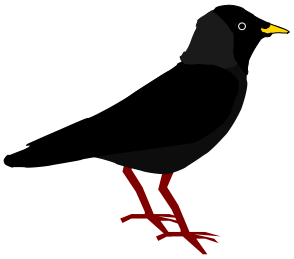
Modern C

Jens Gustedt



Modern C

Jens Gustedt

INRIA, France

ICube, Strasbourg, France

This is the 3rd edition of this book, as of October 15, 2024, in line with the most recent C standard, C23.

The contents of this book is identical to the print and ebook version that is licensed to Manning Publications Co., Shelter Island, New York, USA.

This free version, sample code, links to Manning's print edition and much more is available at

`https://gustedt.gitlabpages.inria.fr/modern-c/`

Acknowledgments

Special thanks go to the people who encouraged the writing of this book by providing me with constructive feedback, including

colleagues and other interested readers, Cédric Bastoul, Lucas Nussbaum, Vincent Loechner, Kliment Yanev, Szabolcs Nagy, Marcin Kowalczuk, Ali Asad Lotia, Richard Palme, Yann Barsamian, Fernando Oleo, Róbert Kohányi, Jean-Michel Gorius, and Martin Uecker; Manning's staff Jennifer Stout, Nitin Gode, and Frances Buontempo; and the impressive number of reviewers provided by Manning for both editions they managed.

Many others have contributed to the success of this book. My sincerest thanks to all of you.

About this book

The C programming language has been around since the early 1970s (see Ritchie [1993]). 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 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, so it is organized in *levels* that reflect familiarity with the C language and programming in general. This structure may go against some habits of the book's readers. In particular, it splits some difficult subjects (such as pointers) across levels so as not to swamp readers too early with the wrong information. We'll explain the book's organization in more detail shortly.

Generally, although many universally applicable ideas will be presented that would also be valid for other programming languages (such as Java, Python, Ruby, C#, and C++), the book primarily addresses concepts and practices that are unique to C or are of particular value when programming in the C language.

C revisions. As the title of this book suggests, today's C is not the same language as the one originally designed by its creator. Right from the start, C has been in a continuous process of adjustment and improvement. Usually, early C is referred to as K&R C (Kernighan and Ritchie C) after the first book that made the language popular (Kernighan and Ritchie [1978]). Since then, it has undergone an important standardization and extension process, now driven by the International Standards Organization (ISO). This led to the publication of a series of C standards in 1989, 1999, 2011, 2018, and 2024, commonly referred to as C89, C99, C11, C17, and C23. The C standards committee put a lot of effort into guaranteeing backward compatibility such that code written for earlier revisions of the language (say, C11), should compile to a semantically equivalent executable with a compiler that implements a newer revision. Unfortunately, this backward compatibility has had the unwanted side effect of not motivating projects that could benefit greatly from the new features to update their code base. To emphasize this progress of revisions, we indicate which standard revision introduced newer features.

This edition. This edition presents a considerable rework given the latest revision, C23, of the C standard. A lot of new material has been added, and many expositions have been straightened out to reflect the new capabilities of the C programming language. So, in this book, we will mainly refer to C23, as defined in C23, but at the time of this writing, compilers hadn't yet implemented this standard completely. If you want to compile the examples in this book, you will need at least a compiler that implements most of C17. For the novelties that C23 introduces, we provide a compatibility header and discuss how to possibly generate a suitable C compiler and C library platform on POSIX systems as a fallback in technical annex 21.4. Beware that this is not meant as a permanent tool but as only a crutch while platforms adapt.

C and C++. 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, including corporate or individual interest, politics, beauty, logic, luck, ignorance, selfishness, ego, sectarianism, and (add your primary motivation here). Thus, the development of C has not been and cannot be ideal. It has flaws and artifacts that can only be understood within their historical 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. Although 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 addition of atomics and threads, have been designed in 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, not C++. Many of the code examples will not even compile with a C++ compiler. So we should not mix sources of both languages.

Takeaway #1 *C and C++ are different: don't mix them, and don't mix them up.*

Note that when you are working through this book, you will encounter many lines marked like that one. These takeaways summarize features, rules, recommendations, and so on. There is a list of these takeaways toward the end of the book, which you might use as a cheat sheet.

Requirements. To be able to profit from this book, you need to fulfill some minimal requirements. If you are uncertain about any of these, please obtain or learn them first; otherwise, you might waste a lot of time.

First, you can't learn a programming language without practicing it, so you *must* have a decent programming environment at your disposal (usually on a PC or laptop), and you *must* master it to some extent. This environment can be integrated (an IDE) or a collection of separate utilities. Platforms vary widely in what they offer, so it is difficult to advise on specifics. On Unix-like environments such as Linux and Apple's macOS, you will find editors such as `emacs` and `vim` and compilers such as `c99`, `c17`, `gcc`, and `clang`.

You must be able to do the following:

(1) Navigate your file system. File systems on computers are usually organized hierarchically in *directories*. You must be able to navigate through these to find and manipulate files.

(2) Edit programming text. This is different from editing a letter in a word-processing environment. Your environment, editor, or whatever it is called should have a basic understanding of the programming language C. You will see that if you open a C file (which usually has the file extension `.c`). It might highlight some keywords or help you indent your code according to the nestedness of `{}` brackets.

(3) Execute a program. The programs you will see here are very basic at first and will not offer you any graphical features. They need to be launched in the *command line*. An example of such a program launched that way is the *compiler*. On Unix-like environments, the command line is usually called a *shell* and is launched in a *console* or *terminal*.

(4) Compile programming text. Some environments provide a menu button or a keyboard shortcut for compilation. An alternative is to launch the compiler in the command line of a terminal. This compiler *must* adhere to recent standards; don't waste your time with a compiler that does not conform.

If you have never programmed before, this book will be tough. Knowing some of the following will help: Basic, C (historical revisions), C++, Fortran, R, bash, JavaScript, Java, MATLAB, Perl, Python, Scilab, and so on. But perhaps you have had some other programming experience, maybe even without noticing. Many technical specifications actually come in some sort of specialized language that can be helpful as an analogy—for example, *HTML* for web pages and *LaTeX* for document formatting.

You should have an idea of the following concepts, although their precise meanings may be a bit different in C than in the context where you learned them:

Variables: Named entities that hold values
Conditionals: Doing something (or not) subject to a precise condition
Iteration: Doing something repeatedly for a specified number of times or until a certain condition is met

Source code. Many of the programming code snippets presented in this book are publicly available (see `https://inria.hal.science/hal-03345464/document`). This allows you to view them in context and to compile and try them out. The archive also contains a `Makefile` with a description of the components that are needed to compile these files. It is centered around Linux or, more generally, POSIX systems, but it may also help you to find out what you need when you are on a different system.

Exercises and challenges. Throughout this book, you'll see exercises meant to get you thinking about the concepts being discussed. These are probably best done directly along with your reading. Then there is another category called "challenges." These are generally more demanding. You will need to do some research even to understand what they are about, and the solutions will not come all by themselves; they will require effort. They will take more time, sometimes hours or, depending on your degree of satisfaction with your work, even days. The subjects covered in these challenges are the fruit of my own personal bias toward "interesting questions" from my personal experience. If you have other problems or projects in your studies or work that cover the same ground, they should do equally well. The important aspect is to train yourself by first searching for help and ideas elsewhere and then to get your hands dirty and get things done. You will only learn to swim if you jump into the water.

Organization. This book is organized in *levels*, numbered from 0 to 3. The starting level 0, named Encounter, will summarize the very basics of programming with C. Its principal role is to remind you of the main concepts we have mentioned and familiarize you with the special vocabulary and viewpoints that C applies.[1] By the end of it, even if you don't have much experience in programming with C, you should be able to understand the structure of simple C programs and start writing your own.

The Acquaintance level 1 details most principal concepts and features such as control structures, data types, operators, and functions. It should give you a deeper understanding of what is 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 are not yet fully introduced.

The Cognition level 2 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.

[1] One of C's special viewpoints is that indexing starts at `0` and not at `1` as in Fortran.

The Experience level 3 then goes into detail about specific topics, such as performance, reentrancy, atomicity, threads, and type-generic programming. These are probably best discovered as you go, which is when you encounter them in the real world. Nevertheless, as a whole, they are necessary to round off the discussion 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.

Author. Jens Gustedt completed his studies in mathematics at the University of Bonn and Berlin Technical University. His research at that time covered the intersection between discrete mathematics and efficient computation. Since 1998, he has been working as a senior scientist at the French National Institute for Computer Science and Control (INRIA), first in the LORIA lab, Nancy, and, since 2013, in the ICube lab, Strasbourg, where he has been a deputy director since 2023.

Throughout his career, most of his scientific research has been accompanied by the development of software—at the beginning, mostly in C++ and then later exclusively in C. He serves AFNOR (the French standards organization) as an expert on the ISO committee JTC1/SC22/WG14. He was co-editor for the C standard document C17 and for the initial phase of C23. He also has a successful blog that deals with programming in C and related topics (`https://gustedt.wordpress.com`).

Contents

LEVEL 0

Encounter

Our mascot for this level is the magpie, one of the most intelligent nonhuman species on earth. They are capable of elaborate social rituals and usage of tools.

This first level of the book may be your first encounter with the programming language C. It provides you with a rough knowledge of C programs, their purposes, their structures, 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 intended to give you a general idea of what C is all about, open up questions, and promote ideas and concepts. These then will be explained in detail throughout the book.

1. Getting started

This section covers

- Introduction to imperative programming
- Compiling and running code

In this section, I will introduce you to a simple program that contains many of the constructs of the C language. If you already have programming experience, you may find some of the discussion feels like needless repetition. If you lack such experience, you might feel overwhelmed by the stream of new terms and concepts.

In either case, be patient. For those of you with programming or C experience, it's very possible that you will encounter subtle details you're not aware of or assumptions you have made about the language that are not valid. For those approaching programming for the first time, be assured that after approximately 10 pages, your understanding will have increased a lot, and you should have a much clearer idea of what programming represents.

An important bit of wisdom for programming in general, and for this book in particular, is summarized in the following quote from the *Hitchhiker's Guide to the Galaxy* by Douglas Adams [1986]:

Takeaway 1 #2 *Don't panic.*

It's not worth it. There are many cross references, links, and bits of side information in the text, and there is an index at the end. Follow those if you have a question. Or just take a break.

Programming in C is about having the computer complete some specific tasks. A C program does that by giving orders, much as we would express such orders in the imperative tense in many human languages, thus the term *imperative programming* for this particular way of organizing computer programs. To get started and see what I am talking about, consider our first program in listing 1.1, which corresponds to the `getting-started.c` source file in the source directory.

1.1. Imperative programming. You probably see that this code is a sort of language, containing some weird words like **`main`**, **`include`**, **`for`**, and so on, which are laid out and colored in a peculiar way and mixed with a lot of strange characters, numbers, and text ("Doing some work") that looks like ordinary English. It is designed to provide a link between us, the human programmers, and a machine, the computer, so we can tell it what to do: we give it "orders."

Takeaway 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 so you can easily cheat and ***jump***^{C} to more explanatory text, at your own risk.[1]

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. The visible result of running this program is to output five lines of text on the command terminal of your computer. On my computer, using this program looks something like this:

[1] Such special terms from C jargon are marked with a superscripted *C*, as shown here.

Listing 1.1. The first example of a C program

```c
/* This may look like nonsense, but really is -*- mode: C -*- */
#include <stdlib.h>
#include <stdio.h>

/* The main thing that this program does. */
int main(int argc, [[maybe_unused]] char* argv[argc+1]) {
  // Declarations
  double A[5] = {
    [0] = 9.0,
    [1] = 2.9,
    [4] = 3.E+25,
    [3] = .00007,
  };

  // Doing some work
  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;
}
```

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 the parts of the text in `getting-started.c` that this program outputs (***prints***^{C}, in C jargon): the part of line 17 between quotes. The real action happens between that line and line 20. C calls this a ***statement***^{C}, which is a bit of a misnomer. Other languages would use the term *instruction*, which describes the purpose better. This particular statement is a ***call***^{C} to a ***function***^{C} named **`printf`**:

. getting-started.c

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

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

- The funny-looking text (between the quotes) is a ***string literal***^{C} that serves as a ***format***^{C} for the output. Within the text are three markers (***format specifiers***^{C}) that indicate the positions in the output where numbers are to be inserted.

These markers start with the percent (`%`) character. This format also contains some special ***escape characters***^{C} that start with a backslash: `\t` and `\n`.

- After a comma character, we find the one-letter word `i`. The thing `i` stands for will be printed in place of the first format specifier, `%zu`.
- Another comma separates the next argument, `A[i]`. The thing this stands for will be printed in place of the second format specifier, the first `%g`.
- Last, again separated by a comma, appears `A[i]*A[i]`, corresponding to the last `%g`.

We will later explain what all of these arguments mean. Just remember that we identified the main purpose of the program (to print some lines on the terminal) and learned it "orders" the **`printf`** function to fulfill that purpose. The rest is some ***sugar***^{C} to specify which and how many numbers will be printed.

1.2. Compiling and running. As shown in the previous subsection, the program text expresses what we want our computer to do. As such, it is just another piece of text that we have written and stored somewhere on our hard disk, but the program text as such cannot 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 ***binary code***^{C} or ***executable***^{C}. What that translated program looks like and how this translation is done are much too complicated to cover in this book.[2] Unfortunately, this book will not be able to explain most of it; that would require yet another whole book. However, for the moment, we don't need to understand more deeply, as we have the tool that does all the work for us.

Takeaway 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, and your refrigerator doesn't speak the same "language" as your set-top box. In fact, that's one of the reasons for C to exist: C provides a level of abstraction for all the different machine-specific languages (usually referred to as an ***assembly language***^{C}).

Takeaway 1.2 #2 *A correct C program is portable between different platforms.*

In this book, we will put a lot of effort into writing "correct" C programs that ensure portability. Unfortunately, there are some platforms that claim to be C but do not conform to the latest standards, and some conforming platforms accept incorrect programs or provide extensions to the C standard that are not widely portable. So, running and testing a program on a single platform will not always guarantee portability.

It is the job of the compiler to ensure that the little program shown earlier (`getting-started.c`), once translated for the appropriate platform, will run correctly on your PC, your phone, your set-top box, and maybe even your refrigerator.

That said, if you have a POSIX system (such as Linux or macOS), there is a good chance that programs named `c99` or `c17` might be present and that it is, in fact, a C compiler. You could try to compile the example program using the following command:

[2] In fact, the *translation* itself is done in several steps that go from textual replacement to proper compilation and linking. Nevertheless, the tool that bundles all this is traditionally called a *compiler* and not a *translator*, which would be more accurate.

Terminal

```
> c17 -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 3] In the example line,

- `c17` is the compiler program.
- `-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}, which contains the C code we have written. Note that the `.c` extension at the end of the filename refers to the C programming language.
- `-lm` tells it to add some standard numerical functions if necessary; we will need those later on.

Now we can ***execute***^{C} our newly created ***executable***^{C}. Type

Terminal

```
> ./getting-started
```

and you should see exactly the same output as I showed you earlier. 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 didn't work, you will 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. [4]The names of compilers vary. Here are some common alternatives that might do the trick:

Terminal

```
> clang -std=c2x -Wall -lm -o getting-started getting-started.c
> gcc -std=c2x -Wall -lm -o getting-started getting-started.c
> icc -std=c2x -Wall -lm -o getting-started getting-started.c
```

The option `-std=c2x` names the standard version "C2x," which was meant for C23 before we knew it would be finished in 2023.

Some of these compilers, even if they are present on your computer, might not compile the program without complaining.[Exs 5]

With the program in listing 1.1, we have been presented with 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 partially works and may produce wrong or unreliable results. Therefore, let's look at the program in listing 1.2. It looks quite similar to the previous listing. .

If you run your compiler on this program, it should give you some ***diagnostic***^{C} information similar to this:

[Exs 3]Try the compilation command in your terminal.

[4]Installing a compiler is necessary if you have a system with a Microsoft operating system. Microsoft's native compilers are now catching up with C17, but I don't know about its plans for C23. Many features we discuss in this book might not work. For a discussion on alternative development environments, see Chris Wellons' blog entry "Four Ways to Compile C for Windows" (`https://nullprogram.com/blog/2016/06/13/`) might still be of interest.

[Exs 5]Start writing a text report about your tests with this book. Note down which command worked for you.

LISTING 1.2. An example of a C program with flaws

```c
/* This may look like nonsense, but really is -*- mode: C -*- */

/* The main thing that this program does. */
void main() {
  // Declarations
  int i;
  double A[5] = {
    9.0,
    2.9,
    3.E+25,
    .00007,
  };

  // Doing some work
  for (i = 0; i < 5; ++i) {
     printf("element␣%d␣is␣%g,␣\tits␣square␣is␣%g\n",
            i,
            A[i],
            A[i]*A[i]);
  }

  return 0;
}
```

Terminal

```
> gcc -std=c2x -Wall -o bad bad.c
bad.c:4:6: warning: return type of 'main' is not 'int' [-Wmain]
    4 | void main() {
      |      ^~~~
bad.c: In function 'main':
bad.c:16:6: warning: implicit declaration of function 'printf' [-Wimpl  it-function-decl ...
   16 |      printf("element %d is %g, \tits square is %g\n", /*@\labe  printf-start-bad ...
      |      ^~~~~~
bad.c:1:1: note: include '<stdio.h>' or provide a declaration of 'prin  '
  +++ |+#include <stdio.h>
    1 | /* This may look like nonsense, but really is -*- mode: C -*-
bad.c:16:6: warning: incompatible implicit declaration of built-in fun  ion 'printf' [ ...
   16 |      printf("element %d is %g, \tits square is %g\n", /*@\labe  printf-start-bad ...
      |      ^~~~~~
bad.c:16:6: note: include '<stdio.h>' or provide a declaration of 'pri  f'
bad.c:22:10: warning: 'return' with a value, in function returning voi  [-Wreturn-type]
   22 |   return 0;                                              /*@\labe  main-return-ba ...
      |          ^
bad.c:4:6: note: declared here
    4 | void main() {
      |      ^~~~
```

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

Clang is even more picky than GCC and gives us similar diagnostics:

Terminal

