *partial knowledge* of the state of all threads of a program to argue about it (its correctness, its performance etc). We will investigate a relation that was introduced by Lamport [1978]. In notations of the C standard, it is the *happened before* relation between two evaluations  $E$  and  $F$ , denoted by  $F \rightarrow E$ . This is a property between events that we observe *a posteriori*. Fully spelled out it would perhaps better be called *knowingly happend before* relation, instead.

One part of it are evaluations in the same thread that are related by the already introduced sequenced before relation:

**Rule 3.20.1.1** *If  $F$  is sequenced before  $E$ , then  $F \rightarrow E$ .*

Between threads, the ordering of events is provided by *synchronization*. There are two types of synchronizations, the first is implied by operations on atomics, the second by certain C library calls. Let us first look into the case of atomics.

Operations on atomics are guaranteed to be locally consistent, that is:

**Rule 3.20.1.2** *The set of modifications of an atomic object  $A$  are performed in an order that is consistent with the sequenced before relation of any threads that deals with  $A$ .*

This sequence is called the *modification order* of  $A$ . E.g for an atomic counter  $c$  that we would only increment, the modification order is characterized by the increasing values that are returned by the sequence of  $c++$  operations.

Each synchronization between threads has a “writer” and a “reader” side. We attach two abstract properties to certain operations on atomics and to certain C library calls that are called *release* semantics (on the writer side), *acquire* semantics (for a reader) or *acquire-release* semantics (for a reader-writer). C library calls with such synchronization properties will be discussed a bit later.

All operations on atomics that we have seen so far and that modify the object are required to have release semantics and all that read have acquire semantics. Only below we will see other atomic operations that have relaxed properties.

**Rule 3.20.1.3** *An acquire operation  $E$  in a thread  $T_E$  synchronizes with a release operation  $F$  in another thread  $T_F$  if  $E$  reads the value that  $F$  has written.*

The idea of the special construction with acquire and release semantic is to force the visibility of effects across such operations. We say that an effect  $X$  is *visible* at evaluation  $E$  if we can consistently replace  $E$  by any appropriate read operation or function call that uses the state affected by  $X$ .

**Rule 3.20.1.4** *If  $F$  synchronizes with  $E$ , all effects  $X$  that have happened before  $F$  must be visible at all evaluations  $G$  that happen after  $E$ .*

Note that several atomic operations can read *and* write atomically in one step. These are called *read-modify-write* operations: compound assignments, increment and decrement, and calls to **atomic\_compare\_exchange\_weak** and **atomic\_flag\_test\_and\_set** functions. Such an operation can synchronize on the read side with one thread and on the write side with others. All such read-modify-write operations that we have seen so far have both, acquire and release semantics.

The happen before relation closes the combination of the relations “sequenced before” and “synchronizes with” transitively. We say that  $F$  knowingly happened before  $E$ , if there are  $n$  and  $E_0 = F, E_1, \dots, E_{n-1}, E_n = E$  such that  $E_i$  is sequenced before  $E_{i+1}$  or synchronizes with it, for all  $i < n$ . Stated otherwise,

**Rule 3.20.1.5** *We only can conclude that one evaluation happened before another if we have a sequenced chain of synchronizations that links them.*

Observe that this happened before relation is a combination of very different concepts. The sequence before relation can in many places be deduced from syntax, in particular if two statements are members of the same basic block. Synchronization is different: besides the two exceptions of thread startup and end, it is deduced through a data dependency on a specific object, such as an atomic or a mutex.

The desired result of all of this on the visibility of effects is:

**Rule 3.20.1.6** *If an evaluation  $F$  happened before  $E$ , all effects that are known to have happened before  $F$  are also known to have happened before  $E$ .*

**20.2. Synchronizing C library calls.** C library functions with synchronizing properties come in pairs, a releasing side and an acquiring side. They are summarized in Table 2. Note that for the first two, we know which events synchronize with which. The call (or

TABLE 2. C library functions that form synchronization pairs

release	acquire	
<b>thrd_create</b> ( <i>..</i> , <i>f</i> , <i>x</i> )	entry to <i>f</i> ( <i>x</i> )	
end of thread <i>id</i>	<b>thrd_join</b> ( <i>id</i> )	any <b>return</b> from <i>f</i> or <b>thrd_exit</b>
<b>call_once</b> (& <i>obj</i> , <i>g</i> )	<b>call_once</b> (& <i>obj</i> , <i>h</i> )	first call with all subsequent calls
mutex release	mutex acquisition	

equivalent) to *f*(*x*) synchronizes with exactly the the call to **thrd\_create** that created its thread. The call to **thrd\_join** synchronizes with exactly the thread with the corresponding *id*.

The other cases are a bit more complicated. For the initialization utility **call\_once**, the return from the very first call **call\_once**(&*obj*, *g*), the one that succeeds to call its function *g*, is a release operation for all subsequent calls with the same object *obj*.

This ensures that all write operations that are performed during the call to `g()` are known to happen before any other call with `obj`. Thereby all other such calls also know that the write operations (the initializations) have been performed.

For a mutex, a release operation can be a call to a mutex function, `mtx_unlock`, or the entry into the wait functions for condition variables, `cnd_wait` or `cnd_timedwait`. An acquire operation on a mutex is the successful acquisition of the mutex via any of the three mutex calls `mtx_lock`, `mtx_trylock` or `mtx_timedlock`, or the return from the the wait functions, `cnd_wait` or `cnd_timedwait`.

**Rule 3.20.2.1** *Critical sections that are protected by the same mutex occur sequentially.*

**Rule 3.20.2.2** *In a critical section that is protected by mutex `mut` all effects of previous critical sections protected by `mut` are visible.*

One of these known effects is always the advancement of the point of execution. In particular, on return from `mtx_unlock` the execution point is outside the critical section, and this effect is known to the next thread that newly acquires the lock.

The wait functions for condition variables differ from acquire-release semantics:

**Rule 3.20.2.3** *`cnd_wait` or `cnd_timedwait` have release-acquire semantics.*

That is before suspending the calling thread they perform a release operation and then, when returning, an acquire operation. The other particularity is that we need a call to `mtx_unlock` to ensure synchronization. The signaling thread will not necessarily synchronize with the waiting thread if it does not place a call to `cnd_signal` or `cnd_broadcast` into a critical section that is protected by the same mutex as the waiter. In particular, non-atomic modifications of objects that constitute the *condition expression* may not become visible to a thread that is woken up by a signal if the modification is not protected by the mutex. There is a simple rule of thumb to ensure synchronization.

**Rule 3.20.2.4** *Calls to `cnd_signal` or `cnd_broadcast` should always occur inside a critical section that is protected by the same mutex as the waiters.*

**20.3. Sequential consistency.** The data consistency for atomic objects that we described above, the one guaranteed by the *happened before* relation, is called *acquire-release consistency*. Whereas the C library calls that we have seen above always synchronize with that kind of consistency, not more and not less, accesses to atomics can be specified with different consistency models.

As you remember, all atomic objects have a *modification order* that is consistent with all sequenced before relations that see these modifications *on the same object*. *Sequential consistency* has even more requirements than that:

**Rule 3.20.3.1** *All atomic operations with sequential consistency occur in one global modification order, regardless of the atomic object they are applied to.*

So, sequential consistency is a very strong requirement. Not only that it enforces acquire-release semantics, that is a causal partial ordering between events, but it rolls out this partial ordering to a total ordering. If you are interested in parallelizing the execution of your program, sequential consistency may not be the right choice, because it forces sequential execution of the atomic accesses.

The standard provides the following functional interfaces for atomic types, that should conform to the description given by their name.



```

void atomic_store(A volatile* obj, C des);
C atomic_load(A volatile* obj);
C atomic_exchange(A volatile* obj, C des);
bool atomic_compare_exchange_strong(A volatile* obj, C *expe, C des);
bool atomic_compare_exchange_weak(A volatile* obj, C *expe, C des);
C atomic_fetch_add(A volatile* obj, M operand);
C atomic_fetch_sub(A volatile* obj, M operand);
C atomic_fetch_and(A volatile* obj, M operand);
C atomic_fetch_or(A volatile* obj, M operand);
C atomic_fetch_xor(A volatile* obj, M operand);
bool atomic_flag_test_and_set(atomic_flag volatile* obj);
void atomic_flag_clear(atomic_flag volatile* obj);

```

Here C is any appropriate data type, A is the corresponding atomic type and M is a type that is compatible with the arithmetic of C. As the name suggest, for the fetch and operator interfaces the call returns the value that \*obj had before the modification of the object. So these interfaces are *not* equivalent to the corresponding compound assignment operator, since that would return the result *after* the modification.

All these functional interfaces provide *sequential consistency*:

**Rule 3.20.3.2** *All operators and functional interfaces on atomics that don't specify otherwise have sequential consistency.*

Observe also that the functional interfaces differ from the operator forms, because they force their arguments to be **volatile** qualified.

**20.4. Other consistency models.** A different consistency model can be requested with a complementary set of functional interfaces. *E.g* an equivalent to the postfix ++ operator with just acquire-release consistency could be specified with

```

unsigned _Atomic at = ATOMIC_VAR_INIT(67);
...
if (atomic_fetch_add_explicit(&at, 1, memory_order_acq_rel)) {
    ...
}

```

**Rule 3.20.4.1** *Every functional interface for atomic objects has a form with **\_explicit** appended which allows to specify its consistency model.*

These interfaces accept additional arguments in the form of symbolic constants of type **memory\_order** that specify the memory semantics of the operation.

- **memory\_order\_seq\_cst** requests sequential consistency. Using this is equivalent to the forms without **\_explicit**.
- **memory\_order\_acq\_rel** is for an operation that has acquire-release consistency. Typically for general atomic types you'd use that for a read-modify-write operation such as **atomic\_fetch\_add** or **atomic\_compare\_exchange\_weak**, or for **atomic\_flag** with **atomic\_flag\_test\_and\_set**.
- **memory\_order\_release** is for an operation that has only release semantic. Typically this would be **atomic\_store** or **atomic\_flag\_clear**.
- **memory\_order\_acquire** is for an operation that has only acquire semantic. Typically this would be **atomic\_load**.
- **memory\_order\_consume** is for an operation that has a weaker form of causal dependency than acquire consistency. Typically this would also be **atomic\_load**.
- **memory\_order\_relaxed** is for an operation that adds no synchronization requirements to the operation. The only guarantee for such an operation is that it is indivisible. A typical use case for such an operation is a performance counter



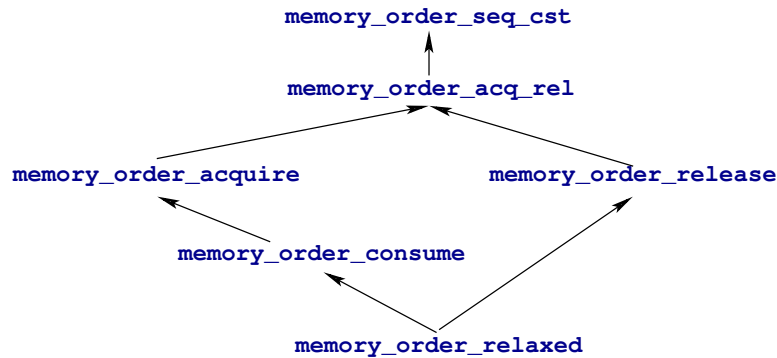


FIGURE 8. Stringency of consistency models

that is used by different threads, but for which we only are interested in a final accumulated count.

The consistency models can be compared with respect to the restrictions they impose to the platform. Fig. 8 show the implication order of the `memory_order` models.

Whereas `memory_order_seq_cst` and `memory_order_relaxed` are admissible for all operations, there are some restrictions for other `memory_order`. Operations that can only occur on one side of a synchronization can only specify an order for that side. Thereby the two operations that only store (`atomic_store` or `atomic_flag_clear`) may not specify acquire semantics. Three operations only perform a load and may not specify release or consume semantics: besides `atomic_load` these are `atomic_compare_exchange_weak` and `atomic_compare_exchange_strong` in case of failure. Thus, the latter two need two `memory_order` arguments for their `_explicit` form, such that they can distinguish the requirements for the success and failure cases:

```

bool
atomic_compare_exchange_strong_explicit(A volatile* obj, C *expe, C des,
                                        memory_order success,
                                        memory_order failure);

bool
atomic_compare_exchange_weak_explicit(A volatile* obj, C *expe, C des,
                                      memory_order success,
                                      memory_order failure);

```

Here, the `success` consistency must be at least as strong as the `failure` consistency, see Fig. 8.

Up to now, we have implicitly assumed that acquire and release sides of a synchronization are symmetric, but they aren't: whereas there always is just one writer of a modification there can be several readers. As moving new data to several processors or cores is expensive, some platforms allow to avoid the propagation of all visible effects that happened before an atomic operation to all threads that read the new value. C's *consume consistency* is designed to map this behavior. We will not go into the details of this model, and you should only use it when you are certain that some effects prior to an atomic read will not affect the reading thread.



## LEVEL 4



## Ambition

*Remark: The presentation of this level is probably not yet in its final form. Please bare with me and don't hesitate to make suggestions.*

I have already noted here and there that C as a standardized language has evolved since its beginnings. Backwards compatibility is a major issue for the C community, so this evolution is slow compared to other programming languages. But nevertheless this evolution is happening, and it recently has brought in important features such as threads, atomics and **\_Generic**.

On this level I will try to give my vision of future developments of C. The idea is that they should be driven by a need, should be easy to implement *and* should only make minimal changes. Certainly these three criteria are subjective, but my hope is that this will somehow lead to some consensus of what is to be included and what would be extensions. The properties of C that I'd like to maintain are:

- C is a statically typed language.<sup>1</sup>
- C is portable to all modern computer architectures.
- A well written program that conforms to one version of the standard, and that doesn't use deprecated features, should remain so for at least one subsequent version of the standard.<sup>2</sup>
- Additions to the standard should be based on well tested extensions.
- Changes that may modify the semantics of conforming programs must be well identified and must have an easy to implement compile time diagnostic. The number of programs that would have modified behavior must be marginal within the existing body of conforming programs.

There are several problem zones that I would like to be tackled. Resolving issues here would make programming in C even easier, without giving up any of the portability or optimization potential.

- Lack of constants for arbitrary types.
- Lack of type generic macro capacities.
- Lack of type generic interfaces for the C library.
- Lack of modularity, reusability and encapsulation.
- Lack of compile time and run time check of function calls.
- An inconsistent and confusing object and memory model.
- An inconsistent **const**-contract.

### Sections of the C standard to be amended

Blocks like this one are used to specify the changes that would have to be made to the C standard. If you are not interested in the very details of that but just in the proposed features you probably can skip these.

<sup>1</sup>VLA are the only permitted extension from that rule.

<sup>2</sup>I.e., it will have a minimal lifetime of about 20 years.

## 21. The **register** overhaul

The **register** storage class is perhaps the least understood, less esteemed and most underestimated tools of the C language. It merits better, since it can be used to force a very economic use of the **&** operator in code that is sensible to optimization. It helps to tackle one of the most important optimization problems that steams from C's "flat" memory model: aliasing. A reformulation of Rule 2.13.2.10 in the language of the C standard would be:

**Rule 4.21.0.1** *Objects that are declared with **register** storage duration can't alias.*

The goals of this proposal are multiple:

- Goal 1** *Use **const** qualified **register** objects as typed compile time constants.*
- Goal 2** *Extend the optimization opportunities of **register** to file scope objects.*
- Goal 3** *Create new optimization opportunities for functions that are local to a TU.*
- Goal 4** *Improve the interplay between objects and functions that are local to a TU.*
- Goal 5** *Impose bounds checking for constant array subscripts.*

To see that the **register** concept can be extended to overcome the lack of general constants in C, let us look at the two following definitions:

```
1  enum { fortytwo = 42, };
2  register int const fortytwo = 42;
```

From the point of view of the C standard, they specify very different things, an enumeration constant and an object. But as long as they are placed inside a function, the two entities are not much distinguishable by code that uses them.

- Both are of type **int**.
- Both have value 42.
- Both may not appear on the left hand side of an assignment.
- Both may not be subject of the address-of operator **&**.
- They have exactly the same spelling.

In C11, there are the following differences to these definitions:

- Definitions with **register** storage are not allowed in file scope.
- A **register** object with **const** qualification is not an *integer constant expression*, ICE, even if the initializer is an ICE.

Other drawbacks for **register** declarations in C11 include:

- Because of implicit address-of operations when accessing them, **register** is not useful for array declarations.
- Because of implicit address-of operations when calling functions, **register**, even if allowed in file scope, would not be useful for function declarations.

In view of that, I will try bring the concepts closer together, for which the separation is quite unproductive and artificial: literals and **const**-qualified **register** objects. Both are just constants in the sense of usual CS terminology.

*A constant is an immutable and non-addressable entity in the program state that is determined during compilation.*

Because it is determined at compile time and immutable, no aspect of such a constant can change during a specific execution of the program. Because it is non-addressable no property of a constant can change between different executions of the same object file. Thus constants will never influence the branching of the program.

Therefore, constants can be present in a object file in very different forms.

- They may be optimized completely and only be implicitly present in the structure of the control flow graph.
- They can be present explicitly as one or several hardware registers, assembler immediates, or special instructions.
- They may be realized in addressable read-only memory.

Which form they take, depends on many factors and a compiler is completely free to chose. Examples:

- A 0 may be realized by an instruction that zeros out a register, *e.g* exclusive-or of the register with itself.
- Depending of its usage, a small integer constant can often be realized as instruction immediate, or fixed address offset.
- A **double** constant may be loaded directly from memory into a special floating point hardware register.

**Overview.** We develop our proposal in several steps, that become more and more intrusive as we go along. First in Section 21.1 we introduce the main feature, file scope **register**, which shows to be simple and straightforward. Section 21.2 then proposes to subsume the main advantage into a new concept, *named constants*, and Section 21.3 integrates these into C's *integer constant expressions*. Sections 21.4 and 21.5 then deal with two important features that are more difficult to integrate into C11, **register** functions and **register** arrays.

**21.1. Introduce **register** storage class in file scope.** Unfortunately, with the current definition the use of **register** is limited to block scope and function prototype scope. There is no technical reason that it couldn't be used in file scope: file scope **register** declarations are already allowed from a viewpoint of syntax, they are explicitly forbidden by a constraint.

21.1.1. *Changes for object types.* Any compiler should be able to implement file scope *objects* with **register** storage class easily. At the minimum they can just be realized similar to file-scope-**static** with two additional features:

- Produce a diagnostic if the address of such an object is taken.
- If such a file scope **register** object is **const** qualified but not **volatile**, allow its use inside any **inline** or **register** function.

Textual changes for this feature that concern objects are minimal. The only problem spots are **const** qualified register objects. Since these are detectable at compile time, we can impose their explicit initialization. Thereby we avoid a clash with the concept of tentative definitions, which should just be forbidden for such **const** qualified objects.

*Existing practice:* In connection with its `__asm__` hardware register extension, the GNU compiler already allows **register** in file scope.

---

### Sections of the C standard to be amended

- 6.7.1 p6
  - Add a mention of aliasing to the optimization potential.
  - Add: *If such an object is of **const** qualified type, it shall be initialized explicitly.*
- 6.7.1 p6 footnote. Transform this footnote into a note.
  - Add after “**auto** declaration”: “(function scope) or **static** declaration (file scope)”
  - Add: *A **const** qualified identifier with **register** storage class can not appear in a tentative definition.*
- 6.9 p2 remove the mention of **register**
- 6.9.1 p4 add **register** to the list
- 6.9.2 p2 add **register** to the cases

### 21.2. Typed constants with **register** storage class and **const** qualification.

Besides helping with aliasing, allowing **register** in file scope has another important consequence, C gains real named constants for all data types. Data types that we declare in C often have some common *values* that should be shared by all users of that type, but unfortunately the C language only provides the possibility to define *objects* that are initialized with the value in question for most of the cases.

Most prominent in C is the all zero initialized value that we always can enforce through a default initializer of a **static** object, others need detailed initializers:

```

1 typedef struct listElem listElem;
2 struct listElem { unsigned val; listElem* next; };
3 static listElem const singleton = { 0 };
4 static listElem const singleOne = { .val = 1 };

```

There are two ways to deal with such **const** qualified objects in file scope. If they are implemented as above with **static** storage class, they may result in one object per compilation unit and the compiler might throw spooky warnings on us if we define them in a header file but don't use them at all.

**Rule 4.21.2.1** *File scope **static const** objects may be replicated in all compilation units that use them.*

**Rule 4.21.2.2** *File scope **static const** objects cannot be used inside **inline** functions with external linkage.*

Another way is to declare them **extern** in a header file

```

1 extern listElem const singleton;

```

and to define them in one of the compilation units:

```

1 listElem const singleton = { 0 };

```

This second method has the big disadvantage that the value of the object is not available in the other units that include the declaration. Therefore we may miss some opportunities for the compiler to optimize our code. For example an initialization function for our structure type may be written as

```

1 inline
2 listElem* listElem_init(listElem* el) {
3     if (el) *el = singleton;
4     return el;
5 }

```

If the value of `singleton` is known, the assignment could be realized without loading it but only with immediats. And `listElem_init` itself could then much easier be inlined where it is called.

**Rule 4.21.2.3** *File scope **extern const** objects may miss optimization opportunities for constant folding and instruction immediates.*

The other disadvantage of both cases (**static** or **extern**) is that the address of the object can be taken inadvertently. So aliasing of such “constants” may in some cases not be excluded by the compiler:

**Rule 4.21.2.4** *File scope **extern** or **static const** objects may miss optimization opportunities because of mispredicted aliasing.*

So being **const** qualified, doesn’t suffice to make an object suitable as constants in the same way literals for integer types are. Ideally, we would like to have a way to declare rvalues of any given type. C currently even has no direct support for all of the standard arithmetic types: all integer literals are at least as large as **int**.

For scalar types a cast can be used to produce a value expression:

```

1 #define myVeritableFalse ((_Bool)+0)
2 #define myVeritableTrue  ((_Bool)+1)
3 #define HELLO ((char const*const) "hello")

```

Or some predefined macro for the special case of complex types:

```

1 #define ORIGIN CMPLXF(0, 0)

```

The only named constants that can serve as genuine integral constant expressions (ICE) as C understands them are of type **int** and are declared through a declaration of an enumeration:

```

1 enum { uchar_upper = ((unsigned) UCHAR_MAX) + 1, };

```

All this, to just define a constant; we thereby define a new type (the unnamed enumeration type) and define a constant that isn’t even of that type, but an **int**.

For composite types, in particular structure types, things are more complicated, because casts are only allowed for arithmetic types. To fabric a value expression that is not an assignable lvalue and such that the address can’t be taken we have to find another way that produces an rvalue for a **struct** type.

Therefore we first define an object for the desired type and then feed it into an expression such that the result is an rvalue. There are only two operators that can return a composite type as a whole, assignment and the ternary operator. We use the ternary operator since it allows to have a type agnostic macro for all types but for arrays and narrow integer types:

```

1 #define RVALUE(X) (1 ? (X) : (X))
2 #define SINGLETON RVALUE((listElem const){ .val = 0, })

```

Such expression are quite ugly and can be hard to read for a human and a debugger. The biggest disadvantage is that such an expression is not suitable for initializers of objects with static storage duration. We'd have to define two different macros, one for the initializer and one for the constant expression:

```
1 #define SINGLETON_INIT      { .val = 0, }
2 #define SINGLETON_CONST_LVALUE (listElem const) SINGLETON_INIT
3 #define SINGLETON           RVALUE(SINGLETON_CONST_LVALUE)
```

In function scope, however, there is a construct that can be used to declare unmutable values of which no address can be taken: **const**-qualified objects with **register** storage class. All the above could be given in block scope with functionally equivalent definitions:

```
1 register listElem const      singleton = { 0 };
2 register listElem const      singleOne = { .val = 1 };
3 register _Bool const         myVeritableFalse = 0;
4 register _Bool const         myVeritableTrue = 1;
5 register char const*const     HELLO = "hello";
6 register float _Complex const ORIGIN = CMPLXF(0, 0);
7 register int const           uchar_upper = (unsigned) UCHAR_MAX +
    1;
8 register listElem const      SINGLETON = singleton;
```

The idea of this proposal is that named constants for all types become available in file scope, because they can then be declared in header files as **const** qualified **register** objects.

