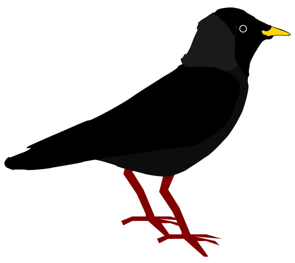


Modern C

Jens Gustedt

INRIA, FRANCE

ICUBE, STRASBOURG, FRANCE

E-mail address: `jens gustedt inria fr`
URL: `http://icube-icps.unistra.fr/index.php/Jens_Gustedt`

This is a preliminary version of this book compiled on March 30, 2015.
It contains feature complete versions of Levels 0, 1 and 2, and the table of contents already gives you a glimpse on what should follow in the two missing ones.

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` (download)

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.

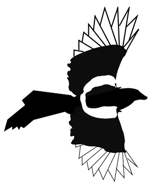
The acquaintance level details most principal concepts and features such as control structures, data types, operators and functions. It should give you a deeper understanding of the things that are going on when you run your programs. This knowledge should be sufficient for an introductory course in 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

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 ovewhelmed 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 182. 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 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 ordinary English phrases. 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**^{C} to more explanatory text, at your own risk.

LISTING 1. A first example of a C program

```
 1 /* This may look like nonsense, but it 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 }
```

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.

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

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

. getting-started.c

```
17      printf("element␣%zu␣is␣%g,␣\tits␣square␣is␣%g\n",
18             i,
19             A[i],
20             A[i]*A[i]);
```

Here, the **printf** **function**^{C} receives four **arguments**^{C}, enclosed in a pair of **parenthesis**^{C}, "`( ... )`":

- The funny-looking text (the blue part) is a so-called **string literal**^{C} that serves as a **format**^{C} for the output. Within the text are three markers (**format specifiers**^{C}), 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**^{C} that start with a backslash, namely `"\t"` and `"\n"`.
- After a comma character we find the word "`i`". The thing that "`i`" stands for will be printed in place of the first format specifier, `"%zu"`.
- Another comma separates the next argument "`A[i]`". The thing that 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**^{C} 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**^{C} or **executable**^{C}. What that translated program looks like is too complicated to explain at this stage. However, for the moment we don't need to understand it, as we have a compiler 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**^{C} on which you will be running your program. There is a simple reason for this: the target binary code is **platform dependent**^{C}, 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
```
> 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.[Exs 1] In the above line

- `c99` is the compiler program.

[Exs 1] Try the compilation command in your terminal.

- `-Wall` tells it to warn us about anything that it finds unusual.
- `-o getting-started` tells it to store the **compiler output**^{C} in a file named `getting-started`.
- `getting-started.c` names the **source file**^{C}, 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**^{C} our newly created **executable**^{C}. Type in:

Terminal
```
> ./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**^{C} 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**^{C} 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:

Terminal
```
> clang -Wall -lm -o getting-started getting-started.c
> gcc -std=c99 -Wall -lm -o getting-started getting-started.c
> 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.[Exs 2]

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**^{C}, something similar to this

Terminal
```
> c99 -Wall -o getting-started-badly getting-started-badly.c
getting-started-badly.c:4:6: warning: return type of ’main’ is not ’in   [-Wmain]
getting-started-badly.c: In function ’main’:
getting-started-badly.c:16:6: warning: implicit declaration of functio  ’printf’ [-Wimplicit-func
getting-started-badly.c:16:6: warning: incompatible implicit declarati   of built-in function ’pr
getting-started-badly.c:22:3: warning: ’return’ with a value, in funct  n returning void [enabled
```

Here we had a lot of long "warning" lines but 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:

[Exs 2] Start a 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 it 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

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

This is a good thing! Its **diagnostic output**^{C} 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.[Exs 3] [Exs 4]

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:

keywords^C**:** In Listing 1 we have used the following keywords or **reserved**^{C} identifiers: **#include**, **int**, **void**, **double**, **for**, and **return**. In our program text, here, they will usually be printed in bold face.

These keywords represent concepts and features that the C language imposes and that cannot be changed.

punctuations^C**:** 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 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.

.

getting-started.c

```
12      [3] = .00007,
```

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^C**:** The construct `/* ... */` that we saw as above tells the compiler that everything inside it is a comment, see *e.g* Line 5.

[Exs 3] 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.

[Exs 4] There is a third difference between the two programs that we didn't mention, yet. Find it.

getting-started.c

```
/* 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^C**:** 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**^{C}.

identifiers^C**:** 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**^{C} (such as `A` and `i`), these are also referred to as **variables**^{C}
- **type**^{C} 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^C**:** 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**^{C}, that is its **declaration**^{C} **int main**(**void**) is followed by a **block**^{C} enclosed in `{ ... }` that describes what that function is supposed to do. In our example this function **definition**^{C} goes from Line 6 to 24. **main** has a special role in all C programs, it must always be present since it is the starting point of the program's execution.

operators^C**:** Of the numerous C operators our program only uses a few:
- `=` for **initialization**^{C} and **assignment**^{C},
- `<` 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**^{C} 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**^{C} that tells it what that identifier is supposed to be. This is where identifiers differ from **keywords**^{C}; 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:

```
int main(void);
double A[5];
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**^{C} **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**^{C}. 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**^{C}, from `0` to `4`.

Each of these declarations starts with a **type**^{C}, 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>`

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

but this is not important for the moment. This information is normally hidden from you in these **include files**^{C} or **header files**^{C}. 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

```
> apropos printf
> man printf
> 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**^{C}) in some part of the program, not everywhere. A **scope**^{C} 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 **??** 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**^{C}. The third type, as used for **main** is called **file scope**^{C}. 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**^{C}.

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:

```
size_t i = 0;
```

is a declaration of `i` that is also a definition with initial **value**^{C} `0`.

`A` is a bit more complex

. getting-started.c

```
  double A[5] = {
    [0] = 9.0,
    [1] = 2.9,
    [4] = 3.E+25,
    [3] = .00007,
  };
```

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**^{C}: 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`.[5]

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

You might have noticed that array positions, **indices**^{C}, 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`*.*

For a function we have a definition (as opposed to only a declaration) if its declaration is followed by braces `{ ... }` containing the code of the function.

[5]We will see later how these number literals with dots `.` and exponents `E+25` work.

```
int main(void) {
  ...
}
```

In our examples so far we have seen two different kinds of objects, **data objects**^{C}, namely `i` and `A`, and **function objects**^{C}, **main** and **printf**.

In contrast to declarations, where several were allowed for the same identifier, definitions must be unique:

Rule 0.2.3.5 *Each object must have exactly one definition.*

This rule concerns data objects as well as function objects.

2.4. Statements. The second part of the **main** function consists mainly of statements. Statements are instructions that tell the compiler what to do with identifiers that have been declared so far. We have

getting-started.c

```
  for (size_t i = 0; i < 5; ++i) {
     printf("element␣%zu␣is␣%g,␣\tits␣square␣is␣%g\n",
            i,
            A[i],
            A[i]*A[i]);
  }

  return EXIT_SUCCESS;
```

We have already discussed the lines that correspond to the call to **printf**. There are also other types of statements: a **for** and a **return** statement, and an increment operation, indicated by the **operator**^{C} `++`.

2.4.1. *Iteration.* The **for** statement tells the compiler that the program should execute the **printf** line a number of times. It is the simplest form of **domain iteration**^{C} that C has to offer. It has four different parts.

The code that is to be repeated is called **loop body**^{C}, it is the `{ ... }` block that follows the **for** `( ... )`. The other three parts are those inside `( ... )` part, divided by semicolons:

(1) The declaration, definition and initialization of the **loop variable**^{C} `i` that we already discussed above. This initialization is executed once before any of the rest of the whole **for** statement.
(2) A **loop condition**^{C}, `i < 5`, that specifies how long the **for** iteration should continue. This one tells the compiler to continue iterating as long as `i` is strictly less than `5`. The loop condition is checked before each execution of the loop body.
(3) Another statement, `++i`, is executed `i` after each iteration. In this case it increases the value of `i` by `1` each time.

If we put all those together, we ask the program to perform the part in the block 5 times, setting the value of `i` to `0`, `1`, `2`, `3`, and `4` respectively in each iteration. The fact that we can identify each iteration with a specific value for `i` makes this an iteration over the **domain**^{C} `0`, ..., `4`. There is more than one way to do this in C, but a **for** is the easiest, cleanest and best tool for the task.

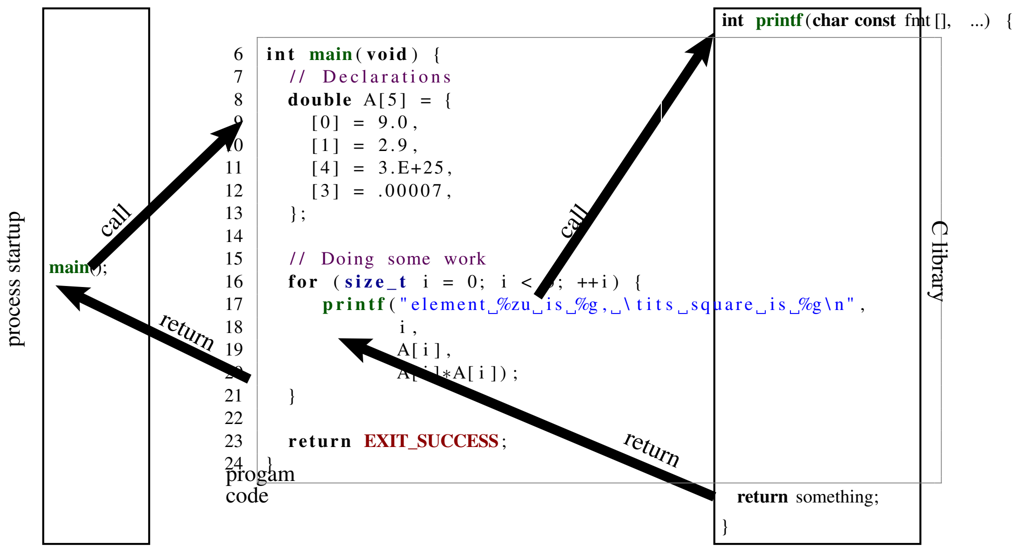
Rule 0.2.4.1 *Domain iterations should be coded with a* **for** *statement.*

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.

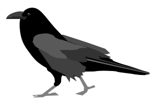


```
 6  int main(void) {
 7    // Declarations
 8    double A[5] = {
 9      [0] = 9.0,
10      [1] = 2.9,
11      [4] = 3.E+25,
12      [3] = .00007,
13    };
14
15    // Doing some work
16    for (size_t i = 0; i < 5; ++i) {
17        printf("element %zu is %g, \tits square is %g\n",
18               i,
19               A[i],
20               A[i]*A[i]);
21    }
22
23    return EXIT_SUCCESS;
24  }
```

FIGURE 1. execution of a small program

In Figure 1 we have a schematic view of the execution of our little program. 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**^{C}, 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**^{C}, 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**^{C} 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

```
char* name;
```

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

```
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

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

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

```
size_t strlen(const char *string);
int main(int argc, char **argv);
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:

```
int main(int argc, char* argv[argc+1]) {
  puts("Hello␣world!");
  if (argc > 1) {
    while (true) {
      puts("some␣programs␣never␣stop");
    }
  } else {
    do {
      puts("but␣this␣one␣does");
    } while (false);
  }
}
```

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**^{C}, 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 20, respectively.

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

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

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**^{C}, and the part in `{ ... }` is called the **dependent block**^{C}.

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:

```
if (i > 25) {
  j = i - 25;
} else {
  j = i;
}
```

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 one of the two **selection statements**^{C}. It selects one of the two possible **code paths**^{C} according to the contents of `( ... )`. The general form is

```
if (condition) statement0-or-block0
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.

```
for (size_t i = 0; i < 5; ++i) {
  if (i) {
    printf("element␣%zu␣is␣%g,␣\tits␣square␣is␣%g\n",
           i,
           A[i],
           A[i]*A[i]);
  }
}
```

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`.[Exs 1]

There are two simple rules for the evaluation a numerical "`condition`":

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

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

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

```
if (i != 0) {
  ...
}
```

as:

```
if (i) {
  ...
}
```

`#include <stdbool.h>`

The type **bool**, specified in `stdbool.h`, is what we should be using to store truth values. Its values are **false** and **true**. Technically, **false** is just another name for `0` and **true** for `1`, but it's important to use **false** and **true** 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:

```
bool b = ...;
...
if ((b != false) == true) {
  ...
}
```

as

```
bool b = ...;
...
if (b) {
  ...
}
```

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:

Rule 1.3.1.4 *All scalars have a truth value.*

Here **scalar**^{C} types include all the numerical types such as **size_t**, **bool** or **int** that we already encountered, and **pointer**^{C} types, that we will come back to in Section 6.2.

[Exs 1] Add the **if** `(i)` condition to the program and compare the output to the previous.

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

```
for (expression-or-declaration; condition; expression)
   statement-or-block
```

This statement is actually quite genereric. Usually "`expression-or-declaration`" is used to state an initial value for the iteration domain. "`condition`" tests if the iteration should continue, "`expression`" updates the iteration variable and "`statement-or-block`" is performed in each iteration. Some advice

- In view of Rule 0.2.4.2 "`expression-or-declaration`" should in most cases be "`declaration`".
- 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:

```
for (size_t i = 10; i; --i) {
  something(i);
}
for (size_t i = 0, stop = upper_bound(); i < stop; ++i) {
  something_else(i);
}
for (size_t i = 9; i <= 9; --i) {
  something_else(i);
}
```

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.[Exs 2]

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**.

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

The following example shows a typical use of the first:

```
#include <tgmath.h>
double const eps = 1E-9;            // desired precision

...
double const a = 34.0;
double x = 0.5;
while (fabs(1.0 - a*x) >= eps) {    // iterate until close
  x *= (2.0 - a*x);                 // Heron approximation
}
```

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

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:

```
  do {                                 // iterate
    x *= (2.0 - a*x);                  // Heron approximation
  } 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:

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

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:

```
for (;;) {
  double prod = a*x;
  if (fabs(1.0 - prod) < eps)        // stop if close enough
     break;
  x *= (2.0 - prod);                 // Heron approximation
}
```

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.

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

In the examples above we made use of a standard macro **fabs**, that comes with the `tgmath.h` header[3]. 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`.[Exs 4][Exs 5][Exs 6]

#include `<stdlib.h>`

LISTING 1.1. A program to compute inverses of numbers

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

/* lower and upper iteration limits centered around 1.0 */
static double const eps1m01 = 1.0 - 0x1P-01;
static double const eps1p01 = 1.0 + 0x1P-01;
static double const eps1m24 = 1.0 - 0x1P-24;
static double const eps1p24 = 1.0 + 0x1P-24;

int main(int argc, char* argv[argc+1]) {
  for (int i = 1; i < argc; ++i) {        // process args
    double const a = strtod(argv[i], 0);  // arg -> double
    double x = 1.0;
    for (;;) {                    // by powers of 2
      double prod = a*x;
      if (prod < eps1m01)         x *= 2.0;
      else if   (eps1p01 < prod) x *= 0.5;
      else break;
    }
    for (;;) {                    // Heron approximation
      double prod = a*x;
      if ((prod < eps1m24) || (eps1p24 < prod))
        x *= (2.0 - prod);
      else break;
    }
    printf("heron:␣a=%.5e,\tx=%.5e,\ta*x=%.12f\n",
           a, x, a*x);
  }
  return EXIT_SUCCESS;
}
```

[3]"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 `eps1m01` and observe the output when you change them slightly.

3.3. Multiple selection. The last control statement that C has to offer is called **switch** statement and, similar to **if**, is also a **selection**^{C}. It is mainly used when cascades of **if**−**else** constructs would be too tedious:

```
if (arg == 'm') {
  puts("this␣is␣a␣magpie");
} else if (arg == 'r') {
  puts("this␣is␣a␣raven");
} else if (arg == 'j') {
  puts("this␣is␣a␣jay");
} else if (arg == 'c') {
  puts("this␣is␣a␣chough");
} else {
  puts("this␣is␣an␣unknown␣corvid");
}
```

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:

```
switch (arg) {
  case 'm': puts("this␣is␣a␣magpie");
            break;
  case 'r': puts("this␣is␣a␣raven");
            break;
  case 'j': puts("this␣is␣a␣jay");
            break;
  case 'c': puts("this␣is␣a␣chough");
            break;
  default: puts("this␣is␣an␣unknown␣corvid");
}
```

#include <stdio.h>

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'` and a **fallback**^{C} case labeled **default**. The default case is triggered if `arg` doesn't match any of the **case** values.[Exs 7]

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

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

and the semantics of it are quite straightforward: the **case** and **default** labels serve as **jump targets**^{C}. 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.

```
switch (count) {
  default:puts("++++␣.....␣+++");
  case 4: puts("++++");
  case 3: puts("+++");
  case 2: puts("++");
  case 1: puts("+");
  case 0:;
```

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

```
}
```

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

```
+++
++
+
```

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.*

4. Expressing computations

We've already made use of some simple examples of **expressions**^{C}. 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 1.4.0.3 *The type* **size_t** *represents values in the range* `[0,` **SIZE_MAX**`]`.

The value of **SIZE_MAX** is quite large, depending on the architecture it should be one of

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

This should be large enough for calculations that are not too sophisticated. The standard header `stdint.h` provides **SIZE_MAX**.

#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**^{C}. The symbols and combinations like `+` or `!=` are called **operators**^{C} and the things to which they are applied are called **operands**^{C}, 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 tables in the appendix; Table 2 lists the operators that operate on values, Table 3 those that operate objects and Table 4 those that operate on types.

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.

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

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**^{C}. 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`.[Exs 8] 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`*.*

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**^{C}. 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.*

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

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**^{C} (right value) and to the object on the left as **lvalue**^{C} (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 known above all have the syntax

```
an_object @= some_expression;
```

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

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

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

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

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**^{C} `++` and the **decrement operator**^{C} `--`:

- `++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 screwed enough write something like

```
a = b = c += ++d;
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**^{C}.