```
> clang -std=c2x -Wall -o bad bad.c
bad.c:4:1: error: 'main' must return 'int'
    4 | void main() {
      | ^~~~
      | int
bad.c:16:6: error: call to undeclared library function 'printf' with t  e 'int (const ch ...
   16 |      printf("element %d is %g, \tits square is %g\n", /*@\labe  printf-start-badly}*/
      |      ^
bad.c:16:6: note: include the header <stdio.h> or explicitly provide a  eclaration for ' ...
bad.c:22:3: error: void function 'main' should not return a value [-Wr  urn-type]
   22 |   return 0;                                              /*@\labe  main-return-badly}*/
      |   ^      ~
3 errors generated.
```

Notice that Clang, unlike GCC, did not produce an executable. It considers all three detected problems to be fatal errors and refuses to carry on. Consider this to be a feature.

Both gave us three diagnostics: they expect a different return type for **`main`**, they expected us to have a line such as line 3 from listing 1.1 to specify where the **`printf`** function comes from, and they detect that the **`return`** in line 22 is not correct for the specification of **`main`** as it is given.

Depending on your platform, you can force your compiler to reject programs that produce such diagnostics. For GCC, such a command-line option would be `-Werror`, and then it would behave as we saw with Clang.

So, the two differences between listings 1.1 and 1.2 turned a good, standards-conforming, portable program into a bad one. We also saw that the compiler was there to help us. It nailed the problem down to the lines in the program that cause trouble. With a bit of experience, you will be able to understand what it is telling you.[Exs 6] [Exs 7]

Takeaway 1.2 #3 *A C program should compile cleanly without warnings.*

[Exs 6] Correct listing 1.2 step by step. Start from the first diagnostic line, fix the code mentioned there, recompile, and so on until you have a flawless program.

[Exs 7] There is a third difference between the two programs that was not yet mentioned. Find it.

Summary

- C is designed to give orders to computers. Therefore, it mediates between us (the programmers) and computers.
- C must be compiled to be executed. The compiler provides the translation between the language that we understand (C) and the specific needs of the particular platform.
- C gives a level of abstraction that provides portability. One C program can be used on many different computer architectures.
- The C compiler is there to help you. If it warns you about something in your program, listen to it.

2. The principal structure of a program

This section covers

- C grammar
- Declaring identifiers
- Defining objects
- Instructing the compiler with statements

Compared to our little examples in the previous section, real programs will be more complicated and contain additional constructs, but their structure will be very similar. Listing 1.1 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 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: declarative parts (what things are), definitions of objects (where things are), and statements (what things are 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 as follows:

Special words: In listing 1.1, we used the following special words:[8]

`#include int maybe_unused char void double for return`

In program text in this book, most of them will be printed in black bold. These special words represent concepts and features that the C language imposes and that cannot be changed.

Punctuation: C uses several types of punctuation to structure the program text.

- There are six kinds of brackets: `{...}`, `(...)`, `[...]`, `[[...]]`, `/*...*/`, and `<...>`. Brackets *group* certain parts of the program together and should always come in pairs. Fortunately, the `<...>` brackets are rare in C and are only used as shown in our example, on the same logical line of text. The other five are not limited to a single line; their contents might span several lines, as they did when we used **`printf`** earlier.
- There are two different separators or terminators: comma and semicolon. When we used **`printf`**, commas *separated* the four arguments of that function. On 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 to C is that the same punctuation characters are used to express different concepts. For example, the pairs `{}` and `[]` are each used for three different purposes in listing 1.1.[Exs 9]

Takeaway 2.1 #1 *Punctuation characters can be used for several different purposes.*

Comments: The construct `/* ... */` that we saw earlier tells the compiler that everything inside it is a *comment*; see, for example, line 5:

[8] In C jargon, these are ***directives***^{C}, ***keywords***^{C}, ***attributes***^{C}, and ***reserved***^{C} identifiers.

[Exs 9] Find the different uses of these two sets of brackets.

getting-started.c

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

Comments are ignored by the compiler, so it is the perfect place to explain and document your code. Such in-place documentation can (and should) greatly improve the readability and comprehensibility of your code. Another form of comment is the so-called C++-style comment, as on line 15. These are marked with `//`. C++-style comments extend from the `//` to the end of the line.

Literals: 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*.

Identifiers: Identifiers 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. Among other things, they may refer to

- ***Data objects***^{C} (such as `A` and `i`). These are also referred to as ***variables***^{C}.
- ***Type***^{C} aliases, such as **`size_t`**, that specify the "kind" 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, such as **`main`** and **`printf`**.
- Constants, such as **`EXIT_SUCCESS`**.

Functions: 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 ***function body***^{C} (indicated by `{ ... }` of a compound statement) 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 C programs, as we will encounter: it must always be present since it is the starting point of the program's execution.

Operators: Of the numerous C operators, our program only uses a few:

- = for ***initialization***^{C} and ***assignment***^{C}
- < for comparison
- ++ to *increment* a variable (to increase its value by `1`)
- ⋆ to multiply two values

Attributes: Attributes such as `[[maybe_unused]]` are placed into double square brackets as shown and provide some supplemental information to the principle structure of the program.[C23]

Just as in natural languages, the lexical elements and the grammar of C programs that we have seen here have to be distinguished from the actual meaning these constructs convey. In contrast to natural languages, this meaning is rigidly specified and usually leaves no room for ambiguity. In the following subsections, we will dig into the three main semantic categories that C distinguishes: declarations, definitions, and statements.

2.2. Declarations. Before we may use a particular identifier in a program, we have to give the compiler a ***declaration***^{C} that specifies what that identifier is supposed to represent. In this way, identifiers differ from keywords: keywords are predefined by the language and must not be declared or redefined.

[C23] This feature is new in C23, so your compiler might not yet implement it.

Takeaway 2.2 #1 *All identifiers in a program have to be declared.*

Several of the identifiers we use are effectively declared in our program: **main**, argc, argv, A, and i. Later on, we will see where the other identifiers (**printf**, **size_t**, and **EXIT_SUCCESS**) come from. We already mentioned the declaration of the **main** function. All five declarations, in isolation as "declarations only," look like this:

```
int main(int, char*[]);
int argc;
[[maybe_unused]] char* argv[];
double A[5];
size_t i;
```

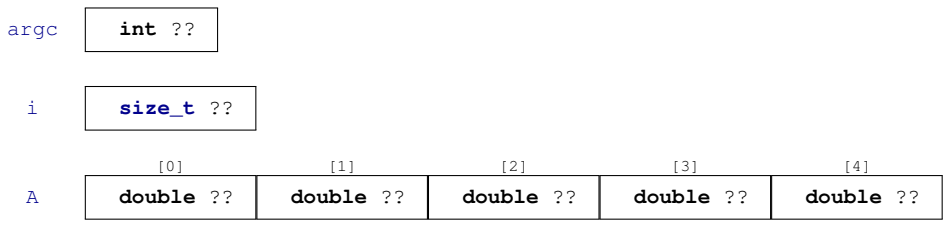
These five declarations follow a pattern. Each has an identifier (**main**, argc, argv, A, or i), and a specification of certain properties that are associated with that identifier:

- i is of ***type***^{C} **size_t**.
- argc is of type **int**.
- **main** is additionally followed by parentheses, `(...)`, 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.
- argv is also followed by brackets, so it also has the properties of an array. The attribute `[[maybe_unused]]` indicates that it is possibly unused, and indeed, we don't see the word argv elsewhere in this program code. There are argc+1 elements of a type denoted by **char***.

Each of these declarations starts with a ***type***^{C} (here, **int**, **char***, **double**, and **size_t**). We will see later what that represents. For the moment, it is sufficient to know that it specifies that four of the identifiers (**main**, argc, A, and i), when used in the context of a statement, will provide some sort of numbers.

For the other, argv, the * at the end of **char*** indicates that it is a ***pointer***^{C}; pointers, though a principal feature of the C language, will only appear much later in this book.

The declarations of argc, i and A declare ***variables***^{C}, which are named items that allow us to store ***values***^{C}. They are best visualized as a kind of box that may contain a "something" of a particular type:



Conceptually, it is important to distinguish the box itself (the *object*), the specification (its *type*), the box contents (its *value*), and the name or label that is written on the box (the *identifier*). In such diagrams, we put `??` if we don't know the actual value of an item.

For the other three identifiers, **`printf`**, **`size_t`**, and **`EXIT_SUCCESS`**, we don't see any declarations. They are, in fact, predeclared identifiers. However, as we saw when we tried to compile listing 1.2, 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 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

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

Because the specifics of these predeclared features are of minor importance, this information is normally hidden from you in ***include files***^{C} or ***header files***^{C}. If you need to know their semantics, it is usually a bad idea to look them up in the corresponding files, as these tend to be barely readable. Instead, search in the documentation that comes with your platform. For the brave, I always recommend a look into the current C standard, as that is where they all come from. For the less courageous, the following commands may help:

Terminal

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

A declaration only describes a feature but does not create it, so repeating a declaration does not do much harm but adds redundancy.

Takeaway 2.2 #2 *Identifiers may have several consistent declarations.*

Clearly, it would become really confusing (for us and the compiler) if there were several contradicting declarations for the same identifier in the same part of the program, so generally this is not allowed. C is quite specific about what "the same part of the program" is supposed to mean: the ***scope***^{C} is a part of the program where an identifier is ***visible***^{C}.

Takeaway 2.2 #3 *Declarations are bound to the scope in which they appear.*

The scopes of identifiers are unambiguously described by the grammar. In listing 1.1, we have declarations in different scopes:

- `A` is visible inside the definition of the function **`main`**, starting at its declaration on line 8 and ending with the closing `}` on line 24 of the innermost `{ ... }` compound statement that contains that declaration.
- `i` has more restricted visibility. It is bound to the **`for`** construct in which it is declared. Its visibility reaches from that declaration on line 16 to the end of the `{ ... }` compound statement that is associated with the **`for`** on line 21.
- **`main`** is not enclosed in any other compound statement, so it is visible from its declaration onward until the end of the file.
- `argc` and `argv` are not enclosed inside `{ ... }`, but they are inside the `( ... )` that marked **`main`** as being a function. They are the ***parameters***^{C} of the

function, and their scope starts at their respective declaration and then spans the whole { ... } body of their function (here, of **main**).

The first two and the last types of scope are called ***block scope***^{C} with a ***block***^{C} being a structure in the grammar that encapsulates such declarations. A function like **main**, together with its parameter list (enclosed in ()) and the whole body (enclosed in {}), forms a single block of its own. The **for** construct forms a ***primary block***^{C}, and the loop body (usually also given with surrounding { ... }) forms a ***secondary block***^{C}. You can see that blocks are ***nested***^{C}: the block of **main** contains the primary block of the **for** loop, which, in turn, contains its secondary block.

The third type of scope, as used for the name **main** itself, which is not inside a (...) or { ... } pair, is called ***file scope***^{C}. Identifiers in file scope are often referred to as ***globals***^{C}.

So, our seemingly simple program has four nested levels of scopes: file scope and three nested blocks.

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}.

Takeaway 2.3 #1 *Declarations specify identifiers, whereas definitions specify objects.*

We will later see that things are a little more complicated in real life, but for now we can make the simplification that we will always initialize our variables. An *initialization* is a grammatical construct that augments a declaration and provides an initial value for the object. For instance,

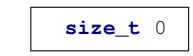
```
size_t i = 0;
```

is a declaration of `i` such that the initial value is `0`.

In C, such a declaration with an initializer also *defines* the object with the corresponding name: that is, it instructs the compiler to provide storage in which the value of the variable can be stored.

Takeaway 2.3 #2 *An object is defined at the same time it is initialized.*

Our box visualization can now be completed with a value, `0` in this example:

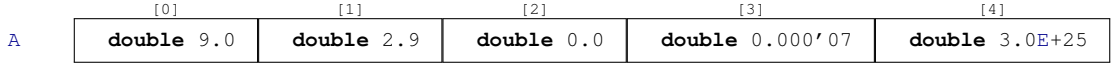


`A` is a bit more complex because it has several components:

. getting-started.c

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

This initializes the `5` items in `A` to the values `9.0`, `2.9`, `0.0`, `0.000'07`, and `3.0E+25`, in that order:



The form of the initializer we see here is called ***designated***^{C}: a pair of brackets with an integer designates which item of the array is initialized with the corresponding value. For example, `[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`.[11]

Takeaway 2.3 #3 *Missing elements in initializers default to `0`.*

You might have noticed that array positions, ***indices***^{C}, start with `0` for the first element, not `1`. Think of an array position as the distance of the corresponding array element from the start of the array.

Takeaway 2.3 #4 *For an array with `n` elements, the first element has index `0`, and 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:

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

In our examples so far, we have seen names for two different features: ***objects***^{C} (`i` and `A`) and ***functions***^{C} (**`main`** and **`printf`**). In contrast to object or function declarations, where several are allowed for the same identifier, definitions of objects or functions must be unique. That is, for a C program to be operational, any object or function used must have a definition (otherwise, the execution would not know where to look for them), and there must be no more than one definition (otherwise, the execution could become inconsistent).

Takeaway 2.3 #5 *Each object or function must have exactly one definition.*

2.4. Statements. The second part of the **`main`** function consists primarily 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

```
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;
```

We have already discussed the lines that correspond to the call to **`printf`**. There are also other types of statements: **`for`** and **`return`** and an increment operation indicated by the ***operator***^{C} `++`. In the following subsection, we will go a bit into the details of three categories of statements: *iterations* (do something several times), *function calls* (delegate execution somewhere else), and *function returns* (resume execution from where a function was called).

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

2.4.1. *Iteration.* The **for** statement tells the compiler that the program should execute the **printf** line a number of times. This is the simplest form of ***domain iteration***^{C} that C has to offer. It has four different parts. The secondary block (here, a compound statement marked by `{ ... }`) that follows the **for** `( ... )` is the code that is to be repeated; it is also called the ***loop body***^{C}. The other three parts are those inside the `( ... )` part, divided by semicolons:

(1) The declaration, definition, and initialization of the ***loop variable***^{C} `i`, which we already discussed. This initialization is executed once before any of the rest of the entire **for** statement.
(2) A ***loop condition***^{C}, `i < 5` specifies how long the **for** iteration should continue. This 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 after each iteration. In this case, it increases the value of `i` by `1` each time.

If we put all of these parts together, we ask the program to perform the code in the secondary block five 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 **for** is the easiest, cleanest, and best tool for the task.

Takeaway 2.4.1 #1 *Domain iterations should be coded with a* ***for*** *statement.*

A **for** statement can be written in several other ways. 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: to help an occasional reader and the compiler understand your code, it is important to know that this variable has the special meaning of an iteration counter for that given **for** loop.

Takeaway 2.4.1 #2 *The loop variable should be defined in the initial part of a* ***for****.*

2.4.2. *Function calls.* *Function calls* are special statements that suspend the execution of the current function (at the beginning, this is usually **main**) and then hand over control to the named function. In our example,

. getting-started.c

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

the called function is **printf**. A function call usually provides more than just the name of the function; it also provides *arguments*. Here, these are the long chain of characters, `i`, `A[i]`, and `A[i]*A[i]`. The *values* of these arguments are passed over to the function. In this case, these values are the information that is printed by **printf**. The emphasis here is on *value*: although `i` is an argument, **printf** will never be able to change `i` itself. Such a mechanism is called *call by value*. Other programming languages also have *call by reference*, a mechanism where the called function can change the value of a variable. C does not implement pass by reference; instead, it has another mechanism to pass the control of a variable to another function: by taking addresses and transmitting pointers. We will see this mechanism much later.

2.4.3. *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 on line 17, the execution of the statements in **main** is temporarily suspended. The execution continues of the **printf** function until a **return** is encountered. After the return from **printf**, the execution of the statements in **main** continues from where it stopped.

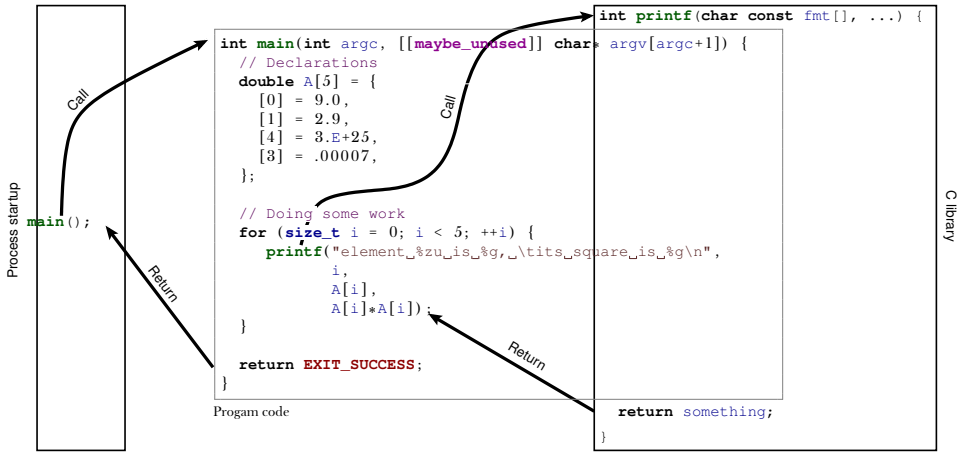


FIGURE 2.1. Execution of a small program

Figure 2.1 shows a schematic view of the execution of our little program: its *control flow*. First, a process-startup routine (on the left) 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.

Summary

- C distinguishes the lexical structure (the punctuators, identifiers, and numbers), the grammatical structure (syntax), and the semantics (meaning) of programs.
- All identifiers (names) must be declared so we know the properties of the concept they represent.
- All objects (things that we deal with) and functions (methods that we use to deal with things) must be defined; that is, we must specify how and where they come to be.
- Statements indicate how things are going to be done: iterations (`for`) repeat variations of certain tasks, function calls (`printf(...)`) delegate a task to a function, and function returns (`return something;`) go back where we came from.

LEVEL 1

Acquaintance

Our mascot for this level, the common raven, is a very sociable corvid and known for its problem-solving capacity. Ravens organize in teams and have been observed playing even as adults.

This level will acquaint you with the C programming language; that is, it will 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 and 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 these sections, you should be able to write short code for everyday needs that is not extremely sophisticated but useful and portable.

Buckle up

In many ways, C is a permissive language; programmers are 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 level 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 learned your C basics in the last millennium or if you were introduced to C on a platform that wasn't upgraded to current ISO C for years.

Experienced C programmers: If you already have some experience with C programming, what follows may take some getting used to or even provoke allergic reactions. If you happen to break out in hives when you read some of the code here, take a deep breath and try to relax, but please *do not skip* these pages.

Inexperienced C programmers: If you are not an experienced C programmer, much of the following discussion may be a bit over your head; for example, we may use terminology that you have not yet even heard of. If so, this is a digression for you, and you may skip to the start of section 3 and come back later when you feel a bit more comfortable. But be sure to do so before the end of this level.

Some of "getting used to" our approach on this level may concern the emphasis and ordering in which we present the material:

- 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 (section 6.1.4), then with their state (being valid or not, section 6.2), and then, on the next level, (section 11), using their entire potential.
- We will focus on the use of arrays whenever possible instead.

You might also be surprised by some style considerations that we will discuss in the following points. On the next level, we will dedicate an entire section (section 9) to these questions, so please be patient and accept them for the moment as they are.

: We bind type modifiers, qualifiers, and attributes 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 or attributes and write

```
char const* const path_name[[deprecated]];
```

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. The attribute `[[deprecated]]` clearly attaches to the identifier `path_name`.

(1) We do not use continued declarations.: They obfuscate the bindings of type declarators. For example,

```
unsigned const*const a, b;
```

Here, `b` has type **`unsigned const`**; that is, the first **`const`** goes to the type, and the second **`const`** only goes to the declaration of `a`. Such rules are highly confusing, and you have more important things to learn.

(2) We use array notation for pointer parameters.: We do so wherever these assume that the pointer can't be null. Some examples are as follows:

```
/* These emphasize that the arguments cannot be null. */
size_t strlen(char const string[static 1]);
int main(int argc, char* argv[argc+1]);
/* Compatible declarations for the same functions.  */
size_t strlen(const char *string);
int main(int argc, char **argv);
```

The first example 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 `argc+1` pointers to **`char`**: referring to the program name, to `argc-1` program arguments, and to one null pointer that terminates the array.

The second set of declarations only adds additional equivalent declarations for features that are already known to the compiler. Because these are declarations and not definitions, such a redeclaration is allowed, but it doesn't add new information in this case.

(3) We use function notation for function pointer parameters.: Along the same lines, we do so whenever we know that a function pointer can't be null:

```
/* This emphasizes that the ``handler'' argument cannot be null. */
int atexit(void handler(void));
/* Compatible declaration for the same function.            */
int atexit(void (*handler)(void));
```

Here, the first declaration of **`atexit`** emphasizes that, semantically, it receives a function named `handler` as an argument and that a null function pointer is not allowed. Technically, the function parameter `handler` is "rewritten" to a function pointer, much as array parameters are rewritten to object pointers, but this is of minor interest for a description of the functionality.

Note, again, that the previous code is valid as it stands and that the second declaration just adds an equivalent declaration for **`atexit`**.

(4) 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 the *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 that in another loop, so we declare the variable within the **`for`** to ensure it stays within the loop's primary block.

(5) 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.

See the following code snippet as an example.