No additional changes to the ones in the previous section are *required*. It might be convenient, though, to add some text to stress the fact that also a hidden modification of such objects is not standard conforming. This may be helpful to avoid that implementations claim the right to map and modify such objects, hiding behind the fact that the standard leaves behavior undefined when a **const** qualified object is modified.

#### Sections of the C standard to be amended

- 6.6 “constant expressions”, p7 general constant expressions, add an item to the end of the list
  - an lvalue of **const** but not **volatile** qualified type that has been declared with **register** storage class.



- Baptize the thing. Therefore add a new paragraph:

A *register constant* is an object that is of **const** but not **volatile** qualified type, that is declared with the **register** storage class, for which the unique declaration is the definition, that is explicitly initialized and for which the initializer only contains constant expressions.(FOOTNOTE1) Such a register constant provides the same value as specified by the initialization throughout all its lifetime.(FOOTNOTE2)

FOOTNOTE1: Register constants can not appear in tentative definitions.

FOOTNOTE2: But for their spelling, for their type qualification, [*for their qualification as an ICE,*] and for their usability in preprocessor expressions, register constants are indistinguishable from literals of the same type and value. Therefore, register constants can be used as named constants of their type.

- **p8**, arithmetic constant expressions: Add *register constants of arithmetic type* to the list.
- **6.7.1** “storage class specifiers” **p6**: new footnote:  
An implementation should only map a **register** declared object to a hardware register or similar device which is subject to changes not effected by the program if the type of the object is **volatile** qualified and therefore is not a register constant.
- **6.7.4** “function specifiers”, **p3**. Allow the use of register constants in all **inline** functions. Therefore at the end add:  
... shall not contain a reference to an identifier with internal linkage unless it is the name of a register constant.

**21.3. Extend ICE to register constants.** Now that we have global constants for all types, we need to integrate this into the rest of the language. We want register constants to behave the same as literals of the underlying type. This is particularly important for constants of integer type, since they are needed to declare array dimensions, alignments, width of bit-fields or values of enumeration constants, and we want all of this to go smoothly.

To make this possible, we have to amend the computation of compile time integer constant, or as the C standard calls them *integer constant expressions*, ICE. In most contexts, attaching the ICE property to a C11 expression will not change the semantics if this new **register** feature is applied.

There are only two such contexts in which **register** objects were previously allowed, and where the meaning changes if the ICE property is added:

- An array declaration declares a VLA if the expression that is used for the size is not an ICE. Thus some VLA that are declared in C11 code would turn into FLA. Such a change would not change the semantics of the program in question.
- Integer expressions of value 0 are only null pointer constants if they are also ICE.

There is no problem by using pointer expressions directly.

```
register int const zero = 0;
```

```
double* p = zero;           // constraint violation in C11
```

In C11, the initialization of `p` is a constraint violation because implicit conversions from integer to pointer expressions are only allowed if the integer expression is an ICE of value 0. So this property for pointers alone would just have code valid that had been invalid C11 code, before.

Only a combination of some rarely used features may effectively result in a change of semantics of a valid C11 program. In particular an expression such as

```
1 (void*) ((x) - (x))
```

is a null pointer constant, only if `x` is an ICE, and then

```
1 (1 ? (int*)0 : (void*) ((x) - (x)))
```

is of type `int*` or `void*` according to that property of `x`. With `_Generic` such types are observable

```
1 #define IS_ICE(X) \
2   _Generic((1 ? (int*)0 : (void*) ((X) - (X))), \
3           int*: 1, \
4           void*: 0)
```

and code paths can be taken differently according to the status of an identifier:

```
1 if (IS_ICE(x)) {
2     // something
3 } else {
4     // different
5 }
```

**Rule 4.21.3.1** *Changing an identifier that represents an integer constant to be an ICE, can change effective types of objects and execution paths.*

So introducing register constants to be ICE, may change the semantic of some programs. Nevertheless, that risk is low, it is detectable and will in most cases have a positive effect on the compiled program:

- A change could only happen for code that currently uses `const` qualified variables of `register` storage class. The usage of `register` is currently extremely low and almost non-existent, `const` qualification of them is even rarer.
- Compilers that would implement this new feature, or plan to implement this in future versions, can easily add a warning feature that detects the case that the only non-ICE sub-expressions inside an integer-type expression `X` originate from objects that are register constants.
- If a size expression of an array becomes an ICE that wasn't one before, the declared array then is a FLA instead of a VLA. This allows a compiler to produce better optimized code (e.g. the change in the stack pointer is a compile time constant) and avoids to trace the size of the array at runtime (e.g. for `sizeof` expressions). Most modern compilers do such optimization, anyhow.
- If an expression is determined by `_Generic` or other code similar to the above to be an ICE (or not) this should usually be used to produce code that uses the fact that the value of the expression can be determined at run time, and that optimizes the program with this information.

### Sections of the C standard to be amended

The change itself is quite easy to perform, because we already have identifiers in C11, namely enumeration constants, that are ICE.

- In all places that it occurs in C11, change the syntax term *enumeration-constant* to *named-constant*. These are 6.4.4, p1, 6.4.4.3, 6.7.2.2 p1, A.1.5.
- 6.4.4.3, previously “enumeration constants”: Change the whole section to read

```

1  6.4.4.3 Named constants
2
3  Syntax
4      named-constant: identifier
5
6  Semantics
7
8  Named constants are identifiers that are declared as
9      register
10     constant or as enumeration constant. An identifier
11     declared
12     as an enumeration constant has type int.
13
14     Forward references: constant expressions (6.6),
15     enumeration
16     specifiers (6.7.2.2).
```

- 6.6 “constant expressions”, p6 “ICE” and footnote: change *enumeration constant* to *named constant of integer type*.
- p8 “arithmetic constant expression”: change *enumeration constant* to *named constants of arithmetic type*.

**21.4. Functions.** The possibility to declare functions with **register** storage class is an interesting fallout of our approach.

- A function declaration of storage class **register** is equivalent to a declaration as **static inline** with the additional property that its address can’t be taken.

This doesn’t mean that implementors of that feature have to implement a completely new function model. Just as currently for **register** variables, a **register** function *may* well reside in memory and internally the compiler can use its address to make a call. But such a mechanism would be to the discretion of the implementation and completely transparent for the programmer.

21.4.1. *Optimization opportunities.* The advantages of **register** functions are:

- Such a function cannot be called from another TU than the one it is defined in and can effectively be inlined without negative effects to other TU. Thus no “*instantiation*” of the symbol of a **register** function is necessary.
- Since the caller of such a function is known to reside in the same TU, the function setup can avoid the reload of certain registers and is not bound to the platform’s function call ABI.

Similar as for **register** objects, these optimizations are currently possible for **static inline** functions for which the compiler can prove that the address never escapes the current TU. A **register** declaration instead of **static inline** guarantees that these optimization opportunities are not lost accidentally.

*Existing practice:* The GNU compiler already has

```
__attribute__((__visibility__("internal")))
```

that allows for similar optimizations, but which is intrinsically unsafe since gcc does not check if a pointer to such a function can escape the current TU.

21.4.2. *Relaxed constraints for TU local objects.* As already mentioned, **inline** declared functions have difficulties with other symbols that have no or internal linkage. Such functions that are not **static** at the same time

- cannot access **static** file scope variables, or
- cannot declare their own block scope **static** variables,

even if these are **const** qualified and known at compile time.

Our proposal simplifies things with that respect.

- All functions that are **static inline** have access to **register** objects that are visible at their point of definition.
- All functions declared **inline** have access to register constants that are visible at their point of definition.

21.4.3. *Changes for function types.* As the function concept is currently formulated, such **register** function could never be called: the standard function call feature takes the address of a function that is called. We propose to change the standard to allow for that by letting the function call operator `()` directly operate on a function. This changes the semantics just in the context of a call. Function to function-pointer would still be mandated in all other contexts and thus would be illegal for functions that are declared with **register**.

#### Sections of the C standard to be amended

- **6.3.2.1 p4** “conversions of function designators”: Add “, or as the postfix expression of a function call” to the list of allowed operations.
- **6.5.2.2** “function calls”:
  - **p 1:** Replace
    - ... pointer to function returning void or returning a complete object type other than an array type.
 by
    - ... function returning void or returning a complete object type other than an array type, or shall have type pointer to such a function.
  - **p5:** Replace
    - ... type pointer to function returning an object type,
    - ...
 by
    - ... type function returning an object type or pointer to such a function, ...

**21.5. Unify designators.** If we want to extend the use of register constants, we want to ensure that the types that can be used for it are not artificially constrained.

With

```
enum sig_stat { unknown = 0, sync = 1, async = 2, };
```

consider the two following alternative declarations of `sig_status` in a header file:

```
extern enum sig_stat const sig_status[];    // values are hidden
```

and

```
static enum sig_stat const sig_status[] = { // not usable in inline
    [SIGABRT] = async,
    [SIGFPE] = sync,
    [SIGILL] = sync,
    [SIGINT] = async,
    [SIGSEGV] = sync,
    [SIGTERM] = async,
};
```

The first hides the contents from the user code, and the fact if `sig_status[SIGIMPL]` is unknown or not<sup>3</sup> can only be checked at runtime, through a memory reference. The second declaration exposes all values to all users, but cannot be used from an **inline** function.

Using the second variant with **register** instead of **static** would solve the problem: all values are visible for all users and all **inline** functions could use it. But unfortunately, with the current version of the standard, arrays that are declared with a **register** storage class can't serve much purpose: we can't even access individual fields without constraint violation. This is because the `[]` designator is defined by taking addresses: referring to an array element through `A[23]` is equivalent an address-of operation, pointer arithmetic and dereferencing, namely `*(&A[0]+23)`. Thus if `A` is declared as register, this is a constraint violation.

This part of the **register** overhaul tries to make **register** arrays useful whenever the expression in the `[]` is an ICE. It does that by attempting a careful review of the three different syntactical contexts, in which `[]` brackets are used.

Syntactically, C uses `[]` brackets in three different places, but in all that places they serve to specify a length (or the lack of) an array dimension or position:

- array declaration,
- array initialization,
- array subscripting.

The expression that is enclosed inside `[]` can be

- a strictly positive integer constant expression (ICE), or empty `[]` to provide a constant value that is determined at compile time
- any other form of integer expression, or empty `[*]` to provide a value that is determined at run time.

All actual C compilers are able to distinguish these two cases. For designated initializers, only the first form is allowed and using the second is a constraint violation. For an array declaration, in the first case the array is FLA, in the second a VLA<sup>4</sup>.

For subscripting, the C standard currently doesn't make the distinction. This is a bit unfortunate because conceptually subscripting an array at fixed location is technically the same as accessing an element of a structure. In this section we aim to amend the C standard such that an subscript expression that is a positive ICE can be applied to a **register** array.

As a fallout, this reorganization ensures bounds checking for FLA and constant subscripts. If an index into an FLA is constant it can be checked against 0 and the array length

<sup>3</sup>for some implementation defined signal `SIGIMPL`

<sup>4</sup>Which may or many not be supported by the compiler.

at translation time. So if it is negative or too large this constitutes a constraint violation under this proposal.

Let us take a look at the following conditional type declaration:

```

3  #ifdef USE_STRUCT
4      typedef struct two two;
5      struct two { double e1; double e2; };
6  # define FRST(X) ((X).e1)
7  # define SEC(X) ((X).e2)
8  #else
9      typedef double two[2];
10 # define FRST(X) ((X)[0])
11 # define SEC(X) ((X)[1])
12 #endif

```

The two types will behave differently in many context, *e.g.* if we pass them as arguments to functions. But in both cases they are composed of two **double** elements that can be accessed through the macros **FRST** and **SEC**, respectively. With either of the choices I would expect a good optimizing compiler to produce exactly the same binary for code like:

```

14 void func0(void) {
15     for (two sp = { 1, 1, };
16         FRST(sp) + SEC(sp) < 10;
17         FRST(sp) += SEC(sp)) {
18         printf("two_values_%g_%g\n", FRST(sp), SEC(sp));
19         SEC(sp) += (FRST(sp) + SEC(sp))/2;
20     }
21 }

```

And the compilers that I have on my machine achieve that, even without special optimization flags turned on. This is possible, because the code only contains offset calculations for fields that are compile time constants: in one case they are given by the field designators `.e1` and `.e2`, in the other by the array designators `[0]` and `[1]`. Since this code doesn't use the relative position of the fields in memory, the compiler can easily treat the two fields as if they were declared as separate variables.

*Existing practice:* The `clang` compiler silently implement this already, although this is not compliant to C11.

## Sections of the C standard to be amended

### 6.5.2 Postfix operators

#### Syntax

postfix-expression:

primary-expression

postfix-expression array-designator

postfix-expression [ expression ]

postfix-expression ( argument-expression-listopt )

postfix-expression struct-union-designator

postfix-expression -> identifier

postfix-expression ++

postfix-expression --

( type-name ) { initializer-list }

( type-name ) { initializer-list , }

array-designator: [ expression ]

struct-union-designator: . identifier

#### Constraints

The *expression* of an *array-designator* shall be an integer constant expression.

#### 6.5.2.1 Array subscripting

#### Constraints

Array subscripting can occur in two different forms. For the first, the first expressions has type *array object of type T and constant length L*, for some type T non-negative integer constant L and is not a variable length array. The integer constant expression of the *array-designator* shall have a non-negative value that is strictly less than L. The result has type T.

Otherwise, after promotion [the first|one] expression shall have type *pointer to complete object type T*, the other expression shall have integer type, and the result has type T.

#### Semantics

A postfix expression followed by an *array-designator* designates the corresponding element of an array object.FOOTNOTE Successive array-designators designate an element of a multidimensional array object.

FOOTNOTE: Because the subscript expression is a integer constant such a designation is similar to the designation of structure or union members.

Otherwise, --- insert the existing text about array subscript semantics ---

EXAMPLE Consider the array object defined by the declaration

```
int x[3][5];
```

Here x is a 3 x 5 array of ints; more precisely, x is an

In Section 6.7.9 and in Annex A, change the syntax definition for *designator* to the following:

```
designator: array-designator struct-union-designator
```

Also change p6 and p7 of that section to directly refer to *array-designator* and *struct-union-designator*, respectively.

## 22. Improve type generic expression programming

A classical example for an implementation of a type generic feature by means of a macro is the computation of a maximum value:

```
1 #define MAX(A, B) ((A) > (B) ? (A) : (B)) // bad, don't do that
```

Such a simple macro has two effects that any macro programming should avoid. First, if an argument expressions has a side effect, this side effect could be evaluated twice, see Section 10.3. Second, the expanded expression behaves differently to its surroundings with respect to sequencing than a function call would. A function call would be “*indeterminately sequenced*” to other evaluations inside the same expression. The expanded expression of the macro is unsequenced. Later, in Section 23.2, we will see how this lack of sequencing can even be dangerous for features of the C library.

If it wouldn't be for the type genericity of the macro, the same functionality could be implemented with an **inline** function:

```
1 inline
2 uintmax_t MAX(uintmax_t a, uintmax_t b) {
3     return a > b ? a : b;
4 }
```

If we want to be type generic and use that method we would have to implement one function per type, *e.g.* all 16 variants for real types and **void\***, and then encapsulate all of these into a big **\_\_Generic** expression. For an equivalent functionality for application types we would have to modify the generic code each time we add a new type to our system.

Gcc and friends have two extensions that largely ease the programming of type generic macros: statement expressions and the **\_\_typeof\_\_** operator. The first are expressions that contain a compound statement and whose return value is the last value that has been evaluated. The second allows to declare temporary local variable of a deduced type.

```
1 #define MAX(A, B) \
2 ({ \
3     __typeof__(A) a = (A); \
4     __typeof__(B) b = (B); \
5     (a > b ? a : b); \
6 })
```

So this introduces a new keyword, **\_\_typeof\_\_** that works similar to **sizeof**, only that its “return” is a type and not a size. With this, we avoid the problem that we would encounter with the ternary expression. In that expression one of the operands is evaluated twice. So if one of the macro arguments has a side effect, that side effect could be duplicated.

```
1 x = MAX(f(r), g(s));
```



There are problems with both approaches. The `__typeof__` constitutes an intrusion into C's syntax. It is not immediate to see that this doesn't create ambiguities. Second, C++ has introduced a similar feature “**auto**” variables that fulfill a similar purpose, but for which the syntax is different.

Statement expressions introduce the new concept of what should be the result type and value of a compound statement. *E.g.* in the example above the last expression to be evaluated is the ternary expression. This is against the rule of C that an expression that is evaluated as statement discards its value and is only evaluated for its side effects.

Also, a similar expression such as

```

1  #define MAX(A, B)          \
2  ({                        \
3      __typeof__(A) a = (A); \
4      __typeof__(B) b = (B); \
5      /* evaluates to a void */ \
6      if (a > b) a;          \
7      else b;                \
8  })

```

does not give the same result: the result type is **void**, here.

In all the evaluation rules are surprising and form a substantial extension to C semantics.

With the proposal that we develop below a generic **MAX** macro could be implemented as in Listing 4.1.

LISTING 4.1. A macro expanding to a anonymous function

```

1  #define MAX(A, B)          \
2  (static (register a, register b)) { \
3      if (a > b) return a; \
4      else return b; \
5  } (A), (B)

```

Here the construct **static (register a, register b)** refers to a function prototype with incomplete types for return and the two arguments. The core of the macro itself is of a form `(T) { something }` that we already know for compound literals:

**Goal 6** *Amend the C standard to be able to use anonymous functions and inferred types for type generic expressions inside macros.*

**22.1. Storage class for compound literals.** This proposal uses storage class specifiers to force the interpretation of certain expressions as declarations. Then we want to infer the type of such a declaration from the context, namely from an initialization, a function argument or from **return** statements.

In the current version of the C standard, compound literals may already have two of different storage durations: static or automatic. The one that is chosen follows some rules but there is also some slackness given to the compiler implementation.

- Compound literals in file scope always have static storage duration.
- Compound literals that are in block scope and that are not **const** qualified have automatic storage duration.

For others, that is in block scope and **const** qualified, it is up to the compiler to choose, and also the compiler is free to realize two such definitions that have the same value as the same object. So in the following

```

1 static double* x = &(double const){ 1.0, };
2
3 void f(void) {
4     double* y = &(double const){ 1.0, };
5     if (x == y)
6         printf("the_same:_%p\n", (void*)x);
7     else
8         printf("different:_%p_and_%p\n", (void*)x, (void*)y);
9 }

```

both branches of the **if** could be taken.

In some cases pointers are also used as a tool to distinguish objects, even if they are constant, if they can originate from different calls to the same function. Also we might want to initialize a block scope **static** variable with the address of a compound literal.

This could easily be achieved by allowing to add storage class specifiers to compound literals:

```

1 void f(void) {
2     static double* x = &(static double const){ 1.0, };
3     auto double* y = &(auto double const){ 1.0, };
4     if (x == y)
5         printf("error:_%p\n", (void*)x);
6     else
7         printf("different:_%p_and_%p\n", (void*)x, (void*)y);
8 }

```

**22.2. Inferred types for variables and functions.** The second ingredient would be to allow objects and functions to have a type that is inferred from the context. This looks a bit like the new **auto** feature from C++, but is simpler than that. Previous versions of C already had a similar feature but which was later deprecated: the implicit **int** rule. Before C99 the following was valid:

```

1 max(a, b){
2     if (a > b) return a;
3     else return b;
4 }

```

All of this was interpreted as if we had written

```

1 int max(int a, int b){
2     if (a > b) return a;
3     else return b;
4 }

```

In fact, declarations as the following

```

1 register ret = a + b;

```

are still valid syntax, only a constraint forbids them.

There are other places where the type of an object is derived from its initializer: arrays.

```

1 double A[] = { [42] = 7., };

```

Here the type of **A** is incomplete until the end of the initializer is reached. Only then the size is determined and the type is fixed to **double**[43].

For C's grammar to deduce that we are declaring an identifier, we either need a type or we need a storage class specifier. With the new feature the following should be valid:

```

1  register a = 0;           // int
2  static  f = 0.0f;        // float
3  static  g = (double const){ 0 }; // double
4  _Atomic unsigned K;
5  auto    k = K;           // unsigned
6  auto    kp = &K;         // _Atomic(unsigned)*
7  auto    A[] = { [42] = 7., }; // double[43]
8  auto    B[] = { [4] = 6u, [2] = 7L, }; // unsigned[5] or long[5]
9  auto    c = { (char const)0 }; // char
10 auto    C[] = { (char const)0 }; // int[1]
11 auto    p = (char const[]) { 0 }; // char const*
12 inline f(int a) {        // unsigned(int)
13     if (a > 0) return a;
14     else      return 0u;
15 }

```

To make this possible, in contrast to the corresponding C++ feature, no change in the syntax is required. In particular, the keyword **auto** just remains with almost the same syntax and semantic as before. No code that is conforming to previous C standards is invalidated, only some syntactical constructs that were a constraint violation before become valid.

To do so, we have to establish some rules to determine the type of an initializer or a function definition, such that we then can reflect that type back to the declaration.

*Self reference.* If the type of an object  $x$  is determined by its initializer, this initializer can't yet refer to the object  $x$ , not even by using the **&**, **sizeof**, **\_Alignof** or **offsetof** operators.

```

1  void* x = &x;           // valid
2  struct T {
3      struct T* next;
4  } y = { .next = &y, }; // valid
5  auto  z = &z;           // invalid

```

*Simple objects.* This is when the declarator has no array designation and the initializer is a single expression, possibly inside braces.

- The type of the expression is not an array type. Then the type of the initializer is that of the expression, all qualifiers dropped.
- The expression is an array type of base type B. The type of the initializer is pointer to B.

*Arrays.* If the declarator has one or several array designation, either incomplete as  $[]$  or complete  $[N]$  for some expression N, the initializer is interpreted as an array initializer. This array initializer is completed with default initializers  $\{0\}$  where necessary to provide initializers for every array element.

The flattened version of this initializer then has braces and an *initializer-list*,  $I_0, I_1, \dots, I_{n-1}$ . Each of the elements  $I_i$  consists of an optional array designation and an initializer  $E_i$  that must be compatible with the to-be-determined base type of the array.

That base type of the array then is determined by applying the rules for the conditional operator  $?:$  transitively to all expressions  $E_i$  that have a type. So supposing that all  $E_i$  are assignment expressions, the base type would have the type of the expression

$$(1?(1?(1\cdots(1?E_0:E_2)\cdots:E_{n-2}):E_{n-1}))$$

Then one of following properties must hold:

- The initializers  $I_i$  are all *assignment-expression*  $E_i$ , possibly enclosed in braces.

- The  $E_i$  are all of arithmetic type. The base type of the array then is the result of computing the *usual arithmetic conversions* of all its expressions  $E_i$ . As part of these conversions, integer promotion takes place, thus the difference in type between variables `c` and `C` in the example above. Also all qualifications that some of the expression  $E_i$  may have are lost.
- At least one of the  $E_i$  has type pointer to  $B$  (possibly qualified) and all other  $E_i$  are pointers to  $B$  (possibly qualified differently), are **void\*** or are null-pointer constants. The base type of the array is then pointer to  $B$ , eventually adapted to the strongest necessary qualification. If one of the pointers is a pointer to an atomic type, all that are not **void\*** or null pointer constants must be so.
- At least one of the initializers is of structure or union type  $T$ . Then all initializers are valid initializers for a type compatible to  $T$ . The base type is then the unqualified type  $T$ .

*Functions.* For a function *definition* with undetermined return type, the type is inferred from the **return** statements of that function.

- If there is at least one execution branch of the function that doesn't hit a **return** statement, the function is of type **void**.
- If at least one of the **return** statements has no expression, the function type also is **void**.
- Otherwise all execution paths end in **return**  $E_i$  for a list of expressions  $E_i$ . The type of the function then is determined as for arrays and is the computed type that is either an arithmetic type, a pointer type or a **struct** or **union** type.