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**^{C} and **postfix decrement**^{C}. 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` according if 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

```
size_t c = (a < b) + (a == b) + (a > b);
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

```
double largeA[N] = { 0 };
...
/*  fill largeA somehow */

size_t sign[2] = { 0, 0 };
for (size_t i = 0; i < N; ++i) {
    sign[(largeA[i] < 1.0)] += 1;
}
```

the array element sign[0] will hold the number of values in largeA that are greater or equal than 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 use the include 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 1. Logical operators

a	**not** a
false	**true**
true	**false**

a **and** b	**false**	**true**
false	**false**	**false**
true	**false**	**true**

a **or** b	**false**	**true**
false	**false**	**true**
true	**true**	**true**

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:

```
double largeA[N] = { 0 };
...
/*  fill largeA somehow */

```

```
size_t isset[2] = { 0, 0 };
for (size_t i = 0; i < N; ++i) {
  isset[!!largeA[i]] += 1;
}
```

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 `0.0` and unequal, respectively.

Operators `&&` and `||` have a particular property that is called **short circuit evaluation**^{C}. 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

```
  if (isgreat(a) && issmall(b))
      ++x;
  if (issmall(c) || issmall(d))
     ++y;
```

then 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

```
  if (isgreat(a))
     if (issmall(b))
        ++x;
  if (issmall(c)) ++y;
     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:

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

#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**^{C}.

```
#include <tgmath.h>

#ifdef __STDC_NO_COMPLEX__
# error "we␣need␣complex␣arithmetic"
#endif

double complex sqrt_real(double x) {
  return (x < 0) ? CMPLX(0, sqrt(-x)) : CMPLX(sqrt(x), 0);
}
```

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 5.5.6.

In the example above we also see conditional compilation that is achieved with **preprocessor directives**^{C}, 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 it 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 if `f(a)` or `g(b)` are 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

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

we wouldn't know which of the 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**^{C} and **data**^{C}.

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 a representation a particular value has should in most cases not be your concern: the compiler is there to organize the translation be 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 something like an **observable state**^{C}, 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 1.5.0.6 *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 really of numbers as 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 idealy that it only depends on the program specification itself.

5.0.2. *Types.* An important step in that direction is the concept of **types**^{C}. 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 1.5.0.7 *All values have a type that is statically determined.*

Rule 1.5.0.8 *A value's type determines its possible operations.*

Rule 1.5.0.9 *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**^{C} of the type.

Rule 1.5.0.10 *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 1.5.0.11 *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 **??**, 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**^{C} that regulates how the program switches from one state to the next. These transitions are determined by value, type and binary representation, only.

Rule 1.5.0.12 (as-if) *Programs execute **as if** they were 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

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

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

```
/* do something without x */
x += 12;
```

or

```
x += 12;
/* 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 allways be consistent with the two seperated 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**^{C}. 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 1.5.0.13 *Type determines optimization opportunities.*

5.1. Basic types. C has a series of basic types and some means of constructing **derived types**^{C} 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 straight forward. 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 1.5.0.6, 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**^{C}, **signed integers**^{C}, **real floating point numbers**^{C} and **complex floating point numbers**^{C}

All these classes contain several types. They differ according to their **precision**^{C}, which determines the valid range of values that are allowed for a particular type. Table 2 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**^{C}. 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.*

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**^{C}. Thus, to chose the "best" type for a given purpose in a

TABLE 2. 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	**_Bool**	**bool**
		unsigned char	
		unsigned short	
		unsigned int	**unsigned**
		unsigned long	
		unsigned long long	
	[un]signed	**char**	
	signed	**signed char**	
		signed short	**short**
		signed int	**signed** or **int**
		signed long	**long**
		signed long long	**long long**
floating point	real	**float**	
		double	
		long double	
	complex	**float _Complex**	**float complex**
		double _Complex	**double complex**
		long double _Complex	**long double complex**

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 5.5.6.

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

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

TABLE 3. Semantic arithmetic types for specialized use cases

uintmax_t	`stdint.h`		maximum width unsigned integer, preprocessor
intmax_t	`stdint.h`		maximum width signed integer, preprocessor
errno_t	`errno.h`	Appendix K	error return instead of **int**
rsize_t	`stddef.h`	Appendix K	size arguments with bounds checking
time_t	`time.h`	**time**`(0)`, **difftime**`(t1, t0)`	calendar time in seconds since epoch
clock_t	`time.h`	**clock**`()`	processor time

The C standard defines a lot of other types, among them other arithmetic types that model special use cases. Table 3 list some of them. The first two represents the type with maximal width that the platform supports.

The second 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**^{C} can be specified:

`123` **decimal integer constant**^{C}. The most natural choice for most of us.

`077` **octal integer constant**^{C}. 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**^{C}. 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**^{C}. 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**^{C}. 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**^{C}. 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**^{C}. 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. An important rule applies:

Rule 1.5.2.1 *Numerical literals are never negative.*

That is if we write something like `-34` or `-1.5E-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 1.5.0.7 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 1.5.2.2 *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 1.5.2.3 *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.[Exs 9]

Rule 1.5.2.4 *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 `0x7FFF` has the value 32767 and thus type **signed**. Other than for the decimal constant, the constant `0x8000` (value 32768 written in hexadecimal) then is an **unsigned** and expression `-0x8000` again is **unsigned**.[Exs 10]

Rule 1.5.2.5 *Don't use octal or hexadecimal constants to express negative values.*

Or if we formulate it postively

Rule 1.5.2.6 *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**.[Exs 11]

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

[Exs 9] 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.

[Exs 10] Show that if in that case the maximum **unsigned** is $2^{16} - 1$ that then `-0x8000` has value 32768, too.

[Exs 11] Show that the expressions `-1U`, `-1UL` and `-1ULL` have the maximum values and type of the three usable unsigned types, respectively.

TABLE 4. 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	**signed**	−2147483647
`2147483648`	+2147483648	**signed long**	−2147483648
`4294967295`	+4294967295	**signed long**	−4294967295
`0x7FFFFFFF`	+2147483647	**signed**	−2147483647
`0x80000000`	+2147483648	**unsigned**	+2147483648
`0xFFFFFFFF`	+4294967295	**unsigned**	+1
`1`	+1	**signed**	−1
`1U`	+1	**unsigned**	+4294967295

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 1.5.2.7 *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 1.5.2.8 *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.

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:

	float	**double**	**long double**
literal	`0.2F`	`0.2`	`0.2L`
value	`0x1.99999AP-3F`	`0x1.999999999999AP-3`	`0xC.CCCCCCCCCCCCCCDP-6L`

Rule 1.5.2.9 *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:

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

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

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

As we have already seen above, arrays that have an **incomplete type**^{C} 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**^{C} `T a = {0}` will almost[12] always do.

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

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**^{C}, 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:

```
char const*const animal[3] = {
  "raven",
  "magpie",
  "jay",
```

[12]The exception are variable length arrays, see Section 6.1.3.

```
};
char const*const pronoun[3] = {
  "we",
  "you",
  "they",
};
char const*const ordinal[3] = {
  "first",
  "second",
  "third",
};
...
for (unsigned i = 0; i < 3; ++i)
    printf("Corvid␣%u␣is␣the␣%s\n", i, animal[i]);
...
for (unsigned i = 0; i < 3; ++i)
    printf("%s␣pural␣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**^{C} 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 be introduced much later than string literals to the C language, and therefore remained as it is for backward compatibility.[13]

[13]A third class of read-only objects exist, temporary objects. We will see them later in Section 13.2.2.

5.4.2. *Enumerations.* C has a simple mechanism for such cases as we see them in the example, namely **enumerations**^{C}:

```
enum corvid { magpie, raven, jay, corvid_num, };
char const*const animal[corvid_num] = {
  [raven] = "raven",
  [magpie] = "magpie",
  [jay] = "jay",
};
...
for (unsigned i = 0; i < corvid_num; ++i)
    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` 1, `jay` 2 and `corvid_num` 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`:

```
enum corvid { magpie, raven, jay, chough, corvid_num, };
char const*const animal[corvid_num] = {
  [chough] = "chough",
  [raven] = "raven",
  [magpie] = "magpie",
  [jay] = "jay",
};
```

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**.

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**^{C} for the type name:

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

To define these constants we can use **integer constant expressions**^{C}, 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.

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

Here, `o52` is an object, **const**-qualified but still, so the expression for `c52` 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.[14]

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**^{C}. A macro is introduced by a **#define** **preprocessor**^{C} declaration:

```
# define M_PI 3.14159265358979323846
```

This declaration has an effect that the identifier `M_PI` is replaced in the then following program code by the **double** constant. Such a declaration 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.

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**^{C} on the replacement side of the macro. Such a compound literal has the form

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

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

```
# define CORVID_NAME /**/          \
(char const*const[corvid_num]){    \
  [chough] = "chough",             \
  [raven] = "raven",               \
  [magpie] = "magpie",             \
  [jay] = "jay",                   \
}
```

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

[14]We will handle the later two concepts in Sections 12.6 and 12.1.

```
for (unsigned i = 0; i < corvid_num; ++i)
    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 aren't suitable for ICE.
- For our purpose here to declare "named constants" the type `T` should be **const-qualified**^{C}. 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 definitively leave **I** alone.

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 representions.

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.*

It mathematical terms they implement a ring, $\mathbb{Z}_N$, the set of integers modulo some number N. The values that are representable are $0, \ldots, N-1$. 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
`#include <limits.h>` **UINT_MAX**, **ULONG_MAX** and **ULLONG_MAX** and they are provided through `limits.h`.
`#include <stdint.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 $b_0, b_1, \ldots, b_{p-1}$ called **bits**^{C}. Each of the bits has a value of 0 or 1. The value of such a number is computed as

$$\sum_{i=0}^{p-1} b_i 2^i. \tag{1}$$

The value p in that binary representation is called the **precision**^{C} of the underlying type. Of the bits b_i that are 1 the one with minimal index i is called the **least significant bit**^{C}, LSB, the one with the highest index is the **most significant bit**^{C}, 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.

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

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 $V = \{0, \ldots, p-1\}$ 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 `^`. 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`	1111111111111111		$\{0,1,2,3,4,5,6,7,8,9,10,$ $11,12,13,14,15\}$
`A`	240	`0x00F0`	0000000011110000		$\{4,5,6,7\}$
`~A`	65295	`0xFF0F`	1111111100001111	$V \setminus A$	$\{0,1,2,3,8,9,10,$ $11,12,13,14,15\}$
`-A`	65296	`0xFF10`	1111111100010000		$\{4,8,9,10,$ $11,12,13,14,15\}$
`B`	287	`0x011F`	0000000100011111		$\{0,1,2,3,4,8\}$
`A|B`	511	`0x01FF`	0000000111111111	$A \cup B$	$\{0,1,2,3,4,5,6,7,8\}$
`A&B`	16	`0x0010`	0000000000010000	$A \cap B$	$\{4\}$
`A^B`	495	`0x01EF`	0000000111101111	$A \Delta B$	$\{0,1,2,3,5,6,7,8\}$

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 `^=`, respectively.[Exs 15][Exs 16][Exs 17][Exs 18]

There is yet another operator that operates on the bits of the value, the complement operator `~`. The complement `~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.[Exs 19][Exs 20]

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.

```c
enum corvid { magpie, raven, jay, chough, corvid_num, };
# define FLOCK_MAGPIE  1U
# define FLOCK_RAVEN 2U
# define FLOCK_JAY     4U
# define FLOCK_CHOUGH  8U
# define FLOCK_EMPTY   0U
# define FLOCK_FULL   15U

int main(void) {
  unsigned flock = FLOCK_EMPTY;

  ...

  if (something) flock |= FLOCK_JAY;

  ...

  if (flock&FLOCK_CHOUGH)
    do_something_chough_specific(flock);

}
```

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,

[Exs 15] Show that $A \setminus B$ can be computed by `A - (A&B)`

[Exs 16] Show that `V + 1` is 0.

[Exs 17] Show that `A^B` is equivalent to `(A - (A&B)) + (B - (A&B))` and `A + B - 2*(A&B)`

[Exs 18] Show that `A|B` is equivalent to `A + B - (A&B)`

[Exs 19] Show that `~B` can be computed by `V - B`

[Exs 20] Show that `-B = ~B + 1`

`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.[Exs 21]

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

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

```
# define FLOCK_MAGPIE  (1U << magpie)
# define FLOCK_RAVEN (1U << raven)
# define FLOCK_JAY      (1U << jay)
# define FLOCK_CHOUGH  (1U << chough)
# define FLOCK_EMPTY     0U
# 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 later to make the semantics of the types and values that you are using clear.

#include `<stdbool.h>`

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 a 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 still many experienced C programmers don't have taken the habit of using 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 2 **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**^{C}. 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**^{C}

[Exs 21] Show that the bits that are "lost" in an operation `x>>n` correspond to the remainder `x % (1ULL << n)`.

- **sign and magnitude**^{C}
- **ones' complement**^{C}
- **two's complement**^{C}

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 simple 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 need to interpret unsigned values as signed ones:

```
bool is_negative(unsigned a) {
  unsigned const int_max = UINT_MAX/2;
  return a > int_max;
}
bool is_signed_less(unsigned a, unsigned b) {
  if (is_negative(b) && !is_negative(a)) return false;
  else return a < b;
}
```

When realize 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**^{C}. The following two loops look quite the same:

```
  for (unsigned i = 1; i; ++i) do_something();
  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 1.5.0.12 allows it to optimize the second loop to

```
  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 never will 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**.[Exs 22]

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.

```
char const* sign_rep[4] =
  {
    [1] = "sign␣and␣magnitude",
    [2] = "ones'␣complement",
    [3] = "two's␣complement",
    [0] = "weird",
  };
enum { sign_magic = -1&3, };
...
printf("Sign␣representation:␣%s.\n", sign_rep[sign_magic]);
```

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

Rule 1.5.5.12 *Use unsigned types for bit operations.*

5.5.6. *Floating point data.* Where integers come near the mathematical concepts of ℕ (unsigned) or ℤ (signed), floating point types are close to ℝ (non-complex) or 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 then `0.0`; the smallest negative **double** value is −**DBL_MAX**.

#include <float.h>

But real numbers (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 π which is "transcendent" and so has an endless expansion in any representation, and which even doesn't repeat in any way.

[Exs 22] Show that **INT_MIN**+**INT_MAX** is -1.

C and other programming languages deal with these difficulties by cutting of 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[23] a floating point value is computed from the following values:

s	sign (± 1)
e	exponent, an integer
$f_1, \ldots, f_p$	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, $emin$ and $emax$ 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 $f_2 = 1$, its value is

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

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.[Exs 24]

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

So basically they loose 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.[Exs 25] 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. We are only able to tell that they are “close”:

Rule 1.5.5.14 *Never compare floating point values for equality.*

The representation of the complex types is straight forward, 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 == `**creal**`(z)+ `**cimag**`(z)*`**I**.[26]

#include `<tgmath.h>`

[23]see Section **??** for the complete view

[Exs 24] Show that all representable floating point values with $e > p$ are multiples of 2^{e-p}.

[Exs 25] Print the results of the following expressions: `1.0E-13 + 1.0E-13` and `(1.0E-13 + (1.0E-13 + 1.0))- 1.0`

[26]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 introduce in Section 12.2. Here, for this level, we will introduce **aggregate data types**^{C}, 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 that confuse 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:

```
double a[16];
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**^{C}. 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:

```
double C[M][N];
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 37 and 44 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 also ruled out also with arrays as object operands, either because they assume arithmetic or because they have a second value operand 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 `[]`[27]. 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**^{C}, FLA, and **variable length arrays**^{C}, 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:

[27]The real C jargon story about arrays and `[]` is a bit more complicated. Let us apply the **as-if** Rule 1.5.0.12 to our explanation. All C program behaves as if the `[]` are directly applied to an array object.

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

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[28] 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* `(`**sizeof** `A)/(`**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**^{C}.

Take the following as an example:

```c
#include <stdio.h>
void swap_double(double a[static 2]) {
  double tmp = a[0];
  a[0] = a[1];
  a[1] = tmp;
}
int main(void) {
  double A[2] = { 1.0, 2.0, };
  swap_double(A);
  printf("A[0]␣=␣%g,␣A[1]␣=␣%g\n");
}
```

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`.

[28] 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**^{C}.

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:

```
char chough0[] = "chough";
char chough1[] = { "chough" };
char chough2[] = { 'c', 'h', 'o', 'u', 'g', 'h', 0, };
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

```
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**^{C}. 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 scripts.

The wast majority of platforms nowadays uses the so-called ASCII[29] 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 use 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.

[29] American 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 introduce with C11. There are **strnlen_s** and **strcpy_s** that additionally deal with the maximum length of their string parameters.[Exs 30]

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

LISTING 1.2. copying a string

```c
#include <string.h>
#include <stdio.h>
int main(int argc, char* argv[argc+1]) {
  size_t const len = strlen(argv[0]); // compute the length
  char name[len+1];                   // create VLA
                                      // ensure place for 0
  memcpy(name, argv[0], len);         // copy the name
  name[len] = 0;                      // ensure 0 character
  if (!strcmp(name, argv[0])) {
    printf("program␣name␣\"%s\"␣successfully␣copied\n",
           name);
  } else {
    printf("coying␣%s␣leads␣to␣different␣string␣%s\n",
           argv[0], name);
  }
}
```

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

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

[Exs 30] Use **memchr** and **memcmp** to implement a bounds checking version of **strcmp**.