```
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);
  }
  return EXIT_SUCCESS;
}
```

(6) We use digit separators for numbers.: The human eye is not very good at perceiving the magnitude of numbers that have many digits. In running text, we therefore use a comma as a thousands separator for large decimal numbers, and we even extend this form of notation to digits after the period, as in, for example, 10,035.677,789. A notable exception from this rule is years, where we do not apply a thousands separator.

Since C23, there is also a *digit separator* for number literals, namely a `'` that appears directly between two consecutive digits, such as in `10'035.677'789`. We use that for all number literals by grouping together three decimal digits (for thousands), four hexadecimal digits (for double bytes), and eight binary digits (for the bits in a byte).

3. Everything is about control

This section covers

- Conditional execution with **`if`**
- Iterating over domains
- Making multiple selections

In our introductory example, listing 1.1, we saw two different constructs that allowed us to control the flow of a program's 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, if so, when the secondary block (`{ printf(...) }`) is executed. C has five conditional *control statements*: **`if`**, **`for`**, **`do`**, **`while`**, and **`switch`**. We will look at these statements in this section: **`if`** introduces a *conditional execution* depending on a Boolean expression; **`for`**, **`do`**, and **`while`** are different forms of *iterations*; and **`switch`** is a *multiple selection* based on an integer value.

C has some other conditionals that we will discuss later: the ***ternary operator***^{C}, denoted by an expression in the form `cond ? A : B` (section 4.5); the compile-time preprocessor conditionals **`#if`**, **`#ifdef`**, **`#ifndef`**, **`#elif`**, **`#elifdef`**, **`#elifndef`**, **`#else`**, **`#endif`** (section 8.1.5); and type generic expressions denoted with the keyword **`_Generic`** (section 18).

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 the example, `i > 25` is called the ***controlling expression***^{C}, and the part in `{ ... }` is called the ***secondary block***^{C}.

On the surface, this form of an **`if`** statement resembles the **`for`** statement that we already encountered. But it works differently than that: there is only one part inside the parentheses, and that determines whether the secondary block is run once or not at all.

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

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

It has another secondary block that is executed if the controlling condition is not fulfilled. Syntactically, this is done by introducing another keyword, **`else`**, which separates the two secondary blocks.

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

```
if (condition) secondary-block0
else secondary-block1
```

The possibilities for `condition` (the controlling expression) 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 4.4.2.

The simplest of such `condition` specifications in an **`if`** statement can be seen in the following example, in a variation of the **`for`** loop from listing 1.1:

```
for (size_t i = 0; i < 5; ++i) {
  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 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 of a numerical `condition`:

Takeaway 3.1 #1 *The value `0` represents logical false.*

Takeaway 3.1 #2 *Any value different from `0` represents logical true.*

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

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

as

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

Which of these two versions is more readable is a question of ***coding style***^{C} and can be subject to fruitless debates. While the first might be easier for occasional readers of C code to read, the latter is often preferred in projects that assume some knowledge about C's type system.

The type **`bool`** should be used to store truth values. Its values are **`false`** and **`true`**. Technically, **`false`** is just a value `0` of type **`bool`** and **`true`** the value `1`. It's important to use **`false`** and **`true`** (and not the numbers) to emphasize that a value is to be interpreted as a condition. We will learn more about the **`bool`** type in section 5.7.4. Note that before C23, to use **`bool`** and its literals **`false`** and **`true`**, you had to use the `<stdbool.h>` header. This is now obsolete, and you should only use it if you suspect that your code may be compiled on an older platform.

<stdbool.h>

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 = ...;
...
```

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

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

as

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

Takeaway 3.1 #3 *Don't compare to* `0`, **`false`**, *or* **`true`**.

Using either the truth value directly (such as `i` above) or the negation (denoted as `!i` in C) makes your code clearer and illustrates one of the basic concepts of the C language.

Takeaway 3.1 #4 *All scalars have a truth value.*

Here, ***scalar***^{C} types include all the numerical types such as **`size_t`**, **`bool`**, and **`int`** that we already encountered and ***pointer***^{C} types; see table 3.1 for the types that are frequently used in this book. We will come back to them in section 6.2.

TABLE 3.1. Scalar types used in this book

Level	Name	Other	Category	Where	**`printf`**
0	**`size_t`**		Unsigned	`<stddef.h>`	`"%zu" "%zx"`
0	**`double`**		Floating	Built in	`"%e" "%f" "%g" "%a"`
0	**`signed`**	**`int`**	Signed	Built in	`"%d"`
0	**`unsigned`**		Unsigned	Built in	`"%b" "%o" "%u" "%x"`
0	**`bool`**	**`_Bool`**	Unsigned	Built in (since C23)	`"%d"` as 0 or 1
1	**`ptrdiff_t`**		Signed	`<stddef.h>`	`"%td"`
1	**`char const*`**		String	Built in	`"%s"`
1	**`char`**		Character	Built in	`"%c"`
1	**`void*`**		Pointer	Built in	`"%p"`
2	**`unsigned char`**		Unsigned	Built in	`"%hhu" "%02hhx"`

3.2. Iterations. Previously, we encountered the **`for`** statement 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 (clause1; condition2; expression3) secondary-block
```

This statement is actually quite generic. Usually, `clause1` is an assignment expression or a variable definition. It serves to state an initial value for the iteration domain. `condition2` tests whether the iteration should continue. Then, `expression3` updates the iteration variable used in `clause1`. It is performed at the end of each iteration. Here's some advice:

- Because we want iteration variables to be defined narrowly in the context for a **`for`** loop (*cf.* takeaway 2.4.1 #2), `clause1` should, in most cases, be a variable definition.
- Because **`for`** is relatively complex with its four different parts and not easy to capture visually, the secondary block should usually be a compound statement indicated by `{ ... }`.

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** looks as though it would go on forever, but it actually counts down from `9` to `0`. In fact, in the next section, we will see that "sizes" in C (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, **while** and **do**:

```
while (condition) secondary-block
do secondary-block while(condition);
```

The following example shows a typical use of **while**. It implements the *Heron approximation* to compute the multiplicative inverse $\frac{1}{x}$ of a number x:

```
#include <tgmath.h>

constexpr double ε = 1E-9;           // Desired precision
...
double const a = 34.0;
double x = 0.5;
while (fabs(1.0 - a*x) >= ε) {       // Iterates until close
  x *= (2.0 - a*x);                  // Heron approximation
}
```

It iterates as long as the given condition evaluates true, namely, as long as the absolute value of the difference between `1.0` and the computed product is smaller than a named constant called ε, which represents the desired minimum precision of the computation.[3] The **do** loop is very similar, except that it checks the condition *after* the secondary block:

```
do {                                 // Iterates
  x *= (2.0 - a*x);                  // Heron approximation
} while (fabs(1.0 - a*x) >= ε);      // Iterates until close
```

This means if the condition immediately evaluates to **false**, a **while** loop will not run its secondary block at all, but the **do** loop will unconditionally run its block at least once before ever looking at the condition. As with the **for** statement, with **do** and **while** it is advisable to use the `{ ... }` variants. There is also a subtle syntactical difference between the two: **do** always needs a semicolon after **while** `(condition)`

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

[3] We will see the details of C23's **constexpr** construct in section 5.6.5.

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 10.2.1.

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 secondary block after the **break** statement:

```
while (true) {
  double prod = a*x;
  if (fabs(1.0 - prod) < ε) {        // Stops 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 **while** then becomes trivial. The same thing can be done using **for**, and there is a tradition among C programmers to write it as follows:

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

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

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

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

<tgmath.h>

In these examples, we use a standard macro **fabs**, which comes with the `<tgmath.h>` header.[4] It calculates the absolute value of a **double**. Listing 3.1 is a complete program that implements the same algorithm, where **fabs** has been replaced by several explicit comparisons against certain fixed numbers: for example, `eps1m24` defined to be $1-1\cdot 2^{-24}$ or `eps1p24` as $1+1\cdot 2^{-24}$. We will see later (section 5.3) how the literals **0x**`1P-24` (standing for the value 2^{-24}) and similar used in these definitions work and how they interact with the **constexpr** construct (section 5.6.5).

LISTING 3.1. Computing multiplicative inverses of numbers

[4] "tgmath" stands for *type generic mathematics* and provides interfaces to numerical functions.

```
2  #include <stdio.h>
3
4  /* lower and upper iteration limits centered around 1.0 */
5  constexpr double eps1m01 = 1.0 - 0x1P-01;
6  constexpr double eps1p01 = 1.0 + 0x1P-01;
7  constexpr double eps1m24 = 1.0 - 0x1P-24;
8  constexpr double eps1p24 = 1.0 + 0x1P-24;
9
10 int main(int argc, char* argv[argc+1]) {
11   for (int i = 1; i < argc; ++i) {        // process args
12     double const a = strtod(argv[i], nullptr);  // arg -> double
13     double x = 1.0;
14     for (;;) {                    // by powers of 2
15       double prod = a*x;
16       if (prod < eps1m01) {
17         x *= 2.0;
18       } else if   (eps1p01 < prod) {
19         x *= 0.5;
20       } else {
21         break;
22       }
23     }
24     for (;;) {                    // Heron approximation
25       double prod = a*x;
26       if ((prod < eps1m24) || (eps1p24 < prod)) {
27         x *= (2.0 - prod);
28       } else {
29         break;
30       }
31     }
32     printf("heron:␣a=%.5e,\tx=%.5e,\ta*x=%.12f\n",
33            a, x, a*x);
34   }
35   return EXIT_SUCCESS;
36 }
```

In the first phase, the product of the current number under investigation `a` with the current estimate `x` is compared to 1.5 and 0.5, and then `x` is multiplied by 0.5 or 2 until the product is close to 1. Then, the Heron approximation, as shown in the code, is used in a second iteration to close in and compute the multiplicative inverse with a high degree of accuracy.

The overall 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 this:

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 5][Exs 6][Exs 7]

<stdlib.h>

CHALLENGE 1 (Sequential sorting algorithms). *Can you do*

(1) A merge sort (with recursion)
(2) A quick sort (with recursion)

on arrays with sort keys such as **`double`** *or strings to your liking?*
Nothing is gained if you don't know whether your programs are correct. Therefore, can you provide a simple test routine that checks whether the resulting array really is sorted?
This test routine should just scan once through the array and should be much, much faster than your sorting algorithms.

3.3. Multiple selection. The last control statement that C has to offer is the **`switch`** statement, which is another ***selection***^{C} statement. 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");
}
```

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 special case labeled **`default`**. The default case is triggered if `arg` doesn't match any of the **`case`** values.[Exs 8]

<stdio.h>

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

[Exs 5] Analyze listing 3.1 by adding **`printf`** calls for intermediate values of `x`.
[Exs 6] Describe the use of the parameters `argc` and `argv` in listing 3.1.
[Exs 7] Print out the values of `eps1m01` and observe the output when you change them slightly.
[Exs 8] Test the example **`switch`** statement in a program. See what happens if you leave out some of the **`break`** statements.

```
switch (expression) secondary-block
```

and its semantics are quite straightforward: the **case** and **default** labels serve as ***jump targets***^{C}. According to the value of the `expression`, control continues at the statement that is labeled accordingly. If we hit a **break** statement, the whole **switch** statement in which it appears is terminated, and then program control is transferred to the next statement after the **switch**. By that specification, **switch** statements can 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:               // prior to C23 this needed an extra ";"
}
```

Once we have jumped into the secondary block, 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` is a triangle with three lines:

Terminal

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

The structure of a **switch** can be more flexible than **if**–**else**, but it is restricted in other ways.

Takeaway 3.3 #1 ***case*** *values must be integer constant expressions.*

Takeaway 3.3 #2 ***case*** *values must be unique for each* ***switch*** *statement.*

In section 5.6.2, 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` in the previous example. In particular, variables such as `count` are only allowed in the **switch** part, not in 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:

```
switch (x) {
  unsigned tmp = 45;
  ...
  case 0: printf("the␣temp␣is␣%u\n", tmp); // tmp may be uninitialized
}
```

Such variables are valid, but their initializer may not have been seen, for the example when `x` is zero. So in this case a program execution where `x` happens to be zero would be erroneous. Initialization (or lack thereof) will be discussed in more detail in section 5.5.

Takeaway 3.3 #3 ***case*** *labels must not jump beyond a variable definition.*

Prior to C23, the placement of the **case** labels was even more restricted. They necessarily had to mark a statement: labels in front of declarations or just in front of the closing brace `}` of a compound statement were not valid.

CHALLENGE 2 (Numerical derivatives). *Something we'll deal with a lot is the concept of numerical algorithms. To get your hands dirty, see whether you can implement the numerical derivative* `double f(double x)` *of a function* `double F(double x)`.
Implement this with an example `F` *for the function you use for this exercise. A good primary choice for* `F` *would be a function for which you know the derivative, such as* **`sin`**, **`cos`**, *or* **`sqrt`**. *Doing so allows you to check your results for correctness.*

CHALLENGE 3 (π). *Compute the N first decimal places of π.*

Summary

- Numerical values can be directly used as conditions for **`if`** statements; 0 represents "false," and all other values are "true."
- There are three different iteration statements: **`for`**, **`do`**, and **`while`**. **`for`** is the preferred tool for domain iterations.
- A **`switch`** statement performs multiple selections. One **`case`** runs into the next if it is not terminated by a **`break`**.

4. Expressing computations

This section covers

- Performing arithmetic
- Modifying objects
- Working with booleans
- The ternary operator
- Setting the evaluation order

We've already used some simple examples of ***expressions***^{C}. These are code snippets that compute a value based on other values. The simplest such expressions are arithmetic expressions, which are similar to those we learned in school. But there are others, notably, comparison operators such as == and `!=`, which we saw earlier.

4.1. Operands and operators. Here, the values and objects on which we will do these computations will be mostly of the type **`size_t`**, which we have already met. 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`**.

Takeaway 4.1 #1 *The type **`size_t`** represents values in the range `[0, SIZE_MAX]`.*

The value of **`SIZE_MAX`** is quite large. Depending on the platform, it is one of

$$\begin{aligned} 2^{16}-1 &= 65{,}535 \\ 2^{32}-1 &= 4{,}294{,}967{,}295 \\ 2^{64}-1 &= 18{,}446{,}744{,}073{,}709{,}551{,}615 \end{aligned}$$

The first value is a minimal requirement; nowadays, such a small value would only occur on some embedded platforms. The other two values are much more commonly used today: the second is still found on some PCs and laptops, and the large majority of newer platforms have the third. Such a choice of value is large enough for calculations that are not too sophisticated. The standard header `<stdint.h>` provides **`SIZE_MAX`** such that you don't have to figure out that value yourself or specialize your program accordingly.

<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}. Symbols and combinations like + and `!=` 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 following tables:

- Table 4.1 lists the operators that operate on values.
- Table 4.2 lists those that operate on objects.
- Table 4.3 lists those that operate on types.

To work with these, you may have to jump from one table to another. For example, if you want to work out an expression such as `a + 5`, where `a` is some variable of type **`unsigned`**, you first have to go to the third line in table 4.2 to see that `a` is evaluated. Then, you can use the third line in table 4.1 to deduce that the value of `a` and `5` are combined in an arithmetic operation: a +. Don't be frustrated if you don't understand everything in these tables. A lot of the concepts that are mentioned have not yet been introduced; they are listed here to form a reference for the entire book.

TABLE 4.1. Value operators: The Form column gives the syntactic form of the operation, where `@` represents the operator and `a` and possibly `b` denote values that serve as operands. For arithmetic and bit operations, the type of the result is a type that reconciles the types of `a` and `b`. For some of the operators, the Nick column gives an alternative form of the operator or lists a combination of operators that has special meaning. Most of the operators and terms will be discussed later.

Operator	Nick	Form	Type restriction a	Type restriction 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`**	`a@b`	Integer	Integer	Integer	Bit
`|`	**`bitor`**					
`^`	**`xor`**					
`<< >>`		`a@b`	Integer	Positive	Integer	Bit
`== < > <= >=`		`a@b`	Scalar	Scalar	`0,1`	Comparison
`!=`	**`not_eq`**	`a@b`	Scalar	Scalar	`0,1`	Comparison
	`!!a`	`a`	Scalar		`0,1`	Logic
`!a`	**`not`**	`@a`	Scalar		`0,1`	Logic
`&& ||`	**`and or`**	`a@b`	Scalar	Scalar	`0,1`	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`	None		**`size_t`**	Size, ICE
`alignof`	**`_Alignof`**	`@(a)`	None		**`size_t`**	Alignment, ICE

4.2. Arithmetic. Arithmetic operators form the first group in table 4.1 of operators that operate on values.

4.2.1. *+, -, and *.* The arithmetic operators *+*, *−*, and *⋆* mostly work as we would expect by computing the sum, the difference, and the product, respectively, of two values:

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

Here, `c` must be equal to `76`, and `d` to `31`. As you can see from this little example, subexpressions can be grouped together with parentheses to enforce a preferred binding of the operator.

In addition, the operators *+* and *−* have unary variants. `-b` gives the negative of `b`: a value `a` such that `b + a` is `0`. `+a` simply provides the value of `a`. The following gives `76` as well:

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

Even though we use an unsigned type for our computation, negation and difference by means of the operator *−* are ***well defined***[C]. That is, regardless of the values we feed into such a subtraction, our computation will always have a valid result. In fact, one of

TABLE 4.2. Object operators: The Form column gives the syntactic form of the operation, where `@` represents the operator, `o` denotes an object, and `a` denotes a suitable additional *value* (if any) that serves as an operand. An additional * in the Type column requires that the object `o` be 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@`	Arithmetic or pointer	Value	Arithmetic
`&=`	**`and_eq`**	`o@a`	Integer	Value	Bit
`|=`	**`or_eq`**				
`^=`	**`xor_eq`**				
`<<= >>=`		`o@a`	Integer	Value	Bit
`.`		`o@m`	**`struct`**	Object	Member
`[]`		`o[a]`	Array*	Object	Member
`&`		`@o`	Any*	Pointer	Address
`sizeof`		`@ o`	Data Object, non-VLA	**`size_t`**	Size, ICE
`sizeof`		`@ o`	VLA	**`size_t`**	size
`alignof`	**`_Alignof`**	`@(o)`	Non-function	**`size_t`**	Alignment, ICE

TABLE 4.3. Type operators: These operators return an integer constant (ICE) of type **`size_t`**. They have function-like syntax with the operands in parentheses.

Operator	Nick	Form	Type of `T`	
`sizeof`		**`sizeof`**`(T)`	Any	Size
`alignof`	**`_Alignof`**	**`alignof`**`(T)`	Any	Alignment
	`offsetof`	**`offsetof`**`(T,m)`	**`struct`**	Member offset

the miraculous properties of **`size_t`** is that `+-*` arithmetic always works where it can. As long as the final mathematical result is within the range `[0, SIZE_MAX]`, then that result will be the value of the expression.

Takeaway 4.2.1 #1 *Unsigned arithmetic is always well defined.*

Takeaway 4.2.1 #2 *The operations +, −, and * on **`size_t`** provide the mathematically correct result if it is representable as a **`size_t`**.*

When the result is not in that range and thus is not ***representable***^{C} as a **`size_t`** value, we speak of arithmetic ***overflow***^{C}. Overflow can happen, for example, 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.

4.2.2. *Division and remainder.* The operators `/` and `%` are a bit more complicated because they correspond to integer division and the remainder operation. You might not be as used to them as you are 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`s are removed from `a`. The operators `/` and `%` come in pairs: if we have `z = a / b`, the remainder `a % b` can be computed as `a - z*b`.

Takeaway 4.2.2 #1 *For unsigned values, `a == (a/b)*b + (a%b)`.*

A familiar example of the `%` operator is the hours on a clock. Say we have a 12-hour clock: 6 hours after 8:00 is 2:00. 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 9] Another similar use for `%` is computation using minutes in an hour, of the form `a % 60`.

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

Takeaway 4.2.2 #2 *Unsigned `/` and `%` are only well defined 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, 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`**.

Takeaway 4.2.2 #3 *Arithmetic on **`size_t`** implicitly computes modulo **`SIZE_MAX`** `+ 1`.*

Takeaway 4.2.2 #4 *In the case of overflow, unsigned arithmetic wraps around.*

This means 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`**; 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
```

The operators `/` and `%` have the nice property that their results are always smaller than or equal to their operands.

Takeaway 4.2.2 #5 *The result of unsigned `/` and `%` is always smaller than the operands.*

Takeaway 4.2.2 #6 *Unsigned `/` and `%` can't overflow.*

4.3. Operators that modify objects. Another important operation that we have already 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 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 sufficient.