Regular function *definitions* cannot have undetermined argument types, but see *anonymous functions*, below.

For a function *declaration* that is not a definition, that has some undetermined types (**return** or arguments), but for which a definition is already visible, the inferred types are those that were specified or inferred at that definition. This is to ensure that the following construct is valid:

```

1 // Header (.h file):
2 inline f(int a) {
3     if (a > 0) return a;
4     else      return 0u;
5 }
6
7 // TU (.c file) that instantiates f
8 extern f(register);

```

*Errors.* In all other cases the type cannot be determined and these are considered constraint violations. Such an error must be diagnosed and the compilation stops.

22.2.1. *Extended use of **auto**.* In addition to extend the use of **auto** and other storage class specifiers to compound literals, the use of **auto** should also be permitted for function parameters. In the current standard there is a constraint that forbids this and only allows for **register**. This is probably because function parameters have automatic storage, anyhow, and there was previously no need to specify more than that.

22.3. **Anonymous functions.** Anonymous functions are the most important change to the standard that we propose here. The idea is simple:

**Goal 7** *Extend the notion of compound literals to function types.*

Syntactically this is quite simple: for a compound literal where the type expression is a function specification, the brace `{ }` enclosed initializer is expected to be a *compound*

*statement*, just as for an ordinary function. The successful presence of gcc’s block expressions as an extension shows that such an addition can be added to C’s syntax tree without much difficulties.

The semantic is a bit more complicated. There are three cases that we have to consider according to the storage class that is chosen for the anonymous function, **static**, **register** or **auto**. If such a storage class is not explicitly specified in the definition, we propose the following:

**static:** Anonymous functions that are defined in file scope are as if defined **static**.

**register:** Anonymous functions in block scope that refer to symbols defined in their function but don’t have their address taken are as if defined **register**.

**auto:** Anonymous functions in block scope that refer to symbols defined in their function and have their address taken are as if defined **auto**.

**unspecified:** For other combinations the choice is unspecified and the implementation may choose any storage duration that is consistent with the specification.

The three different storage classes for functions imply some specific rules that we will develop now.

22.3.1. **static anonymous functions.** As long as the anonymous function is pure, that is doesn’t access symbols that are defined outside of its own block scope, things are simple. Our first example from Listing 4.1 falls into that category.

More generally, this class of anonymous functions can be specified with some restrictions to their access to outside symbols. Of the symbols that are visible at its point of definition, an anonymous function that is of static storage duration may only access

- its parameters,
- symbols that are declared in file scope,
- block scope symbols that are **static**,
- block scope compile time constants and types.

In particular, the latter forbids access to variably modified types.

Such a function can then simply implemented as an ordinary **static** function with an implementation specific name, much as many implementations do for compound literals in file scope. If needed, the address of such a function can be obtained, just as for any other **static** function.

22.3.2. **register anonymous functions.** A **register** anonymous function has no name and no recoverable address, therefore it can only be used once, at the point of its definition. In particular, if it would be defined in file scope, it could never be called, and could thus be omitted from the final executable that is produced.

A function that is just called once can seamlessly be integrated into the calling function, which here is the same function inside which it is defined. Thereby it can use the block scope objects of its caller directly, and no supplementary mechanism to access all visible objects is necessary.

22.3.3. **auto anonymous functions.** This category of functions is the most difficult to implement. Such a function *f* is similar to a **register** anonymous function in that it must have access to the local variables (types etc) of its definition scope in function *F*. In addition, if *f* cannot be realized as **register** its address is taken and used in a context that is not directly visible to the compiler. As a consequence several calls of the enclosing function *F* may be active simultaneously and *f* may have several instances that see different instances of the local variables of *F*.

Such instantiation of an anonymous function that closes all “open” references to symbols (the local variables) have a long history in Computer Science and are known as *closures*, the concept and the term was coined by Landin [1964]. For Goal 6, closures are not imperative. Since the implementation of closures is non-trivial, we could leave their realization as implementation defined choice that could be tested with a feature macro, e.g. `__STD_AUTO_FUNCTIONS__`.

22.3.4. *A partial implementation.* There are several methods to implement closures effectively, the most promising in our context seems to be the one proposed by Breuel [1988]. It implements a closure of an anonymous function  $f(a,b)$ , say, as a two step function call sequence.

At runtime, at every encounter of the definition of  $f(,)$  in  $F$  a small stub function  $f_c(a,b)$  is generated and is stored in some automatic storage. In that function (which is specific to the calling context) we store a pointer  $c$  to the context of the definition. When called  $f_c(a,b)$  retrieves  $c$ . Then it performs a call  $f'(a,b,c)$  to a statically compiled function  $f'$  that receives  $c$  as additional parameter.

This method of Breuel [1988] is used in `gcc` to implement *local functions*. These can be used as a base to what we propose here, by simply packing such a definition of a local function and an evaluation of its symbol into `gcc`'s statement expression `({ ... })`. The use pattern of such a macro `ANONYMOUS` for a simple case where all types are known looks as follows:

```
anonymous_function.c
39 # define ANON_PART3 (...) __VA_ARGS__
40 #else
41 # define ANONYMOUS(RET) ({ RET __anon_func ANON_PART2
42 # define ANON_PART2 (...) (__VA_ARGS__) ANON_PART3
43 # define ANON_PART3 (...) __VA_ARGS__ __anon_func; })
44 #endif
```

So the syntax is not exactly the same as it would just by extending compound literals to functions, but it comes sufficiently close. When we want to use undetermined types, we have to use `gcc`'s `__typeof__` feature.

```
anonymous_function.c
47 ANONYMOUS(uintmax_t) (uintmax_t a, uintmax_t b) \
48 ({ \
49     if (a < b) return b; \
50     else return a; \
51 }) \
52 ((A), (B)) \
53 \
54 #define MAX(A, B) \
55 /* function header */ \
56 /* return type */ \
57 ANONYMOUS(__typeof__(1 ? (A) : (B))) \
58 /*and then argument list */ \
59 (__typeof__(A) a, __typeof__(B) b) \
```

The implementation works with three macros that are chained to compose the whole statement expression.

```
anonymous_function.c
35 /**
36 #ifdef __clang__
37 # define ANONYMOUS(RET) (^ RET ANON_PART2
```

The first one opens a statement expression with `({`, the third one closes it with `})`. Such a macro is fragile, so this is not something that I want to recommend in every day's code. All of this here just serves as a proof of concept.

#### 22.3.5. Other comparable existing C extensions.

Clang. The `clang` compiler has implemented a similar concept to what we propose here called *Blocks*. The syntax is a bit different, but the semantics are similar with two

notable exception. The first is the access to local variables that are visible in the context of the declaration of a clang-block. These local variables have copy semantics, that is their value at the point of definition of the clang-block is copied into a *new* local variable that is bound to the new clang-block. Clang needs to do this, because it also provides means to copy clang-blocks and to keep such clang-blocks alive, when the definition of the originating clang-block goes out of scope. I don't think that C should consider such complicated scoping and life-time rules and therefore should not follow this approach.

On the other hand, clang-blocks already provide type inference for the return type, but seemingly only in a quite reduced form. Here, types in all **return** statements must be *equal*, there is no inference of a common super-type that could fulfill the requirements of all **return**.

Emulation by macros. A sophisticated macro package has been written by Leushenko<sup>5</sup>. It implements anonymous functions and lambdas to a wide extent.

## 23. Improve the C library

### 23.1. Make the presence of all C library headers mandatory.

*Overview.* The transition from one C standard to the next is a difficult exercise for everybody, for compiler and library providers that have to implement new features and for the programmers that want to upgrade their code to these new features. Usually this difficulty is not in the implementation of individual features themselves, but in the coordination of the effort between the different parties.

In particular upgrading a compiler frontend to a new C language standard is often done independently from upgrading different parts of the C library. This is because most modern platforms are not monolithic but very heterogeneous:

- Several compiler frontends are provided by several sources. *E.g* on most POSIX platforms there are versions of the open source compilers `gcc` and `clang`, but also commercial compilers provided by hardware or OS vendors.
- These compiler frontends are used with different C libraries (*e.g* on Linux or Windows systems) or versions thereof.

This proposal tries to improve on that situation and to simplify the life for both ends, C implementors and C programmers.

**Goal 8** *Allow optional C library headers to be incomplete.*

**Goal 9** *Provide feature test macros for completeness of headers.*

**Goal 10** *Provide a version macro for all C library headers.*

**Goal 11** *Distinguish test macros for compiler and library features.*

**Goal 12** *Make all C library headers mandatory.*

*Current situation.* C11's approach to query the `__STDC_VERSION__`, and the feature test macros

- `__STDC_HOSTED__`
- `__STDC_NO_ATOMICS__`
- `__STDC_NO_COMPLEX__`
- `__STDC_NO_THREADS__`

is not a satisfactory solution to deal with library versions: they have to be queried *before* any header file has been included and thus cannot easily deal with different versions of the C library. This is in particular not appropriate during a transition phase between C versions, where newly implemented features should get as much coverage as possible.

<sup>5</sup><https://github.com/Leushenko/C99-Lambda>

The solution to these difficulties is usually to impose complicated preprocessor tests that ensure that a certain header can be included and that provide fallbacks if certain features are not yet implemented. These preprocessor tests usually use specific compiler and library knowledge, in particular version numbers, and are in most cases a maintenance nightmare, for C implementors as well as for C programmers.

Depending on the platform definitions, *most* of the standard library headers can be optional. As of C11, the minimal requirement is only to have `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdalign.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>`, `<stdint.h>`, `<stdnoreturn.h>` in a free standing environment. The presence of all other C library headers can be checked by feature test macros, see Table 7.

header	C11 feature test macro	new
<code>&lt;assert.h&gt;</code>	<code>__STDC_HOSTED__</code>	<code>__STDC_ASSERT__</code>
<code>&lt;complex.h&gt;</code>	<code>__STDC_NO_COMPLEX__</code>	<code>__STDC_COMPLEX__</code>
<code>&lt;errno.h&gt;</code>	<code>__STDC_HOSTED__</code>	<code>__STDC_ERRNO__</code>
<code>&lt;fenv.h&gt;</code>	<code>__STDC_HOSTED__</code>	<code>__STDC_FENV__</code>
<code>&lt;inttypes.h&gt;</code>	<code>__STDC_HOSTED__</code>	<code>__STDC_INTTYPES__</code>
<code>&lt;locale.h&gt;</code>	<code>__STDC_HOSTED__</code>	<code>__STDC_LOCALE__</code>
<code>&lt;math.h&gt;</code>	<code>__STDC_HOSTED__</code>	<code>__STDC_MATH__</code>
<code>&lt;setjmp.h&gt;</code>	<code>__STDC_HOSTED__</code>	<code>__STDC_SETJMP__</code>
<code>&lt;signal.h&gt;</code>	<code>__STDC_HOSTED__</code>	<code>__STDC_SIGNAL__</code>
<code>&lt;stdatomic.h&gt;</code>	<code>__STDC_NO_ATOMICS__</code>	<code>__STDC_STDATOMIC__</code>
<code>&lt;stdio.h&gt;</code>	<code>__STDC_HOSTED__</code>	<code>__STDC_STDIO__</code>
<code>&lt;stdlib.h&gt;</code>	<code>__STDC_HOSTED__</code>	<code>__STDC_STDLIB__</code>
<code>&lt;string.h&gt;</code>	<code>__STDC_HOSTED__</code>	<code>__STDC_STRING__</code>
<code>&lt;tgmath.h&gt;</code>	<code>__STDC_HOSTED__</code>	<code>__STDC_TGMATH__</code>
<code>&lt;threads.h&gt;</code>	<code>__STDC_NO_THREADS__</code>	<code>__STDC_THREADS__</code>
<code>&lt;time.h&gt;</code>	<code>__STDC_HOSTED__</code>	<code>__STDC_TIME__</code>
<code>&lt;uchar.h&gt;</code>	<code>__STDC_HOSTED__</code>	<code>__STDC_UCHAR__</code>
<code>&lt;wchar.h&gt;</code>	<code>__STDC_HOSTED__</code>	<code>__STDC_WCHAR__</code>
<code>&lt;wctype.h&gt;</code>	<code>__STDC_HOSTED__</code>	<code>__STDC_WCTYPE__</code>

TABLE 7. Feature test macros for C library headers with optional features

*Our proposal.* This proposal is intended to invert the situation, namely that the feature test macros can be made dependent of the header file. The “only” requirements for this to be possible, is to mandate the presence of all these (currently 29) headers and to allow them to be mostly empty, eventually.

So in particular, this does not mean that free-standing environments would have to provide all features of the C library, they would just have to provide mostly empty files or have some simple fallback for the include `<>` construct.

As a consequence of our approach, some of the macros that are mandated in the corresponding header files can also be used as feature test macros. E.g *user code* could provide an alternative for assert macros on freestanding environments:

```

1 #include <assert.h>
2 #ifndef static_assert
3 # define static_assert(X, DOC) \
4     typedef char _Dummy[ " " DOC " " && sizeof(char[ (X) ]) ]
5 #endif

```

That is the header can be included unconditionally, the presence of the feature can then be tested and an alternative can be provided.



With our approach updating an existing header file to a new C version should be easy. Before implementing additional features that might be requested in most cases simply adding the feature test macro should suffice. *E.g* for the atomics extension:

```
#ifndef __STDC_LIB_VERSION__
# define __STDC_LIB_VERSION__ __STDC_VERSION__
#endif
#define __STDC_STDATOMIC__ 201112L
#if __STDC_LIB_VERSION__ > __STDC_STDATOMIC__
# define __STDC_LIB_VERSION__ __STDC_STDATOMIC__
#endif
```

This clearly indicates to C programmers that certain feature are not yet available, but that they can rely on features of a previous version of the standard. A free standing environment that does not implement `<setjmp.h>`, say, would just have to add a header

```
#ifndef __STDC_SETJMP__
# define __STDC_SETJMP__ 0L
# undef __STDC_HOSTED__
# define __HOSTED__ 0
#endif
```

or trigger such definitions for any unknown header file that is included with `<>`.

### Sections of the C standard to be amended

#### 6.10.8:

Remove **p1** about stating that the macros remain constant throughout compiling the same TU. This has to be elaborated specifically for the different sections.

Modify the first phrase of **p2** from:

None of these macro names, nor the identifier **defined**, shall be the subject of a **#define** or a **#undef** preprocessing directive.

by

None of these macro names shall be the subject of a **#define** or a **#undef** preprocessing directive other than effected by the inclusion of a standard header.

The identifier **defined** shall not be the subject of a **#define** or a **#undef** preprocessing directive.

*Note: we'd have to think of a better place this constraint for **defined**.*

Add a new paragraph.

Macros that reflect versions of ISO/IEC 9989 shall expand to the value of **\_\_STDC\_VERSION\_\_** of the corresponding version, to 199000L if the version is ISO/IEC 9899:1990, or to 0L if no version is supported.

**6.10.8.1, add:**

The values of the predefined macros `__DATE__`, `__TIME__`, `__STDC__` and `__STDC_VERSION__` remain constant throughout the translation unit.

**Add the following item and footnote to the list in Section 6.10.8.1:**

`__STDC_LIB_VERSION__` The minimal version of ISO/IEC 9899 for which all included library headers are feature complete.FOOTNOTE

FOOTNOTE: The intent is that this macro should only differ from `__STDC_VERSION__` during the transition of an implementation to a new version of ISO/IEC 9899.

**Move the item for `__STDC_HOSTED__` from Section 6.10.8.1 to 6.10.8.3. and add the following text at its end.**

This macro shall be defined whenever one of the standard headers that are optional for a hosted environment is included. Its value shall be 1 as long as the included standard headers provide all mandated features. Its value shall be 0 if any of the included standard headers misses any of the mandated features.

**6.10.8.2, describes language and not library features, therefore they should not change during compilation. Add as new first paragraph**

The values of the predefined macros listed in the following subclause are either defined or not before inclusion of all standard headers and remain unchanged throughout the translation unit.

**6.10.8.3, add a new first paragraph with example:**

Each standard header with name of the form `<xxxxx.h>` that is not listed in clause 4, p6, defines a macro `__STDC_XXXXX__` that expands to the maximum version number of ISO/IEC 9989 to which this header complies. This version is greater or equal to `__STDC_LIB_VERSION__`.

EXAMPLE: In a hosted environment, the header `<locale.h>` defines the macro `__STDC_LOCALE__` with a value that is at least 199000L. In a freestanding environment, this macro can also evaluate to the value 0L to indicate that no conforming implementation of the locale feature is available. In that case, the macro `__STDC_HOSTED__` also is 0 after inclusion of `<locale.h>`.

Then replace the three items

**\_\_STDC\_NO\_ATOMICS\_\_** The integer constant 1, intended to indicate that the implementation does not support atomic types (including the **\_Atomic** type qualifier) and the `<stdatomic.h>` header.

**\_\_STDC\_NO\_COMPLEX\_\_** The integer constant 1, intended to indicate that the implementation does not support complex types or the `<complex.h>` header.

**\_\_STDC\_NO\_THREADS\_\_** The integer constant 1, intended to indicate that the implementation does not support the `<threads.h>` header.

as follows:

**\_\_STDC\_NO\_ATOMICS\_\_**, after inclusion of the `<stdatomic.h>` header, the integer constant 1, intended to indicate that the implementation does not support atomic features including the **\_Atomic** type qualifier.

**\_\_STDC\_NO\_COMPLEX\_\_**, after inclusion of the `<complex.h>` header, the integer constant 1, intended to indicate that the implementation does not support complex types.

**\_\_STDC\_NO\_THREADS\_\_**, after inclusion of the `<threads.h>` header the integer constant 1, intended to indicate that the implementation does not support the threads interface.

**7.1.2, p2**, remove the footnote (currently number 183). Then add at the end the following paragraph and example after the list of the headers.

All these headers shall be present for a conforming implementation. If a header describes an optional feature that can be tested by a predefined macro (see Section 6.10.8) the header defines some or all of the corresponding features and provides the necessary feature test macros after inclusion.

Example: In a freestanding environment the header `<locale.h>` shall be available, but must not necessarily implement all features. After the inclusion the macros **\_\_STDC\_LOCALE\_\_** and **\_\_STDC\_HOSTED\_\_** are defined. If not all locale features of a specific version of ISO/IEC 9989 are implemented by the header they evaluate to 0L and 0, respectively.

*Partial implementations.* Additionally, other of the macros that are specified by library headers can be used as tests for partial features, see Table 10.

This interpretation of the presence of these macros could be enforced with some redactional effort, but we think that it is sufficient to recommend a reasonable usage. We propose the following

### Sections of the C standard to be amended

Add to the end of 7.1.2 p2:

RECOMMENDED PRACTICE. If an implementation does not support all features of a specific header or of a new version of it, it should use the macros that are to be provided by the header as test macros for partial features. Therefore an initialization or manipulation macro for a specific type should only be provided if the data types and its operations are also provided. E.g the presence of **ATOMIC\_VAR\_INIT** should indicate that the **\_Atomic** qualifier and operators for atomic objects are provided, and likewise the presence of the macro **ONCE\_FLAG\_INIT** should indicate the availability of the type **once\_flag** and the function **call\_once**.

header	macro	feature
<complex.h>	<b>I_complex</b>	<b>_Complex</b>
<math.h>	<b>FP_FAST_FMA</b>	<b>fma</b>
	<b>FP_ILOGB0</b>	<b>ilogb</b>
	<b>MATH_ERRNO</b>	<b>math_errhandling</b>
<signal.h>	<b>SIG_DFL</b>	<b>signal</b>
<stdatomic.h>	<b>ATOMIC_VAR_INIT</b>	<b>_Atomic</b>
	<b>ATOMIC_XXXX_LOCK_FREE</b>	type <b>atomic_XXXX</b>
	<b>ATOMIC_FLAG_INIT</b>	<b>atomic_flag</b>
<stdio.h>	<b>BUFSIZ</b>	<b>setbuf</b>
	<b>_IONBF</b>	<b>setvbuf</b>
	<b>FOPEN_MAX</b>	<b>fopen</b>
	<b>TMP_MAX</b>	<b>tmpnam</b>
	<b>SEEK_SET</b>	<b>fseek</b>
<stdlib.h>	<b>EXIT_SUCCESS</b>	<b>exit</b>
	<b>RAND_MAX</b>	<b>rand</b>
<threads.h>	<b>thread_local</b>	<b>_Thread_local</b>
	<b>ONCE_FLAG_INIT</b>	<b>once_flag</b>
	<b>TSS_DTOR_ITERATIONS</b>	<b>tss_t</b>
<time.h>	<b>TIME_UTC</b>	<b>timespec_get</b>
	<b>CLOCKS_PER_SEC</b>	<b>clock</b>

TABLE 10. Macros to test partial features

**23.2. Add requirements for sequence points.** As it stands currently, all functions of the standard C library can also be implemented as macros. The standard takes care to enforce that such a replacement by a macro is functionally equivalent to a function call. *E.g.* it imposes that such macros must be written that they don't evaluate their arguments more than once (to avoid multiplication of side effects) and that the resulting expression must integrate well into any target expression. The latter is usually achieved by adding parenthesis around the expression, such that no operators can bind to the expression in a non-obvious way.

The standard misses one point, though, for which macro evaluation can be different from function evaluation: sequence points. Let us look at a function from the C standard:

```
char* fgets(char*, size_t, FILE* restrict);
```

Here is a code example that is a bit contrived:

```
char line[256];
if (fgets(line, 256, stdin) == fgets(line, 256, stdin)) {
    if (!feof(stdin) && !ferror(stdin)) {
        // do something if we read 2 lines successfully
        // the second line is stored in line
    }
}
```

If these are function calls, we know that they are indeterminately sequenced, so one is cleanly performed before the other. We would just not be interested which of the two calls is executed first. We know that

- If both calls are successful, they both return `&line[0]`.
- If both of them fail, both return 0, but then `feof` or `ferror` would detect that.
- Otherwise, one of the calls returns `&line[0]`, the other 0.

We also know that if both were successful, `line` contains the last of the two lines that were read.

This changes if `fgets` is realized as a macro, where the expansions are just ordinary expressions, handling *e.g.* directly the buffers that are hidden behind a `FILE`. Now this access to the buffers can not be assumed to be sequenced. The compiler may mix buffer accesses of both “calls” to `fgets`. The result returned in `line` could be some mixed state of the two calls and even the calls could fail or not for the wrong reasons.

Things become even more important when we are talking about atomics, where the fact if two operations are sequenced with respect to one another can validate or invalidate the whole state of an execution. An expression such as

```
atomic_fetch_add_explicit(a, 1, memory_order_acq_rel) +
    atomic_fetch_add_explicit(b, 1, memory_order_acq_rel)
```

can have an outcome that is quite different according to the the two calls being sequenced or not.

---

### Sections of the C standard to be amended

In Section 7.1.4, “*Use of library functions*”, change the phrase

```
1 Any invocation of a library function that is implemented
2 as a macro shall expand to code that evaluates each of
3 its arguments exactly once, fully protected by
4 parentheses where necessary, so it is generally safe to
5 use arbitrary expressions as arguments.
```

to the following

```
1 Any invocation of a library function that is implemented
2 as a macro shall expand to code that is functionally
3 equivalent to the function code and shall in terms of
4 observability of state be indistinguishable from such a
5 call.FOOTNOTE[This means in particular that (1) such a
6 macro expansion evaluates each of its arguments exactly
7 once, fully protected by parentheses where necessary, so
8 it is generally safe to use arbitrary expressions as
9 arguments; (2) the replacement expression is protected
10 with parenthesis if necessary to avoid unexpected
11 operator bindings to surrounding code; (3) the
12 sequencing rules of the replacement expression to the
13 evaluation of the macro arguments and to other
14 evaluations must be observed as if a function call was
15 made.]
```

### 23.3. Provide type generic interfaces for string functions.

The following string search functions in `string.h` are potentially dangerous, since they break the **const** contract of application code. They return a pointer that is not **const**-qualified which is pointing to a **const** qualified object that is received as a pointer parameter:

```
void *memchr(void const *s, int c, size_t n);
char *strchr(char const *s, int c);
char *strpbrk(char const *s1, const char *s2);
char *strrchr(char const *s, int c);
char *strstr(char const *s1, const char *s2);
```

If the application code stores the returned pointer in a variable and uses this later this can lead to two types of problems:

- Writing to read only memory. The result would be a runtime crash of the program.
- A subtle unnoticed modification of an object that is received through a **const**-qualified pointer, and thus a break of an interface contract.