```
signed strcoll(char const s0[static 1], char const s1[static 1]);
char* strchr(const char s[static 1], int c);
size_t strspn(const char s1[static 1], const char s2[static 1]);
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:

```
void* memcpy(void* target, void const* source, size_t len);
signed memcmp(void const* s0, void const* s1, size_t len);
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**^{C}. 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 37. 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**^{C}, 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 a 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:

```
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>`

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

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

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

is simple:

yday.c
```
  struct tm today = {
    .tm_year = 2014,
    .tm_mon  = 2,
    .tm_mday = 29,
    .tm_hour = 16,
    .tm_min  = 7,
    .tm_sec  = 5,
  };
```

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: by using the name of the field preceded with a dot `.` suffices to specify were the corresponding data should go.

Accessing the fields of the structure is as simple and has similar "`.`" syntax:

yday.c
```
  printf("this␣year␣is␣%d,␣next␣year␣will␣be␣%d\n",
         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

yday.c
```
struct tm time_set_yday(struct tm t) {
  // tm_mdays starts at 1
  t.tm_yday += DAYS_BEFORE[t.tm_mon] + t.tm_mday - 1;
  // take care of leap years
  if ((t.tm_mon > 1) && leapyear(t.tm_year))
```

```
    ++t.tm_yday;
  return t;
}
```

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, `t`, 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.

. yday.c

```
  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.

#include `<time.h>`

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 such an ancient **struct** 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:

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

So here we see the opaque type **time_t** that we already know from Table 3 for the seconds, and a **long** for the nanoseconds.[31] 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:

. timespec.c

```

double timespec_diff(struct timespec a, struct timespec b){
  double ret = a.tv_sec - b.tv_sec;   // tv_sec can be used
                                      // for arithmetic
  ret += (a.tv_nsec - b.tv_nsec) * 1E-9;
  return ret;
```

[31]Unfortunately even the semantic of **time_t** is different, here. In particular, **tv_sec** may be used in arithmetic.

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

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

int leapyear(unsigned year) {
  /*all years that are divisible by 4 are leap years,
    unless they start a new century, provided they
    don't start a new millennium.  */
  return !(year % 4) && ((year % 100) || !(year % 1000));
}

#define DAYS_BEFORE                                  \
(int const[12]){                                     \
  [0] = 0, [1] = 31, [2] = 59, [3] = 90,             \
  [4] = 120, [5] = 151, [6] = 181, [7] = 212,        \
  [8] = 243, [9] = 273, [10] = 304, [11] = 334,      \
}

struct tm time_set_yday(struct tm t) {
  // tm_mdays starts at 1
  t.tm_yday += DAYS_BEFORE[t.tm_mon] + t.tm_mday - 1;
  // take care of leap years
  if ((t.tm_mon > 1) && leapyear(t.tm_year))
    ++t.tm_yday;
  return t;
}

int main(void) {
  struct tm today = {
    .tm_year = 2014,
    .tm_mon  = 2,
    .tm_mday = 29,
    .tm_hour = 16,
    .tm_min  = 7,
    .tm_sec  = 5,
  };
  printf("this␣year␣is␣%d,␣next␣year␣will␣be␣%d\n",
         today.tm_year, today.tm_year+1);
  today = time_set_yday(today);
  printf("day␣of␣the␣year␣is␣%d\n", today.tm_yday);
}
```

Whereas the function **difftime** is part of the C standard, this functionality here is very simple and isn't based on platform specific implementation. So it can easily be implemented by anyone who needs it. A more complete example for **struct** timespec will be given later on Level 2 on page 161.

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:

```
struct person {
  char name[256];
  struct stardate {
    struct tm date;
    struct timespec precision;
  } bdate;
};
```

The visibility of declaration **struct** `stardate` is the same as for **struct** `person`, there is no “namescope” 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.

```
struct stardate {
  struct tm date;
  struct timespec precision;
};
struct person {
  char name[256];
  struct stardate bdate;
};
```

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, structures not only introduce 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**^{C} that we used in the structure declaration:

```
  typedef struct animalStruct animalStructure;
  typedef struct animalStruct animalStruct;
```

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

```
typedef struct animalStruct animalStruct;
struct animalStruct {
  ...
};
```

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**^{C} is always valid, a so-called **forward declaration**^{C} 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:

```c
typedef double vector[64];
typedef vector vecvec[16];
vecvec A;
typedef double matrix[16][64];
matrix B;
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 15.1.
- **setjmp** and **longjmp** can be used to return unconditionally to a calling context, see Section 18.2.

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. The 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**^{C}, 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:

```
extern double fbar(double x);

...
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**^{C} that comes with the header `stdargs.h`.

`#include <stdargs.h>`

We will see later (**??**) 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 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:

```
int main(void);
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

```
_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 **noreturn**, that comes with the header `stdnoreturn.h`.

`#include <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 communicated 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 $i = 0, \ldots, argc$ 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:

$$\gcd(a, b) = \max\{c \in \mathbb{N} \mid c|a \text{ and } c|b\}$$

If we also assume that $a < b$, we can easily see that two recursive formula hold:

$$\gcd(a, b) = \gcd(a, b - a) \tag{2}$$

$$\gcd(a, b) = \gcd(a, b\%a) \tag{3}$$

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>`

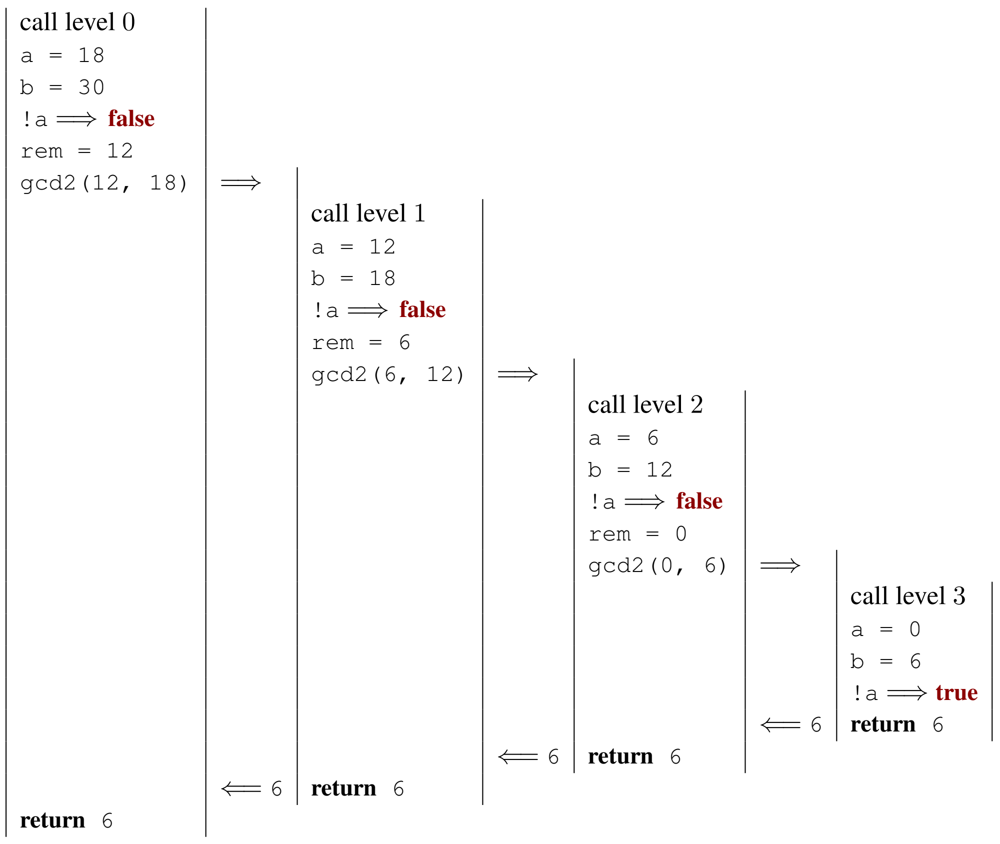
Rule 1.7.3.1 *Make all preconditions for a function explicit.*

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 5. 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 better shouldn't try this.

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 5 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**^{C}:

euclid.h
```
size_t gcd(size_t a, size_t b) {
  assert(a);
  assert(b);
  if (a < b)
    return gcd2(a, b);
  else
    return gcd2(b, a);
}
```

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 Fibonnacci 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

$$F_1 = 1 \tag{4}$$

$$F_2 = 1 \tag{5}$$

$$F_i = F_{i-1} + F_{i-2} \qquad \text{for all } i > 3 \tag{6}$$

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

$$\varphi = \frac{1+\sqrt{5}}{2} = 1.61803... \tag{7}$$

asymptotically we have

$$F_n = \frac{\varphi^n}{\sqrt{5}} \tag{8}$$

So the growth of F_n is exponential.

The recursive mathematical definition can be translated straight forward into a C function:

.

fibonacci.c

```c
size_t fib(size_t n) {
  if (n < 3)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

Here, again, we first check for the termination condition, namely if the argument to the call, `n`, is less then `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 6 shows an example of a call to `fig` 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. [Exs 32]

Implemented like that, the computation of the Fibonacci numbers is quite slow. [Exs 33] 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:

$$T_{\texttt{fib(1)}} = C_0 \tag{9}$$

$$T_{\texttt{fib(2)}} = C_0 \tag{10}$$

$$T_{\texttt{fib(i)}} = T_{\texttt{fib(i-1)}} + T_{\texttt{fib(i-2)}} + C_1 \qquad \text{for all } i > 3 \tag{11}$$

where C_0 and C_1 are some constants that depend on the platform.

[Exs 32] Show that a call `fib(n)` induces F_n leaf calls.

[Exs 33] 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 6. Recursive call `fib(4)`

call level 0				
`n = 4`				
`n<3` $\Longrightarrow$ **false**				
`fib(3)`	$\Longrightarrow$			
		call level 1		
		`n=3`		
		`n<3` $\Longrightarrow$ **false**		
		`fib(2)`	$\Longrightarrow$	
				call level 2
				`n=2`
				`n<3` $\Longrightarrow$ **true**
			$\Longleftarrow$ `1`	**return** `1`
		`fib(1)`	$\Longrightarrow$	
				call level 2
				`n=1`
				`n<3` $\Longrightarrow$ **true**
			$\Longleftarrow$ `1`	**return** `1`
	$\Longleftarrow$ `2`	**return** `1 + 1`		
`fib(2)`	$\Longrightarrow$			
		call level 1		
		`n=2`		
		`n<3` $\Longrightarrow$ **true**		
	$\Longleftarrow$ `1`	**return** `1`		
return `2 + 1`				

It follows that regardless of the platform and our cleverness of the implementation the execution time of the function will always be something like

$$T_{\texttt{fib(i)}} = F_i(C_0 + C_1) \approx \varphi^n \cdot \frac{C_0 + C_1}{\sqrt{5}} = \varphi^n \cdot C_2 \tag{12}$$

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 6, 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

```c
/* Compute Fibonacci number n with help of a cache that may
   hold previously computed values. */
size_t fibCacheRec(size_t n, size_t cache[n]) {
  if (!cache[n-1]) {
    cache[n-1]
      = fibCacheRec(n-1, cache) + fibCacheRec(n-2, cache);
  }
  return cache[n-1];
}
```

```c

size_t fibCache(size_t n) {
  if (n+1 <= 3) return 1;
  /* Set up a VLA to cache the values. */
  size_t cache[n];
  /* A VLA must be initialized by assignment. */
  cache[0] = 1; cache[1] = 1;
  for (size_t i = 2; i < n; ++i)
    cache[i] = 0;
  /* Call the recursive function. */
  return fibCacheRec(n, cache);
}
```

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

$$T_{\texttt{fibCache(n)}} = n \cdot C_3 \qquad (13)$$

for some platform dependent parameter C_3.[Exs 34] 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: [Exs 35]

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

```c
void fib2rec(size_t n, size_t buf[2]) {
  if (n > 2) {
    size_t res = buf[0] + buf[1];
    buf[1] = buf[0];
    buf[0] = res;
    fib2rec(n-1, buf);
  }
}

size_t fib2(size_t n) {
  size_t res[2] = { 1, 1, };
  fib2rec(n, res);
  return res[0];
}
```

Whether or not this is really faster, and how to prove that this version is still correct, is left as an exercise.[Exs 36] [Exs 37]

[Exs 34] Show Equation (13).
[Exs 35] Measure times for calls `fibCache(n)` with the same values as for `fib`.
[Exs 36] Measure times for calls `fib2(n)` with the same values as for `fib`.
[Exs 37] 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 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.[Exs 38]

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 7 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 chose 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**^{C}.

```
#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.

[Exs 38] Write a function `my_strtod` that implements the functionality of **strtod** for decimal floating point constants.

TABLE 7. C library headers

`<assert.h>`	assert run time conditions	8.6
`<complex.h>`	complex numbers	**??**
`<ctype.h>`	character classification and conversion	
`<errno.h>`	error codes	
`<fenv.h>`	floating-point environment.	
`<float.h>`	properties of floating point types	5.5
`<inttypes.h>`	exact width integer types	**??**
`<iso646.h>`	alternative spellings for operators	4.1
`<limits.h>`	properties of integer types	5.0.3
`<locale.h>`	internationalization	
`<math.h>`	type specific mathematical functions	8.1
`<setjmp.h>`	non-local jumps	18.2
`<signal.h>`	signal handling functions	
`<stdalign.h>`	alignment of objects	
`<stdarg.h>`	functions with varying number of arguments	
`<stdatomic.h>`	atomic operations	
`<stdbool.h>`	Booleans	3.1
`<stddef.h>`	basic types and macros	
`<stdint.h>`	exact width integer types	**??**
`<stdio.h>`	input and output	8.2
`<stdlib.h>`	basic functions	
`<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	
`<wchar.h>`	wide string	
`<wctype.h>`	wide character classification and conversion	

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 8 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

```
if (puts("hello world") == EOF) {
  perror("can't output to terminal:");
  exit(EXIT_FAILURE);
}
```

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>`

TABLE 8. 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	`!value`	other values are valid	**fopen**
special error code	`value == code`	other values are valid	**puts**, **clock**, **mktime**, **strtod**, **fclose**
non-zero value	`value`	value otherwise unneeded	**fgetpos**, **fsetpos**
special sucess code	`value != code`	case distinction for failure condition	**thrd_create**
negative value	`value < 0`	positive value is a “counter”	**printf**

Rule 1.8.0.7 *Failure is always an option.*

Rule 1.8.0.8 *Check the return value of library functions for errors.*

An immediate failure of the program is often the best way to ensure that bugs are detected and get fixed in early development.

Rule 1.8.0.9 *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.

```
void puts_safe(char const s[static 1]) {
  static bool failed = false;
  if (!failed && puts(s) == EOF) {
    perror("can't␣output␣to␣terminal:");
    failed = true;
    errno = 0;
  }
}
```

Bounds-checking interfaces. Many of the functions in the C library are vulnerable to **buffer overflow**^{C} if they are called with an inconsistent set of parameters. This lead (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 theses 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:

```
#if !__STDC_LIB_EXT1__
# error "This␣code␣needs␣bounds␣checking␣interface␣Annex␣K"
#endif
#define __STDC_WANT_LIB_EXT1__ 1
```

```

#include <stdio.h>

/* Use printf_s from here on. */
```

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 1.8.0.10 *Identifier names terminating with `_s` are reserved.*

If such a function encounters an inconsistency, a **runtime constraint violation**^{C}, 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 1.8.0.11 *Missed preconditions for the execution platform must abort compilation.*

The classical tool to achieve this are so-called **preprocessor conditionals**^{C} as we have seen them above:

```
#if !__STDC_LIB_EXT1__
# error "This code needs bounds checking interface Annex K"
#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.[Exs 39]

Rule 1.8.0.12 *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 1.8.0.13 *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>`

```
#include <assert.h>
static_assert(sizeof(double) == sizeof(long double),
  "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` for its argument and for which the return value then is of that same type.

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

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 9 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.

`#include <stdio.h>`

8.2.1. *Unformatted output of text.* There is even a more basic function than that, namely **putchar**, that just outputs one single character. The interfaces of this two later functions are as follows:

```
int putchar(int c);
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:

```
int puts_manually(char const s[static 1]) {
  for (size_t i = 0; s[i]; ++i) {
    if (putchar(s[i]) == EOF) return EOF;
  }
  if (putchar('\n') == EOF) return EOF;
  return 0;
}
```

Just take this as example, this is most probably less efficient than the **puts** that your platform provides.

Up to know we only have seen how we can output to the terminal. Often you'd want to write results to some permanent storage, the type **FILE*** for **streams**^{C} provides with an abstraction for this. There are two functions, **fputs** and **fputc**, that generalize the idea of unformatted output to streams.

```
int fputc(int c, FILE* stream);
int fputs(char const s[static 1], FILE* stream);
```

[Exs 39] Write a preprocessor condition that tests if **int** has two's complement sign representation.