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

```
an_object @= some_expression;
```

They are just convenient abbreviations for combining the arithmetic operator `@` and the assignment; see table 4.2. A mostly equivalent form is

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

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

[Exs 9] Implement some computations using a 24-hour clock, such as 3 hours after 10:00 and 8 hours after 20:00.

```
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: for example, `i + = 7` instead of `i += 7` is a syntax error.

Takeaway 4.3 #1 *Operators must have all their characters directly attached to each other.*

We already have seen two other operators that modify objects: 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 the value of the object *after* the modification, but not the object itself. You could, if you were crazy 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}.

Takeaway 4.3 #2 *Side effects in value expressions are evil.*

Takeaway 4.3 #3 *Never modify more than one object in a statement.*

For the increment and decrement operators, there are two other forms: ***postfix increment***^{C} and ***postfix decrement***^{C}. They differ from the one we have seen in the result they provide to the surrounding expression. The prefix versions of these operators (`++a` and `--a`) do the operation first and then return the result, much like the corresponding assignment operators (`a+=1` and `a-=1`); the postfix operations return the value *before* the operation and perform the modification of the object thereafter. For any of them, the effect on the variable is the same: the incremented or decremented value.

All this shows that the evaluation of expressions with side effects may be difficult to follow. Don't do it.

4.4. Boolean context. Several operators yield a value of `0` or `1`, depending on whether some condition is verified; see table 4.1. They can be grouped into two categories: comparisons and logical evaluation.

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

Takeaway 4.4.1 #1 *Comparison operators return the value* **`false`** *or* **`true`**.

Remember that **`false`** and **`true`** are nothing more than fancy names for `0` and `1`, respectively. So, they can be used in arithmetic or for array indexing. In the following code, `c` will always be `1`, and `d` will be `1` if `a` and `b` are equal and `0` otherwise:

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

In the next example, `N` is a big number, and the array element `sign[false]` will hold the number of values in `largeA` that are greater than or equal to `1.0` and `sign[true]` those that are strictly less:

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

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

	[false]	[true]
sign	size_t	size_t

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

4.4.2. *Logic.* Logic operators operate on values that are already supposed to represent a **false** or **true** value. If they do not, the rules described for conditional execution (takeaway 3.1 #1) apply first. The operator *!* (***not***) logically negates its operand, and operators *&&* (***and***) is logical and, and *||* (***or***) is logical or. The results of these operators are summarized in table 4.4.

TABLE 4.4. 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 to the comparison operators, logic operators return truth values.

Takeaway 4.4.2 #1 *Logic operators return the value* ***false*** *or* ***true****.*

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

```
double largeA[N] = { };
...
/* 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 (takeaway 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:

	[false]	[true]
isset	size_t	size_t

The operators `&&` and `||` have a particular property 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:

```
// This never divides by 0.
if (b != 0 && ((a/b) > 1)) {
  ++x;
}
```

Here, the evaluation of `a/b` is omitted conditionally during execution, and thereby a division by zero can never occur. Equivalent code would be

```
if (b) {
  // This never divides by 0.
  if (a/b > 1) {
    ++x;
  }
}
```

4.5. The ternary or conditional operator. The *ternary operator* is similar to an **`if`** statement, but 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;
  }
```

Similar to the operators `&&` and `||`, the second and third operand are evaluated only 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}:

<tgmath.h>

```
#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 called only 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`**.

Complex arithmetic and the tools used for it require the header `<complex.h>`, which is indirectly included by `<tgmath.h>`. They will be introduced later, in section 5.7.8.

<complex.h>
<tgmath.h>

In the previous example, we also see a 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__`** is defined.

4.6. Evaluation order. Of the operators so far, 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 for the operands: the first operand, since it is a condition for the remaining ones, is always evaluated first.

Takeaway 4.6 #1 *`&&`, `||`, `?:`, and `,` evaluate their first operand first.*

The comma (`,`) is the only operator we haven't introduced yet. It evaluates its operands in order, and the result is the value of the right operand. For example, `(f(a), f(b))` first evaluates `f(a)` and then `f(b)`; the result is the value of `f(b)`.

Be aware that the comma *character* plays other syntactical roles in C that do *not* use the same convention about evaluation. For example, the commas that separate initializations do not have the same properties as those that separate function arguments.

The comma operator is rarely useful in clean code, and it is a trap for beginners: `A[i, j]` is *not* a two-dimensional index for matrix `A`, but results in `A[j]`.

Takeaway 4.6 #2 *Don't use the , operator.*

Other operators don't have an evaluation restriction. For example, in an expression such as `f(a)+g(b)`, there is no pre-established order specifying whether `f(a)` or `g(b)` is to be computed first. If either the function `f` or `g` works with side effects (for instance, if `f` modifies `b` behind the scenes), the outcome of the expression will depend on the chosen order.

Takeaway 4.6 #3 *Most operators don't sequence their operands.*

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

The same holds for function arguments. In something like

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

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

Takeaway 4.6 #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.

Takeaway 4.6 #5 *Functions calls within expressions should not have side effects.*

CHALLENGE 4 (Union-Find). *The Union-Find problem deals with the representation of partitions over a base set. We will identify the elements of the base set using the numbers* 0, 1, … *and will represent partitions with a forest data structure where each element knows a "parent" that is another element inside the same partition. Each set in such a partition is identified by a designated element called the* root *of the set.*
We want to perform two principal operations:

- *A* `Find` *operation receives one element of the ground set and returns the* root *of the corresponding set.*
- *A* `Union`[a] *operation receives two elements and merges the two sets to which these elements belong into one.*

Can you implement a forest data structure in an index table of base type **`size_t`** *called* `parent`*? Here, a value in the table* **`SIZE_MAX`** *would mean a position represents a root of one of the trees; another number represents the position of the parent of the corresponding tree.*
One of the important features of starting the implementation is an initialization function that makes `parent` *the singleton partition; that is, the partition where each element is the root of its own private set.*
With this index table, can you implement a `Find` *function that, for a given index, finds the root of its tree?*
Can you implement a `FindReplace` *function that changes all* `parent` *entries on a path to the root (including) to a specific value?*
Can you implement a `FindCompress` *function that changes all* `parent` *entries to the root that has been found?*
Can you implement a `Union` *function that, for two given elements, combines their trees into one? Use* `FindCompress` *for one side and* `FindReplace` *for the other.*

---

[a] C also has a concept called a **`union`**, which we will see later and which is *completely different* than the operation we are currently talking about. Because **`union`** is a keyword, we use capital letters to name the operations here.

Summary

- Arithmetic operators do math. They operate on values.
- Assignment operators modify objects.
- Comparison operators compare values and return `0` or `1`.
- Function calls and most operators evaluate their operands in a nonspecific order. Only `&&`, `||`, and `?:` impose an ordering on the evaluation of their operands.

5. Basic values and data

This section covers

- Understanding the abstract state machine
- Working with types and values
- Initializing variables
- Using named constants
- Binary representations of types

We will now change our focus from "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 on paper using the Hindu-Arabic numeral system. But we have other systems to write numbers: for example, Roman numerals (i, ii, iii, iv, and so on) or textual notation. To know that the word *twelve* denotes the value 12 is a nontrivial step and reminds us that European languages denote numbers not only in decimal but also in other systems. English and German mix with base 12, French with bases 16 and 20. For non-native French speakers like myself, it may be difficult to spontaneously associate *quatre vingt quinze* (four times twenty and fifteen) with the value 95.

Similarly, representations of values on a computer can vary "culturally" from architecture to architecture or are determined by the type the programmer gave to the value. Therefore, we should try to reason primarily about values and not about representations if we want to write portable code.

If you already have some experience in C and in manipulating bytes and bits, you will need to make an effort to actively "forget" your knowledge for most of this section. Thinking about concrete representations of values on your computer will inhibit you more than it helps you.

Takeaway 5 #1 *C programs primarily reason about values and not about their representation.*

The representation that a particular value has should, in most cases, not be your concern. The compiler is there to organize the translation back and forth between values and representations.

In this section, we will see how the different parts of this translation are supposed to work. The ideal world in which you will usually "argue" in your program is C's *abstract state machine* (section 5.1). It gives a vision of the execution of your program, which is mostly independent of the platform on which the program runs. The components of the *state* of this machine, the *objects*, all have a fixed interpretation (their *type*) and a value that varies in time. C's basic types are described in subsection 5.2, followed by descriptions of how we can express specific values for such basic types (subsection 5.3), how types are assembled in expressions (subsection 5.4), how we can ensure that our objects initially have the desired values (subsection 5.5), how we can give names to recurrent values (subsection 5.6), and how such values are represented in the abstract state machine (subsection 5.7).

5.1. The abstract state machine. A C program can be seen as a sort of machine that manipulates values: the particular values that variables of the program have at a given time and intermediate values that are the result of computed expressions. Let us consider a basic example:

```
double x = 5.0;
double y = 3.0;
...
x = (x * 1.5) - y;
printf("x is \%g\n", x);
```

Here we have two variables, `x` and `y`, that have initial values `5.0` and `3.0`, respectively. The third line computes some expressions: the subexpression

```
x
```

that evaluates `x` and provides the value `5.0`

```
(5.0 * 1.5)
```

that results in the value `7.5`

```
y
```

that evaluates `y` and provides the value `3.0`

```
7.5 - 3.0
```

that results in `4.5`;

```
x = 4.5
```

that changes the value of `x` to `4.5`

```
x
```

that evaluates `x` again but now provides the value `4.5` and

```
printf("x is \%g\n", 4.5)
```

that outputs a text line to the terminal.

Not all operations and their resulting values are *observable* from within your program. They are observable only if they are stored in *addressable* memory or written to an output device. In the example, to a certain extent, the **`printf`** statement "observes" what was done on the previous line by evaluating the variable `x` and then writing a string representation of that value to the terminal. But the other subexpressions and their results (such as the multiplication and subtraction) are not observable as such, since we never define a variable that is supposed to hold these values.

Your C compiler is allowed to shortcut any of the steps during a process called ***optimization***^{C} only if it ensures the realization of the end results. Here, in our toy example, there are basically two possibilities. The first is that variable `x` is not used later in the program, and its acquired value is only relevant for our **`printf`** statement. In that case, the only effect of our code snippet is the output to the terminal, and the compiler may well (and will!) replace the whole snippet with the equivalent

```
printf("x is 4.5\n");
```

That is, it will do all the computations at compile time, and the executable that is produced will just print a fixed string. All the remaining code and even the definitions of the variables disappear.

The other possibility is that `x` might be used later. Then a decent compiler would either do something like

```
double x = 4.5;
printf("x␣is␣4.5\n");
```

or maybe

```
printf("x␣is␣4.5\n");
double x = 4.5;
```

because to use `x` at a later point, it is not relevant whether the assignment took place before or after the **`printf`**.

For an optimization to be valid, it is only important that a C compiler produces an executable that reproduces the ***observable states***^{C}. Observable states consist of the contents of some variables (and similar entities that we will see later) and the output as they evolve during the execution of the program. This whole mechanism of change is called the ***abstract state machine***^{C}.

To explain the abstract state machine, we first have to look into the concepts of a *value* (what state are we in), the *type* (what this state represents), and the *representation* (how state is distinguished). As the term *abstract* suggests, C's mechanism allows different platforms to realize the abstract state machine of a given program differently according to their needs and capacities. This permissiveness is one of the keys to C's potential for optimization.

5.1.1. *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, and evaluating a value `0` in a control expression will always trigger the **`false`** branch of the control statement.

So far, most of our examples of values have been some kind of numbers. This is not an accident; it relates to one of the major concepts of C.

Takeaway 5.1.1 #1 *All values are numbers or translate to numbers.*

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

The *data* of a program execution consists of 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 the 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 the result of a computation to depend on the executable (which is platform specific) but ideally to depend only on the program specification itself. An important step to achieve this platform independence is the concept of ***types***^{C}.

5.1.2. *Types.* A type is an additional property that C associates with values. Up to now, we have seen several such types, most prominently **`size_t`**, but also **`double`** and **`bool`**.

Takeaway 5.1.2 #1 *All values have a type that is statically determined.*

Takeaway 5.1.2 #2 *Possible operations on a value are determined by its type.*

Takeaway 5.1.2 #3 *A value's type determines the results of all operations.*

5.1.3. *Binary representation and the abstract state machine.* Unfortunately, the variety of computer platforms is not such that the C standard can completely impose the results of the operations on a given type. Things not completely specified as such by the standard include, for example, the precision to which a `double` floating-point operation is performed (*floating-point representation*).[10] C only imposes properties on 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

For example, the operations on the type `size_t` can be entirely determined when inspecting the value of `SIZE_WIDTH` in addition to the operands.[11] We call the model to represent values of a given type on a given platform the ***binary representation***^{C} of the type.

Takeaway 5.1.3 #1 *A type's binary representation determines the results of all operations.*

Generally, all information we need to determine that model is within reach of any C program. The C library headers provide the necessary information through named values (such as `SIZE_MAX`), operators, and function calls.

Takeaway 5.1.3 #2 *A type's binary representation is observable.*

This binary representation is still a model and thus 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 is the *object representation*. In contrast to the binary representation, the object representation is usually not of 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 different platform models. Much later, in section 12.1, we will see that we can 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.

Takeaway 5.1.3 #3 (as-if) *Programs execute **as if** following the abstract state machine.*

5.1.4. *Optimization.* How a concrete executable manages to follow the description of the abstract state machine 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 constant values such as

[10] Other international standards are more restrictive about these representations. For example, the POSIX standard enforces a particular sign representation, and ISO/IEC/IEEE 60559 [2011] normalizes floating-point representations.

[11] Prior to C23, this value was not available. We needed the value of `SIZE_MAX` (see previous discussion). This value can be deduced from `SIZE_WIDTH`. Similarly, by knowing that the sign representation now is fixed to 2's complement, the minimal and maximal values can be deduced for all integer types.

```
x += 5;
/* Do something else without x in the meantime. */
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: for example, as long as 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 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, overflow of an unsigned type is not a problem, and the result of the condensed operation will always be consistent with the two separate ones. For other types, such as signed integer types (**`signed`**) and floating-point types (**`double`**), an overflow may *raise an exception* and terminate the program. In that case, the optimization cannot be performed.

As we have already mentioned, this allowed slackness between program description and abstract state machine is a very valuable feature, commonly referred to as optimization. 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 of this discussion can be summarized in the following takeaway.

Takeaway 5.1.4 #1 *Type determines optimization opportunities.*

5.2. Basic types. C has a series of basic types and 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 straightforward. There is a first level of specification that is done entirely with keywords of the language, such as **`signed`**, **`int`**, and **`double`**. This first level is mainly organized according to C internals. On top of that is a second level of specification that comes through header files, and we have already seen examples: **`size_t`** and **`bool`**. This second level is organized by type semantics, specifying what properties a particular type brings to the programmer.

We will start with the first-level specification of such types. As we discussed earlier (takeaway 5.1.1 #1), all basic values in C are numbers, but there are different kinds of numbers. As a principal distinction, we have two different classes of numbers, each with two subclasses each: ***unsigned integers***^{C}, ***signed integers***^{C}, ***real floating-point numbers***^{C}, and ***complex floating-point numbers***^{C}. Each of these four classes contains several types. They differ according to their ***precision***^{C}, which determines the valid range of values that are allowed for a particular type.[12] Table 5.1 contains an overview of the 18 base types.

[12] The term *precision* is used here in a restricted sense as the C standard defines it. It is different from the *accuracy* of a floating-point computation.

TABLE 5.1. Base types according to the four main type classes. Types with a gray 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 this table are considered to be distinct types, even if they have the same class and precision.

Class		Systematic name	Other name	Rank
Integers	Unsigned	**bool**	**_Bool**	0
		unsigned char		1
		unsigned short		2
		unsigned int	**unsigned**	3
		unsigned long		4
		unsigned long long		5
	[Un]signed	**char**		1
	Signed	**signed char**		1
		signed short	**short**	2
		signed int	**signed** or **int**	3
		signed long	**long**	4
		signed long long	**long long**	5
Floating point	Real	**float**		
		double		
		long double		
	Complex	**float _Complex**	**float complex**	
		double _Complex	**double complex**	
		long double _Complex	**long double complex**	

As you can see from the table, there are six types that we can't use directly for arithmetic, the ***narrow types***[C]. They are ***promoted***[C] to one of the wider types before they are considered in an arithmetic expression. Nowadays, on any realistic platform, this promotion will be a **signed int** of the same value as the narrow type, regardless of whether the narrow type was signed.

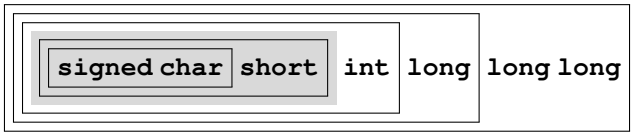
Takeaway 5.2 #1 *Before arithmetic, narrow integers are promoted to* ***signed int****.*

Observe that among the narrow integer types, we have two prominent members: **char** and **bool**. The first is C's type that handles printable characters for text, and the second holds truth values, **false** and **true**. As we said earlier, for C, even these are just some sort of numbers. The 12 remaining, unpromoted types split nicely into the four classes.

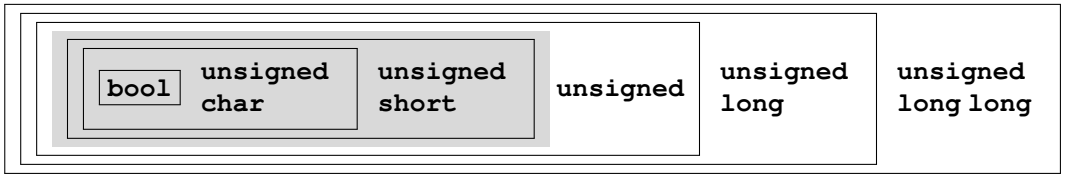
Takeaway 5.2 #2 *Each of the four classes of base types has three distinct unpromoted types.*

Contrary to what many people believe, the C standard doesn't prescribe the precision of these 12 types: it only constrains them. They depend on a lot of factors that are ***implementation defined***[C].

One of the things the standard does prescribe is that the possible ranges of values for the signed types must include each other according to their *rank*:

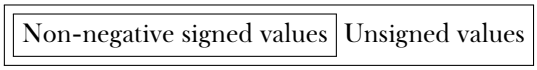


But this inclusion does not need to be strict. For example, on many platforms, the set of values of **int** and **long** are the same, although the types are considered to be different. An analogous inclusion holds for the six unsigned types:



But remember that for any arithmetic or comparison, the narrow unsigned types are promoted to **signed int** and not to **unsigned int**, as this diagram might suggest.

The comparison of the ranges of signed and unsigned types is more difficult. Obviously, an unsigned type can never include the negative values of a signed type. For the non-negative values, we have the following inclusion of the values of types with corresponding rank:



That is, for a given rank, the non-negative values of the signed type fit into the unsigned type. On any modern platform you encounter, this inclusion is strict: the unsigned type has values that do not fit into the signed type. For example, a common pair of maximal values is $2^{31} - 1 = 2,147,483,647$ for **signed int** and $2^{32} - 1 = 4,294,967,295$ for **unsigned int**.

Because the interrelationship between integer types depends on the platform, choosing the "best" type for a given purpose in a portable way can be a tedious task. Luckily, we can get some help from the compiler implementation, which provides us with **typedef**s such as **size_t** that represent certain features.

Takeaway 5.2 #3 *Use **size_t** for sizes, cardinalities, or ordinal numbers.*

Remember that unsigned types are the most convenient because they are the only types that have an arithmetic that is defined consistently with mathematical properties: the modulo operation. They can't raise signals on overflow and can be optimized best. They are described in more detail in subsection 5.7.1.

Takeaway 5.2 #4 *Use **unsigned** for small quantities that can't be negative.*

If your program needs values that may be either positive or negative but do not include fractions, use a signed type (see subsection 5.7.5).

Takeaway 5.2 #5 *Use **signed** for small quantities that bear a sign.*

Takeaway 5.2 #6 *Use **ptrdiff_t** for large differences that bear a sign.*

If you want to do fractional computation with a value such as `0.5` or `3.771'89E+89`, use floating-point types (see subsection 5.7.8).

Takeaway 5.2 #7 *Use **double** for floating-point calculations.*

Takeaway 5.2 #8 *Use **double complex** for complex calculations.*

TABLE 5.2. Some semantic arithmetic types for specialized use cases

Type	Header	Context of definition	Meaning
`size_t`	`<stddef.h>`		Type for "sizes" and cardinalities
`ptrdiff_t`	`<stddef.h>`		Type for size differences
`uintmax_t`	`<stdint.h>`		Unsigned integer type of preprocessor
`intmax_t`	`<stdint.h>`		Signed integer type of preprocessor
`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 5.2 list some of them. The second pair represents the type in which the preprocessor does any of its arithmetic or comparison. Prior to C23 these were the types of maximal width that the compiler supported, but this constraint has been relaxed; under certain circumstances, there may be *extended integer types* that are wider (see section 5.7.6).

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 platform's model of processor clock cycles, so the unit of time is usually much less than a second; `CLOCKS_PER_SEC` can be used to convert such values to seconds.

5.3. Specifying values. We have already seen several ways in which numerical constants (***literals***^{C}) can be specified:

`123` ***Decimal integer literal***^{C}—The most natural choice for most of us.

`077` ***Octal integer literal***^{C}.—This is specified by a sequence of digits, the first being `0` and the following between `0` and `7`. For example, `077` has the value 63. This type of specification merely has historical value and is rarely used nowadays. Only one octal literal is commonly used: `0` itself.

`0xFFFF` ***Hexadecimal integer literal***^{C}—This is specified by starting with **`0x`** followed by a sequence of digits between `0`, ..., `9` and `a` ... `f`. For example, **`0x`**`beaf` has the value 48,815. The `a .. f` and `x` can also be written in capitals, **`0X`**`BEAF`.

`0b1010` ***Binary integer literal***^{C}—This is specified by starting with **`0b`** followed by a sequence of `0` or `1` digits. For example, **`0b`**`1010` has the value 10. The leading **`0b`** may also be written as **`0B`** instead. Binary literals were introduced in C23.

`1.7E-13` ***Decimal floating-point literals***^{C}—These literals are quite familiar as the version with 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 literals***^{C}—These are usually used to describe floating-point values in a form that makes it easy to specify values that have exact representations. The general form **`0X`**`hPe` is interpreted as $h \cdot 2^e$. Here, h is specified as a hexadecimal fraction. The exponent e is still specified as a decimal number.

`'a'` ***Integer character literal***^{C}—These are characters put between `'` apostrophes, such as `'a'` or `'?'`. These have values that are only implicitly fixed by the C standard. For example, `'a'` corresponds to the integer code for the character *a* of the Latin alphabet.

Among character literals, the `\` character has a special meaning. For example, we already have seen `'\\n'` for the newline character.

`"hello"` ***String literals***^{C}—They specify text, such as that needed for the **`printf`** and **`puts`** functions. Again, the `\` character is special, as with character literals.[13]

All literals except the last are numerical constants: they specify numbers.[14] String literals are an exception and can be used to specify text that is known at compile time. Integrating larger text into our code could be tedious, if we weren't allowed to split string literals into chunks:

```
puts("first␣line\n"
     "another␣line\n"
     "first␣and␣"
     "second␣part␣of␣the␣third␣line");
```

Takeaway 5.3 #1 *Consecutive string literals are concatenated.*

The rules for numbers are a little bit more complicated.

Takeaway 5.3 #2 *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 it. We will see shortly 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.

We have already seen (takeaway 5.1.2 #1) that all literals must have not only a value but also a type. Don't mix up the fact that a literal has a positive value with its type, which can be **`signed`**.

Takeaway 5.3 #3 *Decimal integer literals 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.

Takeaway 5.3 #4 *A decimal integer literal has the first of the three signed types that fits.*

This rule can have surprising effects. Suppose that on a platform, the minimal **`signed`** value is $-2^{15} = -32'768$ and the maximum value is $2^{15} - 1 = 32'767$. The literal $32'768$ then doesn't fit into **`signed`** and is thus **`signed long`**. As a consequence, the expression `-32'768` has type **`signed long`**. Thus the minimal value of the type **`signed`** on such a platform cannot be written as a literal.[Exs 15]

Takeaway 5.3 #5 *The same value can have different types.*

[13] If used in the context of the **`printf`** function, another character also becomes "special": the `%` character. If you want to print a literal `%` with **`printf`**, you have to duplicate it.

[14] You may have observed that complex numbers are not included in this list. We will see how to specify them in subsection 5.3.1.

[Exs 15] 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.

Deducing the type of a binary, octal, or hexadecimal literal is a bit more complicated. These can also be of an unsigned type if the value doesn't fit for a signed type. In the earlier example, the hexadecimal literal `0x7FFF` has the value 32,767 and thus is type **signed**. Other than for the decimal literal, the literal `0x8000` (value 32′768 written in hexadecimal) then is an **unsigned**, and expression `-0x8000` again is **unsigned**.[Exs 16]

Takeaway 5.3 #6 *Don't use binary, octal, or hexadecimal literals for negative values.*

As a consequence, there is only one choice left for negative values.

Takeaway 5.3 #7 *Use decimal literals for negative values.*

TABLE 5.3. Examples for literals and their types, under the supposition that **signed** and **unsigned** have the commonly used representation with 32 bits

Literal x	Value	Type	Value of $-x$
`2'147'483'647`	+2,147,483,647	**signed**	−2,147,483,647
`2'147'483'648`	+2,147,483,648	**signed long**	−2,147,483,648
`4'294'967'295`	+4,294,967,295	**signed long**	−4,294,967,295
`0x7FFF'FFFF`	+2,147,483,647	**signed**	−2,147,483,647
`0x8000'0000`	+2,147,483,648	**unsigned**	+2,147,483,648
`0xFFFF'FFFF`	+4,294,967,295	**unsigned**	+1
`1`	+1	**signed**	−1
`1U`	+1	**unsigned**	+4,294,967,295

A common error is to try to assign a hexadecimal literal to a **signed** with the expectation that it will represent a negative value. Consider a declaration such as `int x = 0xFFFF'FFFF`. This is done under the assumption that the hexadecimal value has the same *binary representation* as the signed value −1. On most architectures with 32-bit **signed**, this will be true (but not on all of them). However, then nothing guarantees that the effective value +4,294,967,295 is converted to the value −1. Table 5.3 has some examples of interesting literals, their values, and their types.

So a possible prefix (`0`, `0b` or `0x`) not only specifies the base in which an integer literal is read, but indirectly may also have an influence for the deduced type. This deduced type can be changed by a suffix that is appended to the literal.

TABLE 5.4. Suffixes for integer literals and their types

Suffix	Type
`l` or `L`	At least **long**
`ll` or `LL`	**long long**
`wb` or `WB`	**_BitInt**(N) for sufficiently large N
`u` or `U`	Force unsigned

For example, `1U` has value 1 and type **unsigned**, `1L` is **signed long**, and `1ULL` has the same value 1 but type **unsigned long long**.[Exs 17] Note that we are representing C literals such as `1ULL` in typewriter font and distinguish them from their mathematical value 1, which is in normal font.

[Exs 16] Show that if the maximum **unsigned** is $2^{16} - 1$, then `-0x8000` has value 32,768, too.

[Exs 17] Show that the expressions `-1U`, `-1UL`, and `-1ULL` have the maximum values and type as the three non-promoted unsigned types, respectively.

So with suffixes, integer literals can be forced to have a type with minimal rank. For a decimal integer literal, if there is one `l` or `L` the type is **`long`** if the value fits and **`long long`** otherwise. If there are two (`ll` or `LL`), the type is fixed to **`long long`**. For prefixed integer literals (`0`, **`0b`**, or **`0x`**), these suffixes still may be **`unsigned long`** or **`unsigned long long`**, depending on the value. To force an unsigned type, we can add `u` or `U` to the suffix.

The suffix `wb` or `WB` that was introduced in C23 forces the literal to have a specific bit-precise type. With a possible combination of `u` or `U`, it is the only suffix that can guarantee a type of a specific signedness regardless of the base. We will see these types a bit later in section 5.7.7.

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

Takeaway 5.3 #8 *Different literals can have the same value.*

For integers, this rule looks almost trivial, but for floating-point literals, it 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.

Takeaway 5.3 #9 *The effective value of a decimal floating-point literal may be different from its literal value.*

For example, on my machine, the literal `0.2` has the value

$$0.200{,}000{,}000{,}000{,}000{,}011{,}1$$

As a consequence, the literals `0.2` and `0.200'000'000'000'000'011'1` have the same value.

Hexadecimal floating-point literals were designed to correspond better with the binary representations of floating-point values. In fact, on most modern architectures, such a literal (that does not have too many digits) will exactly correspond to the literal value. Unfortunately, these beasts are almost unreadable for mere humans. For example, consider the two literals

`0x``1.9999'9AP-3` and **`0x`**`C.CCCC'CCCC'CCCC'CCDP-6`.

They correspond to

$$1.600{,}000{,}023{,}84 * 2^{-3} \text{ and } 12.800{,}000{,}000{,}000{,}000{,}000{,}2 * 2^{-6};$$

thus, expressed as decimal floating points, their values are approximatively

$$0.200{,}000{,}002{,}98 \text{ and } 0.200{,}000{,}000{,}000{,}000{,}000{,}003.$$

So the two literals have values that are very close to each other, whereas their representation as hexadecimal floating-point literals seems to put them far apart.

Finally, floating-point literals can be followed by the letter `f` or `F` to denote a **`float`** or by `l` or `L` to denote a **`long double`**. Otherwise, they are of type **`double`**. Be aware that different types of literals generally lead to different values for the same literal. Here is a typical example:

	`float`	**`double`**	**`long double`**
Literal	`0.2F`	`0.2`	`0.2L`
Value	**`0x`**`1.9999'9AP-3F`	**`0x`**`1.9999'9999'9999AP-3`	**`0x`**`C.CCCC'CCCC'CCCC'CCDP-6L`

Takeaway 5.3 #A *Literals have value, type, and binary representations.*

5.3.1. *Complex constants.* Complex types are not necessarily supported by all C platforms. This 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 numerical functions, this is already done implicitly.

`<complex.h>`
`<tgmath.h>`

Unfortunately, C provides no literals to specify literals of a complex type. It only has several macros[18] that may ease the manipulation of these types.

The first possibility to specify complex values is the macro **`CMPLX`**, which comprises two floating-point values, the real and imaginary parts, in one complex value. For example, **`CMPLX`**`(0.5, 0.5)` is a **`double complex`** value with the real and imaginary part of one-half. Analogously, there are **`CMPLXF`** for **`float complex`** and **`CMPLXL`** for **`long double complex`**.

Another, more convenient, possibility is provided by the macro **`I`**, which represents a constant value of type **`float complex`** such that **`I`**`*`**`I`** has the value -1. One-character macro names in uppercase are often used in programs for numbers that are fixed for the whole program. By itself, it is not a brilliant idea (the supply of one-character names is limited), and you should definitely leave **`I`** alone.

Takeaway 5.3.1 #1 ***`I`*** *is reserved for the imaginary unit.*

`I` can be used to specify constants of complex types similar to the usual mathematical notation. For example, `0.5 + 0.5*`**`I`** would be of type **`double complex`** and `0.5F + 0.5F*`**`I`** of **`float complex`**. The compiler implicitly ***converts***^{C} the result to the wider of the types if we mix, for example, **`float`** and **`double`** literals for real and imaginary parts. Another way to encode complex constants is by using complex literals. These are floating point literals with an extra `i`, for example `0.5i` or `0.5IF`. Although widely supported nowadays, unfortunately, this form of literals is not (yet) standardized, so you may not rely on it.

CHALLENGE 5 (complex numbers). *Can you extend the derivative (Challenge 2) to the complex domain: that is, functions that receive and return* ***`double complex`*** *values?*

5.4. Implicit conversions. As we have seen in the examples, the type of an operand has an influence on the type of an operator expression, such as `-1` or `-1U`. Whereas the first is a **`signed int`**, the second is an **`unsigned int`**. The latter might be particularly surprising for beginners because an **`unsigned int`** has no negative values, so the value of `-1U` is a large positive integer.

Takeaway 5.4 #1 *Unary − and + have the type of their promoted operand.*

So, these operators are examples where the type usually does not change. In cases where they do change, we have to rely on C's strategy to do *implicit conversions*: that is, to move a value with a specific type to one that has a desired type. Consider the following examples, again under the assumption that $-2,147,483,648$ and $2,147,483,647$ are the minimal and maximal values of a **`signed int`**, respectively:

```
double         a =  1;             // Harmless; value fits type
signed short   b = -1;             // Harmless; value fits type
signed int     c =  0x8000'0000;   // Dangerous; value too big for type
```

[18] We see what macros are really about in subsection 5.6.3. For now, just take them as names to which the compiler has associated some specific property.

```
signed int     d = -0x8000'0000;   // Dangerous; value too big for type
signed int     e = -2'147'483'648; // Harmless; value fits type
unsigned short g =  0x8000'0000;   // Loses information; has value 0
```

Here, the initializations of a and b are harmless. The respective values are well in the range of the desired types, so the C compiler can convert them silently.

The next two conversions for c and d are problematic. As we have seen, **0x**8000'0000 is of type **unsigned int** and does not fit into a **signed int**. So c receives a value that is implementation-defined, and we have to know what our platform has decided to do in such cases. It could just reuse the bit pattern of the value on the right or terminate the program. As for all implementation-defined features, which solution is chosen should be documented by your platform, but be aware that this can change with new versions of your compiler or may be switched by compiler arguments.

For the case of d, the situation is even more complicated. **0x**8000'0000 has the value 2,147,483,648, and we might expect that -**0x**8000'0000 is just the negative value. But since effectively -**0x**8000'0000 is again 2,147,483,648, the same problem arises as for c.[Exs 19]

Then, e is harmless again. This is because we used -2'147'483'648, a negated decimal literal which has type **signed long** and whose value effectively is as intended (shown earlier). Since this value fits into a **signed int**, the conversion can be done with no problem.

The last example for g is ambiguous in its consequences. A value that is too large for an unsigned type is converted according to the modulus. Here in particular, if we assume that the maximum value for **unsigned short** is $2^{16} - 1$, the resulting value is 0. Whether or not such a "narrowing" conversion is the desired outcome is often difficult to tell.

Takeaway 5.4 #2 *Avoid narrowing conversions.*

Takeaway 5.4 #3 *Don't use narrow types in arithmetic.*

The type rules become even more complicated for operators that have two operands, such as addition and multiplication because these then may have different types. Here are some examples of operations that involve floating-point types:

```
1       + 0.0  // Harmless; double
1       + I    // Harmless; complex float
INT_MAX + 0.0F // May lose precision; float
INT_MAX + I    // May lose precision; complex float
INT_MAX + 0.0  // Usually harmless; double
```

Here, the first two examples are harmless: the value of the integer literal 1 fits well into the type **double** or **complex float**. In fact, for most such mixed operations, whenever the range of one type fits into the range of the other, the result has the type of the wider range.

The next two are problematic because **INT_MAX**, the maximal value for **signed int**, usually will not fit into a **float** or **complex float**. For example, on my machine, **INT_MAX** + 0.0F is the same as **INT_MAX** + 1.0F. The last line shows that for an operation with **double**, this would work fine on most platforms. Nevertheless, on an existing or future platform where **int** is 64 bit, an analogous problem with the precision could occur.

Because there is no strict inclusion of value ranges for integer types, deducing the type of an operation that mixes signed and unsigned values can be nasty:

[Exs 19] Under the assumption that the maximum value for **unsigned int** is **0x**FFFF'FFFF, prove that -**0x**8000'0000 == **0x**8000'0000.

```
-1   < 0    // True, harmless, same signedness
-1L  < 0    // True, harmless, same signedness
-1U  < 0U   // False, harmless, same signedness
-1   < 0U   // False, dangerous, mixed signedness
-1U  < 0    // False, dangerous, mixed signedness
-1L  < 0U   // Depends, dangerous, same or mixed signedness
-1LL < 0UL  // Depends, dangerous, same or mixed signedness
```

The first three comparisons are harmless because even if they mix operands of different types, they do not mix signedness. For these cases, since the ranges of possible values nicely contain each other, C simply converts the other type to the wider one and does the comparison there.

The next two cases are unambiguous but perhaps not what a naive programmer would expect. In fact, for both, all operands are converted to **`unsigned int`**. Thus, both negated values are converted to large unsigned values, and the result of the comparison is **`false`**.

The last two comparisons are even more problematic. On platforms where

`INT_WIDTH` < **`LONG_WIDTH`**,

`0U` is converted to `0L`, and thus the first result is **`true`**. On other platforms with

`INT_WIDTH` == **`LONG_WIDTH`**,

`-1L` is converted to `-1U` (that is, **`UINT_MAX`**), and thus the first comparison is **`false`**. Analogous observations hold for the second comparison of the last two, but be aware that there is a good chance the outcome of the two is not the same.

Examples like the last two comparisons can give rise to endless debates in favor of or against signed or unsigned types, respectively. But they show only one thing: that the semantics of mixing signed and unsigned operands is not always clear. There are cases where either possible choice of an implicit conversion is problematic.

Takeaway 5.4 #4 *Avoid operations with operands of different signedness.*

Takeaway 5.4 #5 *Use unsigned types whenever you can.*

Takeaway 5.4 #6 *Chose your arithmetic types such that implicit conversions are harmless.*

5.5. Initializers. We have seen (subsection 2.3) that the initializer is an important part of an object definition. Initializers help us guarantee that a program execution is always in a defined state so that whenever we access an object, it has a well-known value that determines the state of the abstract machine.

Takeaway 5.5 #1 *All variables should be initialized.*

The only exception to that rule should be made for code that must be highly optimized.[20] 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 its last assignment or its initialization. Superfluous initializations or assignments will simply be optimized out.

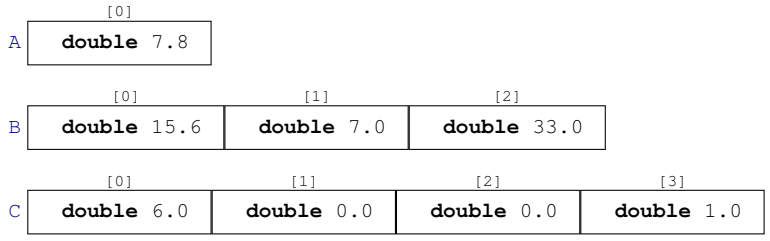
For scalar types such as integers and floating points, an initializer just contains an expression that can be converted to that type. We have seen a lot of examples of that. Optionally, such an initializer expression may be surrounded with `{}`. Here are some examples:

[20] Since C23, even variable-length arrays (VLA; see subsection 6.1.3) may be initialized by a default initializer.

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

Initializers for other types *must* have these `{}`. For example, array initializers contain initializers for the different elements, each of which is followed by a comma:

```
double A[] = { 7.8, };
double B[3] = { 2 * A[0], 7, 33, };
double C[] = { [0] = 6, [3] = 1, };
```



As we have seen, if there is no length specification, the array is said to have an ***incomplete type***^{C}. The type is then completed by the initializer to fully specify the length. Here, `A` has only one element, whereas `C` has four. For the first two initializers, the element to which the scalar initialization applies is deduced from the position of the scalar in the list: for example, `B[1]` is initialized to `7`. The form as for `C` is called designated initializers. These are by far preferable since they make the code more robust against small changes in declarations.

Takeaway 5.5 #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 = { }
```

will always do.

Takeaway 5.5 #3 *`{ }` is a valid initializer for all objects.*

This feature has only been introduced in C23; before that, we had to use `{0}`, and there was a relatively complicated reasoning that made this work. This default initializer may also be used for variable-length arrays (see subsection 6.1.3), which previously had no initializer syntax.

In initializers, we often have to specify values that have a particular meaning for the program.