In any case writing to such an object has undefined behavior.

Tracking provenance of pointers to objects to remember if an object is writable or not is a tedious enterprise. **const**-qualification had been introduced to C to avoid the need of such provenance tracking and C library functions that break this contract are jeopardizing the whole idea.

**Goal 13** *Ensure that all C library interfaces honor the **const** contract.*

As a simple fall out of using type generic functions for `str*` and `mem*` functions is that we also can use these to unify functions for `char*` and `wchar_t`, much as `tgmath.h` provides type generic interfaces for the functions in `math.h`.

**Goal 14** *Provide type generic interfaces for narrow and wide string functions.*

The string-to-number conversion functions from `stdlib.h` and `inttypes.h` such as

```
#include <tgmath.h>
```

```
#include <math.h>
```

```
#include <stdlib.h>
```

```
#include <inttypes.h>
```

```
double strtod(const char * restrict nptr, char ** restrict endptr);
```

exhibit an even worse problem, because `endptr` is a pointer to pointer that is forcibly the wrong type if the origin of `nptr` had been `const` qualified: if `endptr` is non-null and the string is successfully scanned, the function stores `nptr+x` (of type `const char*`) into the pointer `*endptr` (of type `char*`). If the `const` properties of `endptr` are not correctly tracked by the application code, the application can be tempted to write into a `const` qualified object and thereby encounter undefined behavior.

Previous versions of the C standard followed that strategy for the interface specification because they wanted to ensure that these functions can be called for both types of strings, those that are `const` qualified and those that are or not. This was necessary because many of these functions existed before `const` was even integrated into C, backwards compatibility was a prime objective, and adding qualifiers to pointer targets was the only way to allow for qualified and unqualified arguments.

This situation has changed with C11. Because it added type generic selection with the keyword `_Generic`, type generic interfaces can now easily be added to the standard and help applications to keep their `const` contracts more easily.

In C++, from which C inherited `const` qualification, such functions would be overloaded with different interfaces that distinguish according to the `const`-qualification of their argument.

```
1  #ifdef __cplusplus
2  void *memchr(void *s, int c, size_t n);
3  void const*memchr(void const *s, int c, size_t n);
4
5  char *strchr(char *s, int c);
6  char const*strchr(char const *s, int c);
7
8  char *strpbrk(char *s1, const char *s2);
9  char const*strpbrk(char const *s1, const char *s2);
10
11 char *strrchr(char *s, int c);
12 char const*strrchr(char const *s, int c);
13
14 char *strstr(char *s1, const char *s2);
15 char const*strstr(char const *s1, const char *s2);
16
17 double strtod(char *nptr, char **endptr);
18 double strtod(const char *nptr, char const**endptr);
19 #endif
```

With `_Generic` we now have the possibility to have a similar interface for C. We can provide macros of the same name as the standard library functions and provide the semantics of function overloading.

At the same time that we overcome the `const` qualification problem, we may also simplify the use of string functions even more. Most string functions and byte functions are copied by interfaces for wide character strings, that have analogous interfaces with

**wchar\_t** instead of **char**. For them, equivalent interfaces can be provided that also map the functionalities to the corresponding **str** names.

By means of type generic selection, see Section 22, the implementation of such macros is straightforward. The following macro selects one of four function pointers according to the first argument being

- a narrow character pointer, or
- a wide character pointer

and according to **const** qualification.

```

165  /**
166  ** @brief An internal macro to select a const conserving string
167  ** function.
168  **
169  ** This macro simply selects the function that corresponds to
170  ** the
171  ** type of argument @a X. It is not meant to be used directly
172  ** but
173  ** to be integrated in specific macros that overload C library
174  ** functions.
175  **
176  ** The argument @a X must correspond to an array or pointer,
177  ** otherwise an error is triggered in the controlling expression
178  **
179  **
180  ** If any of the 4 cases makes no sense for the type generic
181  ** interface in question, that case can be switched off by
182  ** passing a 0 instead of a function. A use of that case in user
183  ** code then results in a compile time error.
184  **
185  ** @see strchr on how to integrate this into a type generic user
186  ** interface
187  **/
188  # define _TG_TO_CONST(SN, SC, WN, WC, X, ...) \
189  _Generic(&(X[0]), \
190  char*: SN, \
191  char const*: SC, \
192  wchar_t*: WN, \
193  wchar_t const*: WC \
194  ) ((X), __VA_ARGS__)

```

Then we add auxiliar defintions such as the following two:

```

214  _TG_INLINE
215  char const*strchr_const(char const *_s, int _c) {
216  return strchr(_s, _c);
217  }

```

```

219  _TG_INLINE
220  wchar_t const*wcschr_const(wchar_t const *_s, wchar_t _c) {
221  return wcschr(_s, _c);
222  }

```



Here, `_TG_INLINE` can hide some specialized storage class, or attribute such as `static inline`, `register` or `_Attribute(_Always_inline)` if such features are included in C2x.

Then we define `strchr` as a macro:

```

224 # undef strchr
225 # define strchr(...) \
226     _TG_TO_CONST(strchr, strchr_const, wcschr, wcschr_const, \
227                 __VA_ARGS__)

```

tgstring.h

The functions that should be covered by such type generic macros are: `memchr`, `strlen`, `strncat`, `strncmp`, `strncpy`, `strpbrk`, `strrchr`, `strspn`, `strstr`, `strtod`, `strtof`, `strtok`, `strtol`, `strtold`, `strtoll`, `strtoul`, `strtoull`, `strtoumax`, `wcsncmp`, `wcsncpy`, `wcspbrk`, `wcsrchr`, `wcsspn`, `wcsstr`, `wcstod`, `wcstof`, `wcstok`, `wcstol`, `wcstold`, `wcstoll`, `wcstoul`, `wcstoull`, `wmemchr`, and `wmemset`.

## 24. Modules

Since decades, C is one of the most widely used programming languages<sup>6</sup>, and is used successfully for large software projects that are ubiquitous in modern computing devices of all scales. For many programmers, software projects and commercial enterprises C has advantages (relative simplicity, faithfulness to modern architectures, backward and forward compatibility) that largely outweigh its shortcomings. Among these shortcomings, is a lack of two important closely related features: *modularity* and *reusability*. C misses to encapsulate different translation units (TU) properly: all symbols that are part of the interface of a software unit such as functions are shared between all TU that are linked together into an executable.

The common practice to cope with that difficulty is the introduction of naming conventions. Usually software units (*modules*) are attributed a name prefix that is used for all data types and functions that constitute the programmable interface (API). Often such naming conventions (and more generally coding styles) are perceived as a burden. They require a lot of self-discipline and experience, and C is missing features that would support or ease their application.

**24.1. C needs a specific approach.** *Modular C* is a new approach that takes up at that point and proposes to fill the gap. It adds one feature (composed identifiers) to the core language and operates with a handful of directives to glue different modules into one project, providing *modularity* and *reusability*.

*Modular C* is thought to be simple and to conserve most features of the core language and of the habits of its community. It should not develop into a new language of its own, but remain a shallow addition on top of C. Hopefully it would be possible to add it as an optional feature to the standard.

**24.2. All is about naming.** Already C's handling of its own library API already shows all weaknesses of the approach:

- It is intrusive: naming choices for the C library strongly influence possible choices for other projects.
- It is inconsistent: the naming convention shows a history of random choices.
- It is ever-growing: every version adds new constraints.
- It is incomplete: `struct` field names or parameters of `inline` functions are not "officially" reserved

<sup>6</sup><http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>

A typical example when using composed identifiers, and some directives to import (and abbreviate) other modules, would look like this:

```

1  #pragma CMOD separator +
2  #pragma CMOD module head   = proj+structs+list
3  #pragma CMOD import elem   = proj+structs+element
4  #pragma CMOD import io     = C+io
5  #pragma CMOD import printf = C+io+printf
6
7  /* The following declarations are exported */
8  #pragma CMOD declaration
9
10 /* Exports proj+structs+list */
11 struct head {
12     elem* first;
13     elem* last;
14 };
15
16 /* From here on, only exported if external linkage. */
17 #pragma CMOD definition
18
19 void say_hello(void) {
20     io+puts("Hello_world");
21 }
22 static unsigned count;
23 static void say_goodbye(void) {
24     printf("on_exit_we_see_%u", count);
25 }
26 head* init(head* h) {
27     if (h) *h = (head){ 0 };
28     return h;
29 }
30 /* Exports proj+structs+list+top, no conflict with ctype.h */
31 double top(head* h) {
32     return h->first.val;
33 }

```

As you can see the main part of this is still "normal" C, only the interpretation and visibility of some identifiers change. There is a much longer story to tell about the features that come almost free when introducing this naming and import scheme. If you want to know more about *Modern C* you should read on with the research report<sup>7</sup> that I have written about it and then just try the reference implementation<sup>8</sup>:

<https://scm.gforge.inria.fr/anonscm/git/cmod/cmod.git>

There are still a lot of bugs and missing features, please help me to find them. Below you find a list of the advantages that you might get in return.

### 24.3. Modular C features.

**Encapsulation:** We ensure encapsulation by introducing a unique composed module name for each TU that is used to prefix identifiers for export. This creates unique global identifiers that will never enter in conflict with any identifier of another module, or with local identifiers of its own, such as function or macro parameters or block scope variables. Thereby by default the import of a module will not pollute the name space of the importer nor interfere with parameter lists.

<sup>7</sup><https://hal.inria.fr/hal-01169491v2/document>

<sup>8</sup><http://cmod.gforge.inria.fr/>

**Declaration by definition:** Generally, any identifier in a module will be defined (and thereby declared) exactly once. No additional declarations in header files or forward declarations for struct types are necessary.

**Brevity:** An abbreviation feature for module names systematically avoids the use of long prefixed identifiers as they are necessary for usual naming conventions.

**Completeness:** The naming scheme using composed identifiers applies to all file scope identifiers, that are objects, functions, enumeration constants, types and macros.

**Separation:** Implementation and interface of a module are mostly specified through standard language features. The separation between the two is oriented along the C notion of external versus internal linkage. For an inline function, the module that defines it also provides its “instantiation”. Type declarations and macro definitions that are to be exported have to be placed in code sections that are identified with a declaration directive.

**Code Reuse:** We export functionality to other modules in two different ways:

- by interface definition and declaration as described above, similar to what is usually provided through a C header file,
- by sharing of code snippets, similar to X macros or C++’s templates. The later allows to create parameterized data structures or functions easily.

**Acyclic dependency:** Import of modules is not from source but uses a compiled object file. This enforces that the import relation between modules defines a directed acyclic graph. By that it can be updated automatically by tools such as POSIX’ make and we are able to provide infrastructure for orderly startup and shutdown of modules according to the import dependency.

**Exchangeability:** The abbreviation feature allows easy interchange of software components that fulfill the same interface contract.

**Optimization:** Our approach allows for the full set of optimizations that the C compiler provides. It eases the use of inline functions and can be made compatible with link time optimization.

**C library structure:** We provide a more comprehensive and structured approach to the C library.

**Extensibility:** Our approach doesn’t interfere with other extensions of C, such as OpenMP or OpenCL.

**Migration path:** We propose a migration path for existing software to the module model. Legacy interfaces through traditional header files can still be used for external users of modules.

## 25. Simplify the object and value models

C’s object model has several quirks that make it difficult to understand, even by experts. The goal here is to streamline the model by eliminating special cases.

**Goal 15** *Simplify the value and memory models.*

**25.1. Remove objects of temporary lifetime.** Objects of temporary lifetime had been introduced to be able to handle array fields in **struct** types. If *e.g.* the return of a function is such a **struct**, with C11 the array field can only be accessed by taking its address. With the introduction of designators that can access arrays without using their address, see Section 21.5 above, this wouldn’t be necessary anymore.

Code that accesses array fields of returned structures directly is rare, anyhow:

```
1 return functionCall(a, b).array[i];
```

We can reasonably expect that at least the designator can be made ICE.

**25.2. Introduce comparison operator for object types.** Although C has the concept of equality for (almost) all its object types, the `==` operator is not introduced for all these types. It is easy to define this operator for **struct** explicitly by a recursive definition. So let's just do this.

In contrast to that, equality of **union** is **not** well defined, so trying to use `==` for **union** types should remain a constraint violation.

With the introduction of this operator, we also can easily require *value consistency*:

- Force consistency between assignment `=` and comparison (`==` and `!=`) for all data types that allow for these operators.

**25.3. Make `memcpy` and `memcmp` consistent.** The memory (layout) model for types defines how an object of a given type can be accessed through memory. As of today the specification of `memcpy` can be interpreted that it will not successfully write a determinate value to padding bytes in structures. A later call to `memcmp` that is issued without knowledge about the **struct** and its padding, could then fail because it evaluates padding bytes to be unequal.

Although **union**s have values, their field overlay is only described through that layout. This is even the case if the **union** object itself is addressless, *e.g.* because it is an rvalue or declared with **register**.

Similar to the above ideas on values, we require *representation consistency*:

- Force consistency between `memcpy` and `memcmp` for all objects that have an address.

**25.4. Enforce representation consistency for `_Atomic` objects.** This guaranty should also be extended to `_Atomic` objects. All functional interfaces make it clear that this type of object is acted upon for their representation. In any case comparison with `==` would first to an atomic load of the value, and the comparison would then be done with the unqualified value.

The generic `atomic_compare_exchange_weak` etc. interfaces may perform comparison and storage operations. These should be clearly defined as operations on the representation. That is:

- The comparison operation has a result as-if it would be done with `memcmp`.
  - Some values that have multiple representations could test unequal for that reason, although a comparison of the values would lead to equality. This can happen for integer types with padding bits such as `_Bool`.
  - Some floating point values that will never test equal, *e.g.* **NAN**, will test equal as representations.
- The assignment operation is done as-if with `memcpy`. In both cases (update of the expected value or update of the atomic target) this must be compatible with future compare-exchange operations on one of the involved objects.

**25.5. Make string literals `char const[]`.** Modifying string literals (and wide string literals) has no defined behavior, bring their type in sync with this. Modifying old programs that are in order with this might be tedious, but should always be possible. It would be simplified by type generic string functions, see 23.3.

**25.6. Default initialize padding to 0.** Treat padding bytes in **struct** and **union** just as if they were declared as `unsigned char[n]` for appropriate value of `n`.

**25.7. Make `restrict` qualification part of the function interface.** The **restrict** qualifier is semantically different from the other three qualifiers:

- It doesn't reflect a property of the qualified object but of a pointed-to object.
- Such qualified parameters imposes a restriction to the caller of the function, not to the callee.

25.7.1. *Function parameters.* The restriction is problematic for the caller, because the qualification is not part of the prototype. As of C11 and before, the two following lines are compatible function prototypes:

```
1 extern void *memmove (void*restrict, void const*restrict, size_t)
  ;
2 extern void *memmove (void*, void const*, size_t);
```

So on one hand, a **restrict** qualification constrains the caller of a function, but on the other hand this constraint is not part of the interface and can only be deduced from an informal description.

25.7.2. *Function return.* Functions that return pointers don't give any information concerning aliasing. This can lead to missed optimization opportunities when the function actually provides a pointer to a new object that are not yet part of the abstract state. The possibility of using a **restrict** qualifier for such a pointer would clarify this:

```
1 void*restrict malloc(size_t);
```

clearly states that the pointer that is returned by **malloc** cannot alias with any object previously known in the abstract state.

25.7.3. *Proposed additions.*

- Add **restrict** qualification as a discriminator to function parameter types.
- Clarify the role of qualifications of function return types. Omit **const**, **volatile** and **\_Atomic** from such return types, but keep **restrict** qualification as discriminating qualification.

25.8. **References.** Should we introduce references similar to C++? This would certainly improve the handling of array parameters a lot.

## 26. Contexts

26.1. **Introduce evaluation contexts in the standard.** Introduce classes of the expected type for each context of expression evaluation:

- *Type specification context* T expressions in **sizeof** (T), **\_Alignof** (T), **offsetof** (T, ...), casts (T), compound literals (T) { ... } and variable declarations.
- *Type evaluation context* expressions in **sizeof** E, and all expressions in **\_Generic**.
- *Object context* for operators **&**, **++** and **--** and the left operands of all assignment operators.
- *Value contexts*
  - *ICE context* for initialization of static integer objects, **case** expressions, initialization of enumeration constants and designator [ ] operator, including array lengths in FLA declarations.  
It is a constraint violation to provide
    - \* a negative value to a designator [ ] operator.
    - \* a 0 value to a designator [ ] operator that is used for an array declaration.
  - *Boolean context* for initialization or assignment to **\_Bool** objects, and controlling expressions of **if**, **while**, **do-while** and **for**.
  - *Integer context* for initialization or assignment to other non-**\_Bool** integer objects, controlling expressions of **switch** and dynamic [ ] operator.
  - *Floating point context* for initialization or assignment to floating point objects.
  - *Pointer context* for initialization or assignment to pointer objects.
  - *Void context* for evaluation of expressions that are unused.

- *Unspecific context* for function calls that do not impose one of the other context to their arguments, in particular variadic functions.

**26.2. Convert object pointers to `void*` in unspecific context.** All object pointers can be implicitly converted to `void*`. Even if the platform has different formats for some specific object pointers, a necessary conversion should not be of the concern of the programmer when, e.g., printing an address with `printf`. Features in `stdarg.h` should be easily adaptable to such a change.

```
#include <stdarg.h>
```

**26.3. Introduce `nullptr` as a generic null pointer constant and deprecate `NULL`.** Historically, the incompatibility in C++ between `(void*) 0` and other pointer types, had forced the wide use of some forms of integer zero, `0` or `0L` as expansion of `NULL`, also in the C community. This use still has a major problem: `NULL` might or might not have the same width as a pointer, and thus using it in unspecific context such as functions with `va_arg` parameters can lead to serious bugs.

After having imposed such a choice, C++ still wasn't satisfied and moved on to a real solution of their problem and introduced `nullptr`. Basically `nullptr` is what `NULL` had been intended to be from the beginning: a generic null pointer constant, that can't be used in integer contexts.

For C, a simple macro with expansion `((void*) 0)` can serve that goal, so one option could be to force `NULL` to have that value. Unfortunately, currently `NULL` is still defined quite differently on different platforms, and it seems that different assumptions are widely made about what `NULL` would be. Therefore it seems to be simpler to phase the use of `NULL` out and introduce a new feature, that is designed analogous to the C++ feature.

This *could* be implemented as a macro, similar to the following

```
1 typedef void* __nullptr_t;
2 #define nullptr ((__nullptr_t)+0)
```

or could be implemented as a keyword.

## List of Rules

A	C and C++ are different, don't mix them and don't mix them up.	3
B	Don't panic.	1
0.1.1.1	C is an imperative programming language.	1
0.1.2.1	C is a compiled programming language.	3
0.1.2.2	A C program is portable between different platforms.	3
0.1.2.3	A C program should compile cleanly without warnings.	5
0.2.1.1	Punctuation characters can be used with several different meanings.	6
0.2.2.1	All identifiers of a program have to be declared.	7
0.2.2.2	Identifiers may have several consistent declarations.	8
0.2.2.3	Declarations are bound to the scope in which they appear.	8
0.2.3.1	Declarations specify identifiers whereas definitions specify objects.	9
0.2.3.2	An object is defined at the same time as it is initialized.	9
0.2.3.3	Missing elements in initializers default to 0.	9
0.2.3.4	For an array with $n$ the first element has index 0, the last has index $n-1$ .	10
0.2.3.5	Each object must have exactly one definition.	10
0.2.4.1	Domain iterations should be coded with a <b>for</b> statement.	11
0.2.4.2	The loop variable should be defined in the initial part of a <b>for</b> .	11
1.3.1.1	The value 0 represents logical false.	16
1.3.1.2	Any value different from 0 represents logical true.	16
1.3.1.3	Don't compare to 0, <b>false</b> or <b>true</b> .	16
1.3.1.4	All scalars have a truth value.	17
1.3.3.1	<b>case</b> values must be integer constant expressions.	21
1.3.3.2	<b>case</b> labels must not jump beyond a variable definition.	21
1.4.0.1	The type <b>size_t</b> represents values in the range $[0, \text{SIZE\_MAX}]$ .	22
1.4.1.1	Unsigned arithmetic is always well defined.	24
1.4.1.2	Operations +, - and * on <b>size_t</b> provide the mathematically correct result if it is representable as a <b>size_t</b> .	24
1.4.1.3	For unsigned values, $a == (a/b) * b + (a \% b)$ .	24
1.4.1.4	Unsigned / and % are well defined only if the second operand is not 0.	25
1.4.1.5	Arithmetic on <b>size_t</b> implicitly does computation $\% (\text{SIZE\_MAX} + 1)$ .	25
1.4.1.6	In case of overflow, unsigned arithmetic wraps around.	25
1.4.1.7	The result of unsigned / and % is always smaller than the operands.	25
1.4.1.8	Unsigned / and % can't overflow.	25
1.4.2.1	Operators must have all their characters directly attached to each other.	25
1.4.2.2	Side effects in value expressions are evil.	26
1.4.2.3	Never modify more than one object in a statement.	26
1.4.3.1	Comparison operators return the values <b>false</b> or <b>true</b> .	26