TABLE 9. Mathematical functions. Type generic macros are printed in red, real functions in green.

abs, **labs**, **llabs**	$\lvert x\rvert$ for integers
acosh	hyperbolic arc cosine
acos	arc cosine
asinh	hyperbolic arc sine
asin	arc sine
atan2	arc tangent, two arguments
atanh	hyperbolic arc tangent
atan	arc tangent
cbrt	$\sqrt[3]{x}$
ceil	$\lceil x\rceil$
copysign	copy the sign from y to x
cosh	hyperbolic cosine
cos	cosine function, $\cos x$
div, **ldiv**, **lldiv**	quotient and remainder of integer division
erfc	complementary error function, $1 - \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$
erf	error function, $\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$
exp2	2^x
expm1	$e^x - 1$
exp	e^x
fabs	$\lvert x\rvert$ for floating point
fdim	positive difference
floor	$\lfloor x\rfloor$
fmax	floating point maximum
fma	$x \cdot y + z$
fmin	floating point minimum
fmod	remainder of the floating point division
fpclassify	classify floating point value
frexp	significand and exponent
hypot	$\sqrt{x^2 + y^2}$
ilogb	$\lfloor\log_{\texttt{FLT\_RADIX}} x\rfloor$ as integer
isfinite	check if finite
isinf	check if infinite
isnan	check if NaN
isnormal	checks if normal
ldexp	$x \cdot 2^y$
lgamma	$\log_e \Gamma(x)$
log10	$\log_{10} x$
log1p	$\log_e 1 + x$
log2	$\log_2 x$
logb	$\lfloor\log_{\texttt{FLT\_RADIX}} x\rfloor$ as floating point
log	$\log_e x$
modf, **modff**, **modfl**	integer and fractional parts
nan, **nanf**, **nanl**	not-a-number, NaN, of corresponding type
nearbyint	nearest integer using current rounding mode
nextafter, **nexttoward**	next representable floating point value
pow	x^y
remainder	signed remainder of division
remquo	signed remainder and the last bits of the division
rint, **lrint**, **llrint**	nearest integer using current rounding mode
round, **lround**, **llround**	`sign(x)`$\cdot\lfloor\lvert x\rvert + 0.5\rfloor$
scalbn, **scalbln**	$x \cdot$ **FLT_RADIX**^{y}
signbit	checks if negative
sinh	hyperbolic sine
sin	sine function, $\sin x$
sqrt	$\sqrt[2]{x}$
tanh	hyperbolic tangent
tan	tangent function, $\tan x$
tgamma	gamma function, $\Gamma(x)$
trunc	`sign(x)`$\cdot\lfloor\lvert x\rvert\rfloor$

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 a pointer, the identifier **FILE** itself is a so-called **opaque type**^{C}, for which 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:

```
int putchar_manually(int c) {
  return fputc(c, stdout);
}
int puts_manually(char const s[static 1]) {
  if (fputs(s[i], stdout) == EOF) return EOF;
  if (fputc('\n', stdout) == EOF) return EOF;
  return 0;
}
```

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**.

```
FILE* fopen(char const path[static 1], char const mode[static 1]);
FILE* freopen(char const path[static 1], char const mode[static 1], FILE *stream);
```

This can be used as simple as here:

```
int main(int argc, char* argv[argc+1]) {
 FILE* logfile = fopen("mylog.txt", "a");
 if (!logfile) {
   perror("fopen␣failed");
   return EXIT_FAILURE;
 }
 fputs("feeling␣fine␣today\n", logfile);
 return EXIT_SUCCESS;
}
```

TABLE 10. 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 11 for all valid combinations.

mode	memo		file status after **fopen**
`'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**^{C} 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 1.8.0.8 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.

```
fputs("fopen␣failed:␣some-diagnostic\n", stderr);
```

This "some-diagnostic" might (but hasn't 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.

```
errno_t fopen_s(FILE* restrict streamptr[restrict],
                char const filename[restrict], char const mode[
                    restrict]);
errno_t freopen_s(FILE* restrict newstreamptr[restrict],
                  char const filename[restrict], char const mode[
                      restrict]
                  FILE* restrict stream);
```

There are different modes to open a file, `"a"` is only one of several possibilities. Table 10 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 11 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 11. Mode strings for **fopen** and **freopen**. These are the valid combinations of the characters in Table 10.

`"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"` `"wb"` `"a+b"` `"ab+"` `"r+b"` `"rb+"` `"w+b"` `"wb+"`	same as above but for binary file instead of text file
`"wx"` `"w+x"` `"wbx"` `"w+bx"` `"wb+x"`	same as above but error if file exists prior to call

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**.

```
int fclose(FILE* fp);
int fflush(FILE* stream);
```

The primary uses for **freopen** and **fclose** are straight forward: **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

```
int main(int argc, char* argv[argc+1]) {
  if (!freopen("mylog.txt", "a", stdout)) {
    perror("freopen␣failed");
    return EXIT_FAILURE;
  }
  puts("feeling␣fine␣today");
  return EXIT_SUCCESS;
}
```

8.2.3. *Text IO.* Output to text streams is usually **buffered**^{C}, 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**^{C} 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 destination. Listing 1.4 shows an example that writes 10 dots to **stdout** with a delay of approximately one second between all writes.[Exs 40]

[Exs 40] Observe the behavior of the program by running it with 0, 1 and 2 command line arguments.

LISTING 1.4. flushing buffered output

```c
#include <stdio.h>

/* delay execution with some crude code,
   should use thrd_sleep, once we have that */
void delay(double secs) {
  double const magic = 4E8;  // works just on my machine
  unsigned long long const nano = secs * magic;
  for (unsigned long volatile count = 0;
       count < nano;
       ++count) {
    /* nothing here */
  }
}

int main(int argc, char* argv[argc+1]) {
  fputs("waiting␣10␣seconds␣for␣you␣to␣stop␣me", stdout);
  if (argc < 3) fflush(stdout);
  for (unsigned i = 0; i < 10; ++i) {
    fputc('.', stdout);
    if (argc < 2) fflush(stdout);
    delay(1.0);
  }
  fputs("\n", stdout);
  fputs("You␣did␣ignore␣me,␣so␣bye␣bye\n", stdout);
}
```

The most common form of IO buffering for text files in **line buffering**^{C}. In that mode, output is only physically written if the end of a text line is encoutered. 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**^{C} 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.*

The C library additionally also has very limited support to manipulate files within the file system:

TABLE 12. 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 13. 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	**void*** pointer

```
int remove(char const pathname[static 1]);
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:

```
int printf(char const format[static 1], ...);
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 12 for details.

The specifier is not optional and selects the type of output conversion that is performed. See Table 13 for an overview.

The modifier part is important to specify the exact type of the corresponding argument. Table 14 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 15.

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

TABLE 14. Format modifiers for **printf** and similar functions. **float** arguments are first converted to **double**.

character	type	conversion
"hh"	**char** types	integer
"h"	**short** types	integer
""	**signed**, **unsigned**	integer
"l"	**long** integer types	integer
"ll"	**long long** integer types	integer
"j"	**intmax_t**, **uintmax_t**	integer
"z"	**size_t**	integer
"t"	**ptrdiff_t**	integer
"L"	**long double**	floating point

TABLE 15. Format flags for **printf** and similar functions.

character	meaning	conversion
"#"	alternate form, e.g prefix 0x	"aAeEfFgGoxX"
"0"	zero padding	numeric
"-"	left adjustment	any
"␣"	' ' for positive values '-' for negative	signed
"+"	'+' for positive values '-' for negative	signed

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.

```
int printf_s(char const format[restrict], ...);
int fprintf_s(FILE *restrict stream,
              char const format[restrict], ...);
```

Here is a modified example for the re-opening of **stdout**.

```
int main(int argc, char* argv[argc+1]) {
  int ret = EXIT_FAILURE;
  fprintf_s(stderr, "freopen␣of␣%s:", argv[1]);
  if (freopen(argv[1], "a", stdout)) {
    ret = EXIT_SUCCESS;
    puts("feeling␣fine␣today");
  }
  perror(0);
  return ret;
}
```

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**.

```
int fgetc(FILE* stream);
char* fgets(char s[restrict], int n, FILE* restrict stream);
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.

```
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**

```
char* fgets_manually(char s[restrict], int n,
                     FILE*restrict stream) {
  if (!stream) return 0;
  if (!n) return s;
  /* Read at most n-1 characters */
  for (size_t pos = 0; pos < n-1; ++pos) {
    int val = fgetc(stream);
    switch (val) {
      /* EOF signals end-of-file or error */
      case EOF: if (feof(stream)) {
        s[i] = 0;
        /* has been a valid call */
        return s;
      } else {
        /* we are on error */
        return 0;
      }
      /* stop at end-of-line */
      case '\n': s[i] = val; s[i+1] = 0; return s;
      /* otherwise just assign and continue */
      default: s[i] = val;
    }
  }
  s[n-1] = 0;
  return s;
}
```

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

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

enum { buf_max = 32, };

int main(int argc, char* argv[argc+1]) {
  int ret = EXIT_FAILURE;
  char buffer[buf_max] = { 0 };
  for (int i = 1; i < argc; ++i) {        // process args
    FILE* instream = fopen(argv[i], "r"); // as file names
    if (instream) {
      while (fgets(buffer, buf_max, instream)) {
        fputs(buffer, stdout);
      }
      fclose(instream);
      ret = EXIT_SUCCESS;
    } else {
      /* Provide some error diagnostic. */
      fprintf(stderr, "Could␣not␣open␣%s:␣", argv[i]);
      perror(0);
      errno = 0;                        // reset error code
    }
  }
  return ret;
}
```

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**.[Exs 41][Exs 42][Exs 43][Exs 44]

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 41] Under what circumstances this program finishes with success or failure return codes?

[Exs 42] Surprisingly this program even works for files with lines that have more than 31 characters. Why?

[Exs 43] Have the program read from **stdin** if no command line argument is given.

[Exs 44] Have the program precedes all output lines with line numbers if the first command line argument is `"-n"`.

TABLE 16. 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
islower	lower case	`'a'` ··· `'z'`	yes
isupper	upper case	`'A'` ··· `'Z'`	yes
isblank	blank	`'␣'`, `'\t'`	yes
isspace	space	`'␣'`, `'\f'`, `'\n'`, `'\r'`, `'\t'`, `'\v'`	yes
isdigit	decimal	`'0'` ··· `'9'`	no
isxdigit	hexadecimal	`'0'` ··· `'9'`, `'a'` ··· `'f'`, `'A'` ··· `'F'`	no
iscntrl	control	`'\a'`, `'\b'`, `'\f'`, `'\n'`, `'\r'`, `'\t'`, `'\v'`	yes
isalnum	alphanumeric	**isalpha**`(x)||`**isdigit**`(x)`	yes
isalpha	alphabet	**islower**`(x)||`**isupper**`(x)`	yes
isgraph	graphical	`(!`**iscntrl**`(x)) && (x != '␣')`	yes
isprint	printable	`!`**iscntrl**`(x)`	yes
ispunct	punctuation	**isprint**`(x)&&!(`**isalnum**`(x)||`**isspace**`(x))`	yes

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 16 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.

#include `<ctype.h>`

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 17. 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 17. 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"` but corresponds to `"\xdeB""ruyn"`, the character with code `3563` followed by the four characters `'r'`, `'u'`, `'y'` and `'n'`.

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 45] [Exs 46] [Exs 47]

strtoul.c

```c
#include <string.h>

/* Supposes that lower case characters are contiguous. */
_Static_assert('z'-'a' == 25,
               "alphabetic␣characters␣not␣contiguous");
#include <ctype.h>
/* convert an alphanumeric digit to an unsigned */
/* '0' ... '9'  =>  0 ..  9u */
/* 'A' ... 'Z'  => 10 .. 35u */
/* 'a' ... 'z'  => 10 .. 35u */
/* other values =>   greater */
unsigned hexatridecimal(int a) {
  if (isdigit(a)) {
    /* This is guaranteed to work, decimal digits
       are consecutive and isdigit is not
       locale dependent */
    return a - '0';
  } else {
    /* leaves a unchanged if it is not lower case */
    a = toupper(a);
    /* Returns value >= 36 if not latin upper case */
    return (isupper(a)) ? 10 + (a - 'A') : -1;
  }
}
```

Besides the function **strtod**, the C library has **strtoul**, **strtol**, **strtoumax**, **strtoimax**, **strtoull**, **strtoll**, **strtold**, and **strtof** 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**

```c
unsigned long int strtoul(char const nptr[restrict],
                          char** restrict endptr,
                          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.

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

[Exs 45] The second **return** of `hexatridecimal` makes an assumption about the relation between `a` and `'A'`. Which?

[Exs 46] Describe an error scenario in case that this assumption is not fulfilled.

[Exs 47] Fix this bug.

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 `hexatridecimal` in a loop to compute a large integer from a string.

strtoul.c

```c
unsigned long Strtoul_inner(char const s[static 1],
                            size_t i,
                            unsigned base) {
  unsigned long ret = 0;
  while (s[i]) {
    unsigned c = hexatridecimal(s[i]);
    if (c >= base) break;
    /* maximal representable value for 64 bit is
       3w5e11264sgsf in base 36 */
    if (ULONG_MAX/base < ret) {
      ret = ULONG_MAX;
      errno = ERANGE;
      break;
    }
    ret *= base;
    ret += c;
    ++i;
  }
  return ret;
}
```

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

```c
unsigned long Strtoul(char const s[static 1], unsigned base) {
  if (base > 36u) {               /* test if base           */
    errno = EINVAL;               /* extends specification */
    return ULONG_MAX;
  }
  size_t i = strspn(s, "␣\f\n\r\t\v"); /* skip spaces    */
  bool switchsign = false;        /* look for a sign        */
  switch (s[i]) {
  case '-' : switchsign = true;
  case '+' : ++i;
  }
  if (!base || base == 16) {      /* adjust the base        */
    size_t adj = find_prefix(s, i, "0x");
    if (!base) base = (unsigned[]){ 10, 8, 16, }[adj];
    i += adj;
  }
  /* now, start the real conversion */
  unsigned long ret = Strtoul_inner(s, i, base);
  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.[Exs 48]

To skip the spaces, Strtoul uses **strspn**, one of the string search functions that are provided by string.h. This functions 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 at 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>

```
time_t time(time_t *t);
double difftime(time_t time1, time_t time0);
time_t mktime(struct tm tm[1]);
size_t strftime(char s[static 1], size_t max,
                char const format[static 1],
                struct tm const tm[static 1]);
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 use 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 represent in that way should lie between Gregorian years 0 and 9999.
- **tm_sec** is in the range from 0 to 60, including. The later 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,
- **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

[Exs 48] Implement a function find_prefix as needed by Strtoul.

(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**^{C} 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:

```
errno_t asctime_s(char s[static 1], rsize_t maxsize,
                  struct tm const timeptr[static 1]);
errno_t ctime_s(char s[static 1], rsize_t maxsize,
                const time_t timer[static 1]);
struct tm *gmtime_s(time_t const timer[restrict static 1],
                    struct tm result[restrict static 1]);
struct tm *localtime_s(time_t const timer[restrict static 1],
                       struct tm result[restrict static 1]);
```

The following picture shows how all these functions interact:

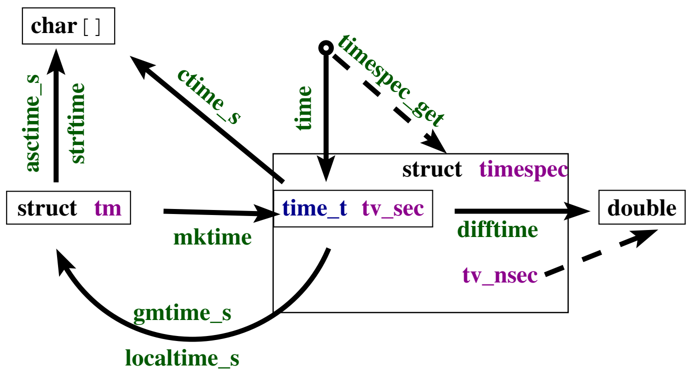


FIGURE 1. Time conversion functions

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 18. 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.[49] Relativity or other time adjustments could be avoided by using a so-called "monotonic clock" that would only be refering 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.

```c
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_PROCESS_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**^{C}: a list of name-value pairs of strings (often called **environment variables**^{C}) that can transmit specific information from the runtime environment. There is a historic function **getenv** to access this list:

```c
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:

```c
bool havenv(char const name[static 1]) {
  return getenv(name);
}
```

Instead, we use the secured function **getenv_s**: Annex K

[49]Beware that objects that move fast relative to Earth such as satelites or space crafts may perceive relativistic time shifts compared to UTC.