5.6. Named constants. A common issue, even in small programs, is that they use special values for some purposes 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,[21] on which we would like to perform some operations:

[21] This uses a *pointer*, type `char const*const`, to refer to strings. We will see later how this particular technique works.

```
char const*const bird[3] = {
  "raven",
  "magpie",
  "jay",
};
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, bird[i]);
...
for (unsigned i = 0; i < 3; ++i)
    printf("%s␣plural␣pronoun␣is␣%s\n", ordinal[i], pronoun[i]);
```

Here, we use the constant `3` in several places with three different "meanings" that are not very correlated. For example, an addition to our set of corvids would require two separate code changes. In a real setting, there might be many more places in the code that depend on this particular value, and in a large code base, this can be very tedious to maintain.

Takeaway 5.6 #1 *All constants with a particular meaning must be named.*

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

Takeaway 5.6 #2 *All constants with different meanings must be distinguished.*

At the start, C had surprisingly little means to specify named constants, and its terminology even caused a lot of confusion about which constructs effectively lead to compile-time constants. So we first have to get the terminology straight (subsection 5.6.1) before we look into the only proper named constants that C provided until C23: enumeration constants (subsection 5.6.2). The latter will help us to replace the different versions of `3` in our example with something more explanatory. A second, generic mechanism complements this feature with simple text replacement: macros (subsection 5.6.3). Macros only lead to compile-time constants if their replacements are composed of literals of base types, as we have seen. Finally we distinguish a concept of unnamed temporary objects called compound literals (subsection 5.6.4) and C23's new **`constexpr`** objects that also can serve as proper named or unnamed constants (subsection 5.6.5).

5.6.1. *Read-only objects.* Don't confuse the term *constant*, which has a very specific meaning in C, with objects that can't be modified. For example, in the previous code, `bird`, `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 `bird`, 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:

Takeaway 5.6.1 #1 *An object of* **`const`***-qualified type is read-only.*

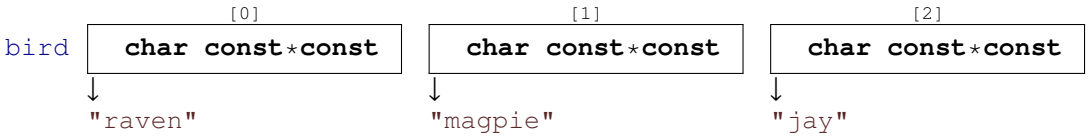
That doesn't mean the compiler or run-time system may not perhaps change the value of such an object: other parts of the program may see that object without the qualification and change it. The fact that you cannot write the summary of your bank account directly (but only read it) doesn't mean it will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

Takeaway 5.6.1 #2 *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 was introduced to the C language much later than string literals, and therefore it remained as it is for backward compatibility.[22]

Arrays such as `bird` also use another technique to handle string literals. They use a ***pointer***^{C} type, **char const*const**, to "refer" to a string literal. A visualization of such an array looks like this:



That is, the string literals themselves are not stored inside the array `bird` but in some other place, and `bird` only refers to those places. We will see much later, in subsections 6.2 and 11, how this mechanism works.

Since C23, there is another construct indicated by the keyword **constexpr**, that results in read-only objects. But in contrast to objects that are simply **const**-qualified, these are guaranteed to never change, and their value is known at compile time. The main difference can be seen in the following example:

```
extern double const factor;
constexpr double π = 3.141'592'653'589'793'238'46;
```

The declaration of `factor` only tells us (and the compiler) that somewhere there is a **double** object that we don't have the right to change. When and where is value is determined is unspecified. On the other hand, the value of π is given together with the declaration and will be stable for the whole compilation of the program. We will see **constexpr** in more detail later.[23]

5.6.2. *Enumerations.* C has a simple mechanism to name small integers as we needed them in the example, called ***enumerations***^{C}:

```
enum corvid { magpie, raven, jay, corvid_num, };
char const*const bird[corvid_num] = {
  [raven]  = "raven",
  [magpie] = "magpie",
  [jay]    = "jay",
};
...
for (unsigned i = 0; i < corvid_num; ++i)
    printf("Corvid␣%u␣is␣the␣%s\n", i, bird[i]);
```

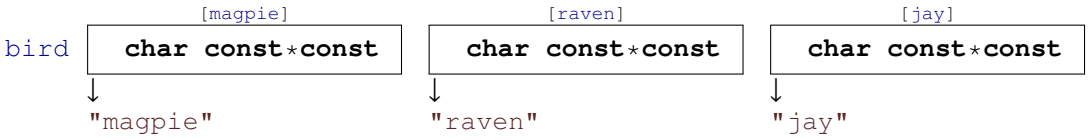
This declares a new integer type **enum** `corvid` for which we know four different values. As you might have guessed, positional values start from `0` onward, so in our

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

[23] Note also that using a non-Latin character such as π should be supported on modern platforms. Consider upgrading to a newer one if your compiler does not support this.

example we have `raven` with value 0, `magpie` with 1, `jay` with 2, and `corvid_num` with 3. This last 3 is obviously the 3 we are interested in.

Takeaway 5.6.2 #1 *Enumeration constants have either an explicit or a positional value.*



Notice that this uses a different order for the array entries than before, and this is one of the advantages of the approach with enumerations: we do not have to manually track the order we used in the array. The ordering that is fixed in the enumeration type does that automatically.

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

Listing 5.1. An enumeration type and related array of strings

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

As for most other narrow types, there is not really much interest in declaring variables of an enumeration type as given here; for indexing and arithmetic, they would be converted to **`signed`** or **`unsigned`**, anyway. Even the enumeration constants themselves aren't necessarily of the enumeration type.

Takeaway 5.6.2 #2 *If all enumeration constants of a simple enumeration type fit into* ***`signed int`****, they have that type.*

So for small values, the interest really lies in the constants, not in the newly created type. We can thus name any **`signed int`** constant that we need, without even providing a ***tag***^{C} for the type name:

```
enum { p0 = 1, p1 = 2*p0, 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 must its value be determinable at compile time (no function call allowed), but also no evaluation of an object must participate as an operand to the value:

```
signed const     o42 = 42;
constexpr signed c42 = 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.
  d52 = c42 + 10, // Ok: c42 is a named constant.
};
```

Here, `o42` is an object, albeit **`const`**-qualified, so the expression for `c52` is not an "integer constant expression." With `c42`, we see that **`constexpr`**, in turn, can be used freely in such a context.

Takeaway 5.6.2 #3 *An integer constant expression must only evaluate objects that are declared with* **constexpr**.

So, principally, an ICE may consist of any operations with integer literals, enumeration constants, **constexpr** objects, **alignof**[24] and **offsetof** subexpressions, and eventually some **sizeof** subexpressions.[25]

Before C23, even if the value was an ICE, to be able to use it to define an enumeration constant, we had to ensure that the value fit into a **signed int**. This has changed with C23.

Takeaway 5.6.2 #4 *If enumeration constants do not fit into* **signed int**, *if possible, the enumeration type is adjusted such that it can store all enumeration constants.*

Takeaway 5.6.2 #5 *If enumeration constants do not fit into* **signed int**, *the constants have the enumeration type.*

Note that it may actually happen that there is no type that can hold all the values for the constants:

```
enum tooLarge { minimus = LLONG_MIN, maximus = ULLONG_MAX, };
```

Unless the compiler finds an extended integer type that is wider than **signed long long**, this line will likely not compile.

The fact that enumeration types are adjusted can be convenient when we are not really interested in the type:

```
enum wide { minimal = LONG_MIN, maximal = LONG_MAX, };
typedef enum wide wide;
```

Here, it depends on the platform if **long** is wider than **signed**, so the *underlying type* of `wide` may be either of them, depending on the circumstances. C23 also introduced new syntax to force the underlying type to a specific one.

```
enum wider : long { minimer = LONG_MIN, maximer = LONG_MAX, };
typedef enum wider wider;
```

A colon followed by an integer type indicates the underlying type and also forces that the enumeration constants have the type of enumeration, even if the values would fit into **signed**:

```
enum narrow : unsigned char { zero, one, };
typedef enum narrow narrow;
```

The property that an enumeration type is adjusted such that it fits all its constants could have surprising effects for the users of the type and should probably not be abused. So it is preferable to specify the underlying integer type explicitly, whenever that is possible.

Takeaway 5.6.2 #6 *If the enumeration constants potentially do not all fit into* **signed int**, *specify the underlying integer type of an enumeration type.*

This is particularly important if the underlying type could be a signed or an unsigned type, as in the following:

```
enum large { down = 0, up = 0xFFFF'FFFF, }; // ambigouous, don't use
typedef enum large large;
```

[24] Since C23; previously **_Alignof**.

[25] We will handle the latter two concepts in subsections 12.7 and 12.1.

Here, depending on the width of **signed int**, the constant **0x**FFFF'FFFF could have any of types **signed**, **unsigned**, **signed long**, **unsigned long**, **signed long long**, or **unsigned long long**, and thus the underlying integer type could become any of those. For an occasional reader, it would be better to clearly state the intent:

```
enum eInt : signed int { dInt = 0, uInt = 0xFFFF'FFFF, };
typedef enum eInt eInt;
enum eSig : typeof(4'294'967'295) { dSig = 0, uSig = 4'294'967'295, };
typedef enum eSig eSig;
enum e32 : uint32_t { d32 = 0, u32 = 0xFFFF'FFFF, };
typedef enum e32 e32;
```

The first definition for eInt would only compile if **signed int** has a width of more than 32. The second for eSig uses the **typeof** feature (which will be introduced in more detail much later in section 18) to state explicitly that we want the type to be the one of the decimal constant. That type is always signed. The third for e32 uses the type definition **uint32_t** (see following discussion) to indicate that the sought type is an unsigned type with a width of at least 32.

5.6.3. *Macros.* Prior to C23, there was no other mechanism to declare constants in the strict sense of the C language of other types than **signed int**. Instead, C proposes another powerful mechanism that introduces textual replacement of the program code: ***macros***^{C}. A macro is introduced by a ***preprocessor***^{C} **#define**:

```
# define M_PI 3.141'592'653'589'793'238'46
```

This macro definition has the effect that the identifier M_PI is replaced in the following program code by the **double** constant. Such a macro definition consists of five 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 3.141'592'653'589'793'238'46
(5) A terminating newline character

With this trick, we can declare textual replacement for constants of **unsigned**, **size_t**, and **double**. In fact, the implementation-imposed bound of **size_t**, **SIZE_MAX**, is defined, as well as many of the other system features we have already seen:

EXIT_SUCCESS, **not_eq**, **complex** ...

Here, in this book, such C standard macros are generally printed in **dark red**.

The spelling of these examples from the C standard is not representative for the conventions that are generally used in a large majority of software projects. Most of them have quite restrictive rules such that macros visually stick out from their surroundings.

Takeaway 5.6.3 #1 *Macro names are in all caps.*

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

5.6.4. *Compound literals.* For types that don't have literals that describe their constants, things get even more complicated. For macros, we have to use ***compound literals***^{C} on the replacement side. Such a compound literal has the form

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

That is, a type in parentheses, followed by an initializer. Here's an example:

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

With that, we could leave out the `bird` array and rewrite our **`for`** loop:

```
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 declare something that behaves similarly to a constant of a chosen type, it isn't a constant in the sense that we have previously discussed.

Takeaway 5.6.4 #1 *A compound literal defines an object.*

Overall, this form of macro has some pitfalls:

- Compound literals, as introduced so far, 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 slack to generate good binary code for such a macro replacement.
- There *must* be at least one space character 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.

Takeaway 5.6.4 #2 *Don't hide a terminating semicolon inside a macro.*

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

Takeaway 5.6.4 #3 *Right-indent continuation markers for macros to the same column.*

As you can see in the example, this helps to visualize the entire spread of the macro definition easily.

5.6.5. *The* ***constexpr*** *construct.* All these techniques are not very helpful in contexts in which, say, we would need named constants for complicated types that are to be used as an initializer in file scope. Here, an initializer has to be a constant expression. C23 introduced the **`constexpr`** construct, which may be applied to declarations and also to compound literals. A declaration equivalent to our previously introduced macro `M_PI` is the following:

```
constexpr double π = 3.141'592'653'589'793'238'46;
```

Using such a **`constexpr`** has the advantage that the constant is checked at compile time where it is declared: a conversion that leads to a change of value is an error. For example,

```
constexpr unsigned πflat = 3.141'592'653'589'793'238'46; // error
```

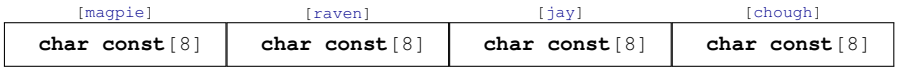
results in a compiler error because significant digits after the decimal point on the right are lost when converting to the type **unsigned** on the left.

Takeaway 5.6.5 #1 *The initializer of a* ***constexpr*** *must fit exactly.*

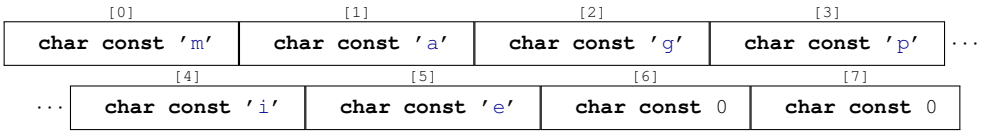
constexpr can also be used for compound literals:

```
# define CORVID_NAMES /**/          \
(constexpr char[8][corvid_num]){    \
  [chough] = "chough",              \
  [raven] = "raven",                \
  [magpie] = "magpie",              \
  [jay] = "jay",                    \
}
```

Observe that here we changed to an array of `corvid_num` arrays of 8 characters each. Each of these 8 character arrays is initialized with the indicated values and then is filled with zeros to the end. So, schematically, the compound literal as a whole looks as follows:



For example, the first of these array elements, which is itself an array, looks like this:



For all these characters, by using **constexpr**, the compiler now knows that they are not intended to change during execution, and the **const** qualification is implied. That knowledge could be used to make our program more efficient, either faster (in some sense) or by using less space. If the array, hidden behind the macro, is indexed directly (such as in `CORVID_NAMES[raven]`), the whole array is not needed. Only the corresponding string literal (here `"raven"`) could be used directly by the compiler. Even more, the compiler would be allowed to use the same string literal `"raven"` for all the occurrences with index `raven`, the same for all with `magpie`, etc.

5.7. Binary representions. The *binary representation* of a type is a *model* that describes the possible values for that type. It is not the same as the in-memory *object representation* that describes the more or less physical storage of values of a given type.

Takeaway 5.7 #1 *The same value may have different binary representations.*

5.7.1. *Unsigned integers.* We have seen that unsigned integer types are those arithmetic types for which the standard arithmetic operations have a nice, closed mathematical description, as we have seen in takeaway 4.2.2 #4 they are closed under arithmetic operations. In mathematical terms, they implement a *ring*, $\mathbb{Z}_N$, the set of integers modulo some number N. The values that are representable are $0, \dots, N-1$. The maximum value $N-1$ completely determines such an unsigned integer type and is made available through a macro with terminating **_MAX** in the name. For the basic unsigned integer types, these are **UINT_MAX**, **ULONG_MAX**, and **ULLONG_MAX** , and they are provided through `<limits.h>`. As we have seen, the one for **size_t** is **SIZE_MAX** from `<stdint.h>`. <limits.h> <stdint.h>

The binary representation for non-negative integer values is always exactly what the term indicates: such a number is represented by binary digits $b_0, b_1, \dots, b_{p-1}$ called

TABLE 5.5. Bounds for scalar types used in this book

Name	[min, max]	Where	Typical
`size_t`	[0, `SIZE_MAX`]	`<stdint.h>`	$[0,\ 2^w - 1]$, $w = 32$ or 64
`double`	[±`DBL_MIN`, ±`DBL_MAX`]	`<float.h>`	$[\pm 2^{-w-2}, \pm 2^w]$, $w = 1{,}024$
`signed`	[`INT_MIN`, `INT_MAX`]	`<limits.h>`	$[-2^w, 2^w - 1]$, $w = 31$
`unsigned`	[0, `UINT_MAX`]	`<limits.h>`	$[0, 2^w - 1]$, $w = 32$
`bool`	[`false`, `true`]	`<stdbool.h>`	$[0, 1]$
`ptrdiff_t`	[`PTRDIFF_MIN`, `PTRDIFF_MAX`]	`<stdint.h>`	$[-2^w, 2^w - 1]$, $w = 31$ or 63
`char`	[`CHAR_MIN`, `CHAR_MAX`]	`<limits.h>`	$[-128, 127]$ or $[0, 255]$
`unsigned char`	[0, `UCHAR_MAX`]	`<limits.h>`	$[0, 255]$

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, which, for unsigned types, is also the same as the *width*. For all unsigned types, these values can be determined from the corresponding macro, such as `UINT_WIDTH`, `ULONG_WIDTH`, and `ULLONG_WIDTH`. Bit b_0 is called the ***least-significant bit***^{C}, *LSB*, and b_{p-1} is the ***most-significant bit***^{C} (*MSB*).

Of the bits b_i that are 1, the one with minimal index i is called the ***least-significant bit set***^{C}, and the one with the highest index is the ***most-significant bit set***^{C}. For example, 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 least-significant bit set i is b_4, and the most-significant bit set is b_7. From (1), we see immediately that 2^p is the first value that cannot be represented with the type. Thus, $N = 2^p$ and the following observation holds.

Takeaway 5.7.1 #1 *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 haven't argued about the signedness of the type. These rules apply equally to signed and unsigned types. Only for unsigned types, we are lucky, and what we have said so far completely suffices to describe such an unsigned type.

Takeaway 5.7.1 #2 *Arithmetic on an unsigned integer type is determined by its precision.*

Finally, table 5.5 shows the bounds of some of the commonly used scalars throughout this book.

5.7.2. *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: 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 bit sets: `|`, `&`, 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 choose $A = 240$, representing $\{4, 5, 6, 7\}$, and $B = 287$, the bit set $\{0, 1, 2, 3, 4, 8\}$; see table 5.6. For the result of these operations, the total size of the base set, and thus the precision p, is not needed. As for the

TABLE 5.6. Effects of bitwise operators

Bit op	Value	Hex	$b_{15} \ldots b_8\ b_7 \ldots b_0$	Set op	Set
`V`	65,535	`0xFFFF`	`0b11111111'11111111`		{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
`A`	240	`0x00F0`	`0b00000000'11110000`		{4, 5, 6, 7}
`~A`	65,295	`0xFF0F`	`0b11111111'00001111`	$V \setminus A$	{0, 1, 2, 3, 8, 9, 10, 11, 12, 13, 14, 15}
`-A`	65,296	`0xFF10`	`0b11111111'00010000`		{4, 8, 9, 10, 11, 12, 13, 14, 15}
`B`	287	`0x011F`	`0b00000001'00011111`		{0, 1, 2, 3, 4, 8}
`A|B`	511	`0x01FF`	`0b00000001'11111111`	$A \cup B$	{0, 1, 2, 3, 4, 5, 6, 7, 8}
`A&B`	16	`0x0010`	`0b00000000'00010000`	$A \cap B$	{4}
`A^B`	495	`0x01EF`	`0b00000001'11101111`	$A \Delta B$	{0, 1, 2, 3, 5, 6, 7, 8}

arithmetic operators, there are corresponding assignment operators `&=`, `|=`, and `^=`, respectively.[Exs 26][Exs 27][Exs 28][Exs 29]

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 30][Exs 31]

All of these operators can be written with identifiers: ***bitor***, ***bitand***, ***xor***, ***or_eq***, ***and_eq***, ***xor_eq***, and ***compl*** if you include header `<iso646.h>`. A typical usage of bit sets is for *flags*, variables that control certain settings of a program:

<iso646.h>

```
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, 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 whether a particular corvid is present.

[Exs 26] Show that $A \setminus B$ can be computed by `A - (A&B)`.

[Exs 27] Show that `V + 1` is 0.

[Exs 28] Show that `A^B` is equivalent to `(A - (A&B)) + (B - (A&B))` and `A + B - 2*(A&B)`.

[Exs 29] Show that `A|B` is equivalent to `A + B - (A&B)`.

[Exs 30] Show that `~B` can be computed by `V - B`.

[Exs 31] Show that `-B = ~B + 1`.

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 you memorize the particular spelling of these operators. On the other hand, keep in mind that the operators `||` and `&&` have short-circuit evaluation, so be sure to distinguish them clearly from the bit operators.

<stdbit.h>

Since C23, another set of bit operations is provided through the `<stdbit.h>` header. These include the functionality to count the bits with value 1 (the size of a set) with **stdc_count_ones**, to detect whether there is exactly one bit with value 1 (if a set is a singleton) with **stdc_has_single_bit** to provide the bit with the highest number that holds the value 1 with **stdc_bit_width** or to return the singleton set with the highest numbered element **stdc_bit_floor**. Since this header is new with C23, your platform might not yet have it. Another new feature is the preprocessor test **__has_include**, which you can use to query whether a header file can be found or not:

```
#if !__has_include(<stdbit.h>)
# error "this␣file␣needs␣the␣<stdbit.h>␣header"
#endif
```

5.7.3. *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. For example, for $A = 240$, the set $\{4, 5, 6, 7\}$, `A << 2` is $240 \cdot 2^2 = 240 \cdot 4 = 960$, which represents the set $\{6, 7, 8, 9\}$. Resulting bits that don't fit into the binary representation for the type are simply omitted. In our example, `A << 9` would correspond to set $\{13, 14, 15, 16\}$ (and value 122880), but since there is no bit 16, the resulting set is $\{13, 14, 15\}$, value 57344.

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

Takeaway 5.7.3 #1 *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 toward the less-significant bits. Analogously, this corresponds to an integer division by a power of 2. Bits in positions less than or equal to the shift value are omitted from the result. Observe that for this operation, the precision of the type isn't important.[Exs 32] There are also corresponding assignment operators <<= and >>=.

The primary use of the left-shift operator << is specifying powers of 2. In our example, we can now replace the **#define**s:

```
#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.