1.4.3.2	Logic operators return the values <b>false</b> or <b>true</b> .	27
1.4.5.1	&&,   , ?: and , evaluate their first operand first.	28
1.4.5.2	Don't use the , operator.	28
1.4.5.3	Most operators don't sequence their operands.	28
1.4.5.4	Function calls don't sequence their argument expressions.	29
1.4.5.5	Functions that are called inside expressions should not have side effects.	29
1.5.0.1	All values are numbers or translate to such.	30
1.5.0.2	All values have a type that is statically determined.	31
1.5.0.3	Possible operations on a value are determined by its type.	31
1.5.0.4	A value's type determines the results of all operations.	31
1.5.0.5	A type's binary representation determines the results of all operations.	31
1.5.0.6	A type's binary representation is observable.	31
1.5.0.7	Programs execute <b>as if</b> following the abstract state machine.	31
1.5.0.8	Type determines optimization opportunities.	32
1.5.1.1	Each of the 4 classes of base types has 3 distinct unpromoted types.	33
1.5.1.2	Use <b>size_t</b> for sizes, cardinalities or ordinal numbers.	33
1.5.1.3	Use <b>unsigned</b> for small quantities that can't be negative.	33
1.5.1.4	Use <b>signed</b> for small quantities that bear a sign.	33
1.5.1.5	Use <b>ptrdiff_t</b> for large differences that bear a sign.	33
1.5.1.6	Use <b>double</b> for floating point calculations.	33
1.5.1.7	Use <b>double complex</b> for complex calculations.	34
1.5.2.1	Consecutive string literals are concatenated.	35
1.5.2.2	Numerical literals are never negative.	35
1.5.2.3	Decimal integer constants are signed.	35
1.5.2.4	A decimal integer constant has the first of the 3 signed types that fits it.	35
1.5.2.5	The same value can have different types.	35
1.5.2.6	Don't use octal or hexadecimal constants to express negative values.	36
1.5.2.7	Use decimal constants to express negative values.	36
1.5.2.8	Different literals can have the same value.	36
1.5.2.9	The effective value of a decimal floating point constant may be different from its literal value.	36
1.5.2.10	Literals have value, type and binary representation.	37
1.5.3.1	All variables should be initialized.	37
1.5.3.2	Use designated initializers for all aggregate data types.	37
1.5.3.3	{ 0 } is a valid initializer for all object types that are not VLA.	38
1.5.4.1	All constants with particular meaning must be named.	38
1.5.4.2	All constants with different meaning must be distinguished.	38
1.5.4.3	An object of <b>const</b> -qualified type is read-only.	39
1.5.4.4	String literals are read-only.	39
1.5.4.5	Enumeration constants have either an explicit or positional value.	39
1.5.4.6	Enumeration constants are of type <b>signed int</b> .	40
1.5.4.7	An integer constant expression doesn't evaluate any object.	40
1.5.4.8	Macro names are in all caps.	40
1.5.4.9	A compound literal defines an object.	41
1.5.4.10	Don't hide a terminating semicolon inside a macro.	41
1.5.4.11	Right-indent continuation markers for macros to the same column.	41

as-if



1.5.4.12	<b>I</b> is reserved for the imaginary unit.	42
1.5.5.1	The same value may have different binary representations.	42
1.5.5.2	Unsigned arithmetic wraps nicely.	42
1.5.5.3	The maximum value of any integer type is of the form $2^p - 1$ .	42
1.5.5.4	Arithmetic on an unsigned integer type is determined by its precision.	43
1.5.5.5	The second operand of a shift operation must be less than the precision.	44
1.5.5.6	Positive values are represented independently from signedness.	45
1.5.5.7	Once the abstract state machine reaches an undefined state no further assumption about the continuation of the execution can be made.	46
1.5.5.8	It is your responsibility to avoid undefined behavior of all operations.	46
1.5.5.9	Signed arithmetic may trap badly.	46
1.5.5.10	In twos' complement representation <b>INT_MIN</b> < - <b>INT_MAX</b> .	46
1.5.5.11	Negation may overflow for signed arithmetic.	46
1.5.5.12	Use unsigned types for bit operations.	47
1.5.5.13	If the type <code>uintN_t</code> is provided it is an unsigned integer type with exactly $N$ bits width and precision.	47
1.5.5.14	If the type <code>intN_t</code> is provided it is signed, with two's complement representation, has a width of exactly $N$ bits and a precision of $N - 1$ .	47
1.5.5.15	If types with the required properties exist for values of 8, 16, 32 or 64, types <code>uintN_t</code> and <code>intN_t</code> respectively must be provided.	47
1.5.5.16	For any of the fixed-width types that are provided, <b>_MIN</b> (only signed), maximum <b>_MAX</b> and literals <b>_C</b> macros are provided, too.	48
1.5.5.17	Floating point operations are neither <i>associative</i> , <i>commutative</i> or <i>distributive</i> .	49
1.5.5.18	Never compare floating point values for equality.	49
1.6.1.1	Arrays are not pointers.	50
1.6.1.2	An array in a condition evaluates to <b>true</b> .	51
1.6.1.3	There are array objects but no array values.	51
1.6.1.4	Arrays can't be compared.	51
1.6.1.5	Arrays can't be assigned to.	51
1.6.1.6	VLA can't have initializers.	51
1.6.1.7	VLA can't be declared outside functions.	51
1.6.1.8	The length of an FLA is determined by an ICE or an initializer.	51
1.6.1.9	An array length specification must be strictly positive.	51
1.6.1.10	An array with a length not an integer constant expression is an VLA.	52
1.6.1.11	The length of an array <code>A</code> is <code>(sizeof A) / (sizeof A[0])</code> .	52
1.6.1.12	The innermost dimension of an array parameter to a function is lost.	52
1.6.1.13	Don't use the <b>sizeof</b> operator on array parameters to functions.	52
1.6.1.14	Array parameters behave <i>as-if</i> the array is <i>passed by reference</i> <sup>C</sup> .	52
1.6.1.15	A string is a 0-terminated array of <b>char</b> .	53
1.6.1.16	Using a string function with a non-string has undefined behavior.	54
1.6.2.1	Pointers are opaque objects.	55
1.6.2.2	Pointers are valid, null or indeterminate.	55
1.6.2.3	Initialization or assignment with 0 makes a pointer null.	55
1.6.2.4	In logical expressions, pointers evaluate to <b>false</b> iff they are null.	55
1.6.2.5	Indeterminate pointers lead to undefined behavior.	55
1.6.2.6	Always initialize pointers.	55

1.6.3.1	Omitted <b>struct</b> initializers force the corresponding field to 0.	57
1.6.3.2	A <b>struct</b> initializer must initialize at least one field.	57
1.6.3.3	<b>struct</b> parameters are passed by value.	58
1.6.3.4	Structures can be assigned with = but not compared with == or !=.	58
1.6.3.5	A structure layout is an important design decision.	58
1.6.3.6	All <b>struct</b> declarations in a nested declaration have the same scope of visibility.	60
1.6.4.1	Forward-declare a <b>struct</b> within a <b>typedef</b> using the same identifier as the tag name.	60
1.6.4.2	Identifier names terminating with <b>_t</b> are reserved.	61
1.7.1.1	All functions must have prototypes.	63
1.7.1.2	Functions only have one entry but several <b>return</b> .	63
1.7.1.3	A function <b>return</b> must be consistent with its type.	63
1.7.1.4	Reaching the end of the { } block of a function is equivalent to a <b>return</b> statement without expression.	63
1.7.1.5	Reaching the end of the { } block of a function is only allowed for <b>void</b> functions.	63
1.7.2.1	Use <b>EXIT_SUCCESS</b> or <b>EXIT_FAILURE</b> as return values of <b>main</b> .	64
1.7.2.2	Reaching the end of the { } block of <b>main</b> is equivalent to a <b>return EXIT_SUCCESS</b> ;	64
1.7.2.3	Calling <b>exit</b> (s) is equivalent evaluation of <b>return</b> s in <b>main</b> .	64
1.7.2.4	<b>exit</b> never fails and never returns to its caller.	64
1.7.2.5	All cmdline arguments are transferred as strings.	64
1.7.2.6	Of the arguments to <b>main</b> , <b>argv[0]</b> holds the name of the program invocation.	64
1.7.2.7	Of the arguments to <b>main</b> , <b>argv[argc]</b> is 0.	64
1.7.3.1	Make all preconditions for a function explicit.	65
1.7.3.2	In a recursive function, first check the termination condition.	65
1.7.3.3	Ensure the preconditions of a recursive function in a wrapper function.	66
1.7.3.4	Multiple recursion may lead to exponential computation times.	68
1.7.3.5	A bad algorithm will never lead to a performing implementation.	69
1.7.3.6	Improving an algorithm can dramatically improve performance.	69
1.8.0.1	Failure is always an option.	71
1.8.0.2	Check the return value of library functions for errors.	72
1.8.0.3	Fail fast, fail early and fail often.	72
1.8.0.4	Identifier names terminating with <b>_s</b> are reserved.	73
1.8.0.5	Missed preconditions for the execution platform must abort compilation.	73
1.8.0.6	Only evaluate macros and integer literals in a preprocessor condition.	73
1.8.0.7	In preprocessor conditions unknown identifiers evaluate to 0.	73
1.8.2.1	Opaque types are specified through functional interfaces.	76
1.8.2.2	Don't rely on implementation details of opaque types.	76
1.8.2.3	<b>puts</b> and <b>fputs</b> differ in their end of line handling.	76
1.8.2.4	Text input and output converts data.	79
1.8.2.5	There are three commonly used conversion to encode end-of-line.	79
1.8.2.6	Text lines should not contain trailing white space.	80
1.8.2.7	Parameters of <b>printf</b> must exactly correspond to the format specifiers.	80

1.8.2.8	Use "%d", "%X" and "%a" for conversions that have to be read, later.	81	
1.8.2.9	Don't use <b>gets</b> .	82	
1.8.2.10	<b>fgetc</b> returns <b>int</b> to be capable to encode a special error status, <b>EOF</b> , in addition to all valid characters.	83	
1.8.2.11	End of file can only be detected <i>after</i> a failed read.	83	
1.8.3.1	The interpretation of numerically encoded characters depends on the execution character set.	85	
1.8.6.1	Regular program termination should use <b>return</b> from <b>main</b> .	92	
1.8.6.2	Use <b>exit</b> from a function that may terminate the regular control flow.	92	
1.8.6.3	Don't use other functions for program termination than <b>exit</b> , unless you have to inhibit the execution of library cleanups.	92	
1.8.6.4	Use as many <b>assert</b> as you may to confirm runtime properties.	93	
1.8.6.5	In production compilations, use <b>NDEBUG</b> to switch off all <b>assert</b> .	93	
Level 2		95	
C	All C code must be readable.	95	
2.9.0.1	Short term memory and the field of vision are small.	95	
2.9.0.2	Coding style is not a question of taste but of culture.	95	
2.9.0.3	When you enter an established project you enter a new cultural space.	95	
2.9.1.1	Choose a consistent strategy for white space and other text formatting.	96	
2.9.1.2	Have your text editor automatically format your code correctly.	96	
2.9.2.1	Choose a consistent naming policy for all identifiers.	96	
2.9.2.2	Any identifier that is visible in a header file must be conforming.	97	
2.9.2.3	Don't pollute the global name space.	97	
2.9.2.4	Names must be recognizable and quickly distinguishable.	98	
2.9.2.5	Naming is a creative act.	99	
2.9.2.6	File scope identifiers must be comprehensive.	99	
2.9.2.7	A type name identifies a concept.	99	
2.9.2.8	A global constant identifies an artifact.	99	
2.9.2.9	A global variable identifies state.	99	
2.9.2.10	A function or functional macro identifies an action.	99	
2.10.0.1	Function interfaces describe <i>what</i> is done.	100	what
2.10.0.2	Interface comments document the purpose of a function.	100	what for
2.10.0.3	Function code tells <i>how</i> things are done.	100	how
2.10.0.4	Code comments explain <i>why</i> things are done as they are.	100	why
2.10.0.5	Separate interface and implementation.	100	
2.10.0.6	Document the interface – Explain the implementation.	100	
2.10.1.1	Document interfaces thoroughly.	101	
2.10.1.2	Structure your code in units that have strong semantic connections.	102	
2.10.2.1	Implement literally.	103	
2.10.2.2	Control flow must be obvious.	103	
2.10.3.1	Macros should not change control flow in a surprising way.	103	
2.10.3.2	Function like macros should syntactically behave like function calls.	105	
2.10.4.1	Function parameters are passed by value.	105	
2.10.4.2	Global variables are frowned upon.	105	
2.10.4.3	Express small tasks as pure functions whenever possible.	106	

	2.11.1.1	Using <code>*</code> with an indeterminate or null pointer has undefined behavior.	109
	2.11.2.1	A valid pointer addresses the first element of an array of the base type.	109
	2.11.2.2	The length an array object cannot be reconstructed from a pointer.	110
	2.11.2.3	Pointers are not arrays.	110
	2.11.2.4	Pointers have truth.	111
	2.11.2.5	A pointed-to object must be of the indicated type.	111
	2.11.2.6	A pointer must point to a valid object, one position beyond a valid object or be null.	111
	2.11.2.7	Only subtract pointers to elements of an array object.	112
	2.11.2.8	All pointer differences have type <code>ptrdiff_t</code> .	112
	2.11.2.9	Use <code>ptrdiff_t</code> to encode signed differences of positions or sizes.	112
	2.11.4.1	Don't hide pointers in a <code>typedef</code> .	115
	2.11.5.1	The two expressions <code>A[i]</code> and <code>*(A+i)</code> are equivalent.	115
	2.11.5.2	Evaluation of an array <code>A</code> returns <code>&amp;A[0]</code> .	115
	2.11.6.1	In a function declaration any array parameters rewrites to a pointer.	115
	2.11.6.2	Only the innermost dimension of an array parameter is rewritten.	116
	2.11.6.3	Declare length parameters before array parameters.	116
	2.11.7.1	Don't use <b>NULL</b> .	117
	2.11.8.1	A function <code>f</code> without following opening <code>(</code> decays to a pointer to its start.	117
	2.11.8.2	Function pointers must be used with their exact type.	119
	2.11.8.3	The function call operator <code>(...)</code> applies to function pointers.	120
	2.12.0.1	Pointer types with distinct base types are distinct.	121
	2.12.1.1	<code>sizeof(char)</code> is 1 by definition.	121
	2.12.1.2	Every object <code>A</code> can be viewed as <code>unsigned char[sizeof A]</code> .	121
	2.12.1.3	Pointers to character types are special.	121
	2.12.1.4	Use the type <code>char</code> for character and string data.	121
	2.12.1.5	Use the type <code>unsigned char</code> as the atom of all object types.	121
	2.12.1.6	The <code>sizeof</code> operator can be applied to objects and object types.	121
	2.12.2.1	The in-memory order of the representation digits of a numerical type is implementation defined.	123
	2.12.2.2	On most architectures <b>CHAR_BIT</b> is 8 and <b>UCHAR_MAX</b> is 255.	123
Aliasing	2.12.3.1	With the exclusion of character types, only pointers of the same base type may alias.	124
	2.12.3.2	Avoid the <code>&amp;</code> operator.	124
	2.12.4.1	Any object pointer converts to and from <code>void*</code> .	125
	2.12.4.2	Converting an object pointer to <code>void*</code> and then back to the same type is the identity operation.	125
<i>avoid</i> <sup>2*</sup>	2.12.4.3	Avoid <code>void*</code> .	125
	2.12.5.1	Chose your arithmetic types such that implicit conversions are harmless.	125
	2.12.5.2	Don't use narrow types in arithmetic.	126
	2.12.5.3	Use unsigned types whenever you may.	126
	2.12.5.4	Don't use casts.	126
Effective Type	2.12.6.1	Objects must be accessed through their effective type or through a pointer to a character type.	127

2.12.6.2	Any member of an object that has an effective <b>union</b> type can be accessed at any time, provided the byte representation amounts to a valid value of the access type.	127
2.12.6.3	The effective type of a variable or compound literal is the type of its declaration.	127
2.12.6.4	Variables and compound literals must be accessed through their declared type or through a pointer to a character type.	127
2.13.1.1	Don't cast the return of <b>malloc</b> and friends.	131
2.13.1.2	Objects that are allocated through <b>malloc</b> are uninitialized.	131
2.13.1.3	<b>malloc</b> indicates failure by returning a null pointer value.	134
2.13.1.4	For every <b>malloc</b> there must be a <b>free</b> .	137
2.13.1.5	For every <b>free</b> there must be a <b>malloc</b> .	137
2.13.1.6	Only call <b>free</b> with pointers as they are returned by <b>malloc</b> .	137
2.13.2.1	Identifiers only have visibility inside their scope, starting at their declaration.	138
2.13.2.2	The visibility of an identifier can be shadowed by an identifier of the same name in a subordinate scope.	138
2.13.2.3	Every definition of a variable creates a new distinct object.	138
2.13.2.4	Read-only object literals may overlap.	139
2.13.2.5	Objects have a lifetime outside of which they can't be accessed.	139
2.13.2.6	Referring to an object outside of its lifetime has undefined behavior.	139
2.13.2.7	Objects with static storage duration are always initialized.	140
2.13.2.8	Unless they are VLA or temporary objects, automatic objects have a lifetime corresponding to the execution of their block of definition.	140
2.13.2.9	The <b>&amp;</b> operator is not allowed for variables declared with <b>register</b> .	141
2.13.2.10	Variables declared with <b>register</b> can't alias.	141
2.13.2.11	Declare local variables in performance critical code as <b>register</b> .	141
2.13.2.12	For an object that is not a VLA, lifetime starts when the scope of the definition is entered, and it ends when that scope is left.	142
2.13.2.13	Initializers of automatic variables and compound literals are evaluated each time the definition is met.	142
2.13.2.14	For a VLA, lifetime starts when the definition is encountered, ends when the visibility scope is left.	142
2.13.2.15	Objects of temporary lifetime are read-only.	142
2.13.2.16	Temporary lifetime ends at the end of the enclosing full expression.	142
2.13.3.1	Objects of static or thread storage duration are initialized per default.	142
2.13.3.2	Objects of automatic or allocated storage duration must be initialized explicitly.	143
2.13.3.3	Systematically provide an initialization function for each of your data types.	143
2.14.1.1	The string <b>strto...</b> conversion functions are not <b>const</b> -safe.	150
2.14.1.2	The <b>memchr</b> and <b>strchr</b> search functions are not <b>const</b> -safe.	150
2.14.1.3	The <b>strspn</b> and <b>strcspn</b> search functions are <b>const</b> -safe.	151
2.14.1.4	<b>sprintf</b> makes no provision against buffer overflow.	152
2.14.1.5	Use <b>snprintf</b> when formatting output of unknown length.	153
2.14.3.1	Multibyte characters don't contain null bytes.	156
2.14.3.2	Multibyte strings are null terminated.	156
2.15.0.1	Labels for <b>goto</b> are visible in the whole function that contains them.	165

2.15.0.2	<b>goto</b> can only jump to a label inside the same function.	165
2.15.0.3	<b>goto</b> should not jump over variable initializations.	165
Level 3		167
D	Premature optimization is the root of all evil.	167
3.16.0.1	Optimizers are clever enough to eliminate unused initializations.	168
3.16.0.2	The different notations of pointer arguments to functions result in the same binary code.	168
3.16.0.3	Not taking addresses of local variables helps the optimizer because it inhibits aliasing.	168
3.16.1.1	Inlining can open a lot of optimization opportunities.	169
3.16.1.2	Adding a compatible declaration without <b>inline</b> keyword ensures the emission of the function symbol in the current TU.	170
3.16.1.3	An <b>inline</b> function definition is visible in all TU.	170
3.16.1.4	An <b>inline</b> <i>definition</i> goes in a header file.	171
3.16.1.5	An additional <i>declaration</i> without <b>inline</b> goes in exactly one TU.	171
3.16.1.6	Only expose functions as <b>inline</b> if you consider them to be stable.	171
3.16.1.7	All identifiers that are local to an <b>inline</b> function should be protected by a convenient naming convention.	171
3.16.1.8	<b>inline</b> functions can't access <i>identifiers</i> of <b>static</b> functions.	171
3.16.1.9	<b>inline</b> functions can't define or access <i>identifiers</i> of <b>static</b> objects.	171
3.16.2.1	A <b>restrict</b> -qualified pointer has to provide exclusive access.	172
3.16.2.2	A <b>restrict</b> -qualification constrains the caller of a function.	172
E	Don't speculate about performance of code, verify it rigorously.	173
3.16.3.1	Complexity assessment of algorithms needs proofs.	173
3.16.3.2	Performance assessment of code needs measurement.	173
3.16.3.3	All measurements introduce bias.	173
3.16.3.4	Instrumentation changes compile time and runtime properties.	173
3.16.3.5	Collecting higher order moments of measurements to compute variance and skew is simple and cheap.	179
3.16.3.6	Run time measurements must be hardened with statistics.	179
3.17.0.1	Whenever possible, prefer an <b>inline</b> function to a functional macro.	179
3.17.0.2	A functional macro shall provide a simple interface to a complex task.	180
3.17.1.1	Macro replacement is done in an early translation phase, before any other interpretation is given to the tokens that compose the program.	181
3.17.1.2	If the name of functional macro is not followed by ( ) it is not expanded.	181
3.17.3.1	The line number in <u>  <b>LINE</b>  </u> may not fit into an <b>int</b> .	186
3.17.4.1	When passed to a variadic parameter, all arithmetic types are converted as for arithmetic operations, with the exception of <b>float</b> arguments which are converted to <b>double</b> .	190
3.17.4.2	A variadic function has to receive valid information about the type of each argument in the variadic list.	190
3.17.4.3	Using variadic functions is not portable, unless each argument is forced to a specific type.	191
3.17.4.4	Avoid variadic functions for new interfaces.	191
3.17.4.5	The <b>va_arg</b> mechanism doesn't give access to the length of the <b>va_list</b> .	192
3.17.4.6	A variadic function needs a specific convention for the length of the list.	192

3.17.5.1	The result type of a <b>_Generic</b> expression depends on the type of chosen expression.	194
3.17.5.2	Using <b>_Generic</b> with <b>inline</b> functions adds optimization opportunities.	194
3.17.5.3	The type expressions in a <b>_Generic</b> expression should only be unqualified types, no array types and no function types.	194
3.17.5.4	The type expressions in a <b>_Generic</b> expression must refer to mutually incompatible types.	196
3.17.5.5	The type expressions in a <b>_Generic</b> expression cannot be a pointer to VLA.	196
3.17.5.6	All choices <i>expression1 ... expressionN</i> in a <b>_Generic</b> must be valid.	197
3.18.1.1	Side effects in functions can lead to indeterminate results.	201
3.18.1.2	The specific operation of any operator is sequenced after the evaluation of all its operands.	202
3.18.1.3	The effect of updating an object by any of the assignment, increment or decrement operators is sequenced after the evaluation of its operands.	202
3.18.1.4	A function call is sequenced with respect to all evaluations of the caller.	202
3.18.1.5	Initialization list expressions for array or structure types are indeterminately sequenced.	202
3.18.2.1	Each iteration defines a new instance of a local object.	203
3.18.2.2	<b>goto</b> should only be used for exceptional changes in control flow.	203
3.18.3.1	Each function call defines a new instance of a local object.	204
3.18.4.1	<b>longjmp</b> never returns to the caller.	206
3.18.4.2	When reached through normal control flow, a call to <b>setjmp</b> marks the call location as a jump target and returns 0.	206
3.18.4.3	Leaving the scope of a call to <b>setjmp</b> invalidates the jump target.	206
3.18.4.4	A call to <b>longjmp</b> transfers control directly to the position that was set by <b>setjmp</b> as if that had returned the condition argument.	207
3.18.4.5	A 0 as <b>condition</b> parameter to <b>longjmp</b> is replaced by 1.	208
3.18.4.6	<b>setjmp</b> may only be used in simple comparisons inside controlling expression of conditionals.	208
3.18.4.7	Optimization interacts badly with calls to <b>setjmp</b> .	208
3.18.4.8	Objects that are modified across <b>longjmp</b> must be <b>volatile</b> .	209
3.18.4.9	<b>volatile</b> objects are reloaded from memory each time they are accessed.	209
3.18.4.10	<b>volatile</b> objects are stored to memory each time they are modified.	209
3.18.4.11	The <b>typedef</b> for <b>jmp_buf</b> hides an array type.	209
3.18.5.1	C's signal handling interface is minimal and should only be used for elementary situations.	210
3.18.5.2	Signal handlers can kick in at any point of execution.	213
3.18.5.3	After return from a signal handler, execution resumes exactly where it was interrupted.	213
3.18.5.4	A C statement may correspond to several processor instructions.	214
3.18.5.5	Signal handlers need types with uninterruptible operations.	214
3.18.5.6	Objects of type <b>sig_atomic_t</b> should not be used as counters.	214
3.18.5.7	Unless specified otherwise, C library functions are not asynchronous signal safe.	215
3.19.2.1	Pass thread specific data through function arguments.	220