TABLE 18. **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	

```
errno_t getenv_s(size_t * restrict len,
                 char value[restrict],
                 rsize_t maxsize,
                 char const name[restrict]);
```

TABLE 19. Categories for the **setlocale** function

LC_COLLATE	string comparison through **strcoll** and **strxfrm**
LC_CTYPE	character classification and handling functions, see Section 8.3.
LC_MONETARY	monetary formatting information, **localeconv**
LC_NUMERIC	decimal-point character for formatted I/O, **localeconv**
LC_TIME	**strftime**, see Section 8.4
LC_ALL	all of the above

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:

```
void printenv(char const name[static 1]) {
  if (getenv(name)) {
    char value[256] = { 0, };
    if (getenv_s(0, value, sizeof value, name)) {
      fprintf(stderr,
              "%s:␣value␣is␣longer␣than␣%zu\n",
              name, sizeof value);
    } else {
      printf("%s=%s\n", name, value);
    }
  } else {
    fprintf(stderr, "%s␣not␣in␣environment\n", name);
  }
}
```

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**^{C} 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>`

```
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 19 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**.

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:

```
_Noreturn void quick_exit(int status);
_Noreturn void _Exit(int status);
_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**^{C} that are to be executed at program termination. Two functions can be used for that:

```
int atexit(void func(void));
int at_quick_exit(void func(void));
```

These have a syntax that we have not yet seen: **function parameters**^{C}. E.g the first reads "function **atexit** that returns an **int** and that receives a function `func` as a parameter".[50]

We will not go into details here, an example just shows how this can be used:

```
void sayGoodBye(void) {
  if (errno) perror("terminating with error condition");
  fputs("Good Bye\n", stderr);
}

int main(int argc, char* argv[argc+1]) {
  atexit(sayGoodBye);
  ...
}
```

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,

[50]In fact, in C such a notion of a function parameter `func` to a function **atexit** is equivalent to passing a **function pointer**^{C}. 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.

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 2.9.0.6 *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 2.9.0.7 *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 2.9.0.8 *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 2.9.0.9 *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[1] 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)[2]. 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:

```
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**^{C} if its meaning is fixed by the C standard and you may not redefined 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**^{C}, **application programmable interface**^{C}, 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>

```
double p99_memory_sum(size_t p00_n, size_t p00_i,
                      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:

[1]https://www.gnu.org/software/emacs/
[2]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
`llllllllOll`	`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**^{C}, using `internalCapitalsToBreakWords`
- **snake case**^{C}, using `internal_underscores_to_break_words`
- **Hungarian notation**^{C}[3], 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 2.9.0.7, they easily clog up a whole, precious line of programming text with unreadable expressions:

```
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.*

[3]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, "dictonary" 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 π), 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 multiples 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:

```c
u = fun4you(u, i, 33, 28);  // ;)
++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

```c
/* 33 and 28 are suitable because they are coprime. */
u = nextApprox(u, i, 33, 28);
/* Theorem 3 ensures that we may move to the next step. */
++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 2.10.0.11 (what) *Function interfaces describe what is done.*

Rule 2.10.0.12 (what for) *Interface comments document the purpose of a function.*

Rule 2.10.0.13 (how) *Function code tells how things are done.*

Rule 2.10.0.14 (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 2.10.0.15 *Separate interface and implementation.*

This rule is reflected in the use of two different kinds of C source files, **header files**^{C}, usually ending with `".h"` and **translation units**^{C}, `".c"`.

Syntactical comments have two distinct roles in those two kinds of source files that should well be separated.

Rule 2.10.0.16 *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[4]. 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.

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:

heron_k.h

```c
/**
 ** @brief use the Heron process to approximate @a a to the
 ** power of `1/k`
 **
 ** Or in other words this computes the @f$k^{th}@f$ root of @a a.
 ** As a special feature, if @a k is `-1` it computes the
 ** multiplicative inverse of @a a.
 **
 ** @param a must be greater than `0.0`
 ** @param k should not be `0` and otherwise be between
 ** `DBL_MIN_EXP*FLT_RDXRDX` and
 ** `DBL_MAX_EXP*FLT_RDXRDX`.
 **
 ** @see FLT_RDXRDX
 **/
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_RDXRDX` and **DBL_MAX_EXP**`*FLT_RDXRDX`.

See also: FLT_RDXRDX

```c
double heron(double a, signed k);
```

[4]http://www.stack.nl/ dimitri/doxygen/

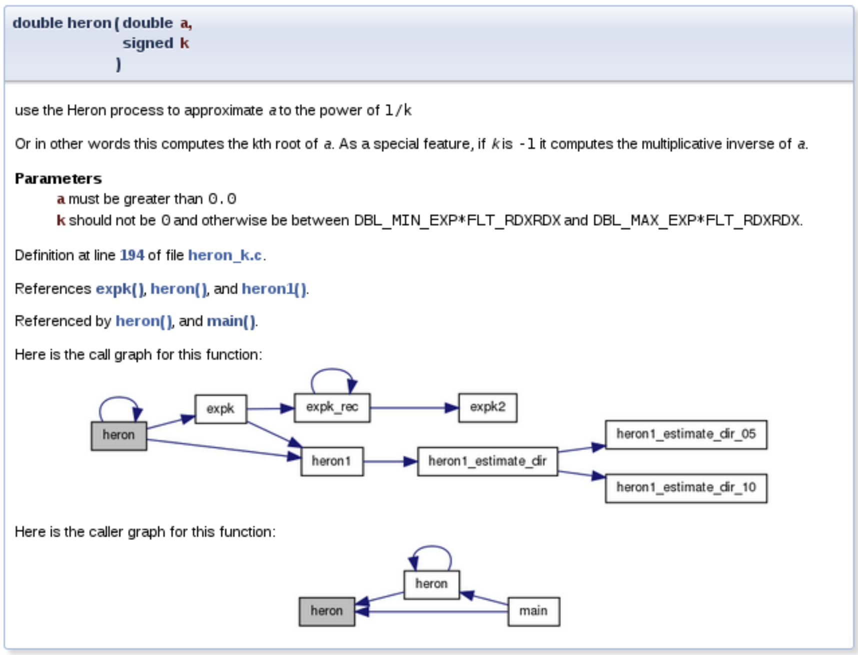
heron_k.h

`FLT_RDXRDX`: the radix base 2 of **FLT_RADIX**

This is needed internally for some of the code below.

```
# define FLT_RDXRDX 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.


```
double heron ( double  a,
               signed  k
             )
```

use the Heron process to approximate *a* to the power of `1/k`

Or in other words this computes the kth 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_RDXRDX and DBL_MAX_EXP*FLT_RDXRDX.

Definition at line **194** of file **heron_k.c**.

References **expk()**, **heron()**, and **heron1()**.

Referenced by **heron()**, and **main()**.

Here is the call graph for this function:

Here is the caller graph for this 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

```
#ifndef BRIAN_H
#define BRIAN_H 1
#include <time.h>

/** @file
```

```
 ** @brief Following Brian the Jay
 **/

typedef struct brian brian;
enum chap { sct, en, };
typedef enum chap chap;

struct brian {
  struct timespec ts; /**< point in time */
  unsigned counter;   /**< wealth        */
  chap masterof;      /**< occupation    */
};

/**
 ** @brief get the data for the next point in time
 **/
brian brian_next(brian);

...
#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**^{C}, 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**[5] 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 flyspecked 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.

[5]These will be discussed in Sections 13.2.2 and 15.1.

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 shouldn't change control flow in a surprising way.*

Notorious examples that pop up in discussion with beginners from time to time are things like:

```
#define begin {
#define end }
#define forever for (;;)
#define ERRORCHECK(CODE) if (CODE) return -1

forever
  begin
  // do something
  ERRORCHECK(x);
  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:

```
if (a) ERRORCHECK(x);
else puts("a␣is␣0!");
```

These lines are rewritten as

```
if (a) if (x) return -1;
else puts("a␣is␣0!");
```

The **else**-clause (so-called **dangling else**^{C}) is attached to the innermost **if**, which we don't see. So this is equivalent to

```
if (a) {
  if (x) return -1;
  else puts("a␣is␣0!");
}
```

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.

```
#define ERROR_RETURN(CODE)   \
do {                         \
  if (CODE) return -1;       \
} while (false)
```

This macro by itself is probably not as obvious, but its use has no surprises.

```
if (a) ERROR_RETURN(x);
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

```
if (a) do {
  if (CODE) return -1;
} while (false);
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.[Exs 6] 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.[Exs 7]

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**^{C} 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**^{C}:

- The function has no other effects than returning a value.
- The function return value only depends on its parameters.

[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)`?

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.
- The function keeps a persistent internal state between calls[8]
- The function does IO[9]

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 the they return a value of type `rat`. The others, working with pointers to `rat` will used later on in Section 11.3.

This is a straightforward 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.

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 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.[Exs 10] [Exs 11]

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

[8]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.

[9]Such an IO would e.g occur by using **printf**.

[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.

LISTING 2.1. A type for computation with rational numbers.

```c
#ifndef RATIONALS_H
# define RATIONALS_H 1
# include <stdbool.h>
# include "euclid.h"

typedef struct rat rat;

struct rat {
  bool sign;
  size_t num;
  size_t denom;
};

/* Functions that return a value of type rat. */
rat rat_get(long long num, unsigned long long denom);
rat rat_get_normal(rat x);
rat rat_get_extended(rat x, size_t f);
rat rat_get_prod(rat x, rat y);
rat rat_get_sum(rat x, rat y);


/* Functions that operate on pointers to rat. */
void rat_destroy(rat * rp);
rat * rat_init(rat * rp,
              long long num,
              unsigned long long denom);
rat * rat_normalize(rat * rp);
rat * rat_extend(rat * rp, size_t f);
rat * rat_sumup(rat * rp, rat y);
rat * rat_rma(rat * rp, rat x, rat y);

#endif
```

.

rationals.c

```c
rat rat_get(long long num, unsigned long long denom) {
  rat ret = {
    .sign = (num < 0),
    .num = (num < 0) ? -num : num,
    .denom = denom,
  };
  return ret;
}
```

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 (**put ref here**) we will learn how to use that strategy efficiently by using so-called `inline` functions or link time optimization that many modern tool chains provide.

.
rationals.c
```c
rat rat_get_normal(rat x) {
  size_t c = gcd(x.num, x.denom);
  x.num /= c;
  x.denom /= c;
  return x;
}
```

.
rationals.c
```c
rat rat_get_extended(rat x, size_t f) {
  x.num *= f;
  x.denom *= f;
  return x;
}
```

.
rationals.c
```c
rat rat_get_prod(rat x, rat y) {
  rat ret = {
    .sign = (x.sign != y.sign),
    .num = x.num * y.num,
    .denom = x.denom * y.denom,
  };
  return rat_get_normal(ret);
}
```

.
rationals.c
```c
rat rat_get_sum(rat x, rat y) {
  size_t c = gcd(x.denom, y.denom);
  size_t ax = y.denom/c;
  size_t bx = x.denom/c;
  x = rat_get_extended(x, ax);
  y = rat_get_extended(y, bx);
  assert(x.denom == y.denom);

  if (x.sign == y.sign) {
    x.num += y.num;
  } else if (x.num > y.num) {
    x.num -= y.num;
  } else {
    x.num = y.num - x.num;
    x.sign = !x.sign;
  }
  return rat_get_normal(x);
}
```

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**^{C} operator `&` allows us to refer to an object through its **address**^{C}. A call to our function could look as

```
double_swap(&d0, &d1);
```

The type that the address-of operator returns is a **pointer type**^{C}, marked with a `*`. In

```
void double_swap(double* p0, double* p1) {
  double tmp = *p0;
  *p0 = *p1;
  *p1 = tmp;
}
```

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**^{C} 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:

```
void double_swap(double p0[static 1], double p1[static 1]) {
  double tmp = p0[0];
  p0[0] = p1[0];
  p1[0] = tmp;
}
```

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:

```
double sum0(size_t len, double const* a) {
  double ret = 0.0;
  for (size_t i = 0; i < len; ++i) {
    ret += *(a + i);
  }
  return ret;
}
```

```
double sum1(size_t len, double const* a) {
  double ret = 0.0;
  for (double const* p = a; p < a+len; ++p) {
    ret += *p;
  }
  return ret;
}
```

```
double sum2(size_t len, double const* a) {
  double ret = 0.0;
  for (double const*const aStop = a+len; a < aStop; ++a) {
    ret += *a;
  }
  return ret;
}
```

These functions can then be called analogously to the following:

```
double A[7] = { 0, 1, 2, 3, 4, 5, 6, };
double s0_7 = sum0(7, &A[0]);    // for the whole
double s1_6 = sum0(6, &A[1]);    // for last 6
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:

```
double sum0(size_t len, double const a[len]);
double sum1(size_t len, double const a[len]);
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

```
char const* name = 0;

// do something that eventually sets name

if (name) {
  printf("today's␣name␣is␣%s\n", name);
} else {
  printf("today␣we␣are␣anonymous\n");
}
```

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.

```
double A[2] = { 0.0, 1.0, };
double * p = &A[0];
printf("element␣%g\n", *p); // referencing object
++p;                        // valid pointer
printf("element␣%g\n", *p); // referencing object
++p;                        // valid pointer, no object
printf("element␣%g\n", *p); // referencing non-object
                            // 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:

```c
double A[2] = { 0.0, 1.0, };
double * p = &A[0];
printf("element␣%g\n", *p); // referencing object
p += 2;                      // valid pointer, no object
printf("element␣%g\n", *p); // referencing non-object
                             // undefined behavior
```

Whereas this last example may already crash at the increment operation:

```c
double A[2] = { 0.0, 1.0, };
double * p = &A[0];
printf("element␣%g\n", *p); // referencing object
p += 3;                      // invalid pointer arithmetic
                             // 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**^{C}, 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:

```c
double A[4] = { 0.0, 1.0, 2.0, -3.0, };
double * p = &A[1];
double * q = &A[3];
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.

#include `<stddef.h>`

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 structs. 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

```c
.

void timespec_normalize(struct timespec * a){
  long const giga = 1000000000L;
  if (a->tv_nsec >= giga) {
    a->tv_sec += a->tv_nsec / giga;
    a->tv_nsec = a->tv_nsec % giga;
  } else if (a->tv_nsec < 0) {
    long sec = -(a->tv_nsec) / giga;
    long nsec = -(a->tv_nsec) % giga;
    a->tv_sec -= (sec+1);
    a->tv_nsec = giga - nsec;
  }
```

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.

. rationals.c

```
void rat_destroy(rat * rp) {
  if (rp) *rp = (rat){ 0 };
}
```

. rationals.c

```
rat * rat_init(rat * rp,
               long long num,
               unsigned long long denom) {
  if (rp) *rp = rat_get(num, denom);
  return rp;
}
```

. rationals.c

```
rat * rat_normalize(rat * rp) {
  if (rp) *rp = rat_get_normal(*rp);
  return rp;
}
```

. rationals.c

```
rat * rat_extend(rat * rp, size_t f) {
  if (rp) *rp = rat_get_extended(*rp, f);
  return rp;
}
```

These first four functions are straight forward 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.

rationals.c

```
rat * rat_sumup(rat * rp, rat y) {
  size_t c = gcd(rp->denom, y.denom);
  size_t ax = y.denom/c;
  size_t bx = rp->denom/c;
  rat_extend(rp, ax);
  y = rat_get_extended(y, bx);
  assert(rp->denom == y.denom);

  if (rp->sign == y.sign) {
    rp->num += y.num;
  } else if (rp->num > y.num) {
    rp->num -= y.num;
  } else {
    rp->num = y.num - rp->num;
    rp->sign = !rp->sign;
  }
  return rat_normalize(rp);
}
```

rationals.c

```
rat * rat_rma(rat * rp, rat x, rat y) {
  return rat_sumup(rp, rat_get_prod(x, y));
}
```

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`.[Exs 12]

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**^{C} 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:

```
/* forward declaration of struct toto */
struct toto;
struct toto * toto_get(void);
void toto_destroy(struct toto *);
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**.

```
/* forward declaration of struct toto_s and user type toto */
typedef struct toto_s * toto;
toto toto_get(void);
void toto_destroy(toto);
void toto_doit(toto, unsigned);
```

[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 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**.

This is not the same as just introducing a **typedef** name for the **struct** as we have done before.

```
/* forward declaration of struct toto and typedef toto */
typedef struct toto toto;
toto * toto_get(void);
void toto_destroy(toto *);
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 attact 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**^{C}:

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

```
  size_t strlen(char const* s);
  char*  strcpy(char* target, char const* source);
  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 as tells a whole story. This becomes even more

```
double double_copy(size_t len,
                   double target[len],
                   double const source[len]);
```

interesting if we handle two-dimensional arrays. A typical matrix multiplication could look as follows:

```
void matrix_mult(size_t n, size_t k, size_t m,
                 double C[n][m],
                 double A[n][k],
                 double B[k][m]) {
   for (size_t i = 0; i < n; ++i) {
     for (size_t j = 0; j < m; ++j) {
       C[i][j] = 0.0;
       for (size_t l = 0; l < k; ++l) {
         C[i][j] += A[i][l]*B[l][j];
       }
     }
   }
}
```

The prototype is equivalent to the less readable

```
void matrix_mult(size_t n, size_t k, size_t m,
                 double (C[n])[m],
                 double (A[n])[k],
                 double (B[k])[m]);
```

and

```
void matrix_mult(size_t n, size_t k, size_t m,
                 double (*C)[m],
                 double (*A)[k],
                 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 straight forward 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**^{C} that corresponds to a 0 value of any pointer type.[13] Here

```
double const*const nix = 0;
double const*const nax = nix;
```

"`nix`" and "`nax`" would be such a pointer object of value 0. But unfortunately a **null pointer constant**^{C} 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`	**unsigned**

`0`
`'\0'`
enumeration constant of value `0`
	signed

`0UL`
`0L`
	unsigned long
	signed long

`0ULL`
`0LL`
	unsigned long long
	signed long

(void*)0	**void**∗

Commonly used values are `0`, `0L` and `(`**void**`*)0`.[14]

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 **??**. 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.

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[15] to retrieve their

[13]Note the different capitalization of "null" versus **NULL**.

[14]In theory, there are even more possible expansions for **NULL**, such as `((`**char**`)+0)` or `((`**short**`)-0)`.