5.7.4. *Boolean values.* The Boolean data type in C is also considered an unsigned type. Remember that it has only values **false** and **true**, corresponding to 0 and 1,

[Exs 32] Show that the bits that are "lost" in an operation `x>>n` correspond to the remainder `x % (1ULL << n)`.

so there are no negative values. Before C23, the names **bool**[33] as well as the constants **false** and **true** only came through the inclusion of <stdbool.h>. If you have to maintain an old code base or have to ensure backwards compatibility for older systems, you should still use that include; on systems that don't need it, it shouldn't cause any harm. <stdbool.h>

Treating **bool** as an unsigned type is a stretch of the concept. Assignment to a variable of that type doesn't follow the modulus rule of takeaway 4.2.2 #3, but a special rule for Boolean values (takeaway 3.1 #1).

You will probably rarely need **bool** variables. They are only useful if you 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 unfortunately many experienced C programmers still don't use it.

5.7.5. *Signed integers.* Signed types are a bit more complicated than unsigned types. A C implementation has to decide about two points:

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

Signed and unsigned types come in pairs according to their integer rank, with the notable two exceptions from table 5.1: **char** and **bool**. The binary representation of the signed type is constrained by the inclusion diagram that we previously saw.

Takeaway 5.7.5 #1 *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 why the maximum value for any integer type can be expressed so easily (takeaway 5.7.1 #1): signed types also have a precision, p, that determines the maximum value of the type.

The next thing the standard prescribes is that signed types have one additional bit, the ***sign bit***^{C}. If it is 0, we have a positive value; if it is 1, the value is negative. Historically, there have been different concepts of how such a sign bit can be used to obtain a negative number, but C23 clarified that such that nowadays only the ***two's complement***^{C} is allowed for the ***sign representations***^{C}.

Previously, there also had been ***sign and magnitude***^{C} and ***ones' complement***^{C}, but nowadays they only have historical or exotic relevance: for sign and magnitude, the magnitude is taken as positive values, and the sign bit simply specifies that there is a minus sign. Ones' complement takes the corresponding positive value and complements all bits. Both representations have the disadvantage that two values evaluate to 0: there is a positive and a negative 0. Because there is no active platform that has these representations, they have now been removed from the C standard; you should only encounter them in history books or in maliciously intended recruitment tests.

The two's complement representation performs exactly the same arithmetic as we have seen for unsigned types, but the upper half of unsigned values (those with a high-order bit of 1) is interpreted as being negative. The following two functions are basically all that is needed to interpret unsigned values as signed values:

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

[33] Previously, the basic type was called _Bool. For backward compatibility with older sources, that name is still maintained, but you should not use it in new code.

TABLE 5.7. Negation for 16-bit unsigned integer types

Op	Value	b_{15} ... b_0
`A`	240	`0b00000000'11110000`
`~A`	65,295	`0b11111111'00001111`
`+1`	1	`0b00000000'00000001`
`-A`	65,296	`0b11111111'00010000`

Table 5.7 shows an example of how the negative of value `240` can be constructed. For unsigned types, `-A` can be computed as `~A + 1`.[Exs 34][Exs 35][Exs 36] Two's complement representation performs exactly the same bit operation for signed types as for unsigned types. It only *interprets* representations that have the high-order bit as being negative.

When done that way, 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 much 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 loop: the counter is incremented up to **`UINT_MAX`** and 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 loop, everything 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 the 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 (takeaway 5.1.3 #3) allows it to optimize the second loop to

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

That's right, an *infinite loop*. The only possibility that the code is valid is that `do_something` has a side effect so the program execution makes progress. Also, it may assume that the point after the loop is never reached, either the loop runs continues indefinitively or because it will reach an internal state that terminates execution. In section 15.4, we will discuss such situations in more detail.

Takeaway 5.7.5 #2 *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 it pleases for the operation itself ("Undefined? So, let's define it"), but it may also assume that it will never reach such a state and draw conclusions from that.

Commonly, a program that has reached an undefined state is referred to as "having" or "showing" ***undefined behavior***^{C}. This wording is a bit unfortunate; in many such cases, a program does not "show" any visible signs of weirdness. In the contrary, bad things will be going on that you will not even notice for a long time.

Takeaway 5.7.5 #3 *It is your responsibility to avoid undefined behavior of all operations.*

[Exs 34] Prove that for unsigned arithmetic, `A + ~A` is the maximum value.

[Exs 35] Prove that for unsigned arithmetic, `A + ~A` is -1.

[Exs 36] Prove that for unsigned arithmetic, `A + (~A + 1) == 0`.

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

In the sequel, we will avoid talking about undefined behavior and generally refer to *program failure* because that is what is important for your program. Such a failure is, in general, unreliable (often referred to as *byzantine*); that is, none of the components of the execution are reliable anymore. The possible range of effects includes it going unnoticed to doing real harm to your platform or your data. We will dedicate a whole section of the book to program failure (section 15).

Takeaway 5.7.5 #4 *If the program state reaches an operation with undefined behavior, the execution has failed.*

Basically, what we have 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 don't need it, avoid it. We say that a program performs a ***trap***^{C} (or just *traps*) if it is terminated abruptly before its usual end.

Takeaway 5.7.5 #5 *Signed arithmetic may trap badly.*

One of the things that might already overflow for signed types is negation. We have seen that **INT_MAX** has all bits with the exception of the sign bit set to 1. **INT_MIN** then has the "next" representation: the sign bit set to 1 and all other bits set to 0. The corresponding value is not −**INT_MAX**.[Exs 37]

Takeaway 5.7.5 #6 ***INT_MIN*** < −***INT_MAX***

Or, stated otherwise, the positive value −**INT_MIN** is out of bounds since the *value* of the operation is larger than **INT_MAX**.

Takeaway 5.7.5 #7 *Negation may overflow for signed arithmetic.*

For signed types, bit operations work with the binary representation. The shift operations then become really messy. The semantics of what such an operation is for a negative value is not clear.

Takeaway 5.7.5 #8 *Use unsigned types for bit operations.*

5.7.6. *Fixed-width integer types.* The width (and thus the precision) of the integer types that we have seen so far can be inspected by using macros from `<limits.h>`, such as **UINT_WIDTH** and **LONG_WIDTH**. The C standard only guarantees a minimal width for them. For the unsigned types, these are

<limits.h>

type	minimal precision
bool	1
unsigned char	8
unsigned short	16
unsigned	16
unsigned long	32
unsigned long long	64

[Exs 37] Show that **INT_MIN**+**INT_MAX** is −1.

Under usual circumstances, these guarantees should give you enough information; but under some technical constraints, such guarantees might not be sufficient, or you might want to emphasize a particular precision. This may be the case if you want to use an unsigned quantity to represent a bit set of a known maximal size. If you know that 32 bits will suffice for your set, depending on your platform, you might want to choose **unsigned** (if it happens to have 32 bits) or **unsigned long** (if **unsigned** is too narrow) to represent it.

<stdint.h>

The C standard provides names for *exact-width integer types* in <stdint.h>. As the name indicates, they are of an exact prescribed "width," which for provided unsigned types is guaranteed to be the same as their precision.

Takeaway 5.7.6 #1 *If the type* **uint***N*_t *is provided, it is an unsigned integer type with exactly* N *bits of width and precision.*

Takeaway 5.7.6 #2 *If the type* **int***N*_t *is provided, it is signed and has a width of exactly* N *bits and a precision of* $N - 1$.

The **typedef** must be provided if types with the corresponding properties exist.[38]

Takeaway 5.7.6 #3 *If the types with the required properties exist for a value* N, **int***N*_t *and* **uint***N*_t *must be provided.*

Nowadays, platforms usually provide **uint8_t**, **uint16_t**, **uint32_t**, and **uint64_t** unsigned types and **int8_t**, **int16_t**, **int32_t**, and **int64_t** signed types, with more types being added, such as **uint128_t** and **int128_t**. Their presence and bounds can be tested with the macros **UINT8_WIDTH**, ..., **UINT128_WIDTH** for unsigned types and **INT8_WIDTH**, ..., **INT128_WIDTH**, respectively.[Exs 39]

To encode literals of the requested type, there are the macros **UINT8_C**, ..., **UINT64_C**, and **INT8_C**, ..., **INT128_C**, respectively. For example, on platforms where **uint64_t** is **unsigned long**, **INT64_C**(1) usually expands to something like 1UL.

Takeaway 5.7.6 #4 *For any of the fixed-width types that are provided, width* **_WIDTH**, *minimum* **_MIN** *(only signed), maximum* **_MAX**, *and literals* **_C** *macros are provided, too.*

Since we cannot know the type behind such a fixed-width type, it would be difficult to guess the correct format specifier to use for **printf** and friends. Since C23, the "w*N*" length specifiers can be used for this task, where N is the width of the type:[40]

```
uint32_t n = 78;
int64_t big = (-UINT64_C(1))>>1;     // Same value as INT64_MAX
printf("n is %w32u, and big is %w64d\n", n, big);
```

The availability of the width macros (and some other specification tricks) since C23 makes it possible that types can be provided that cannot even be fully handled by the macro-preprocessor. In particular, most modern desktop computers have hardware support for 128 bit types, and these can now be exposed to C integer types as **int128_t** and **uint128_t**. This can be particularly interesting if you have to handle large bit sets:

[38] Before C23, this was only guaranteed for values 8, 16, 32, and 64. Note that the only types that are guaranteed to exist then are **uint***N*_t and **int***N*_t for N that equals CHAR_BITS, even on platforms where CHAR_BITS is not 8.

[Exs 39] If they exist, the values of all these macros are prescribed by the properties of the types. Think of a closed formulas in N for these values.

[40] Previously, only macro substitutes were provided by the header <inttypes.h>. For example, for $N = 64$, there are **PRIb64**, **PRId64**, **PRIi64**, **PRIo64**, **PRIu64**, **PRIx64**, and **PRIX64**, for **printf** formats "%b", "%d", "%i", "%o", "%u", "%x" and "%X", respectively.

```
#if ULLONG_WIDTH < UINT128_WIDTH    // don't use UINT128_MAX
typedef uint128_t wideType;
#else
typedef unsigned long long wideType;
#endif
```

The width is guaranteed to be a relatively small number, so it will always be possible to provide preprocessor conditionals with that.

5.7.7. *Bit-precise integer types.* The exact-width integer types that we have seen previously only exist for a specific number of bits, generally only for powers of two. For quantities that should fit into a precise number of bits C23 introduced bit-precise integer types. They are specified with the **_BitInt** keyword:

```
unsigned _BitInt(3) u3 = 7wbu; // values          0, ..., 3, ... ,7
signed   _BitInt(3) s3 = 3wb;  // values -4, ..., 0, ..., 3
         _BitInt(3) s3 = 3wb;  // same
```

Here we see that these types also have literals, namely number literals that have suffixes of `wb` or `WB`, possibly combined with `u` or `U`, in their suffix. These have the particularity that they have the type with the least width that can represent the value. For example,

- `7wbu` needs 3 bits to represent the value 7 and is unsigned, so the type is **unsigned _BitInt**`(3)`.
- `3wb` needs 2 bits for the value 3 and reserves 1 bit for the sign, so it has type **signed _BitInt**`(3)`.
- `3wbu` needs 2 bits for the value but doesn't need a sign bit, so it has type **signed _BitInt**`(2)`.

These types always compute within the maximal width of the operands. For example, in

```
u3 + 1wbu
```

the first operand has a width of 3 and the second, a width of 1. So, the result of the operation has type **unsigned _BitInt**`(3)`, and for our choice of values, the mathematical result 8 of the addition wraps around for a final value of 0.

One possible use for these types is for specific constants that we want to have for a certain width:

```
constexpr unsigned _BitInt(3) max3u = -1;            // 0b111
constexpr unsigned _BitInt(4) max4u = -1;            // 0b1111
constexpr unsigned _BitInt(4) high4u= max4u - max3u; // 0b1000
constexpr   signed _BitInt(4) max4s = max3u;         // 0b0111
constexpr   signed _BitInt(4) min4s = ~max4s;        // 0b1000
```

The types exist for all widths from `1` (for unsigned types) or `2` (for signed types) up to the value **BITINT_MAXWIDTH**, defined in `<limits.h>`; this maximal width is always greater than or equal to the width of **unsigned long long**, **ULLONG_WIDTH**. This allows us to specify all integer literals with exact value and signedness. For example, the following monster is a signed integer literal with a precision of 127 bit and thus of type **_BitInt**`(128)`: <limits.h>

```
0x7FFF'FFFF'FFFF'FFFF'FFFF'FFFF'FFFF'FFFFwb
```

So if that type exists (**BITINT_MAXWIDTH** ≥ 128), it can be used for the initialization of variables of type **int128_t**, even if literals for that type itself are not supported.

5.7.8. *Floating-point data.* Whereas integers come near the mathematical concepts of $\mathbb{N}$ (unsigned) or $\mathbb{Z}$ (signed), floating-point types are close to $\mathbb{R}$ (non-complex) or $\mathbb{C}$ (complex). The way they differ from these mathematical concepts is twofold. First, there is a size restriction on what is presentable. This is similar to what we have seen for integer types. The include file `<float.h>`, for example, has constants **DBL_MIN** and **DBL_MAX** that provide us with the minimal and maximal values for **double**. But be aware that here, **DBL_MIN** is the smallest number that is strictly greater than `0.0`; the smallest negative **double** value is −**DBL_MAX**.

<float.h>

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

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

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

s	Sign (± 1)
e	Exponent, an integer
$f_1, \ldots, f_p$	values 0 or 1, the mantissa bits

For the exponent, we have $e_{min} \le e \le e_{max}$. p, the number of bits in the mantissa, is called *precision*. The floating-point value is then given by this 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}, $-1'021$), and **DBL_MAX_EXP** (e_{max}, $1'024$).

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

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

which corresponds to the decimal value $-0.156,25$. 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 41]

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

Takeaway 5.7.8 #1 *Floating-point operations are neither associative, commutative, nor distributive.*

So, basically, floating points lose all the nice algebraic properties 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 42] For example, 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 whether two computations have the "same" result. Such

[Exs 41] Show that all representable floating-point values with $e > p$ are multiples of 2^{e-p}.

[Exs 42] Print the results of the following expressions: `1.0E-13 + 1.0E-13` and `(1.0E-13 + (1.0E-13 + 1.0)) - 1.0`.

investigations are often cutting-edge research questions, so we cannot expect to be able to assert equality or inequality. We are only able to tell that the results are "close."

Takeaway 5.7.8 #2 *Never compare floating-point values for equality.*

The representation of the complex types is straightforward and identical to an array of two elements of the corresponding real floating-point type. To access the real and imaginary part of a complex number, two type generic macros also come with the header `<tgmath.h>`: **`creal`** and **`cimag`**. For any `z` of one of the three complex types, we have that `z == creal(z) + cimag(z)*I`.[43] `<tgmath.h>`

[43] We will learn about such function-like macros in section 8.1.2.

Summary

- C programs run in an *abstract state machine* that is mostly independent of the specific computer where it is launched.
- All basic C types are kinds of numbers, but not all of them can be used directly for arithmetic.
- Values have a type and a binary representation.
- When necessary, types of values are implicitly converted to fit the needs of particular places where they are used.
- Variables must be explicitly initialized before their first use.
- Integer computations give exact values as long as there is no overflow.
- Floating-point computations only give approximated results that are cut off after a certain number of binary digits.

6. Derived data types

This section covers

- Grouping objects into arrays
- Using pointers as opaque types
- Combining objects into structures
- Giving types new names with **`typedef`**

All other data types in C are derived from the basic types that we know now. There are four strategies for deriving data types. Two of them are called *aggregate data types* because they combine multiple instances of one or several other data types:

Arrays: These combine items that all have the same base type (subsection 6.1).

Structures: These combine items that may have different base types (subsection 6.3).

The two other strategies to derive data types are more involved:

Pointers: Entities that refer to an object in memory.
Pointers are by far the most involved concept, and we will delay a full discussion of them to section 11. Here, in subsection 6.2, we will only discuss them as opaque data types, without even mentioning the real purpose they fulfill.

Unions: These overlay items of different base types in the same memory location.
Unions require a deeper understanding of C's memory model and are not of much use in a programmer's everyday life, so they are only introduced later, in subsection 12.2.

There is a fifth strategy that introduces new names for types: **`typedef`** (subsection 6.4). Unlike the previous four, this does not create a new type in C's type system, but only creates a new name for an existing type. In that way, it is similar to the definition of macros with **`#define`**, thus the choice for the keyword for this feature.

6.1. Arrays. Arrays allow us to group objects of the same type into an encapsulating object. We will see pointer types later (section 11), but many people who come to C are confused about arrays and pointers. This is completely normal. Arrays and pointers are closely related in C, and to explain them, we face a *chicken 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: we will start with arrays and stay with them as long as possible before introducing pointers. This may seem "wrong" to some of you, but remember that everything stated here must be viewed based on the *as-if* rule (takeaway 5.1.3 #3). We will first describe arrays in a way that is consistent with C's assumptions about the abstract state machine.

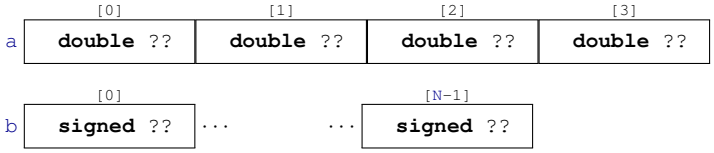
Takeaway 6.1 #1 *Arrays are not pointers.*

Later, we will see how these two concepts relate, but for the moment, it is important to read this section without prejudice about arrays; otherwise, you will delay your ascent to a better understanding of C.

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

```
double a[4];
signed b[N];
```

Here, `a` comprises four subobjects of type **`double`**, and `b` comprises `N` of type **`signed`**. We visualize arrays with diagrams like the following, with a sequence of boxes of their base type:

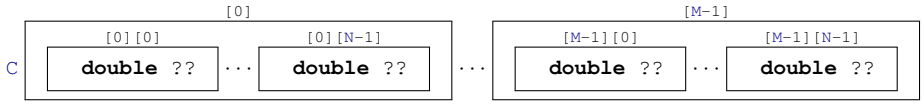


The dots ··· here indicate that there may be an unknown number of similar items between the two boxes.