3.19.2.2	Keep thread specific state in local variables.	221
3.19.2.3	Use <b>thread_local</b> if initialization can be determined at compile time.	221
3.19.3.1	Every mutex must be initialized with <b>mtx_init</b> .	223
3.19.3.2	A thread that holds a non-recursive mutex must not call any of the mutex lock function for it.	223
3.19.3.3	A recursive mutex is only released after the holding thread issues as many calls to <b>mtx_unlock</b> as it has acquired locks.	223
3.19.3.4	A locked mutex must be released before the termination of the thread.	223
3.19.3.5	A thread must only call <b>mtx_unlock</b> on a mutex that it holds.	223
3.19.3.6	Each successful mutex lock corresponds to exactly one call to <b>mtx_unlock</b> .	223
3.19.3.7	A mutex must be destroyed at the end of its lifetime.	223
3.19.4.1	On return from a <b>cnd_t</b> wait, the expression must be checked, again.	225
3.19.4.2	A condition variable can only be used simultaneously with one mutex.	225
3.19.4.3	A <b>cnd_t</b> must be initialized dynamically.	226
3.19.4.4	A <b>cnd_t</b> must be destroyed at the end of its lifetime.	226
3.19.5.1	Returning from <b>main</b> or calling <b>exit</b> terminates all threads.	226
3.19.5.2	While blocking on <b>mtx_t</b> or <b>cnd_t</b> a thread frees processing resources.	228
3.20.0.1	Every evaluation has an effect.	229
3.20.1.1	If $F$ is sequenced before $E$ , then $F \rightarrow E$ .	229
3.20.1.2	The set of modifications of an atomic object $A$ are performed in an order that is consistent with the sequenced before relation of any threads that deals with $A$ .	229
3.20.1.3	An acquire operation $E$ in a thread $T_E$ synchronizes with a release operation $F$ in another thread $T_F$ if $E$ reads the value that $F$ has written.	230
3.20.1.4	If $F$ synchronizes with $E$ , all effects $X$ that have happened before $F$ must be visible at all evaluations $G$ that happen after $E$ .	230
3.20.1.5	We only can conclude that one evaluation happened before another if we have a sequenced chain of synchronizations that links them.	230
3.20.1.6	If an evaluation $F$ happened before $E$ , all effects that are known to have happened before $F$ are also known to have happened before $E$ .	230
3.20.2.1	Critical sections that are protected by the same mutex occur sequentially.	231
3.20.2.2	In a critical section that is protected by mutex <code>mut</code> all effects of previous critical sections protected by <code>mut</code> are visible.	231
3.20.2.3	<b>cnd_wait</b> or <b>cnd_timedwait</b> have release-acquire semantics.	231
3.20.2.4	Calls to <b>cnd_signal</b> or <b>cnd_broadcast</b> should always occur inside a critical section that is protected by the same mutex as the waiters.	231
3.20.3.1	All atomic operations with sequential consistency occur in one global modification order, regardless of the atomic object they are applied to.	231
3.20.3.2	All operators and functional interfaces on atomics that don't specify otherwise have sequential consistency.	232
3.20.4.1	Every functional interface for atomic objects has a form with <b>_explicit</b> appended which allows to specify its consistency model.	232
Level 4		235
4.21.0.1	Objects that are declared with <b>register</b> storage duration can't alias.	236



- 4.21.2.1 File scope **static const** objects may be replicated in all compilation units that use them. 238
- 4.21.2.2 File scope **static const** objects cannot be used inside **inline** functions with external linkage. 238
- 4.21.2.3 File scope **extern const** objects may miss optimization opportunities for constant folding and instruction immediates. 239
- 4.21.2.4 File scope **extern** or **static const** objects may miss optimization opportunities because of mispredicted aliasing. 239
- 4.21.3.1 Changing an identifier that represents an integer constant to be an ICE, can change effective types of objects and execution paths. 242



## Listings

<b>Listing 1:</b> A first example of a C program .....	2
<b>Listing 2:</b> An example of a C program with flaws .....	5
<b>Listing 1.1:</b> A program to compute inverses of numbers .....	19
<b>Listing 1.2:</b> copying a string .....	54
<b>Listing 1.3:</b> A sample program manipulating <b>struct tm</b> .....	59
<b>Listing 1.4:</b> flushing buffered output .....	79
<b>Listing 1.5:</b> Implementing <b>fgets</b> in terms of <b>fgetc</b> .....	82
<b>Listing 1.6:</b> A program to concatenate text files .....	83
code/heron_k.h .....	101
code/heron_k.h .....	101
<b>Listing 2.1:</b> A type for computation with rational numbers. ....	106
code/crash.c .....	128
code/crash.c .....	129
code/circular.h .....	131
code/circular.h .....	131
code/circular.h .....	131
code/circular.h .....	132
code/circular.h .....	132
code/circular.h .....	132
code/circular.h .....	132
code/circular.h .....	132
code/circular.h .....	133
code/circular.h .....	133
<b>Listing 2.2:</b> An example for shadowing by local variables .....	138
<b>Listing 2.3:</b> An example for shadowing by an <b>extern</b> variable .....	139
code/literals.c .....	139
code/numberline.c .....	149
code/numberline.c .....	151
code/numberline.c .....	152
code/mbstrings-main.c .....	156
code/mbstrings.h .....	156
code/mbstrings.h .....	159

code/mbstrings.h .....	160
code/mbstrings.h .....	161
code/numberline.c .....	163
<b>Listing 3.5:</b> gcc's assembler for <code>stats_collect2(c)</code> .....	177
<b>Listing 3.7:</b> gcc's version of the first loop of Listing 3.6.....	177
<b>Listing 3.8:</b> clang's version of the first loop of Listing 3.6. ....	178
<b>Listing 3.9:</b> gcc's version of the second loop of Listing 3.6. ....	178
<b>Listing 3.10:</b> gcc's version of the third loop of Listing 3.6. ....	178
<b>Listing 3.11:</b> A simple recursive descent parser for code indentation.....	204
<b>Listing 3.12:</b> The user interface for the recursive descent parser.....	207
<b>Listing 4.1:</b> A macro expanding to a anonymous function.....	249

## Bibliography

- Alpine, 2015. URL <http://alpinelinux.org/>.
- Thomas M. Breuel. Lexical closures for C++. In *Proceedings of the USENIX C++ Conference*, 1988. URL <http://www-cs-students.stanford.edu/~blynn/files/lexic.pdf>.
- Tiobe Software BV, 2015. URL <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>. monthly since 2000.
- Clang, 2015. URL <http://clang.llvm.org/>.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2 edition, 2001.
- Edsger W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Commun. ACM*, 11(3):147–148, March 1968. ISSN 0001-0782. doi: 10.1145/362929.362947. URL <http://doi.acm.org/10.1145/362929.362947>.
- Ulrich Drepper. Futexes are tricky. Red Hat Inc., rev. 1.6, 2011. URL <http://www.akkadia.org/drepper/futex.pdf>.
- Martin Gardner. Mathematical Games – The fantastic combinations of John Conway’s new solitaire game "life". *Scientific American*, 223:120–123, October 1970.
- gcc. GNU compiler collection, 2015. URL <https://gcc.gnu.org/>.
- glibc. GNU C library, 2015. URL <https://www.gnu.org/software/libc/>.
- Doug Gregor. Modules. Apple Inc., Dec 2012. URL <http://llvm.org/devmtg/2012-11/Gregor-Modules.pdf>.
- Jens Gustedt. Futex based locks for c11’s generic atomics. Technical Report RR-XXXX, INRIA, 2015. URL <https://hal.inria.fr/hal-0123xxxx>.
- Jens Gustedt. Futex based locks for c11’s generic atomics, extended abstract. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM, 2016. accepted for publication.
- Darren Hart. A futex overview and update. *LWN.net*, 2009. URL <https://lwn.net/Articles/360699/>.
- D. Richard Hipp. Makeheaders, 1993. URL <http://www.hwaci.com/sw/mkhdr/>.
- Andrew J. Hutton, Stephanie Donovan, C. Craig Ross, Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Proceedings of the Ottawa Linux Symposium*, pages 479–495, 2002. URL <https://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf>.
- IBM System/370 Extended Architecture, Principles of Operation*. IBM, 1983. SA22-7085. JTC1/SC22/WG14, editor. *Programming languages - C*. Number ISO/IEC 9899. ISO, cor. 1:2012 edition, 2011. URL <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf>.
- Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
- Brian W. Kernighan and Dennis M. Ritchie. The C programming language. *Encyclopedia of Computer Science*, 1980.
- Brian W. Kernighan and Dennis M. Ritchie. The state of C. *BYTE*, 13(8):205–210, August 1988a.

- Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988b.
- Donald Knuth. Structured programming with go to statements. In *Computing Surveys*, volume 6. 1974.
- Donald E. Knuth. *The Art of Computer Programming. Volume 1: Fundamental Algorithms*. Addison-Wesley, 3rd edition, 1997.
- Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
- P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964. doi: 10.1093/comjnl/6.4.308. URL <http://comjnl.oxfordjournals.org/content/6/4/308.abstract>.
- Maged M. Michael. ABA prevention using single-word instructions. Technical Report RC23089, IBM Research, 2004.
- MUSL. libc, 2015. URL <http://musl-libc.org>.
- Robert H. B. Netzer and Barton P. Miller. What are race conditions? some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, March 1992. ISSN 1057-4514. doi: 10.1145/130616.130623. URL <http://doi.acm.org/10.1145/130616.130623>.
- T. Nishizeki, K. Takamizawa, and N. Saito. Computational complexities of obtaining programs with minimum number of GO TO statements from flow charts. *Trans. Inst. Elect. Commun. Eng. Japan*, 60(3):259–260, 1977.
- Philippe Pébay. Formulas for robust, one-pass parallel computation of covariances and arbitrary-order statistical moments. Technical Report SAND2008-6212, SANDIA, 2008. URL <http://prod.sandia.gov/techlib/access-control.cgi/2008/086212.pdf>.
- Rob Pike. Go at google. In *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12*, pages 5–6, 2012. doi: 10.1145/2384716.2384720. URL <http://doi.acm.org/10.1145/2384716.2384720>. see also <http://talks.golang.org/2012/splash.article>.
- POSIX. *ISO/IEC/IEEE Information technology – Portable Operating Systems Interface (POSIX®) Base Specifications*, volume 9945:2009. ISO, Geneva, Switzerland, 2009. Issue 7.
- Dennis M. Ritchie. Variable-size arrays in C. *Journal of C Language Translation*, 2(2): 81–86, September 1990.
- Dennis M. Ritchie. The development of the C language. In Thomas J. Bergin, Jr. and Richard G. Gibson, Jr., editors, *Proceedings, ACM History of Programming Languages II*, Cambridge, MA, April 1993. URL <http://cm.bell-labs.com/cm/cs/who/dmr/chist.html>.
- Dennis M. Ritchie, Brian W. Kernighan, and Michael E. Lesk. The C programming language. Comp. Sci. Tech. Rep. No. 31, Bell Laboratories, Murray Hill, New Jersey, October 1975. Superseded by Kernighan and Ritchie [1988b].
- Dennis M. Ritchie, Steven C. Johnson, Michael E. Lesk, and Brian W. Kernighan. Unix time-sharing system: The C programming language. *Bell Sys. Tech. J.*, 57(6):1991–2019, 1978.
- Keith Schwarz. Advanced preprocessor techniques, 2009. URL [http://www.keithschwarz.com/csl061/spring2009/handouts/080\\_Preprocessor\\_2.pdf](http://www.keithschwarz.com/csl061/spring2009/handouts/080_Preprocessor_2.pdf).
- Charles Simonyi. Meta-programming: a software production model. Technical Report CSL-76-7, PARC, 1976. URL <http://www.parc.com/content/attachments/meta-programming-csl-76-7.pdf>.
- Saurabh Srivastava, Michael Hicks, Jeffrey. S Foster, and Patrick Jenkins. Modular information hiding and type-safe linking for C. *IEEE Transactions on Software Engineering*,

- 34(3):357–376, 2008.
- Mikkel Thorup. Structured programs have small tree-width and good register allocation. *Information and Computation*, 142:318–332, 1995.
- Linus Torvalds et al. Linux kernel coding style, 1996. URL <https://www.kernel.org/doc/Documentation/CodingStyle>. evolved mildly over the years.
- John von Neumann. First draft of a report on the EDVAC, 1945. internal document of the ENIAC project, see also von Neumann [1993].
- John von Neumann. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):28–75, 1993. URL <http://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=85>. Edited and corrected by Michael D. Godfrey.
- B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.





# Index

- `.L_AGAIN`, 146, 147
- `.L_ELSE`, 146
- `.L_END`, 147
- `.L_NO`, 146
- `.L_N_GT_0`, 147
- `.L_YES`, 146
- `_Alignas`, 129, 168
- `_Alignof`, 23, 24, 40, 128, 129, 251, 269
- `_Bool`, 17, 33, 44, 195, 239, 240, 268, 269
- `_C`, 48, 273
- `_Complex`, 33, 240, 260
- `_Exit`, 62, 92, 93, 211, 213, 215
- `_Generic`, 15, 180, 192, 193, 193–197, 235, 242, 248, 263, 264, 269, 279
- `_IONBF`, 260
- `_MAX`, 42, 48, 273
- `_MIN`, 48, 273
- `_Noreturn`, 64, 64, 92, 199, 200, 206, 215, 227
- `_Static_assert`, 73, 85, 93
- `_Thread_local`, 139, 221, 260
- `__DATE__`, 185, 258, 289
- `__FILE__`, 120, 185, 289
- `__LINE__`, 185–187, 190, 196, 278, 289
- `__STDC_HOSTED__`, 255–259
- `__STDC_ISO_10646__`, 158
- `__STDC_LIB_EXT1__`, 72, 73
- `__STDC_MB_MIGHT_NEQ_WC__`, 158
- `__STDC_NO_ATOMICS__`, 214, 255, 256, 259
- `__STDC_NO_COMPLEX__`, 28, 28, 41, 255, 256, 259
- `__STDC_NO_THREADS__`, 255, 256, 259
- `__STDC_VERSION__`, 255, 257, 258
- `__STDC_WANT_LIB_EXT1__`, 72
- `__STDC__`, 258
- `__TIME__`, 185, 258, 289
- `__VA_ARGS__`, 143, 144, 187–190, 198, 254, 264, 265, 289
- `__cplusplus`, 263
- `__func__`, 185, 289
- `_explicit`, 232, 233, 280
- `_t`, 7, 61, 97, 274
- `void` function, 62
- `void` parameter list, 62
- `__DATE__`, 185
- `__FILE__`, 185
- `__LINE__`, 185
- `__TIME__`, 185
- `__VA_ARGS__`, 187
- `__func__`, 185
- `abort`, 62, 92, 93, 93, 199, 211, 215
- `abs`, 75
- abstract representation, 31
- `acos`, 74, 75, 188
- `acosh`, 74, 75
- acquire-release consistency, 231
- address, 109, 109
- `AGAIN`, 141, 146, 147
- aliasing, 124, 172
- `alignas`, 129, 129, 168, 169
- `aligned_alloc`, 130, 140
- alignment, 127
- `alignof`, 23, 24, 129, 169
- `and`, 23, 27, 27, 299
- `and_eq`, 23, 43, 299
- Annex K, 72, 73, 77, 81, 82, 88, 90
- API, 70, 97, 99
- application programmable interface, 70
- argument list, 180
  - variable, 63
- arguments, 2, 3
- array, 8
  - fixed length, 51
  - multidimensional, 50
  - variable length, 51
- array index, 9
- `ASCEND`, 203, 204
- `asctime_s`, 88, 89
- `asin`, 74, 75
- `asinh`, 74, 75
- assembler, 145
- `assert`, 65–67, 73, 93, 93, 108, 112, 114, 256, 275
- `assert.h`, 65, 73, 93
- assignment, 7
- `at_quick_exit`, 92, 93, 118, 213, 215
- `atan`, 74, 75
- `atan2`, 74, 75
- `atanh`, 74, 75
- `atexit`, 14, 92, 93, 93, 117, 118, 227
- `atomic_compare_exchange_strong`, 232, 233
- `atomic_compare_exchange_strong_explicit`, 233
- `atomic_compare_exchange_weak`, 230, 232, 233, 268

- `atomic_compare_exchange_weak_explicit`, 233
- `atomic_exchange`, 232
- `atomic_fetch_add`, 232
- `atomic_fetch_add_explicit`, 232, 261
- `atomic_fetch_and`, 232
- `atomic_fetch_or`, 232
- `atomic_fetch_sub`, 232
- `atomic_fetch_xor`, 232
- `atomic_flag`, 200, 214, 229, 232, 260
- `atomic_flag_clear`, 232, 233
- `ATOMIC_FLAG_INIT`, 260
- `atomic_flag_test_and_set`, 230, 232
- `atomic_load`, 232, 233
- `ATOMIC_LONG_LOCK_FREE`, 216, 217
- `atomic_store`, 232, 233
- `ATOMIC_VAR_INIT`, 232, 260
- `auto`, 140, 140, 206, 238, 249–253
- `basic_blocks`, 206–209, 213
- behavior, 4
- behavior, undefined, 45
- binary code, 3
- binary mode IO, 162
- bit, 42
  - least significant, 42
  - most significant, 42
  - sign, 45
- `bitand`, 23, 43, 299
- `bitor`, 23, 43, 299
- block, 7
  - basic, 199
  - dependent, 15
- `break`, 18
- `bsearch`, 118, 118, 119
- buffer overflow, 72
- buffered IO, 78
- `BUFSIZ`, 260
- bus error, 128
- bytes, 121
- C library, 11, 70
  - header
    - `assert.h`, 65, 73, 93
    - `complex.h`, 28, 41, 97
    - `ctype.h`, 84
    - `errno.h`, 34, 158, 163
    - `float.h`, 48
    - `inttypes.h`, 48, 263
    - `iso646.h`, 26, 43
    - `limits.h`, 42, 47
    - `locale.h`, 91
    - `math.h`, 74, 263
    - `setjmp.h`, 206
    - `signal.h`, 210, 212
    - `stdalign.h`, 129
    - `stdarg.h`, 191, 270
    - `stdarg.h`, 63
    - `stdatomic.h`, 214
    - `stdbool.h`, 16, 44
    - `stddef.h`, 34, 112
    - `stdint.h`, 22, 34, 42, 47
    - `stdio.h`, 8, 20, 71, 74
    - `stdlib.h`, 8, 19, 77, 118, 130, 131, 263
    - `stdnoreturn.h`, 64
    - `string.h`, 53, 87, 262
    - `tgmath.h`, 19, 28, 41, 49, 74, 179, 192, 263
    - `threads.h`, 221
    - `time.h`, 34, 56, 58, 87, 97
    - `wchar.h`, 157
    - `wctype.h`, 158
  - call, 2
    - leaf, 67
  - `call_once`, 230, 260
  - `calloc`, 130, 140, 167
  - camel case, 98
  - `carg`, 74, 74
  - `case`, 20
  - cast, 13, 130, 184
  - casts, 126
  - `cbrt`, 74, 75
  - `ceil`, 74, 75
  - `char16_t`, 162
  - `char32_t`, 162
  - `CHAR_BIT`, 122, 123, 123, 276
  - `CHAR_MAX`, 42
  - `CHAR_MIN`, 42
  - character set
    - basic, 53
  - `cimag`, 49, 74, 128
  - circular, 131
    - `circular_append`, 131
    - `circular_delete`, 132
    - `circular_destroy`, 132
    - `circular_element`, 132
    - `circular_getlength`, 133
    - `circular_init`, 132
    - `circular_new`, 132
    - `circular_pop`, 131
    - `circular_resize`, 133
  - `CLEANUP`, 166
  - `clock`, 34, 72, 89, 89, 260
  - `clock_t`, 34, 89
  - `CLOCKS_PER_SEC`, 34, 89, 260
  - `cmp_lit`, 146
  - `CMPLX`, 28, 42
  - `CMPLXF`, 42, 239, 240
  - `cnd_broadcast`, 225, 231, 280
  - `cnd_destroy`, 226
  - `cnd_init`, 226
  - `cnd_signal`, 222, 224–226, 231, 280
  - `cnd_t`, 221, 224–226, 228, 280
  - `cnd_timedwait`, 224, 225, 231, 280
  - `cnd_wait`, 225, 226, 231, 280
  - code path, 15
  - code point, 157
  - coding error, 158
  - comma operator, 105
  - comment, 6
  - compiled, 3, 271
  - compiler, 3, 4
  - compiler output, 4
  - `compl`, 23, 43, 299
  - `complex`, 28, 33, 40, 41, 42, 128, 129, 256, 260, 272
  - complex point

- real, 32
- `complex.h`, 28, 41, 97
- compound literals, **129**
- conj**, **74**
- constant
  - floating point
    - decimal, 34
    - hexadecimal, 34
  - integer
    - character, 34
    - decimal, 34
    - octal, 34
- constraint violation
  - runtime, 165
- consume consistency, **233**
- continue**, 18, **19**, 19, 103, 222
- control flow, 11
- control statements, **15**
- conversion
  - implicit, **184**
- copysign**, **74**, **75**
- corvid
  - chough, 39, 167
  - jay, 39, 95
  - magpie, 1, 39
  - nutcracker, 235
  - raven, 13, 39
- cos**, **74**, **75**
- cosh**, **74**, **75**
- cproj**, **74**
- creal**, **49**, **74**, 74, 128
- critical
  - data, **221**
  - section, **221**
- critical section, **200**
- ctime\_s**, 88
- `ctype.h`, 84
- dangling **else**, 104
- data, 30, **30**
- DBL\_MANT\_DIG**, **48**
- DBL\_MAX**, 42, **48**
- DBL\_MAX\_EXP**, **48**, 101
- DBL\_MIN**, 42, **48**
- DBL\_MIN\_EXP**, **48**, 101
- decay, 115
- declaration, 7
- declaration order, 38
- default**, **20**
- default initializer, 37
- defined, 7
- defined**, 257
- definition, 7, 9
- descend**, 203–209, 213
- designated, 9
- detach, **227**
- diagnostic, 4
- diagnostic output, 5
- difftime**, **34**, 58, **87**, 87, 88
- directives, 6
- div**, **75**
- do**, **18**
- domain, 10

- domain error, 28
- `draw_sep`, 156
- eax**, 145–147
- ebp**, 147
- ebx**, 147
- ecx**, 145–147
- edi**, 145–147
- EDOM**, 163, 164
- edx**, 145–147
- EFAULT**, 148, 163, 164, 166
- effect, **229**
  - visible, **230**
- effective types, **127**
- EILSEQ**, 158, 163
- EINVAL**, 86
- else**, **15**
- `enable_alignment_check`, 129
- end\_line**, 204, **206**
- Endian
  - big, 123
  - little, 123
- ENOMEM**, 163, 164
- entry point, 145
- enumeration, 39
- environment list, 89
- environment variable, 89
- EOF**, **71**, 71, 72, 74, 76, 82, **83**, 83, 163, 164, 166, **206**, 222, 275
- eofOut**, 205, **206**, 207
- EOVERFLOW**, 148, 163, 164, 166
- epoch, 88
- ERANGE**, 86, 163
- erf**, **74**, **75**
- erfc**, **74**, **75**
- errno**, **72**, 72, 83, 86, 92, 103, 148, 158, 160, 163–166, 212, 256
- `errno.h`, 34, 158, 163
- errno\_t**, **34**, 77, 88, 91
- escape characters, 3
- esi**, 145–147
- evaluation
  - indeterminately sequenced, **201**
  - sequenced, **201**
  - unsequenced, **200**
- exact-width integer types, **47**
- executable, 3, 4
- execute, 4
- execution**, 205
- EXIT\_FAILURE**, **64**, 64, 71, 76, 78, 81, 83, 198, 211, 274
- EXIT\_SUCCESS**, 2, 7, 8, 10, 11, 14, 20, 40, **64**, 64, 76, 78, 81, 83, 260, 274
- EXIT\_SUCCESS**, 11
- exp**, **74**, **75**
- exp2**, **74**, **75**
- expm1**, **74**, **75**
- expression, 22
  - controlling, 15
  - integer constant, **40**
- extern**, 62, 118, 120, **138**, 138–140, 182, 216, 238, 239, 245, 252, 269, 281, 283