[15]Only objects that are declared with keyword **register** don't have an address, see Section **??** on Level 2

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 2.12.0.2 *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**^{C}. 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 them 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.*

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.

endianess.c
```c
.
#include <inttypes.h>

typedef union unsignedInspect unsignedInspect;
union unsignedInspect {
  unsigned val;
  unsigned char bytes[sizeof(unsigned)];
};
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.

```
((0xAA << (CHAR_BIT*3))
    |(0xBB << (CHAR_BIT*2))
    |(0xCC << CHAR_BIT)
    |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**^{C} of an **unsigned** value, namely all its representation digits.

. endianess.c
```
  printf("value is 0x%.08X\n", twofold.val);
  for (size_t i = 0; i < sizeof twofold.bytes; ++i)
    printf("byte[%zu]: 0x%.02hhX\n", i, twofold.bytes[i]);
```

On my own computer, I receive a result as indicated below:[16]

Terminal
```
~/build/modernC% ./code/endianess
value is 0xAABBCCDD
byte[0]: 0xDD
byte[1]: 0xCC
byte[2]: 0xBB
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**^{C}, the later one **big Endian**^{C}. Both orders are commonly used by modern processor types. Some processors are even able to switch between the two orders on the fly.

[16]Test that code on your own machine.

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 waste majority of platforms have these properties[17], 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.[Exs 18][Exs 19][Exs 20]

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.

```c
double blub(double const* a, double* b);

int main(void) {
  double c = 35;
  double d = 3.5;
  printf("blub␣is␣%g\n", blub(&c, &d));
  printf("after␣blub␣the␣sum␣is␣%g\n", c + d);
}
```

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:

```c
double blub(double const* a, double* b) {
  double myA = *a;
  *b = 2*myA;
  return *a;       // may be myA or 2*myA
}
```

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**^{C}. 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:

[17]In particular all POSIX systems.

[Exs 18] Design a similar **union** type to investigate the bytes of a pointer type, **double**`*` say.

[Exs 19] With such a **union**, investigate the addresses of to consecutive elements of an array.

[Exs 20] Compare addresses of the same variable between different executions.

Rule 2.12.3.1 *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:

```
size_t blob(size_t const* a, double* b) {
  size_t myA = *a;
  *b = 2*myA;
  return *a;       // must be myA
}
```

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*^{2}*) *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

```
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:

```
unsigned char highC = 1;
unsigned high hignU = (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 2, 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 to 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.

```
float f = 37.0;         // conversion: to float
double a = f;           // conversion: back to double
float* pf = &f;         // exact type
float const* pdc = &f;  // conversion: adding a qualifier
void* pv = &f;          // conversion: pointer to void*
float* pfv = pv;        // conversion: pointer from void*
float* pd = &a;         // error: incompatible pointer type
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**^{C}.[21] 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:

endianess.c

```
  unsigned val = 0xAABBCCDD;
  unsigned char* valp = (unsigned char*)&val;
  for (size_t i = 0; i < sizeof val; ++i)
    printf("byte[%zu]:␣0x%.02hhX\n", i, valp[i]);
```

In that direction (from "pointer to object" to a "pointer to character type") a cast is mostly harmless.

12.6. Alignment. The inverse direction (from "pointer to character type" to "pointer to object") is not harmless at all. This has to do with another property of C's memory model, **alignment**^{C}: objects of most non-character types can't start at any arbitrary byte position, they usually start at a **word boundary**^{C}. 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:

```
#include <stdio.h>
#include <inttypes.h>
#include <complex.h>
#include "crash.h"

void enable_alignment_check(void);
typedef complex double cdbl;

int main(void) {
  enable_alignment_check();
  /* An overlay of complex values and bytes. */
  union {
    cdbl val[2];
    unsigned char buf[sizeof(cdbl[2])];
```

[21]A cast of an expression X to type T has the form (T)X. Think of it as in "to cast a spell".

```c
  } toocomplex = {
    .val = { 0.5 + 0.5*I, 0.75 + 0.75*I, },
  };
  printf("size/alignment:␣%zu/%zu\n",
         sizeof(cdbl), _Alignof(cdbl));
  /* Run over all offsets, and crash on misalignment. */
  for (size_t offset = sizeof(cdbl); offset; offset /=2) {
    printf("offset\t%zu:\t", offset);
    fflush(stdout);
    cdbl* bp = (cdbl*)(&toocomplex.buf[offset]); // align!
    printf("%g\t+%gI\t", creal(*bp), cimag(*bp));
    fflush(stdout);
    *bp *= *bp;
    printf("%g\t+%gI", creal(*bp), cimag(*bp));
    fputc('\n', stdout);
  }
}
```

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
```
~/.../modernC/code (master % u=) 14:45 <516>$ ./crash
size/alignment: 16/8
offset        16:        0.75        +0.75I        0        +1.125I
offset        8:        0.5        +0I        0.25        +0I
offset        4:        Bus error
```

The program crashes with an error indicated as **bus error**^{C}, which is a shortcut for something like “data bus alignment error”. The real problem line is

.

crash.c
```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 little function `enable_alignment_check` at the beginning to enable that 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.[22] So consider crashing a feature. See mentioned blog entry for an interesting discussion on this topic.

In the above code example we also see a new operator, **_Alignof**, that provides us with the alignment of a type or object. There are chances that you will rarely find the occasion to use that in real live code.

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**^{C} 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 variables that are declared with **static**) or are bound to a block of statements inside a function.[23]

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**^{C}.

The following set of functions, available with `stdlib.h`, has been designed to provide such an interface to allocatable objects: **#include** <stdlib.h>

```
#include <stdlib.h>
void* malloc(size_t size);
void free(void* ptr);
void* calloc(size_t nmemb, size_t size);
void* realloc(void* ptr, size_t size);
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.

[22]For the code that is used inside that function, please consult the source code of `crash.h` to inspect it.

[23]In fact, this is a bit of a simplification, we will see the gory details below.

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.

```
size_t length = livingPeople();
double* largeVec = malloc(length * sizeof *largeVec);
for (size_t i = 0; i < length; ++i) {
  largeVec[i] = 0.0;
}
...

free(largeVec);
```

Here we allocate a large vector of **double** one for each living person.[Exs 24] 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 encouter strictly takes the size of the type of the object that we want to create, namely an array of `length` elements of type **double**.

```
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.*

Not only is such a cast superfluous, doing an explicit conversion can even be counterproductive when we forget to include the header file `stdlib.h`. Older C compilers then

#include `<stdlib.h>`

```
/* If we forget to include stdlib.h, many compilers
   still assume: */
int malloc();           // wrong function interface!
...
double* largeVec = (void*)malloc(sizeof(double[length]));
                                  |
                             int <--
                              |
                        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.*

[Exs 24] 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?

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.h

`circular_resize`: Resize to capacity *max_len*.

```
circular* circular_resize(circular* c, size_t max_len);
```

circular.h

`circular_getlength`: Return the number of elements stored.

```
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

```
/** @brief the hidden implementation of the circular buffer type
    */
struct circular {
  size_t start;    /**< position of element 0 */
  size_t len;      /**< number of elements stored */
  size_t max_len;  /**< maximum capacity */
  double* tab;      /**< array holding the data */
};
```

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	~~garb~~	~~garb~~	6.0	7.7	81.0	99.0	~~garb~~	~~garb~~	~~garb~~	~~garb~~
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	~~garb~~	~~garb~~	~~garb~~	~~garb~~	~~garb~~	~~garb~~	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 quite straight forward:

. circular.c

```
circular* circular_init(circular* c, size_t max_len) {
  if (c) {
    if (max_len) {
      *c = (circular){
        .max_len = max_len,
        .tab = malloc(sizeof(double[max_len])),
      };
        // allocation failed
      if (!c->tab) c->max_len = 0;
    } else {
      *c = (circular){ 0 };
    }
  }
  return c;
}
```

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

```c
void circular_destroy(circular* c) {
  if (c) {
    free(c->tab);
    circular_init(c, 0);
  }
}
```

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

```c
static size_t circular_getpos(circular* c, size_t pos) {
  pos += c->start;
  pos %= c->max_len;
  return pos;
}
```

Obtaining a pointer to an element of the buffer is now quite simple.

. circular.c

```c
double* circular_element(circular* c, size_t pos) {
  double* ret = 0;
  if (c) {
    if (pos < c->max_len) {
      pos = circular_getpos(c, pos);
      ret = &c->tab[pos];
    }
  }
  return ret;
}
```

With all of that information, you should now be able to implement all but one of the function interfaces nicely.[Exs 25] 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:

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.

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**:

[Exs 25] Write implementations of the missing functions.

. circular.c

```c
circular* circular_resize(circular* c, size_t nlen) {
  if (c) {
    size_t len = c->len;
    if (len > nlen) return 0;
    size_t olen = c->max_len;
    if (nlen != olen) {
      size_t ostart = circular_getpos(c, 0);
      size_t nstart = ostart;
      double* otab = c->tab;
      double* ntab;
      if (nlen > olen) {
```

. circular.c

```c
      }
      *c = (circular){
        .max_len = nlen,
        .start = nstart,
        .len = len,
        .tab = ntab,
      };
    }
  }
  return c;
}
```

. circular.c

```c
        ntab = realloc(c->tab, sizeof(double[nlen]));
        if (!ntab) return 0;
```

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:

. circular.c

```c
      // two separate chunks
      if (ostart+len > olen) {
        size_t ulen = olen - ostart;
        size_t llen = len - ulen;
        if (llen <= (nlen - olen)) {
          /* cpy the lower one up after the old end */
          memcpy(ntab + olen, ntab,
                 llen*sizeof(double));
        } else {
          /* mv the upper one up to the new end */
          nstart = nlen - ulen;
          memmove(ntab + nstart, ntab + ostart,
                  ulen*sizeof(double));
        }
      }
```

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	~~garb~~	~~garb~~	~~garb~~	~~garb~~	~~garb~~	~~garb~~	6.0	7.7			
old position	2	3							0	1			
new position	~~2~~	~~3~~							0	1	2	3	
new content	~~81.0~~	~~99.0~~	~~garb~~	~~garb~~	~~garb~~	~~garb~~	~~garb~~	~~garb~~	6.0	7.7	81.0	99.0	~~garb~~
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	~~garb~~	~~garb~~	~~garb~~	~~garb~~	~~garb~~	~~garb~~	6.0	7.7	
old position	2	3							0	1	
new position	2	3								0	1
new content	81.0	99.0	~~garb~~	~~garb~~	~~garb~~	~~garb~~	~~garb~~	~~garb~~	~~6.0~~	6.0	7.7
table index	0	1	2	3	4	5	6	7	8	9	10

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 save. 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.[Exs 26]

[Exs 26] Implement shrinking of the table: it is important to reorganize the table contents before calling **realloc**.

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);  /* refering 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);    /* refering to double x */
17    ...
18  }
```

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**^{C}, 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.

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**^{C} by another variable, also named `x`.

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  }
```

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[27], 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.[Exs 28]

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.

```
char const A[] = { 'e', 'n', 'd', '\0', };
char const B[] = { 'e', 'n', 'd', '\0', };
char const* c = "end";
char const* d = "end";
char const* e = "friend";
char const* f = (char const[]){ 'e', 'n', 'd', '\0', };
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.*

[27]In fact, such an object can be defined at file scope in another translation unit.

[Exs 28] Which value is printed by this program?

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**^{C} 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 *Refering 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**^{C} for objects in C, **static**^{C} when it is determined at compile time, **automatic**^{C} when it is automatically determined at run time, **allocated**^{C} when it is explicitly determined by function calls **malloc** and friends, and **thread**^{C} 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 relatively straight forward: 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* p
2    = &(double){ 1.0, };
3 int main(void) {
4   static B;
5 }
```

This defines four objects of static storage duration, those identified with `A`, `p` and `B` and a compound literal defined in Line 2. 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.

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
```
1: p and q are unequal, *p is 0
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**^{C}. In this case in instructs to continue execution at the place of **label**^{C} `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.

LISTING 2.4. A contrived example for the use of a compound literal

```
void fgoto(unsigned n) {
  unsigned j = 0;
  unsigned* p = 0;
  unsigned* q;
 AGAIN:
  if (p) printf("%u:␣p␣and␣q␣are␣%s,␣*p␣is␣%u\n",
                j,
                (q == p) ? "equal" : "unequal",
                *p);
  q = p;
  p = &((unsigned){ j, });
  ++j;
  if (j <= n) goto AGAIN;
}
```

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

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?

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 *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 2.13.2.13 *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:

```c
struct demo { unsigned ory[1]; };
struct demo mem(void);

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 2.13.2.14 *Objects of temporary lifetime are read-only.*

Rule 2.13.2.15 *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 got 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:

```c
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:

```
rat* rat_new(long long numerator,
             unsigned long long denominator) {
  return rat_init(malloc(sizeof(rat)),
                  numerator,
                  denominator);
}
```

The initialization using that function becomes

```
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

```
#define P99_NEW(T, ...) T ## _init(malloc(sizeof(T)), __VA_ARGS__
    )
```

With this we could have written the initialization above as

```
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:

```
rat* rat_vinit(size_t n, rat p[n]) {
  if (p)
    for (size_t i = 0; i < n; ++i)
      rat_init(p+i, 0, 1);
  return p;
}
```

With such a function, again, initialization becomes straight forward:

```
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

```
rat* rat_vnew(size_t size) {
  return rat_vinit(size, malloc(sizeof(rat[size])));
}
```

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[29]. 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.

The intermediate programming language family that is usually used to describe machine instructions as they are understood by your CPU are called **assembler**^{C}, 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.[Exs 30] 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**^{C} 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**^{C} that identify certain points in the program. E.g `fgoto` is **entry point**^{C} 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**^{C}, 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.

[29]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].

[Exs 30] Find out which compiler arguments produce assembler output for your platform.

LISTING 2.5. An assembler version of the **fgoto** function

```
        .type   fgoto, @function
fgoto:
        pushq   %rbp                    # save base pointer
        movq    %rsp, %rbp              # load stack pointer
        subq    $48, %rsp               # adjust stack pointer
        movl    %edi, -36(%rbp)         # fgoto#0 => n
        movl    $0, -4(%rbp)            # init j
        movq    $0, -16(%rbp)           # init p
.L_AGAIN:
        cmpq    $0, -16(%rbp)           # if (p)
        je      .L_ELSE
        movq    -16(%rbp), %rax         #  p ==> rax
        movl    (%rax), %edx            # *p ==> edx
        movq    -24(%rbp), %rax         # (   == q)?
        cmpq    -16(%rbp), %rax         # (p  ==  )?
        jne     .L_YES
        movl    $.L_STR_EQ, %eax        # yes
        jmp     .L_NO
.L_YES:
        movl    $.L_STR_NE, %eax        # no
.L_NO:
        movl    -4(%rbp), %esi          # j     ==> printf#1
        movl    %edx, %ecx              # *p    ==> printf#3
        movq    %rax, %rdx              # eq/ne ==> printf#2
        movl    $.L_STR_FRMT, %edi      # frmt  ==> printf#0
        movl    $0, %eax                # clear eax
        call    printf
.L_ELSE:
        movq    -16(%rbp), %rax         # p ==|
        movq    %rax, -24(%rbp)         #      ==> q
        movl    -4(%rbp), %eax          # j ==|
        movl    %eax, -28(%rbp)         #      ==> cmp_lit
        leaq    -28(%rbp), %rax         # &cmp_lit ==|
        movq    %rax, -16(%rbp)         #             ==> p
        addl    $1, -4(%rbp)            # ++j
        movl    -4(%rbp), %eax          # if (j
        cmpl    -36(%rbp), %eax         #       <= n)
        jbe     .L_AGAIN                # goto AGAIN
        leave                           # rearange stack
        ret                             # return
```

	... printf	fgoto							caller ...
position		-48 ...	−36 ...	−28 ...	−24 ...	−16 ...	−8 ...	−4 ...	rbp ...
meaning			n	cmp_lit	q	p		j	
after init	~~garb~~	~~garb~~	2	~~garb~~	~~garb~~	0	~~garb~~	0	
after iter 0	~~garb~~	~~garb~~	2	0	0	rbp-28	~~garb~~	1	
after iter 1	~~garb~~	~~garb~~	2	1	rbp-28	rbp-28	~~garb~~	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 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.

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 1.5.0.12 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.[Exs 31][Exs 32]

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 a 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.

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** an writes these same numbers in a normalized way to **stdout**, namely as comma separated hexadecimal numbers.

[Exs 31] 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 32] Even the optimized version leaves room for improvement, the inner part of the loop can still be shortend. Write a C program that explores this potential when compiled with full optimization.

LISTING 2.6. An optimized assembler version of the **fgoto** function

```
        .type   fgoto, @function
fgoto:
        pushq   %rbp                    # save base pointer
        pushq   %rbx                    # save rbx register
        subq    $8, %rsp                # adjust stack pointer
        movl    %edi, %ebp              # fgoto#0 => n
        movl    $1, %ebx                # init j, start with 1
        xorl    %ecx, %ecx              # 0    ==> printf#3
        movl    $.L_STR_NE, %edx        # "ne" ==> printf#2
        testl   %edi, %edi              # if (n > 0)
        jne     .L_N_GT_0
        jmp     .L_END
.L_AGAIN:
        movl    %eax, %ebx              # j+1  ==> j
.L_N_GT_0:
        movl    %ebx, %esi              # j    ==> printf#1
        movl    $.L_STR_FRMT, %edi      # frmt ==> printf#0
        xorl    %eax, %eax              # clear eax
        call    printf
        leal    1(%rbx), %eax           # j+1  ==> eax
        movl    $.L_STR_EQ, %edx        # "eq" ==> printf#2
        movl    %ebx, %ecx              # j    ==> printf#3
        cmpl    %ebp, %eax              # if (j <= n)
        jbe     .L_AGAIN                # goto AGAIN
.L_END:
        addq    $8, %rsp                # rewind stack
        popq    %rbx                    # restore rbx
        popq    %rbp                    # restore rbp
        ret                             # return
```

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 `fprintnumbers` 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:

lbuf	is supposed to be a string
np	if non-null, the count of numbers is stored in `*np`
base	value from 0 to 36, with the same interpretation as for **strtoul**

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);
```

. numberline.c

```c
int fprintnumbers_opt(FILE*restrict stream,
                      char const form[restrict static 1],
                      char const sep[restrict static 1],
                      size_t len, size_t nums[restrict len]) {
  if (!stream)       return -EFAULT;
  if (len && !nums)  return -EFAULT;
  if (len > INT_MAX) return -EOVERFLOW;

  int err = errno;
  size_t const seplen = strlen(sep);

  size_t tot = 0;
  size_t mtot = len*(seplen+10);
  char* buf = malloc(mtot);

  if (!buf) return error_cleanup(ENOMEM, err);

  for (size_t i = 0; i < len; ++i) {
    tot += sprintf(&buf[tot], form, nums[i]);
    ++i;
    if (i >= len) break;
    if (tot > mtot-20) {
      mtot *= 2;
      char* nbuf = realloc(buf, mtot);
      if (buf) {
        buf = nbuf;
      } else {
        tot = error_cleanup(ENOMEM, err);
        goto CLEANUP;
      }
```

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

```c
static
size_t numberline_inner(char const*restrict act,
                        size_t numb[restrict], int base){
  size_t n = 0;
  for (char* next = 0; act[0]; act = next) {
    numb[n] = strtoull(act, &next, base);
    if (act == next) break;
    ++n;
  }
  return n;
}
```

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 a 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.