- fabs**, 18, 19, **74**, **74**, **75**, 193, 194
- fabsf**, 193
- fabsl**, 193
- fallback, 21
- false**, 14, 16, 26, **27**, 27, 30, 40, 42, **44**, 45, 55, 66, 68, 72, 86, 99, 104, 105, 182–187, 196, 271–273
- fclose**, 72, **78**, 78, 83
- fdim**, **74**, **75**
- feof**, 82, **83**, 222, 261
- ferror**, 261
- fflush**, **78**, 78, 79, 128, 207
- fgetc**, 82, 82, 83, 275, 283
- fgetline, 151
- fgetpos**, 72
- gets**, **82**, 82, 83, 151, 204, 261, 283
- goto**, 141, 146, 147
- Fibonacci numbers, **67**
- FILE**, **74**, 76–78, 80–83, 148, 151, 162–164, 166, 172, 189, 191, 261
- file position, **162**
- FLA, *see* fixed length array
- float.h, 48
- floating point
  - real, 32
- floating point multiply add, **74**
- floor**, **74**, **75**
- FLT\_RDXRDX, 101
- FLT\_RADIX**, **75**, 101
- flushing a stream, 78
- fma**, **74**, **74**, **75**, 260
- fmax**, **74**, **75**
- fmin**, **74**, **75**
- fmod**, **74**, **75**
- fopen**, 72, **76**, 76–78, 83, 260
- FOPEN\_MAX**, 260
- fopen\_s**, 77
- for**, 17
- format, 3
- format specifiers, 3
- forward declaration, 60
- FP\_FAST\_FMA**, 260
- FP\_ILOGB0**, 260
- fpclassify**, **75**
- fprintf\_s**, 81, 82
- fprintrnumbers, 163
- fputc**, **74**, 76, 79, 128, 156, 189
- fputs**, **74**, 76, 77, 79, 82, 83, 92, 156, 162, 164–166, 172, 204, 274
- fread**, 162, 162
- free**, 130, 130, 134, 137, 140, 149, 164–166, 277
- freopen**, 76, 77, **78**, 78, 81
- freopen\_s**, 77
- frexp**, **74**, **74**, **75**
- fscanf**, 153
- fseek**, 162, 162, 163, 260
- fsetpos**, 72
- ftell**, 162, 162, 163
- function, 2, 3, 7
  - asynchronous signal safe, **214**
  - pure, 105, **171**
  - recursive, **65**
  - variadic, **190**
- function parameters, 92
- function pointer, **14**, 92
- pointer
  - function, 117
- fwrite**, 162, 162
- getchar**, **82**, 82, 221
- getenv**, **89**, 89, 91
- getenv\_s**, **90**, 91
- gets**, 82, 275
- gets\_s**, 82, 82
- globals, **9**, **99**
- glyph, **157**
- gmtime\_s**, 88, **89**, 89
- greatest common divisor, **65**
- handlers, 92
- happend before, 230
- happened before, **229**, 230
- header files, 8, 100
- heron, 101
- historic interfaces, 19, 32, 34, 39, 44, 45, 58, 60, 72, 74, 76, 83, 84, 89, 121, 150, 199, 209
- Hungarian notation, 98
- hypot**, **74**, **75**
- I**, **41**, **42**, **49**, 96, 97, 260, 273
- identifier, 7
  - reserved, 97
- identifiers, 7
- if**, **15**
- ifdef**, **28**, 28, 158, 183, 191, 211, 215, 216, 246, 254, 263
- ilogb**, **74**, **75**, 260
- imperative, 1, 271
- implementation dependent, 33
- in memory representation, 122
- include files, 8
- include guards, 103
- indices, 8
- initialization, 7
- initializer
  - default, 37
  - designated, **50**
- inlining, **169**
- instructions, 145
- INT16\_C**, **47**
- INT16\_MAX**, **47**
- INT16\_MIN**, **47**
- int16\_t**, **47**
- INT32\_C**, **47**
- INT32\_MAX**, **47**
- INT32\_MIN**, **47**
- int32\_t**, **47**
- INT64\_C**, **47**
- INT64\_MAX**, **47**
- INT64\_MIN**, **47**
- int64\_t**, **47**, 48
- INT8\_C**, **47**
- INT8\_MAX**, **47**
- INT8\_MIN**, **47**
- int8\_t**, **47**
- INT\_MAX**, 42, 46, 118, 148, 163–166, 186, 273
- INT\_MIN**, 42, 46, 273

- integer
  - unsigned, 22
- integers
  - signed, 32
  - unsigned, 32
- interrupt
  - handler, **210**
  - hardware, **209**
  - software, **209**
  - vector, **210**
- interrupted**, 204, 205, **206**, 207
- intmax\_t**, **34**, 81, **85**, 154, 163
- inttypes.h, 48, 263
- is**, 164
- isalnum**, **84**, 84
- isalpha**, **84**, 84
- isblank**, **84**, 84
- iscntrl**, **84**, 84
- isdigit**, **84**, 84, 85
- isfinite**, 75
- isgraph**, **84**, 84
- isinf**, 75
- isless**, 99
- islower**, **84**, 84
- isnan**, 75
- isnormal**, 75
- iso646.h, 26, 43
- isprint**, **84**, 84
- ispunct**, **84**, 84
- isspace**, **84**, 84
- isupper**, **84**, 84, 85
- isxdigit**, **84**, 84
- iteration domain, 10
- j**, 141, 146, 147
- jargon, **1**
- jmp\_buf**, 204, **206**, 206–209, 279
- join, **218**
- jump, **1**, **202**
- jump statement, 141
- jump targets, 21
- keyword
  - \_Alignas**, **129**, 168
  - \_Alignof**, 23, 24, 40, 128, **129**, 251, 269
  - \_Bool**, 17, 33, **44**, 195, 239, 240, 268, 269
  - \_Complex**, 33, 240, 260
  - \_Generic**, 15, 180, **192**, **193**, 193–197, 235, 242, 248, 263, 264, 269, 279
  - \_Noreturn**, **64**, 64, 92, 199, 200, 206, 215, 227
  - \_Static\_assert**, **73**, 85, 93
  - \_Thread\_local**, 139, **221**, 260
  - auto**, **140**, 140, 206, 238, 249–253
  - break**, **18**
  - case**, **20**
  - continue**, 18, **19**, 19, 103, 222
  - default**, **20**
  - defined**, 257
  - do**, **18**
  - else**, **15**
  - extern**, 62, 118, 120, **138**, 138–140, 182, 216, 238, 239, 245, 252, 269, 281, 283
  - for**, **17**
  - if**, **15**
  - ifdef**, **28**, 28, 158, 183, 191, 211, 215, 216, 246, 254, 263
  - pragma**, 266
  - register**, **140**
  - return**, **63**
  - static**, **140**
  - switch**, **20**
  - undef**, 257, 265
  - while**, **18**
- keywords, 6, 7
- knowingly happen before, **229**
- label, 141
- labels, 145, **202**
- labs**, 75
- LC\_ALL**, 92
- LC\_COLLATE**, 92
- LC\_CTYPE**, 92
- LC\_MONETARY**, 92
- LC\_NUMERIC**, 92
- LC\_TIME**, 92
- LDBL\_MAX**, 195
- ldexp**, 74, 74, 75
- ldiv**, 75
- LEFT**, 203, 204
- lgamma**, 74, 75
- library
  - constant
    - memory\_order\_acq\_rel**, 232, 233, 261
    - memory\_order\_acquire**, 232, 233
    - memory\_order\_consume**, 228, 232, 233
    - memory\_order\_relaxed**, 232, 233
    - memory\_order\_release**, 232, 233
    - memory\_order\_seq\_cst**, 232, 233
    - mtx\_plain**, 223
    - mtx\_recursive**, 223
    - mtx\_timed**, 222, 223
  - function
    - \_Exit**, 62, 92, **93**, 211, 213, 215
    - abort**, 62, 92, **93**, 93, 199, 211, 215
    - abs**, 75
    - aligned\_alloc**, **130**, 140
    - asctime\_s**, 88, **89**
    - at\_quick\_exit**, 92, **93**, 118, 213, 215
    - atexit**, 14, 92, **93**, 93, 117, 118, 227
    - atomic\_compare\_exchange\_strong**, 232, 233
    - atomic\_compare\_exchange\_strong\_explicit**, 233
    - atomic\_compare\_exchange\_weak**, 230, 232, 233, 268
    - atomic\_compare\_exchange\_weak\_explicit**, 233
    - atomic\_exchange**, 232
    - atomic\_fetch\_add**, 232
    - atomic\_fetch\_add\_explicit**, 232, 261
    - atomic\_fetch\_and**, 232
    - atomic\_fetch\_or**, 232
    - atomic\_fetch\_sub**, 232
    - atomic\_fetch\_xor**, 232

- `atomic_flag_clear`, 232, 233
- `atomic_flag_test_and_set`, 230, 232
- `atomic_load`, 232, 233
- `atomic_store`, 232, 233
- `bsearch`, 118, 118, 119
- `call_once`, 230, 260
- `calloc`, 130, 140, 167
- `carg`, 74, 74
- `clock`, 34, 72, 89, 89, 260
- `cnd_broadcast`, 225, 231, 280
- `cnd_destroy`, 226
- `cnd_init`, 226
- `cnd_signal`, 222, 224–226, 231, 280
- `cnd_timedwait`, 224, 225, 231, 280
- `cnd_wait`, 225, 226, 231, 280
- `conj`, 74
- `cproj`, 74
- `ctime_s`, 88
- `difftime`, 34, 58, 87, 87, 88
- `div`, 75
- `fabsf`, 193
- `fabsl`, 193
- `fclose`, 72, 78, 78, 83
- `feof`, 82, 83, 222, 261
- `ferror`, 261
- `fflush`, 78, 78, 79, 128, 207
- `fgetc`, 82, 82, 83, 275, 283
- `fgetpos`, 72
- `fgets`, 82, 82, 83, 151, 204, 261, 283
- `fopen`, 72, 76, 76–78, 83, 260
- `fopen_s`, 77
- `fprintf_s`, 81, 82
- `fputc`, 74, 76, 79, 128, 156, 189
- `fputs`, 74, 76, 77, 79, 82, 83, 92, 156, 162, 164–166, 172, 204, 274
- `fread`, 162, 162
- `free`, 130, 130, 134, 137, 140, 149, 164–166, 277
- `freopen`, 76, 77, 78, 78, 81
- `freopen_s`, 77
- `fscanf`, 153
- `fseek`, 162, 162, 163, 260
- `fsetpos`, 72
- `ftell`, 162, 162, 163
- `fwrite`, 162, 162
- `getchar`, 82, 82, 221
- `getenv`, 89, 89, 91
- `getenv_s`, 90, 91
- `gets_s`, 82, 82
- `gmtime_s`, 88, 89, 89
- `isalnum`, 84, 84
- `isalpha`, 84, 84
- `isblank`, 84, 84
- `iscntrl`, 84, 84
- `isdigit`, 84, 84, 85
- `isgraph`, 84, 84
- `islower`, 84, 84
- `isprint`, 84, 84
- `ispunct`, 84, 84
- `isspace`, 84, 84
- `isupper`, 84, 84, 85
- `isxdigit`, 84, 84
- `labs`, 75
- `ldiv`, 75
- `llabs`, 75
- `lldiv`, 75
- `localeconv`, 92
- `localtime_s`, 88, 89, 89
- `longjmp`, 62, 165, 200, 204, 205, 205, 206, 206–209, 213–215, 279
- `main`, 5
- `mblen`, 158, 159
- `mbrtowc`, 159, 160
- `mbsrtowcs`, 157–159
- `memchr`, 53, 54, 54, 55, 87, 150, 262, 263, 265, 277
- `memcmp`, 7, 53, 54, 54, 55, 268
- `memcpy`, 7, 53, 54, 55, 136, 137, 152, 154, 166, 172, 180, 181, 268
- `memmove`, 136, 137, 172, 269
- `mktime`, 72, 87, 88, 88, 89
- `modf`, 75
- `modff`, 75
- `modfl`, 75
- `mtx_destroy`, 222, 223
- `mtx_init`, 222, 223, 280
- `mtx_lock`, 221–223, 226, 228, 231
- `mtx_timedlock`, 222, 223, 231
- `mtx_trylock`, 222, 231
- `mtx_unlock`, 221–224, 226, 231, 280
- `nan`, 75
- `nanf`, 75
- `nanl`, 75
- `perror`, 71, 71, 72, 76–78, 81, 83, 92
- `printf`, 5
- `printf_s`, 73, 81
- `puts`, 14, 20, 21, 35, 70–72, 74, 76, 78, 79, 81, 82, 104, 266, 274
- `qsort`, 118, 118, 119
- `quick_exit`, 62, 92, 93, 211, 213, 215
- `raise`, 62, 207, 212, 213
- `rand`, 260
- `realloc`, 130, 136, 138, 140, 148, 150, 165, 166
- `remove`, 80
- `rename`, 80
- `rewind`, 147
- `scanf`, 153, 153, 154, 186
- `setbuf`, 260
- `setjmp`, 62, 165, 199, 200, 205, 206, 206–209, 213–215, 256, 279
- `setlocale`, 91, 91, 92, 155, 156, 162
- `setvbuf`, 260
- `signal`, 200, 211, 212, 215, 256, 260
- `snprintf`, 70, 152, 153, 164, 165, 277
- `sprintf`, 148, 152, 166, 277
- `sscanf`, 153
- `strchr`, 54, 55, 87, 150, 151, 157, 262–265, 277
- `strcmp`, 54, 54, 55, 99, 115
- `strcoll`, 54, 55, 92
- `strcpy`, 53, 54, 55, 115, 156
- `strcpy_s`, 54
- `strcspn`, 54, 55, 87, 151, 154, 277
- `strftime`, 87, 88, 89, 89, 90, 92

- `strlen`, 13, 14, **53**, 54, 55, 115, 148, 152, 156, 157, 164, 166, 265
- `strncat`, 265
- `strncmp`, 265
- `strncpy`, 265
- `strnlen_s`, 54
- `strpbrk`, **87**, 262, 263, 265
- `strrchr`, **87**, 262, 263, 265
- `strspn`, **54**, 55, 86, **87**, 151, 154, 157, 265, 277
- `strstr`, **87**, 157, 262, 263, 265
- `strtod`, 19, 20, 64, 70, 72, **85**, 154, 263, 265
- `strtof`, **85**, 265
- `strtointmax`, **85**
- `strtok`, **87**, 265
- `strtol`, **85**, 154, 198, 265
- `strtold`, **85**, 265
- `strtoll`, **85**, 198, 265
- `strtoul`, **85**, 85, 86, 149, 154, 198, 199, 265
- `strtoull`, **85**, 149, 150, 198, 227, 265
- `strtoumax`, **85**, 265
- `strxfrm`, 92
- `thrd_create`, 72, 200, 218, 219, 227, 230
- `thrd_current`, 227
- `thrd_detach`, 227, 228
- `thrd_equal`, 227
- `thrd_exit`, 200, 227, 230
- `thrd_join`, 218–220, 228, 230
- `thrd_sleep`, 228
- `thrd_yield`, 228
- `time`, **34**, 59, **87**, 87, 88, **89**, 89, 102, 256, 260
- `timespec_get`, 87, 88, **89**, 89, 173, 174, 178, 179, 224, 260
- `tmpnam`, 260
- `tolower`, **84**
- `toupper`, **84**, 85
- `tss_create`, **221**
- `tss_delete`, **221**
- `tss_get`, **221**
- `tss_set`, **221**
- `ungetc`, 219, 222
- `va_arg`, 191, 192, 270, 278
- `va_copy`, 191
- `va_end`, 191, 192
- `va_start`, 191, 192
- `vfprintf`, 192
- `wcschr`, 264
- `wcsncmp`, 265
- `wcsncpy`, 265
- `wcspbrk`, 265
- `wcsrchr`, 265
- `wcsspn`, 265
- `wcsstr`, 265
- `wctod`, 265
- `wctof`, 265
- `wctok`, 265
- `wctol`, 265
- `wctold`, 265
- `wctoll`, 265
- `wctoul`, 265
- `wctoull`, 265
- `wctype`, 256
- `wmemchr`, 265
- `wmemset`, 265
- macro
  - `__L_AGAIN`, 146, 147
  - `__L_ELSE`, 146
  - `__L_END`, 147
  - `__L_NO`, 146
  - `__L_N_GT_0`, 147
  - `__L_YES`, 146
  - `__IONBF`, 260
  - `__DATE__`, 185, 258, 289
  - `__FILE__`, 120, 185, 289
  - `__LINE__`, 185–187, 190, 196, 278, 289
  - `__STDC_HOSTED__`, 255–259
  - `__STDC_ISO_10646__`, 158
  - `__STDC_LIB_EXT1__`, 72, 73
  - `__STDC_MB_MIGHT_NEQ_WC__`, 158
  - `__STDC_NO_ATOMICS__`, 214, 255, 256, 259
  - `__STDC_NO_COMPLEX__`, **28**, **28**, **41**, 255, 256, 259
  - `__STDC_NO_THREADS__`, 255, 256, 259
  - `__STDC_VERSION__`, 255, 257, 258
  - `__STDC_WANT_LIB_EXT1__`, 72
  - `__STDC__`, 258
  - `__TIME__`, 185, 258, 289
  - `__VA_ARGS__`, 143, 144, 187–190, 198, 254, 264, 265, 289
  - `__cplusplus`, 263
  - `__func__`, 185, 289
  - `acos`, **74**, **75**, 188
  - `acosh`, **74**, **75**
  - `alignas`, **129**, **129**, **168**, **169**
  - `alignof`, 23, 24, **129**, **169**
  - `and`, 23, **27**, 27, 299
  - `and_eq`, 23, **43**, 299
  - `asin`, **74**, **75**
  - `asinh`, **74**, **75**
  - `assert`, 65–67, 73, **93**, 93, 108, 112, 114, 256, 275
  - `atan`, **74**, **75**
  - `atan2`, **74**, **75**
  - `atanh`, **74**, **75**
  - `ATOMIC_FLAG_INIT`, 260
  - `ATOMIC_LONG_LOCK_FREE`, 216, **217**
  - `ATOMIC_VAR_INIT`, 232, 260
  - `bitand`, 23, **43**, 299
  - `bitor`, 23, **43**, 299
  - `BUFSIZ`, 260
  - `cbrt`, **74**, **75**
  - `ceil`, **74**, **75**
  - `CHAR_BIT`, 122, **123**, 123, 276
  - `CHAR_MAX`, 42
  - `CHAR_MIN`, 42
  - `cimag`, **49**, **74**, 128
  - `CLOCKS_PER_SEC`, **34**, **89**, 260
  - `CMPLX`, 28, **42**
  - `CMPLXF`, **42**, 239, 240
  - `compl`, 23, **43**, 299
  - `complex`, 28, 33, 40, **41**, **42**, 128, 129, 256, 260, 272

- copysign**, 74, 75
- cos**, 74, 75
- cosh**, 74, 75
- creal**, 49, 74, 74, 128
- DBL\_MANT\_DIG**, 48
- DBL\_MAX**, 42, 48
- DBL\_MAX\_EXP**, 48, 101
- DBL\_MIN**, 42, 48
- DBL\_MIN\_EXP**, 48, 101
- EDOM**, 163, 164
- EFAULT**, 148, 163, 164, 166
- EILSEQ**, 158, 163
- EINVAL**, 86
- ENOMEM**, 163, 164
- EOF**, 71, 71, 72, 74, 76, 82, 83, 83, 163, 164, 166, 206, 222, 275
- EOVERFLOW**, 148, 163, 164, 166
- ERANGE**, 86, 163
- erf**, 74, 75
- erfc**, 74, 75
- errno**, 72, 72, 83, 86, 92, 103, 148, 158, 160, 163–166, 212, 256
- EXIT\_FAILURE**, 64, 64, 71, 76, 78, 81, 83, 198, 211, 274
- EXIT\_SUCCESS**, 2, 7, 8, 10, 11, 14, 20, 40, 64, 64, 76, 78, 81, 83, 260, 274
- EXIT\_SUCCESS**, 11
- exp**, 74, 75
- exp2**, 74, 75
- expm1**, 74, 75
- fabs**, 18, 19, 74, 74, 75, 193, 194
- false**, 14, 16, 26, 27, 27, 30, 40, 42, 44, 45, 55, 66, 68, 72, 86, 99, 104, 105, 182–187, 196, 271–273
- fdim**, 74, 75
- fgoto**, 141, 146, 147
- FILE**, 74, 76–78, 80–83, 148, 151, 162–164, 166, 172, 189, 191, 261
- floor**, 74, 75
- FLT\_RADIX**, 75, 101
- fma**, 74, 74, 75, 260
- fmax**, 74, 75
- fmin**, 74, 75
- fmod**, 74, 75
- FOPEN\_MAX**, 260
- FP\_FAST\_FMA**, 260
- FP\_ILOGB0**, 260
- fpclassify**, 75
- frexp**, 74, 74, 75
- hypot**, 74, 75
- I**, 41, 42, 49, 96, 97, 260, 273
- ilogb**, 74, 75, 260
- INT16\_C**, 47
- INT16\_MAX**, 47
- INT16\_MIN**, 47
- INT32\_C**, 47
- INT32\_MAX**, 47
- INT32\_MIN**, 47
- INT64\_C**, 47
- INT64\_MAX**, 47
- INT64\_MIN**, 47
- INT8\_C**, 47
- INT8\_MAX**, 47
- INT8\_MIN**, 47
- INT\_MAX**, 42, 46, 118, 148, 163–166, 186, 273
- INT\_MIN**, 42, 46, 273
- isfinite**, 75
- isinf**, 75
- isless**, 99
- isnan**, 75
- isnormal**, 75
- LC\_ALL**, 92
- LC\_COLLATE**, 92
- LC\_CTYPE**, 92
- LC\_MONETARY**, 92
- LC\_NUMERIC**, 92
- LC\_TIME**, 92
- LDBL\_MAX**, 195
- ldexp**, 74, 74, 75
- lgamma**, 74, 75
- llrint**, 74, 75
- llround**, 74, 74, 75
- log**, 74, 75
- log10**, 74, 75
- loglp**, 74, 75
- log2**, 74, 75
- logb**, 74, 75
- LONG\_MAX**, 163
- LONG\_MIN**, 47
- lrint**, 74, 75
- lround**, 74, 74, 75
- math\_errhandling**, 260
- MATH\_ERRNO**, 260
- MB\_LEN\_MAX**, 160
- NAN**, 268
- NDEBUG**, 93, 93, 182, 183, 275
- nearbyint**, 74, 74, 75
- nextafter**, 74, 75
- nexttoward**, 74, 75
- noreturn**, 64
- not**, 23, 27, 27, 299
- not\_eq**, 23, 26, 40
- NULL**, 7, 116, 117, 191, 270, 276
- offsetof**, 24, 40, 251, 269
- ONCE\_FLAG\_INIT**, 260
- or**, 23, 27, 27, 187, 299
- or\_eq**, 23, 43, 299
- pow**, 74, 75
- PRId64**, 48, 48
- PRi64**, 48
- PRIo64**, 48
- PRi32**, 48
- PRi64**, 48
- PRi64**, 48
- PRi64**, 48
- PRi64**, 48
- PTRDIFF\_MAX**, 42
- PTRDIFF\_MIN**, 42
- putc**, 70
- putchar**, 70, 74, 74, 76, 204, 206
- RAND\_MAX**, 260
- remainder**, 74, 75
- remquo**, 74, 75
- rint**, 74, 74, 75
- round**, 74, 74, 75
- scalbln**, 74, 75