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

```c
size_t* numberline(size_t size, char const lbuf[restrict size],
                   size_t*restrict np, int base){
  size_t* ret = 0;
  size_t n = 0;
  /* Check for validity of the string, first. */
  if (memchr(lbuf, 0, size)) {
    /* The maximum number of integers encoded. To see that this
       may be as much look at the sequence 08 08 08 08 ... */
    ret = malloc(sizeof(size_t[1+(2*size)/3]));

    n = numberline_inner(lbuf, ret, base);

    /* Supposes that shrinking realloc will always succeed. */
    size_t len = n ? n : 1;
    ret = realloc(ret, sizeof(size_t[len]));
  }
  if (np) *np = n;
  return ret;
}
```

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.

```c
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

```
char* fgetline(size_t size, char s[restrict size],
               FILE*restrict stream){
  s[0] = 0;
  char* ret = fgets(s, size, stream);
  if (ret) {
    /* s is writable so can be pos. */
    char* pos = strchr(s, '\n');
    if (pos) *pos = 0;
    else ret = 0;
  }
  return ret;
}
```

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.[Exs 33]

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 27.1 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 `fprintnumbers` in Section 15, below. For our purpose here we restrict ourselves to the discussion of function `sprintnumbers`, that is a bit simpler because it 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

`sprintnumbers`: 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 sprintnumbers(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 `sprintnumbers` 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.

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.*

[Exs 33] Improve the **main** of the example such that it is able to cope with arbitrarily long input lines.

. numberline.c

```c
int sprintnumbers(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]) {
  char* p = buf;   /* next position in buf */
  size_t const seplen = strlen(sep);
  if (len) {
    size_t i = 0;
    for (;;) {
      p += sprintf(p, form, nums[i]);
      ++i;
      if (i >= len) break;
      memcpy(p, sep, seplen);
      p += seplen;
    }
  }
  memcpy(p, "\n", 2);
  return (p-buf)+1;
}
```

That is, if we pass an insufficient buffer as a first argument, bad things will happen. Here inside `sprintnumbers`, 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.

```c
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:

. numberline.c

```c
    /* Count the chars for the numbers. */
    for (size_t i = 0; i < len; ++i)
      tot += snprintf(0, 0, form, nums[i]);
```

We will later see how this is used in the context of `fprintnumbers`.

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
`'d'`	decimal	signed integer	yes	**strtol**, base `10`
`'i'`	decimal, octal or hex	signed integer	yes	**strtol**, base `0`
`'u'`	decimal	unsigned integer	yes	**strtoul**, base `10`
`'o'`	octal	unsigned integer	yes	**strtoul**, base `8`
`'x'`	hexadecimal	unsigned integer	yes	**strtoul**, base `16`
`'aefg'`	floating point	floating point	yes	**strtod**
`'%'`	`'%'` character	no assignment	no	
`'c'`	character (sequence)	character	no	**memcpy**
`'s'`	non-white-space	string	yes	**strcspn** with `"␣\f\n\r\t\v"`
`'['`	scan set	string	no	**strspn** or **strcspn**
`'p'`	address	**void**	yes	
`'n'`	character count	signed integer	no	

14.2. Formatted input. Similar as the **printf** family of functions for formatted output, the C library as 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**:

```c
double a[3];
/* Read and process an entire line with three double values */
if (scanf("␣%lg␣%lg␣%lg␣", &a[0], &a[1], &a[2]) < 3) {
   printf("not␣enough␣input␣values!\n");
}
```

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.
- 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.

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
"hh"	**char** types
"h"	**short** types
""	**signed**, **unsigned**, **float**, **char** arrays and strings
"l"	**long** integer types, **double**, **wchar_t** characters and strings
"ll"	**long long** integer types
"j"	**intmax_t**, **uintmax_t**
"z"	**size_t**
"t"	**ptrdiff_t**
"L"	**long double**

- 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.[Exs 34] 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.[35] 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.

[Exs 34] 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.

[35]The character encoding that is dominantly used for the basic character set referred to as ASCII, American standard code for information interchange.

During the first years of market expansion out to the world different computer manufactors, 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:

mbstrings-main.c

```c
  setlocale(LC_ALL, "");
  /* Multibyte character printing only works after the locale
     has been switched. */
  draw_sep(TOPLEFT "␣©␣2014␣jɛnz␣ˈgʊz,tɛt␣", TOPRIGHT);
```

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 jɛnz ˈgʊz,tɛt ──────────────────────────┐
```

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:

mbstrings-main.c

```c
#define VBAR "\u2502"      /**< a vertical bar character   */
#define HBAR "\u2500"      /**< a horizontal bar character */
#define TOPLEFT "\u250c"   /**< topleft corner character   */
#define TOPRIGHT "\u2510"  /**< topright corner character  */
```

And an adhoc function that nicely formats an output line:

mbstrings-main.c

`draw_sep`: Draw multibyte strings *start* and *end* separated by a horizontal line.

```c
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␣ínóff"` 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`.

mbstrings.h

`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 occured.

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`[36]. 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[37] attempts to provide a unified framework for character codes. It provides a huge table[38] 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 Α.

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

[36]The header `uchar.h` also provides this type.

[37]http://www.joelonsoftware.com/articles/Unicode.html

[38]Today Unicode has about 110000 code points.

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[39]: 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 an 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

.

mbstrings.c

```
size_t mbsrlen(char const*s, mbstate_t const*restrict state) {
  if (!state) state = MBSTATE;
  mbstate_t st = *state;
  size_t mblen = mbsrtowcs(0, &s, 0, &st);
  if (mblen == -1) errno = 0;
  return mblen;
}
```

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:

```
size_t mbsrtowcs(wchar_t*restrict dst, char const**restrict src,
                 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 use to encode exactly one character of the extended character set and these wide characters 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:

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,

[39]http://dotslashzero.net/2014/05/21/the-interesting-state-of-unicode-in-c/

.

mbstrings.h

```
#ifndef __STDC_ISO_10646__
# error "wchar_t␣wide␣characters␣have␣to␣be␣Unicode␣code␣points"
#endif
#ifdef __STDC_MB_MIGHT_NEQ_WC__
# error "basic␣character␣codes␣must␣agree␣on␣char␣and␣wchar_t"
#endif
```

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`.

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**. Part of the code of `mbsrlen` is actually a "repair" of that error strategy by setting **errno** to `0`, again.

Let's now see into a second function that will help us to handle mbs:

mbstrings.h

`mbsrdup`: 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* mbsrdup(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 straight forward:

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**:

```
size_t mbrtowc(wchar_t*restrict pwc,
               const char*restrict s, size_t len,
               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

.
mbstrings.c
```
wchar_t* mbsrdup(char const*s, mbstate_t const*restrict state) {
  size_t mblen = mbsrlen(s, state);
  if (mblen == -1) return 0;
  mbstate_t st =  state ? *state : *MBSTATE;
  wchar_t* S = malloc(sizeof(wchar_t[mblen+1]));
  /* We know that s converts well, so no error check */
  if (S) mbsrtowcs(S, &s, mblen+1, &st);
  return S;
}
```

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 occured.

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
```
size_t mbrtow(wchar_t*restrict C, char const c[restrict static
    1],
               mbstate_t*restrict state) {
  if (!state) state = MBSTATE;
  size_t len = -2;
  for (size_t maxlen = MB_LEN_MAX; len == -2; maxlen *= 2)
    len = mbrtowc(C, c, maxlen, state);
  if (len == -1) errno = 0;
  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

```
char const* mbsrwc(char const s[restrict static 1], mbstate_t*
    restrict state,
                   wchar_t C, size_t occurrence) {
  if (!C || C == WEOF) return 0;
  if (!state) state = MBSTATE;
  char const* ret = 0;

  mbstate_t st = *state;
  for (size_t len = 0; s[0]; s += len) {
    mbstate_t backup = st;
    wchar_t S = 0;
    len = mbrtow(&S, s, &st);
    if (!S) break;
    if (C == S) {
      *state = backup;
      ret = s;
      if (!occurrence) break;
      --occurrence;
    }
  }
  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[40] 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.

```
size_t fread(void* restrict ptr, size_t size, size_t nmemb,
             FILE* restrict stream);
size_t fwrite(void const*restrict ptr, size_t size, size_t nmemb,
              FILE* restrict stream);
int fseek(FILE* stream, long int offset, int whence);
long int ftell(FILE* stream);
```

The use of **fread** and **fwrite** is relatively straight forward. Each stream has a current file position for reading an 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.[41]

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:

[40]http://www.nubaria.com/en/blog/?p=289

[41]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 endianess. 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.[Exs 42]

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 `fprintnumbers`, that continues the series of functions that we discussed in Section 14.1.

numberline.c

`fprintnumbers`: 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 fprintnumbers(FILE*restrict stream,
                  char const form[restrict static 1],
                  char const sep[restrict static 1],
                  size_t len, size_t numb[restrict len]);
```

`#include <errno.h>`

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.

[Exs 42] Write a function `fseekmax` that uses **intmax_t** instead of **long** and that achieves large seek values by combining calls to **fseek**.

. numberline.c

```c
#include <limits.h>
#include <errno.h>
#ifndef EFAULT
# define EFAULT EDOM
#endif
#ifndef EOVERFLOW
# define EOVERFLOW (EFAULT-EOF)
# if EOVERFLOW > INT_MAX
#  error EOVERFLOW constant is too large
# endif
#endif
#ifndef ENOMEM
# define ENOMEM (EOVERFLOW+EFAULT-EOF)
# if ENOMEM > INT_MAX
#  error ENOMEM constant is too large
# endif
#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

```c
int fprintnumbers(FILE*restrict stream,
                  char const form[restrict static 1],
                  char const sep[restrict static 1],
                  size_t len, size_t nums[restrict len]) {
  if (!stream)       return -EFAULT;
  if (len && !nums)  return -EFAULT;
  if (len > INT_MAX) return -EOVERFLOW;

  size_t tot = (len ? len : 1)*strlen(sep);
  int err = errno;
  char* buf = 0;

  if (len) {
    /* Count the chars for the numbers. */
    for (size_t i = 0; i < len; ++i)
      tot += snprintf(0, 0, form, nums[i]);
    /* We return int so we have to constrain the max size. */
    if (tot > INT_MAX) return error_cleanup(EOVERFLOW, err);
  }

  buf = malloc(tot+1);
  if (!buf) return error_cleanup(ENOMEM, err);

  sprintnumbers(tot, buf, form, sep, len, nums);
  /* print whole line in one go */
  if (fputs(buf, stream) == EOF) tot = EOF;
  free(buf);
  return tot;
}
```

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 Sections 8 and **??**, **runtime constraint violations**^{C}.

Dynamic runtime errors are a bit more difficult to handlė. 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

```
static inline int error_cleanup(int err, int prev) {
  errno = prev;
  return -err;
}
```

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**.

15.1. The use of goto for cleanup. 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 2.15.1.1 *Labels for* **goto** *are visible in the whole function that contains them.*

Rule 2.15.1.2 **goto** *can only jump to a label inside the same function.*

Rule 2.15.1.3 **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 18.2 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

```c
int fprintnumbers_opt(FILE*restrict stream,
                      char const form[restrict static 1],
                      char const sep[restrict static 1],
                      size_t len, size_t nums[restrict len]) {
  if (!stream)       return -EFAULT;
  if (len && !nums)  return -EFAULT;
  if (len > INT_MAX) return -EOVERFLOW;

  int err = errno;
  size_t const seplen = strlen(sep);

  size_t tot = 0;
  size_t mtot = len*(seplen+10);
  char* buf = malloc(mtot);

  if (!buf) return error_cleanup(ENOMEM, err);

  for (size_t i = 0; i < len; ++i) {
    tot += sprintf(&buf[tot], form, nums[i]);
    ++i;
    if (i >= len) break;
    if (tot > mtot-20) {
      mtot *= 2;
      char* nbuf = realloc(buf, mtot);
      if (buf) {
        buf = nbuf;
      } else {
        tot = error_cleanup(ENOMEM, err);
        goto CLEANUP;
      }
    }
    memcpy(&buf[tot], sep, seplen);
    tot += seplen;
    if (tot > INT_MAX) {
      tot = error_cleanup(EOVERFLOW, err);
      goto CLEANUP;
    }
  }
  buf[tot] = 0;

  /* print whole line in one go */
  if (fputs(buf, stream) == EOF) tot = EOF;
 CLEANUP:
  free(buf);
  return tot;
}
```

APPENDIX A

TABLE 1. Scalar types used in this book

	name	other	category	where	[min, max]	where	typical	**printf**
0	**size_t**		unsigned	<stddef.h>	[0, **SIZE_MAX**]	<stdint.h>	$[0, 2^w-1]$, $w = 32, 64$	"%zu" "%zx"
0	**double**		floating	builtin	[±**DBL_MIN**, ±**DBL_MAX**]	<**float**.h>	$[\pm 2^{-w-2}, \pm 2^w]$, $w = 1024$	"%e" "%f" "%g" "%a"
0	**signed**	**int**	signed	builtin	[**INT_MIN**, **INT_MAX**]	<limits.h>	$[-2^w, 2^w-1]$, $w = 31$	"%d"
0	**unsigned**		unsigned	builtin	[0, **UINT_MAX**]	<limits.h>	$[0, 2^w-1]$, $w = 32$	"%u" "%x"
0	**bool**	**_Bool**	unsigned	<stdbool.h>	[**false**, **true**]	<stdbool.h>	$[0, 1]$	
1	**ptrdiff_t**		signed	<stddef.h>	[... MIN, ... MAX]	<stdint.h>	$[-2^w, 2^w-1]$, $w = 31, 63$	"%td"
1	**char const***		string	builtin				"%s"
1	**char**		character	builtin	[**CHAR_MIN**, **CHAR_MAX**]	<limits.h>	$[0, 2^w-1]$, $w = 7, 8$	"%c"
1	**void***		pointer	builtin				"%p"
2	**unsigned char**		unsigned	builtin	[0, **UCHAR_MAX**]	<limits.h>	$[0, 255]$	"%hhu" "%02hhx"

TABLE 2. Value operators: “form” gives the syntactic form of the operation where `@` represents the operator and `a` and eventually `b` denote values that serve as operands. Unless noted with “`0,1` value”, the type of the result is the same type as `a` (and `b`),

			type restriction			
operator	nick	form	`a`	`b`	result	
		`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
`~`	**compl**	`@a`	integer		integer	bit
`& | ^`	**bitand** **bitor** **xor**	`a@b`	integer	integer	integer	bit
`<< >>`		`a@b`	integer	positive	integer	bit
`== != < > <= >=`	**not_eq**	`a@b`	scalar	scalar	`0,1` value	comparison
	`!!a`	`a`	scalar		`0,1` value	logic
`!a`	**not**	`@a`	scalar		`0,1` value	logic
`&& ||`	**and or**	`a@b`	scalar	scalar	`0,1` value	logic
`.`		`a@m`	**struct**		value	member
`*`		`@a`	pointer		object	reference
`[]`		`a[b]`	pointer	integer	object	member
`->`		`a@m`	**struct** pointer		object	member
`()`		`a(b ...)`	function pointer		value	call
sizeof		`@a`	any		**size_t**	size

TABLE 3. Object operators: "form" gives the syntactic form of the operation where `@` represents the operator and `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
`+= -= *= /=`		`o@a`	arithmetic	value	arithmetic
`+= -=`		`o@a`	pointer	value	arithmetic
`%=`		`o@a`	integer	value	arithmetic
`++ --`		`@o o@`	real or pointer	value	arithmetic
`&= |= ^=`	**and_eq** **or_eq** **xor_eq**	`o@a`	integer	value	bit
`<<= >>=`		`o@a`	integer	value	bit
`.`		`o@m`	**struct**	object	member
`[]`		`o[a]`	array*	object	member
`&`		`@o`	any*	pointer	address
sizeof		`@o`	non-function	**size_t**	size

TABLE 4. Type operators: these operators all return a value of type **size_t**. All have function-like syntax with the operands in parenthesis.

operator	nick	form	type	
sizeof		**sizeof** `(T)`	any	size
_Alignof	**alignof**	**_Alignof** `(T)`	any	aligment
	offsetof	**offsetof** `(T,m)`	**struct**	member offset

Reminders

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.	6
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`.	9
0.2.3.5	Each object must have exactly one definition.	10
0.2.4.1	Domain iterations should be coded with a **for** statement.	10
0.2.4.2	The loop variable should be defined in the initial part of a **for**.	11
1.3.1.1	The value `0` represents **false**.	16
1.3.1.2	Any value different from `0` represents **true**.	16
1.3.1.3	Don't compare to `0`, **false** or **true**.	16
1.3.1.4	All scalars have a truth value.	16
1.3.3.1	**case** values must be integer constant expressions.	21
1.3.3.2	**case** labels must not jump beyond a variable definition.	21
1.4.0.3	The type **size_t** represents values in the range `[0,` **SIZE_MAX**`]`.	22
1.4.1.1	Unsigned arithmetic is always well defined.	23
1.4.1.2	Operations `+`, `-` and `*` on **size_t** provide the mathematically correct result if it is representable as a **size_t**.	23
1.4.1.3	For unsigned values, `a == (a/b)*b + (a%b)`.	23
1.4.1.4	Unsigned `/` and `%` are well defined only if the second operand is not `0`.	23
1.4.1.5	Arithmetic on **size_t** implicitly does computation `%(`**SIZE_MAX**`+1)`.	23
1.4.1.6	In case of overflow, unsigned arithmetic wraps around.	23
1.4.1.7	The result of unsigned `/` and `%` is always smaller than the operands.	23
1.4.1.8	Unsigned `/` and `%` can't overflow.	23
1.4.2.1	Operators must have all their characters directly attached to each other.	24
1.4.2.2	Side effects in value expressions are evil.	24
1.4.2.3	Never modify more than one object in a statement.	24
1.4.3.1	Comparison operators return the values **false** or **true**.	25

	1.4.3.2	Logic operators return the values **false** or **true**.	25
	1.4.5.1	`&&`, `||`, `?:` and `,` evaluate their first operand first.	27
	1.4.5.2	Don't use the `,` operator.	27
	1.4.5.3	Most operators don't sequence their operands.	27
	1.4.5.4	Function calls don't sequence their argument expressions.	27
	1.4.5.5	Functions that are called inside expressions should not have side effects.	27
	1.5.0.6	All values are numbers or translate to such.	28
	1.5.0.7	All values have a type that is statically determined.	29
	1.5.0.8	A value's type determines its possible operations.	29
	1.5.0.9	A value's type determines the results of all operations.	29
	1.5.0.10	A type's binary representation determines the results of all operations.	29
	1.5.0.11	A type's binary representation is observable.	29
as-if	1.5.0.12	Programs execute **as if** they were following the abstract state machine.	29
	1.5.0.13	Type determines optimization opportunities.	30
	1.5.1.1	Each of the 4 classes of base types has 3 distinct unpromoted types.	30
	1.5.1.2	Use **size_t** for sizes, cardinalities or ordinal numbers.	31
	1.5.1.3	Use **unsigned** for small quantities that can't be negative.	31
	1.5.1.4	Use **signed** for small quantities that bear a sign.	31
	1.5.1.5	Use **ptrdiff_t** for large differences that bear a sign.	31
	1.5.1.6	Use **double** for floating point calculations.	31
	1.5.1.7	Use **double complex** for complex calculations.	32
	1.5.2.1	Numerical literals are never negative.	33
	1.5.2.2	Decimal integer constants are signed.	33
	1.5.2.3	A decimal integer constant has the first of the 3 signed types that fits it.	33
	1.5.2.4	The same value can have different types.	33
	1.5.2.5	Don't use octal or hexadecimal constants to express negative values.	33
	1.5.2.6	Use decimal constants to express negative values.	33
	1.5.2.7	Different literals can have the same value.	34
	1.5.2.8	The effective value of a decimal floating point constant may be different from its literal value.	34
	1.5.2.9	Literals have value, type and binary representation.	34
	1.5.3.1	All variables should be initialized.	35
	1.5.3.2	Use designated initializers for all aggregate data types.	35
	1.5.3.3	`{0}` is a valid initializer for all object types that are not VLA.	35
	1.5.4.1	All constants with particular meaning must be named.	36
	1.5.4.2	All constants with different meaning must be distinguished.	36
	1.5.4.3	An object of **const**-qualified type is read-only.	36
	1.5.4.4	String literals are read-only.	36
	1.5.4.5	Enumeration constants have either an explicit or positional value.	37
	1.5.4.6	Enumeration constants are of type **signed int**.	37
	1.5.4.7	An integer constant expression doesn't evaluate any object.	38
	1.5.4.8	Macro names are in all caps.	38
	1.5.4.9	A compound literal defines an object.	39
	1.5.4.10	Don't hide a terminating semicolon inside a macro.	39
	1.5.4.11	Right-indent continuation markers for macros to the same column.	39
	1.5.4.12	**I** is reserved for the imaginary unit.	39

1.5.5.1	The same value may have different binary representations.	39
1.5.5.2	Unsigned arithmetic wraps nicely.	40
1.5.5.3	The maximum value of any integer type is of the form $2^p - 1$.	40
1.5.5.4	Arithmetic on an unsigned integer type is determined by its precision.	40
1.5.5.5	The second operand of a shift operation must be less than the precision.	42
1.5.5.6	Positive values are represented independently from signedness.	42
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.	43
1.5.5.8	It is your responsibility to avoid undefined behavior of all operations.	43
1.5.5.9	Signed arithmetic may trap badly.	44
1.5.5.10	In twos' complement representation **INT_MIN** `<` `-`**INT_MAX**.	44
1.5.5.11	Negation may overflow for signed arithmetic.	44
1.5.5.12	Use unsigned types for bit operations.	44
1.5.5.13	Floating point operations are neither associative, commutative or distributive.	45
1.5.5.14	Never compare floating point values for equality.	45
1.6.1.1	Arrays are not pointers.	46
1.6.1.2	An array in a condition evaluates to **true**.	47
1.6.1.3	There are array objects but no array values.	47
1.6.1.4	Arrays can't be compared.	47
1.6.1.5	Arrays can't be assigned to.	47
1.6.1.6	VLA can't have initializers.	47
1.6.1.7	VLA can't be declared outside functions.	47
1.6.1.8	The length of an FLA is determined by an ICE or an initializer.	47
1.6.1.9	An array length specification must be strictly positive.	47
1.6.1.10	An array with a length not an integer constant expression is an VLA.	48
1.6.1.11	The length of an array `A` is `(`**sizeof** `A)/(`**sizeof** `A[0])`.	48
1.6.1.12	The innermost dimension of an array parameter to a function is lost.	48
1.6.1.13	Don't use the **sizeof** operator on array parameters to functions.	48
1.6.1.14	Array parameters behave as-if the array is **passed by reference**^{C}.	48
1.6.1.15	A string is a `0`-terminated array of **char**.	49
1.6.1.16	Using a string function with a non-string has undefined behavior.	50
1.6.2.1	Pointers are opaque objects.	51
1.6.2.2	Pointers are valid, null or indeterminate.	51
1.6.2.3	Initialization or assignment with `0` makes a pointer null.	51
1.6.2.4	In logical expressions, pointers evaluate to **false** iff they are null.	51
1.6.2.5	Indeterminate pointers lead to undefined behavior.	51
1.6.2.6	Always initialize pointers.	51
1.6.3.1	Omitted **struct** initializers force the corresponding field to `0`.	53
1.6.3.2	A **struct** initializer must initialize at least one field.	53
1.6.3.3	**struct** parameters are passed by value.	54
1.6.3.4	Structures can be assigned with `=` but not compared with `==` or `!=`.	54
1.6.3.5	A structure layout is an important design decision.	54
1.6.3.6	All **struct** declarations in a nested declaration have the same scope of visibility.	56

1.6.4.1	Forward-declare a **struct** within a **typedef** using the same identifier as the tag name.	56
1.6.4.2	Identifier names terminating with `_t` are reserved.	57
1.7.1.1	All functions must have prototypes.	59
1.7.1.2	Functions only have one entry but several **return**.	59
1.7.1.3	A function **return** must be consistent with its type.	59
1.7.1.4	Reaching the end of the `{}` block of a function is equivalent to a **return** statement without expression.	59
1.7.1.5	Reaching the end of the `{}` block of a function is only allowed for **void** functions.	59
1.7.2.1	Use **EXIT_SUCCESS** or **EXIT_FAILURE** as return values of **main**.	60
1.7.2.2	Reaching the end of the `{}` block of **main** is equivalent to a **return EXIT_SUCCESS**`;`.	60
1.7.2.3	Calling **exit**`(s)` is equivalent evaluation of **return** `s` in **main**.	60
1.7.2.4	**exit** never fails and never returns to its caller.	60
1.7.2.5	All commandline arguments are transferred as strings.	60
1.7.2.6	Of the arguments to **main**, `argv[0]` holds the name of the program invocation.	60
1.7.2.7	Of the arguments to **main**, `argv[argc]` is `0`.	60
1.7.3.1	Make all preconditions for a function explicit.	61
1.7.3.2	In a recursive function, first check the termination condition.	61
1.7.3.3	Ensure the preconditions of a recursive function in a wrapper function.	62
1.7.3.4	Multiple recursion may lead to exponential computation times.	64
1.7.3.5	A bad algorithm will never lead to a performing implementation.	65
1.7.3.6	Improving an algorithm can dramatically improve performance.	65
1.8.0.7	Failure is always an option.	68
1.8.0.8	Check the return value of library functions for errors.	68
1.8.0.9	Fail fast, fail early and fail often.	68
1.8.0.10	Identifier names terminating with `_s` are reserved.	69
1.8.0.11	Missed preconditions for the execution platform must abort compilation.	69
1.8.0.12	Only evaluate macros and integer literals in a preprocessor condition.	69
1.8.0.13	In preprocessor conditions unknown identifiers evaluate to `0`.	69
1.8.2.1	Opaque types are specified through functional interfaces.	72
1.8.2.2	Don't rely on implementation details of opaque types.	72
1.8.2.3	**puts** and **fputs** differ in their end of line handling.	72
1.8.2.4	Text input and output converts data.	75
1.8.2.5	There are three commonly used conversion to encode end-of-line.	75
1.8.2.6	Text lines should not contain trailing white space.	75
1.8.2.7	Parameters of **printf** must exactly correspond to the format specifiers.	76
1.8.2.8	Use `"%+d"`, `"%#X"` and `"%a"` for conversions that have to be read, later.	77
1.8.2.9	Don't use **gets**.	78
1.8.2.10	**fgetc** returns **int** to be capable to encode a special error status, **EOF**, in addition to all valid characters.	79
1.8.2.11	End of file can only be detected after a failed read.	79
1.8.3.1	The interpretation of numerically encoded characters depends on the execution character set.	81
1.8.6.1	Regular program termination should use **return** from **main**.	88

1.8.6.2	Use **exit** from a function that may terminate the regular control flow.	88	
1.8.6.3	Don't use other functions for program termination than **exit**, unless you have to inhibit the execution of library cleanups.	88	
1.8.6.4	Use as many **assert** as you may to confirm runtime properties.	89	
1.8.6.5	In production compilations, use **NDEBUG** to switch off all **assert**.	89	
Level 2		91	
2.9.0.6	All C code must be readable.	91	
2.9.0.7	Short term memory and the field of vision are small.	91	
2.9.0.8	Coding style is not a question of taste but of culture.	91	
2.9.0.9	When you enter an established project you enter a new cultural space.	91	
2.9.1.1	Choose a consistent strategy for white space and other text formatting.	92	
2.9.1.2	Have your text editor automatically format your code correctly.	92	
2.9.2.1	Choose a consistent naming policy for all identifiers.	92	
2.9.2.2	Any identifier that is visible in a header file must be conforming.	93	
2.9.2.3	Don't pollute the global name space.	93	
2.9.2.4	Names must be recognizable and quickly distinguishable.	94	
2.9.2.5	Naming is a creative act.	95	
2.9.2.6	File scope identifiers must be comprehensive.	95	
2.9.2.7	A type name identifies a concept.	95	
2.9.2.8	A global constant identifies an artifact.	95	
2.9.2.9	A global variable identifies state.	95	
2.9.2.10	A function or functional macro identifies an action.	95	
2.10.0.11	Function interfaces describe what is done.	96	what
2.10.0.12	Interface comments document the purpose of a function.	96	what for
2.10.0.13	Function code tells how things are done.	96	how
2.10.0.14	Code comments explain why things are done as they are.	96	why
2.10.0.15	Separate interface and implementation.	96	
2.10.0.16	Document the interface – Explain the implementation.	96	
2.10.1.1	Document interfaces thoroughly.	97	
2.10.1.2	Structure your code in units that have strong semantic connections.	98	
2.10.2.1	Implement literally.	99	
2.10.2.2	Control flow must be obvious.	99	
2.10.3.1	Macros should shouldn't change control flow in a surprising way.	100	
2.10.3.2	Function like macros should syntactically behave like function calls.	101	
2.10.4.1	Function parameters are passed by value.	101	
2.10.4.2	Global variables are frowned upon.	101	
2.10.4.3	Express small tasks as pure functions whenever possible.	102	
2.11.1.1	Using * with an indeterminate or null pointer has undefined behavior.	105	
2.11.2.1	A valid pointer addresses the first element of an array of the base type.	106	
2.11.2.2	The length an array object cannot be reconstructed from a pointer.	106	
2.11.2.3	Pointers are not arrays.	106	
2.11.2.4	Pointers have truth.	107	
2.11.2.5	A pointed-to object must be of the indicated type.	107	
2.11.2.6	A pointer must point to a valid object, one position beyond a valid object or be null.	107	

2.11.2.7	Only subtract pointers to elements of an array object.	108
2.11.2.8	All pointer differences have type **ptrdiff_t**.	108
2.11.2.9	Use **ptrdiff_t** to encode signed differences of positions or sizes.	108
2.11.4.1	Don't hide pointers in a **typedef**.	111
2.11.5.1	The two expressions `A[i]` and `*(A+i)` are equivalent.	111
2.11.5.2	Evaluation of an array `A` returns `&A[0]`.	111
2.11.6.1	In a function declaration any array parameters rewrites to a pointer.	111
2.11.6.2	Only the innermost dimension of an array parameter is rewritten.	112
2.11.6.3	Declare length parameters before array parameters.	113
2.11.7.1	Don't use **NULL**.	113
2.12.0.2	Pointer types with distinct base types are distinct.	114
2.12.1.1	**sizeof**`(`**char**`)` is `1` by definition.	114
2.12.1.2	Every object `A` can be viewed as **unsigned char**`[`**sizeof** `A]`.	114
2.12.1.3	Pointers to character types are special.	114
2.12.1.4	Use the type **char** for character and string data.	114
2.12.1.5	Use the type **unsigned char** as the atom of all object types.	114
2.12.1.6	The **sizeof** operator can be applied to objects and object types.	114
2.12.2.1	The in-memory order of the representation digits of a numerical type is implementation defined.	115
2.12.2.2	On most architectures **CHAR_BIT** is `8` and **UCHAR_MAX** is `255`.	116
2.12.3.1	With the exclusion of character types, only pointers of the same base type may alias.	117
2.12.3.2	Avoid the `&` operator.	117
2.12.4.1	Any object pointer converts to and from **void**`*`.	117
2.12.4.2	Converting an object pointer to **void**`*` and then back to the same type is the identity operation.	117
$avoid^{2}*$ 2.12.4.3	Avoid **void**`*`.	117
2.12.5.1	Chose your arithmetic types such that implicit conversions are harmless.	118
2.12.5.2	Don't use narrow types in arithmetic.	118
2.12.5.3	Use unsigned types whenever you may.	118
2.12.5.4	Don't use casts.	119
2.13.1.1	Don't cast the return of **malloc** and friends.	122
2.13.1.2	Objects that are allocated through **malloc** are uninitialized.	122
2.13.1.3	**malloc** indicates failure by returning a null pointer value.	126
2.13.1.4	For every **malloc** there must be a **free**.	129
2.13.1.5	For every **free** there must be a **malloc**.	129
2.13.1.6	Only call **free** with pointers as they are returned by **malloc**.	129
2.13.2.1	Identifiers only have visibility inside their scope, starting at their declaration.	130
2.13.2.2	The visibility of an identifier can be shadowed by an identifier of the same name in a subordinate scope.	130
2.13.2.3	Every definition of a variable creates a new distinct object.	130
2.13.2.4	Read-only object literals may overlap.	131
2.13.2.5	Objects have a lifetime outside of which they can't be accessed.	131
2.13.2.6	Refering to an object outside of its lifetime has undefined behavior.	131
2.13.2.7	Objects with static storage duration are always initialized.	132

Listings

Bibliography

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`.

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.

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.

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`.

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.

Index