- scalbn**, 74, 74, 75
- SEEK\_CUR**, 162
- SEEK\_END**, 162
- SEEK\_SET**, 162, 260
- SIG\_DFL**, 211, 212, 260
- SIG\_ERR**, 212
- SIG\_IGN**, 211, 212
- SIGABRT**, 211, 213, 245
- SIGFPE**, 210, 245
- SIGILL**, 210, 245
- SIGINT**, 211, 213, 245
- signbit**, 75
- SIGSEGV**, 210, 245
- SIGTERM**, 211, 213, 245
- sin**, 74, 74, 75
- sinh**, 74, 75
- SIZE\_MAX**, 22, 24, 25, 31, 40, 42, 99, 161, 271
- sqrt**, 28, 74, 74, 75
- static\_assert**, 73, 73, 93, 256
- stderr**, 76, 76, 77, 92
- stdin**, 82, 82, 83, 148, 153, 153, 172, 203, 204, 206, 219, 222, 261
- stdout**, 70, 76, 76, 78, 79, 81, 83, 128, 148, 156, 204, 205, 206, 207
- tan**, 74, 75
- tanh**, 74, 75
- tgamma**, 74, 75
- thread\_local**, 139, 221, 260, 280
- TIME\_UTC**, 89, 174, 178, 224, 260
- TMP\_MAX**, 260
- true**, 14, 16, 18–20, 26, 27, 27, 40, 42, 44, 45, 46, 51, 66, 68, 72, 86, 99, 219, 220, 222, 224, 271–273
- trunc**, 74, 74, 75
- TSS\_DTOR\_ITERATIONS**, 260
- UCHAR\_MAX**, 42, 122, 123, 123, 239, 240, 276
- UINT16\_C**, 47
- UINT16\_MAX**, 47
- UINT32\_C**, 47
- UINT32\_MAX**, 47
- UINT64\_C**, 47, 48
- UINT64\_MAX**, 47
- UINT8\_C**, 47
- UINT8\_MAX**, 47
- UINT\_MAX**, 42, 42, 45–47, 125
- ULLONG\_MAX**, 42
- ULONG\_MAX**, 42, 86
- WEOF**, 161
- xor**, 23, 43, 299
- xor\_eq**, 23, 43, 299
- obsolete
  - ASCEND**, 203, 204
  - CLEANUP**, 166
  - eofOut**, 205, 206, 207
  - execution**, 205
  - gets**, 82, 275
  - interrupted**, 204, 205, 206, 207
  - LEFT**, 203, 204
  - NEW\_LINE**, 203, 204
  - plusL**, 204, 205, 207–209
  - plusR**, 204, 205, 207
  - RIGHT**, 203, 204, 207
  - tooDeep**, 204, 205, 206–208
- particle
  - \_C**, 48, 273
  - \_MAX**, 42, 48, 273
  - \_MIN**, 48, 273
  - \_explicit**, 232, 233, 280
  - \_t**, 7, 61, 97, 274
  - is**, 164
  - mem**, 53, 142, 263
  - SIG**, 211
  - str**, 53, 97, 157, 181, 182, 263, 264
  - to**, 187
  - wcs**, 157
- tag
  - AGAIN**, 141, 146, 147
  - cmp\_lit**, 146
  - j**, 141, 146, 147
  - n**, 141, 146, 147
  - p**, 141, 146
  - q**, 141, 146
  - timespec**, 58–60, 87, 88, 89, 97, 99, 102, 112, 175, 178, 222, 224, 225, 228
  - tm**, 56–60, 87–89, 283
  - tm\_hour**, 56, 57, 59
  - tm\_isdst**, 56, 57, 88
  - tm\_mday**, 56–59, 87
  - tm\_min**, 56, 57, 59
  - tm\_mon**, 56–59, 87
  - tm\_sec**, 56, 57, 59, 87
  - tm\_wday**, 56, 57, 87, 88
  - tm\_yday**, 56–59, 88
  - tm\_year**, 56–59, 87
  - tv\_nsec**, 58, 88, 89, 112, 174
  - tv\_sec**, 58, 88, 97, 112, 113, 224
- type
  - atomic\_flag**, 200, 214, 229, 232, 260
  - basic\_blocks**, 206–209, 213
  - char16\_t**, 162
  - char32\_t**, 162
  - clock\_t**, 34, 89
  - cnd\_t**, 221, 224–226, 228, 280
  - descend**, 203–209, 213
  - eax**, 145–147
  - ebp**, 147
  - ebx**, 147
  - ecx**, 145–147
  - edi**, 145–147
  - edx**, 145–147
  - end\_line**, 204, 206
  - errno\_t**, 34, 77, 88, 91
  - esi**, 145–147
  - int16\_t**, 47
  - int32\_t**, 47
  - int64\_t**, 47, 48
  - int8\_t**, 47
  - intmax\_t**, 34, 81, 85, 154, 163
  - jmp\_buf**, 204, 206, 206–209, 279
  - mbstate\_t**, 156–161
  - memory\_order**, 232, 233
  - mtx\_t**, 200, 221–226, 228, 229, 280
  - once\_flag**, 229, 260

- `ptrdiff_t`, 17, 33, **34**, 42, 81, **112**, 154, 272, 276
- `r12`, 177, 178
- `r13`, 178
- `rax`, 145, 146, 177, 178
- `rbp`, 145–147
- `rbx`, 147, 178
- `rcx`, 145
- `rdi`, 177
- `rdx`, 145, 146
- `rip`, 177, 178
- `rsize_t`, **34**, 82, 88, 91
- `rsp`, 145–147, 177, 178
- `sh_count`, 211, 214, 216, 217
- `sh_counted`, 217
- `sh_enable`, 212
- `sig_atomic_t`, **200**, 214, 279
- `signal_handler`, 211, 212
- `size_t`, 2, 7–10, 13–15, 17–19, 22–28, 30–33, **34**, 40, 42, 54, 55, 61, 65–69, 74, 81, 82, 86, 87, 91, 96, 97, 99, 106–110, 112–116, 118, 119, 122, 124, 126–128, **130**, 130, 132–136, 144, 148–154, 156–164, 166, 168, **172**, 189, 192, 198, 202–204, 216, 220, 226, 262, 263, 269, 271, 272
- `size_t`, 11
- `skipspace`, 204, **206**
- `thrd_start_t`, 218
- `thrd_t`, 218, 219, 226–228
- `time_t`, **34**, 58, **87**, 87–89
- `tss_dtor_t`, **221**
- `tss_t`, **221**, 260
- `uint16_t`, **47**
- `uint32_t`, **47**, 48
- `uint64_t`, **47**, 174, 178
- `uint8_t`, **47**
- `uintmax_t`, **34**, 81, **85**, 154, 248, 254
- `va_list`, 191, 192, 278
- `wchar_t`, 154, 157–161, 263, 264
- `xmm0`, 177
- `xmm1`, 177
- `xmm2`, 177
- `xmm3`, 177
- `xmm4`, 177
- lifetime, 129, 139
- limits.h, 42, 47
- line buffering, 79
- literal, 7
  - compound, 41
  - integer
    - hexadecimal, 34
  - string, 35
- literals, 34
- llabs**, **75**
- lldiv**, **75**
- llrint**, **74**, **75**
- llround**, **74**, **74**, **75**
- locale, 91
- locale.h, 91
- localeconv**, 92
- localtime\_s**, 88, **89**, 89
- lock free, **200**
- lock-free, **217**
- log**, **74**, **75**
- log10**, **74**, **75**
- log1p**, **74**, **75**
- log2**, **74**, **75**
- logb**, **74**, **75**
- LONG\_MAX**, 163
- LONG\_MIN**, 47
- longjmp**, 62, 165, 200, 204, **205**, 205, **206**, 206–209, 213–215, 279
- loop
  - variable, 10, 11
- loop body, 10
- loop condition, 10
- lrint**, **74**, **75**
- lround**, **74**, **74**, **75**
- LSB, **42**
- lvalue, 25
- macro, 40
  - function-like, **41**
  - functionlike, 70
  - variadic, **187**
- macro call, **180**
- main**, 5
- math.h, 74, 263
- math\_errhandling**, 260
- MATH\_ERRNO**, 260
- MB\_LEN\_MAX**, 160
- mblen**, 158, 159
- mbrtow**, 160
- mbrtowc**, 159, 160
- mbsrdup**, 159
- mbsrlen**, 156
- mbsrtowcs**, 157–159
- mbsrwc**, 161
- mbstate\_t**, 156–161
- mem**, 53, 142, 263
- memchr**, **53**, **54**, 54, 55, **87**, 150, 262, 263, 265, 277
- memcmp**, 7, **53**, **54**, 54, 55, 268
- memcpy**, 7, **53**, 54, 55, 136, 137, 152, 154, 166, **172**, 180, 181, 268
- memmove**, 136, 137, **172**, 269
- memory allocation
  - dynamic, 130
- memory leak, 137
- memory\_order**, 232, 233
- memory\_order\_acq\_rel**, 232, 233, 261
- memory\_order\_acquire**, 232, 233
- memory\_order\_consume**, 228, 232, 233
- memory\_order\_relaxed**, 232, 233
- memory\_order\_release**, 232, 233
- memory\_order\_seq\_cst**, 232, 233
- mktime**, 72, 87, **88**, 88, 89
- modf**, **75**
- modff**, **75**
- modfl**, **75**
- modification order, **229**
- modularity, **62**
- MSB, **42**
- mtx\_destroy**, 222, 223
- mtx\_init**, 222, 223, 280

- `mtx_lock`, 221–223, 226, 228, 231
- `mtx_plain`, 223
- `mtx_recursive`, 223
- `mtx_t`, 200, 221–226, 228, 229, 280
- `mtx_timed`, 222, 223
- `mtx_timedlock`, 222, 223, 231
- `mtx_trylock`, 222, 231
- `mtx_unlock`, 221–224, 226, 231, 280
- multibyte character, **156**
- multibyte string, **156**
- mutual exclusion, **221**
- n**, 141, 146, 147
- namespace
  - tag, 40
- NAN**, 268
- `nan`, 75
- `nanf`, 75
- `nanl`, 75
- NDEBUG**, 93, 93, 182, 183, 275
- `nearbyint`, 74, 74, **75**
- NEW\_LINE**, 203, 204
- `nextafter`, 74, **75**
- `nexttoward`, 74, **75**
- `noreturn`, **64**
- `not`, 23, **27**, 27, 299
- `not_eq`, 23, **26**, 40
- NULL**, 7, 116, 117, 191, 270, 276
- null pointer, 55
- null pointer constant, 117
- numberline, 149
- object
  - data, 7, 10
  - function, 10
- `offsetof`, 24, 40, 251, 269
- `once_flag`, 229, 260
- ONCE\_FLAG\_INIT**, 260
- opaque
  - structure, 114
- opaque type, 76
- open file, 77
- operand, 22
- operation
  - uninterruptible, 214
- operator, 7, 10, 22
  - `& &`, **27**
  - `&=`, **43**
  - `&`, **43**
  - `&` unary, **109**
  - `()`, **62**
  - `*=`, **25**
  - `*` binary, **22**
  - `*` unary, **109**
  - `++`, **25**
  - `+=`, **25**
  - `+` binary, **22**
  - `+` unary, **24**
  - `--`, **25**
  - `-=`, **25**
  - `->`, **113**
  - `-` binary, **22**
  - `-` unary, **24**
  - `..`, **57**
  - `/=`, **25**
  - `/`, **24**
  - `< <=`, **44**
  - `< <`, **44**
  - `<=`, **26**
  - `<`, **26**
  - `==`, **26**
  - `=`, **25**, **58**
  - `>=`, **26**
  - `> >=`, **44**
  - `> >`, **44**
  - `>`, **26**
  - `[]`, **50**
  - `%=`, **25**
  - `%`, **24**
  - `~`, **43**
  - `^=`, **43**
  - `^`, **43**
  - `==`, 16
  - `=`, 16
  - `||`, **27**
  - `!`, **27**
  - `!=`, **26**
  - `...`, **187**, **190**
  - `:`, *see* ternary
  - `?`, *see* ternary
  - `##`, **186**
  - `#`, **186**
  - `,`, **28**
  - `and_eq`, **43**
  - `and`, **27**
  - `bitand`, **43**
  - `bitor`, **43**
  - `compl`, **43**
  - `not`, **27**
  - `or_eq`, **43**
  - `or`, **27**
  - `xor_eq`, **43**
  - `xor`, **43**
  - concatenation, **186**
  - object-of, 109
  - postfix
    - decrement, 26
    - increment, 26
  - prefix
    - decrement, 25
    - increment, 25
  - ternary, 15, **28**
  - optimization, 32
  - `or`, 23, **27**, 27, 187, 299
  - `or_eq`, 23, **43**, 299
  - overflow, 24, 25
  - p**, 141, 146
  - parameter list, **180**
  - parenthesis, 2
  - parsing state, **158**
  - pass by reference, 209
  - passed by reference, 52, 273
  - pererror**, 71, 71, 72, 76–78, 81, 83, 92
  - platform, 3
  - platform dependent, 3

- plusL**, 204, **205**, 207–209
- plusR**, 204, **205**, 207
- pointer, 17, 109
  - null, 55
- pointer difference, 112
- pointer null, 116
- portable, 3, 271
- pow**, **74**, **75**
- pragma**, 266
- precision, 32, 42
  - floating point, **48**
- preprocessing, **180**
- preprocessor, 40
- preprocessor conditionals, 73
- preprocessor directives, 28
- PRId64**, **48**, 48
- PRi64**, **48**
- print, 3
- printf**, 5
- printf\_s**, 73, 81
- prints, 2
- PRio64**, **48**
- PRiu32**, 48
- PRiu64**, **48**
- PRIX64**, **48**
- PRIx64**, **48**
- prototype, 62
- PTRDIFF\_MAX**, 42
- PTRDIFF\_MIN**, 42
- ptrdiff\_t**, 17, 33, **34**, 42, 81, **112**, 154, 272, 276
- punctuation, 6
- putc**, 70
- putchar**, 70, **74**, 74, 76, 204, **206**
- puts**, 14, 20, 21, 35, 70–72, 74, 76, 78, 79, 81, 82, 104, 266, 274
- q**, 141, 146
- qsort**, **118**, 118, 119
- qualifier, 38
  - const**, 41
- quick sort, **119**
- quick\_exit**, 62, 92, **93**, 211, 213, 215
- r12**, 177, 178
- r13**, 178
- race condition, **220**
- raise**, 62, 207, 212, 213
- rand**, 260
- RAND\_MAX**, 260
- rax**, 145, 146, 177, 178
- rbp**, 145–147
- rbx**, 147, 178
- rcx**, 145
- rdi**, 177
- rdx**, 145, 146
- read-modify-write, **230**
- realloc**, **130**, 136, 138, 140, 148, 150, 165, 166
- recursion, **65**
  - infinite, **65**
- reference
  - array, 52
- register**, 140
- release, **229**
- remainder**, **74**, **75**
- remove**, 80
- remquo**, **74**, **75**
- rename**, 80
- replacement text, **180**
- representation
  - binary, 31
  - floating point, **31**
  - object, **31**
  - ones' complement, 45
  - sign, **31**, 45
  - sign and magnitude, 45
  - two's complement, 45
- reserved, 6
- return, **11**
- return**, **63**
- rewind**, 147
- RIGHT**, 203, 204, 207
- ring, **42**
- rint**, **74**, 74, **75**
- rip**, 177, 178
- round**, **74**, 74, **75**
- rsizet**, **34**, 82, 88, 91
- rsp**, 145–147, 177, 178
- rvalue, 25
- scalar, 17
- scalbln**, **74**, **75**
- scalbn**, **74**, 74, **75**
- scanf**, **153**, 153, 154, 186
- scope, 8, 18
  - block, 9
  - file, 9
- SEEK\_CUR**, **162**
- SEEK\_END**, **162**
- SEEK\_SET**, **162**, 260
- selection, 20
- sequence before, 230
- sequence points, **200**
- sequenced, **199**
- sequenced before, 229, 230
- sequential consistency, **231**, **232**
- set intersection, **43**
- set union, **43**
- setbuf**, 260
- setjmp**, 62, 165, 199, 200, 205, **206**, 206–209, 213–215, 256, 279
- setjmp.h, 206
- setlocale**, **91**, 91, 92, 155, 156, 162
- setvbuf**, 260
- sh\_count**, 211, 214, 216, 217
- sh\_counted**, 217
- sh\_enable**, 212
- shadowed variable, 138
- shared, **220**
- short circuit evaluation, 27, 44
- side effect, 26
- SIG**, 211
- sig\_atomic\_t**, **200**, 214, 279
- SIG\_DFL**, 211, 212, 260
- SIG\_ERR**, 212
- SIG\_IGN**, 211, 212

- SIGABRT**, 211, 213, 245
- SIGFPE**, 210, 245
- SIGILL**, 210, 245
- SIGINT**, 211, 213, 245
- signal, **209**
  - asynchronous, 205
  - signal handler, 62
  - numbers, **210**
  - synchronous, **209**
- signal**, 200, 211, 212, 215, 256, 260
- signal handler, **209**
- signal.h, 210, 212
- signal\_handler**, 211, 212
- signbit**, 75
- SIGSEGV**, 210, 245
- SIGTERM**, 211, 213, 245
- sin**, 74, 74, 75
- sinh**, 74, 75
- SIZE\_MAX**, 22, 24, 25, 31, 40, 42, 99, 161, 271
- size\_t**, 2, 7–10, 13–15, 17–19, 22–28, 30–33, 34, 40, 42, 54, 55, 61, 65–69, 74, 81, 82, 86, 87, 91, 96, 97, 99, 106–110, 112–116, 118, 119, 122, 124, 126–128, **130**, 130, 132–136, 144, 148–154, 156–164, 166, 168, **172**, 189, 192, 198, 202–204, 216, 220, 226, 262, 263, 269, 271, 272
- size\_t**, 11
- skip space**, 204, **206**
- snake case, 98
- snprintf**, 70, 152, 153, 164, 165, 277
- source file, 4
- sprintf**, 148, 152, 166, 277
- sprintf numbers, 152
- sqrt**, 28, **74**, 74, 75
- sscanf**, **153**
- SSE, **177**
- stack, 145
- state, **30**
  - observable, 30
- state machine
  - abstract, 31
- statement, 2
  - selection, 15
- statements, **10**
- static**, **140**
- static\_assert**, 73, 73, 93, 256
- stdalign.h, 129
- stdarg.h, 191, 270
- stdargs.h, 63
- stdatomic.h, 214
- stdbool.h, 16, 44
- stddef.h, 34, 112
- stderr**, 76, 76, 77, 92
- stdin**, 82, 82, 83, 148, **153**, 153, 172, 203, 204, **206**, 219, 222, 261
- stdint.h, 22, 34, 42, 47
- stdio.h, 8, 20, 71, 74
- stdlib.h, 8, 19, 77, 118, 130, 131, 263
- stdnoreturn.h, 64
- stdout**, 70, **76**, 76, 78, 79, 81, 83, 128, 148, 156, 204, 205, **206**, 207
- storage durations, 139
  - allocated, 139
  - automatic, 139
  - static, 139
  - thread, 139
- str**, 53, 97, 157, 181, 182, 263, 264
- strchr**, 54, 55, **87**, 150, 151, 157, 262–265, 277
- strcmp**, 54, 54, 55, 99, 115
- strcoll**, 54, 55, 92
- strcpy**, 53, 54, 55, 115, 156
- strcpy\_s**, 54
- strcspn**, 54, 55, **87**, 151, 154, 277
- stream, 74
- strftime**, 87, 88, **89**, 89, 90, 92
- string literal, 3
- string.h, 53, 87, 262
- stringification, **186**
- strings, 53
- strlen**, 13, 14, **53**, 54, 55, 115, 148, 152, 156, 157, 164, 166, 265
- strncat**, 265
- strncmp**, 265
- strncpy**, 265
- strnlen\_s**, 54
- strpbrk**, 87, 262, 263, 265
- strrchr**, 87, 262, 263, 265
- strspn**, 54, 55, 86, **87**, 151, 154, 157, 265, 277
- strstr**, 87, 157, 262, 263, 265
- strtod**, 19, 20, 64, 70, 72, **85**, 154, 263, 265
- strtodf**, **85**, 265
- strtouimax**, **85**
- strtok**, 87, 265
- strtoul**, **85**, 154, 198, 265
- strtold**, **85**, 265
- strtoll**, **85**, 198, 265
- strtoul**, **85**, 85, 86, 149, 154, 198, 199, 265
- strtoull**, **85**, 149, 150, 198, 227, 265
- strtoumax**, **85**, 265
- structure
  - field, 56, **57**
- strxfrm**, 92
- sugar, 3
- switch**, **20**
- symmetric difference, **43**
- synchronization, **229**
- synchronizes with, 230
- system call, **173**
- tag, 60
- tag name, 60
- tan**, **74**, **75**
- tanh**, **74**, **75**
- task, **217**
- text mode IO, **162**
- tgamma**, **74**, **75**
- tgmath.h, 19, 28, 41, 49, 74, 179, 192, 263
- thrd\_create**, 72, 200, 218, 219, 227, 230
- thrd\_current**, 227
- thrd\_detach**, 227, 228
- thrd\_equal**, 227
- thrd\_exit**, 200, 227, 230
- thrd\_join**, 218–220, 228, 230
- thrd\_sleep**, 228
- thrd\_start\_t**, 218
- thrd\_t**, 218, 219, 226–228

- `thrd_yield`, 228
- thread, 217
- thread specific storage, 221
- `thread_local`, 139, 221, 260, 280
- `threads.h`, 221
- `time`, 34, 59, 87, 87, 88, 89, 89, 102, 256, 260
- `time.h`, 34, 56, 58, 87, 97
- `time_t`, 34, 58, 87, 87–89
- `TIME_UTC`, 89, 174, 178, 224, 260
- `timespec`, 58–60, 87, 88, 89, 97, 99, 102, 112, 175, 178, 222, 224, 225, 228
- `timespec_get`, 87, 88, 89, 89, 173, 174, 178, 179, 224, 260
- `tm`, 56–60, 87–89, 283
- `tm_hour`, 56, 57, 59
- `tm_isdst`, 56, 57, 88
- `tm_mday`, 56–59, 87
- `tm_min`, 56, 57, 59
- `tm_mon`, 56–59, 87
- `tm_sec`, 56, 57, 59, 87
- `tm_wday`, 56, 57, 87, 88
- `tm_yday`, 56–59, 88
- `tm_year`, 56–59, 87
- `TMP_MAX`, 260
- `tmpnam`, 260
- `to`, 187
- `tolower`, 84
- `tooDeep`, 204, 205, 206–208
- `toupper`, 84, 85
- trailing semicolon, 105
- trailing white space, 79
- translation unit, 100
- traps, 209
- `true`, 14, 16, 18–20, 26, 27, 27, 40, 42, 44, 45, 46, 51, 66, 68, 72, 86, 99, 219, 220, 222, 224, 271–273
- `trunc`, 74, 74, 75
- `tss_create`, 221
- `tss_delete`, 221
- `TSS_DTOR_ITERATIONS`, 260
- `tss_dtor_t`, 221
- `tss_get`, 221
- `tss_set`, 221
- `tss_t`, 221, 260
- TU, 100
- `tv_nsec`, 58, 88, 89, 112, 174
- `tv_sec`, 58, 88, 97, 112, 113, 224
- type, 7, 8, 30
  - aggregate, 50
  - derived, 32
  - incomplete, 37
  - narrow, 32
- `UCHAR_MAX`, 42, 122, 123, 123, 239, 240, 276
- `UINT16_C`, 47
- `UINT16_MAX`, 47
- `uint16_t`, 47
- `UINT32_C`, 47
- `UINT32_MAX`, 47
- `uint32_t`, 47, 48
- `UINT64_C`, 47, 48
- `UINT64_MAX`, 47
- `uint64_t`, 47, 174, 178
- `UINT8_C`, 47
- `UINT8_MAX`, 47
- `uint8_t`, 47
- `UINT_MAX`, 42, 42, 45–47, 125
- `uintmax_t`, 34, 81, 85, 154, 248, 254
- `ULLONG_MAX`, 42
- `ULONG_MAX`, 42, 86
- `undef`, 257, 265
- `ungetc`, 219, 222
- Unicode, 157
- uninterruptible operation, 214
- unsigned, 13
- UTF-16, 161
- UTF-32, 161
- UTF-8, 161
- `va_arg`, 191, 192, 270, 278
- `va_copy`, 191
- `va_end`, 191, 192
- `va_list`, 191, 192, 278
- `va_start`, 191, 192
- value, 9, 30, 30
- variable
  - condition, 224
  - global, 105
- variables, 7
- variadic functions, 180
- `vfprintf`, 192
- visible, 8
- VLA, *see* variable length array, 142
- `wchar.h`, 157
- `wchar_t`, 154, 157–161, 263, 264
- `wcs`, 157
- `wcschr`, 264
- `wcsncmp`, 265
- `wcsncpy`, 265
- `wcspbrk`, 265
- `wcsrchr`, 265
- `wcsspn`, 265
- `wcsstr`, 265
- `wcstod`, 265
- `wcstof`, 265
- `wcstok`, 265
- `wcstol`, 265
- `wcstold`, 265
- `wcstoll`, 265
- `wcstoul`, 265
- `wcstoull`, 265
- `wctype`, 256
- `wctype.h`, 158
- `WEOF`, 161
- `while`, 18
- wide character, 157
- wide character strings, 157
- wide characters, 157
- `wmemchr`, 265
- `wmemset`, 265
- word boundary, 127
- wrapper, 66
- `xmm0`, 177
- `xmm1`, 177
- `xmm2`, 177

`xmm3`, 177  
`xmm4`, 177  
`xor`, 23, 43, 299  
`xor_eq`, 23, 43, 299