The type that composes an array may itself again be an array, forming a ***multidimensional array***[C]. The declarations for those 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]`. This means we have to read a nested array declaration from inside out to describe its structure:



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

Initializers can use *designated initializers* (also using `[]` notation) to pick the specific position to which an initialization applies. The example code in listing 5.1 contains such initializers. During development, designated initializers help to make our code robust against small changes in array sizes or positions.

6.1.2. *Array operations.* Arrays are really just objects of a different type than we have seen so far.

Takeaway 6.1.2 #1 *An array in a condition evaluates to* ***`true`***.

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

Takeaway 6.1.2 #2 *There are array objects but no array values.*

So, arrays can't be operands for the value operators in table 4.1, and there is no arithmetic declared on arrays themselves.

Takeaway 6.1.2 #3 *Arrays can't be compared.*

Arrays also can't be on the value side of the object operators in table 4.2. Most of the object operators are likewise ruled out from having arrays as object operands, either because they assume arithmetic or because they have a second value operand that would have to be an array, too.

Takeaway 6.1.2 #4 *Arrays can't be assigned to.*

From table 4.2, we also know that there are only four operators left that work on arrays as object operators. And we know the operator `[]`.[44] The *array decay* operation, the address operator `&`, and the **`sizeof`** operator will be introduced later.

6.1.3. *Array length.* There are two categories of arrays: ***constant-length arrays***^{C} (CLAs) and ***variable-length arrays***^{C} (VLAs). The first are a concept that has been present in C since the beginning; this feature is shared with many other programming languages. The second was introduced in C99 and is relatively unique to C. It has some restrictions on its usage.

Takeaway 6.1.3 #1 *VLAs only can have default initializers.*

Takeaway 6.1.3 #2 *VLAs can't be declared outside functions.*

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

Takeaway 6.1.3 #3 *The length of an FLA is determined by an integer constant expression or by an initializer.*

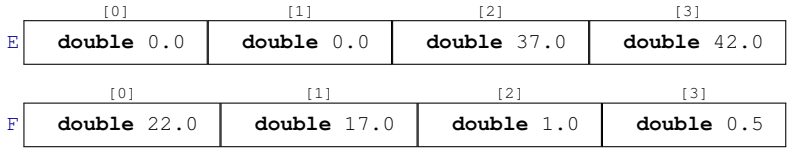
For the first of these alternatives, the length is known at compile time through an ICE (introduced in subsection 5.6.2). There is no type restriction for the ICE: any integer type will do.

Takeaway 6.1.3 #4 *An array-length specification must be strictly positive.*

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

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

Here, `E` and `F` 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 is still an FLA:



All other array variable declarations lead to VLAs.

Takeaway 6.1.3 #5 *If the length is not an integer constant expression, an array is a VLA.*

The status of VLAs has been changing in the history of C: they were introduced as mandatory in C99 and passed to optional in C11. For C23, VLA as such are optional for automatic objects (queried with the macro **`__STDC_NO_VLA__`**) but their types, pointers to them, and thus VLA parameters are again mandatory.

The length of an array can be computed with the **`sizeof`** operator. That operator provides the size of any object,[45]so the length of an array can be calculated using simple division.[46]

[44]The real C jargon story about arrays and `[]` is a bit more complicated. Let us apply the **as-if** rule (takeaway 5.1.3 #3) to our explanation. All C programs behave *as if* the `[]` are directly applied to an array object.

[45]Later, we will see what the unit of measure for such sizes is.

[46]Note also that the **`sizeof`** operator comes in two different syntactical forms. If applied to an object, as it is here, it does not need parentheses, but they would be needed if we applied it to a type.

Takeaway 6.1.3 #6 *The length of an array* `A` *is* `(sizeof A) / (sizeof A[0])`.

That is, it is the total size of the array object, 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 saw for the prototype of **`printf`**, such parameters may have `[]` that make them look like arrays. Because we cannot produce *array values* (takeaway 6.1.2 #2), array parameters cannot be passed by value, and thus array parameters as such would not make much sense. Because of that, these parameters lose information and behave much differently than we might expect.

Takeaway 6.1.4 #1 *The innermost dimension of an array parameter to a function is lost.*

Takeaway 6.1.4 #2 *Don't use the* **`sizeof`** *operator on array parameters to functions.*

Takeaway 6.1.4 #3 *Array parameters behave* as if *the array is* ***passed by reference***^{C}.

Unfortunately, for this level, you just have to accept these facts as is; it will only be possible to explain the mechanism when pointers are fully introduced.

Take the example shown in listing 6.1.

Listing 6.1. A function with an array parameter

```
#include <stdio.h>

void swap_double(double a[static 2]) {
  auto 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", A[0], A[1]);
}
```

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`.

Challenge 6 (Linear algebra). *Some of the most important problems for which arrays are used stem from linear algebra.*
Can you write functions that do vector-to-vector or matrix-to-vector products at this point? What about Gaussian elimination or iterative algorithms for matrix inversion?

6.1.5. *Strings are special.* There is a special kind of array that we have encountered several times and that, in contrast to other arrays, even has literals: ***strings***^{C}.

Takeaway 6.1.5 #1 *A string is a* `0`*-terminated array of* **`char`**.

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

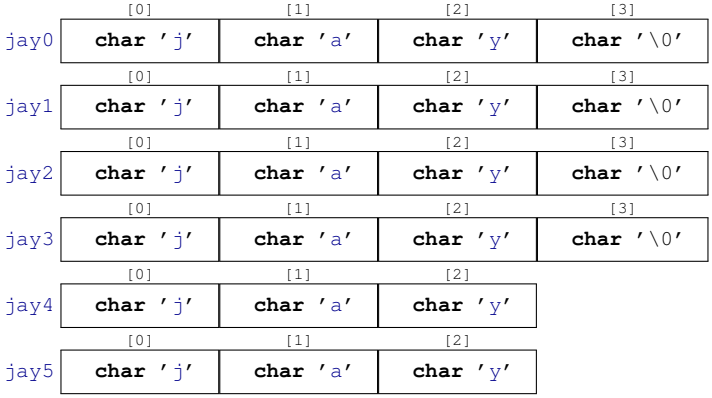
Like all arrays, strings can't be assigned to, but they can be initialized from string literals:

```
char jay0[] = "jay";
char jay1[] = { "jay" };
char jay2[] = { 'j', 'a', 'y', 0, };
char jay3[4] = { 'j', 'a', 'y', };
```

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

```
char jay4[3] = { 'j', 'a', 'y', };
char jay5[3] = "jay";
```

These both cut off after the `'y'` character and so are not `0`-terminated:



We briefly saw 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 the characters of the Latin alphabet, Arabic digits, and punctuation characters that we use for coding in C. It usually doesn't contain special characters (for example, *ä*, *á*) or characters from completely different writing systems.

The vast majority of platforms nowadays use American Standard Code for Information Interchange (ASCII) 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, which use this encoding 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 require an array argument start their names with `mem`, and those that also require that their arguments are strings start with `str`. Listing 6.2 uses some of the functions that are described next. <string.h>

Functions that operate on **char** arrays are as follows:

- **memcpy**`(target, source, len)` can be used to copy one array to another. These have to be known to be distinct arrays. The number of **char**s to be copied must be given as a third argument `len`.
- **memcmp**`(s0, s1, len)` compares two arrays in lexicographic order. That is, it first scans the initial segments 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`.

Next are the string functions:

LISTING 6.2. Using some of the string functions

```
#include <string.h>
#include <stdio.h>
int main(int argc, char* argv[argc+1]) {
  size_t const len = strlen(argv[0]); // Computes the length
  // Initialized VLA, C23
  // terminates array with 0 character
  char name[len+1] = { };
  // Copies the program name
  memcpy(name, argv[0], len);
  if (!strcmp(name, argv[0])) {
    printf("program␣name␣\"%s\"␣successfully␣copied\n",
           name);
  } else {
    printf("copying␣%s␣leads␣to␣different␣string␣%s\n",
           argv[0], name);
  }
}
```

- **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 it is 0-terminated.
- **strcpy**(target, source) works similarly 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.
- **strdup**(source) and **strndup**(source, len) (since C23) work similar as **strcpy**, but they first allocate storage for the copy. We will see much later in section 13.1 how such allocated storage works. Again, for **strdup**, the argument source must be 0-terminated; for **strndup**, this requirement is a bit relaxed because the function will never read beyond the $\texttt{len}^{\text{th}}$ character of source.
- **strcmp**(s0, s1) compares two arrays in lexicographic order, similarly to **memcmp**, but it may not take some specialties of the language environment into account, for example at which position of the alphabet the character "ä" is considered. 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 lexicographic order, respecting language-specific environment settings. We will learn how to properly set this in subsection 8.7.
- **strchr**(s, c) is similar to **memchr**, but the string s must be 0-terminated.
- **strspn**(s0, s1) returns the length of the initial segment in s0 that consist of characters that also appear in s1.
- **strcspn**(s0, s1) returns the length of the initial segment in s0 that consist of characters that do not appear in s1.

Takeaway 6.1.5 #2 *Using a string function with a non-string leads to program failure.*

In real life, common symptoms for such misuse include

- High execution 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 all your strings really are strings. If you know the length of the character array, but you do not know whether it is 0-terminated, **memchr** and pointer arithmetic (see section 11) can be used as a safe replacement for **strlen**. Analogously, if a character array is not known to be a string, it is better to copy it using **memcpy**.[Exs 47]

In the discussion so far, I have been hiding an important detail from you: the prototypes of the functions. For the string functions, they can be written as

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

Other than the bizarre return type of **strcpy**, **strchr**, **strdup** and **strndup**, this looks reasonable.[48] The parameter arrays are arrays of unknown length, so the `[static 1]` correspond to arrays of at least one **char**. **strlen**, **strspn**, and **strcspn** will return a size, and **strcmp** will return 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 in level 2, section 11, that we will see why and how these new concepts of pointers and **void** type occur.

CHALLENGE 7 (Adjacency matrix). *The adjacency matrix of a graph G is a matrix `A` that holds a value **true** or **false** in element `A[i][j]` if there is an arc from node `i` to node `j`. At this point, can you use an adjacency matrix to conduct a breadth-first search in a graph G? Can you find connected components? Can you find a spanning tree?*

CHALLENGE 8 (Shortest path). *Extend the idea of an adjacency matrix of a graph G to a* distance matrix *`D` that holds the distance when going from point `i` to point `j`. Mark the absence of a direct arc with a very large value, such as **SIZE_MAX**.*
Can you find the shortest path between two nodes `x` and `y` given as an input?

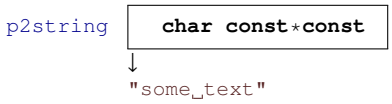
[Exs 47] Use **memchr** and **memcmp** to implement a bounds-checking version of **strcmp**.

[48] Note that since C23, **strchr** is a type-generic macro (see 18.1.7) that can consistently be used with a **char*** or **char const*** argument without violating a **const**-contract.

6.2. Pointers as opaque types. We now have seen the concept of pointers pop up in several places, in particular as a **void*** argument and return type and as **char const*const** to manipulate references to string literals. Their main property is that they do not directly contain the information that we are interested in: rather, they refer, or *point*, to the data. C's syntax for pointers always has the peculiar *:

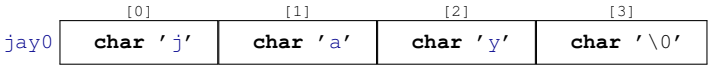
```
char const*const p2string = "some text";
```

It can be visualized like this:



Compare this to the earlier array jay0, which itself contains all the characters of the string that we want it to represent:

```
char jay0[] = "jay";
```



In this first exploration, we only need to know some simple properties of pointers. The binary representation of a pointer is completely up to the platform and is not our business.

Takeaway 6.2 #1 *Pointers are opaque objects.*

This means we will only be able to deal with pointers through the operations that the C language allows for them. As I said, most of these operations will be introduced later; in our first attempt, we will only need initialization, assignment, and evaluation.

One particular property of pointers that distinguishes them from other variables is their state.

Takeaway 6.2 #2 *Pointers are valid, null, or invalid.*

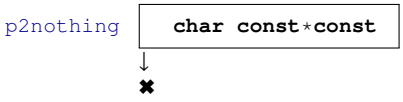
For example, our variable p2string is always valid because it points to the string literal "some text". Due to the second **const**, this association can never be changed.

Takeaway 6.2 #3 *Initialization or assignment with* ***nullptr*** *makes a pointer null.*

Take the following as an example:

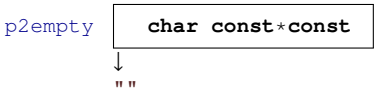
```
char const*const p2nothing = nullptr;
```

We visualize this special situation like this:



Note that this is different from pointing to an empty string:

```
char const*const p2empty = "";
```



Usually, we refer to a pointer in the null state as a ***null pointer***^{C}. Surprisingly, disposing of null pointers is really a feature.

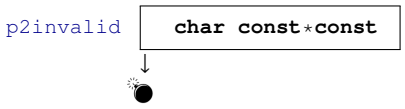
Takeaway 6.2 #4 *In logical expressions, pointers evaluate to* **`false`** *if they are null.*

Note that such tests can't distinguish valid pointers from invalid ones. So, the really "bad" state of a pointer is invalid, since this state is not observable.

Takeaway 6.2 #5 *Invalid pointers lead to program failure.*

An example of an invalid pointer could look like this:

```
char const*const p2invalid;
```



Because it is uninitialized, its state is indeterminate. Any evaluation of it would lead to an invalid value and leave your program in an undefined state (takeaway 5.7.5 #2). Thus, if we can't ensure that a pointer is valid, we *must* at least ensure that it is set to null.

Takeaway 6.2 #6 *Always initialize pointers.*

6.3. Structures. As we 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, ... element is acceptable. If it is not or if we have to combine objects of different type, then *structures*, introduced by the keyword **`struct`** come into play.

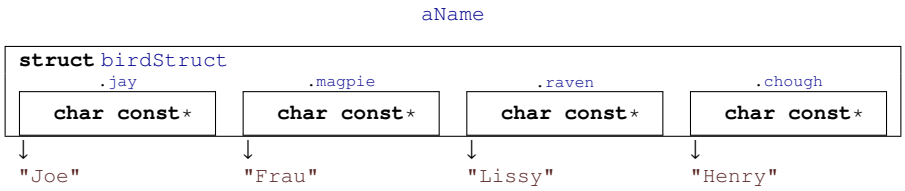
6.3.1. *Simple structures to access fields by name.* As a first example, let us revisit the corvids from section 5.6.2. There, we used a trick with an enumeration type to keep track of our interpretation of the individual elements of an array name. C structures allow for a more systematic approach by giving names to *members* (or *fields*) in an aggregate:

```
struct birdStruct {
  char const* jay;
  char const* magpie;
  char const* raven;
  char const* chough;
};
struct birdStruct const aName = {
  .chough = "Henry",
  .raven = "Lissy",
  .magpie = "Frau",
  .jay = "Joe",
};
```

That is, from line 1 to 6, we have the declaration of a new type, denoted as **`struct`** `birdStruct`. This structure has four ***members***^{C}, whose declarations look exactly like normal variable declarations. So instead of declaring four elements that are bound together in an array, here we name the different members and declare types for them. Such declaration of a structure type only explains the type; it is not (yet) the declaration of an object of that type and, even less, a definition for such an object.

Then, starting on line 7, we declare and define a variable (called `aName`) of the new type. In the initializer and in later usage, the individual members are designated

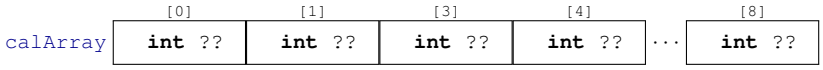
using a notation with a dot (`.`). Instead of `bird[raven]`, as in subsection 5.6.1, for the array we use `aName.raven` for the structure:



Please note that in this example, the individual members again only *refer* to the strings. For example, the member `aName.magpie` refers to an entity `"Frau"` that is located outside the box and is not considered part of the **`struct`** itself.

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 months and years can have different lengths, and so on. One possible way to organize such data for the nine different data that we need for such time stamps (see the following discussion) could be an array:

```
typedef int calArray[9];
```



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:

<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 members*, such as **`tm_sec`** for the seconds and **`tm_year`** for the year. Encoding a date, such as the date of this writing,

Terminal

```
0   > LC_TIME=C date -u
1  Wed Apr  3 10:00:47 UTC 2019
```

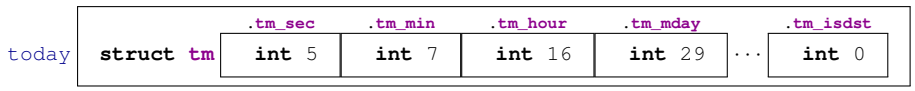
is relatively simple:

yday.c

```
29  struct tm today = {
30    .tm_year = 2019-1900,
31    .tm_mon  = 4-1,
32    .tm_mday = 3,
33    .tm_hour = 10,
34    .tm_min  = 0,
35    .tm_sec  = 47,
```

```
36  };
```

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



Note that this visualization of today has an extra "box" compared to calArray. Indeed, a proper **struct** type creates an additional level of abstraction. This **struct tm** is a proper type in C's type system.

Accessing the members of the structure is just as simple and has similar . syntax:

yday.c

```
37  printf("this␣year␣is␣%d,␣next␣year␣will␣be␣%d\n",
38         today.tm_year+1900, today.tm_year+1900+1);
```

A reference to a member such as today.**tm_year** can appear in an expression just like any variable of the same base type.

There are three other members in **struct tm** that we didn't even mention in our initializer list: **tm_wday**, **tm_yday**, and **tm_isdst**. Since we didn't mention them, they are automatically set to 0.

Takeaway 6.3.1 #1 *Omitted **struct** initializers force the corresponding member to 0.*

This can even go to the extreme that none of the members are initialized. Previously (takeaway 5.5 #3), we saw that there is a default initializer that works for all data types: {}.

So when we initialize **struct tm** as we did here, the data structure is not consistent; the **tm_wday** and **tm_yday** members don't have values that would correspond to the values of the remaining members. A function that sets this member to a value that is consistent with the others could be something like

yday.c

```
19  struct tm time_set_yday(struct tm t) {
20    // tm_mdays starts at 1.
21    t.tm_yday += DAYS_BEFORE[t.tm_mon] + t.tm_mday - 1;
22    // Takes care of leap years
23    if ((t.tm_mon > 1) && leapyear(t.tm_year+1900))
24      ++t.tm_yday;
25    return t;
26  }
```

It uses the number of days of the months preceding the current one, the **tm_mday** member, 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 member of the parameter of the function, t, and not of the original object.

Takeaway 6.3.1 #2 ***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

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

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

Takeaway 6.3.1 #3 *Structures can be assigned.*

Takeaway 6.3.1 #4 *Structures can not be compared with == or !=.*

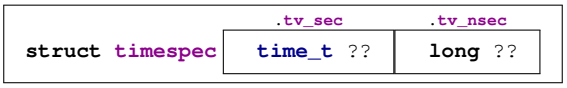
Listing 6.3 shows the complete example code for the use of **struct tm**. It doesn't contain a declaration of the historical **struct tm** since this is provided through the standard header <time.h>. Nowadays, the types for the individual members would probably be chosen differently. But many times in C we have to stick with design decisions that were made many years ago.

Takeaway 6.3.1 #5 *A structure layout is an important design decision.*

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

6.3.2. *Structures with fields of different types.* 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]
};
```



Here, we see the opaque type **time_t** that we saw in table 5.2 for the seconds, and a **long** for the nanoseconds.[49] Again, the reasons for this choice are historical; nowadays the chosen types would perhaps be a bit different. To compute the difference between two **struct timespec** times, we can easily define a function.

Whereas the function **difftime** is part of the C standard, such functionality here is very simple and isn't based on platform-specific properties. So, it can easily be implemented by anyone who needs it.[Exs 50]

6.3.3. *Nested structures.* Any data type other than a VLA is allowed as a member in a structure. So structures can also be nested in the sense that a member of a **struct** can again be 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;
};
```

A structural view is shown in figure 6.1. Here the gray boxes correspond to possible padding, a concept which we will see in the following discussion.

[49] Unfortunately, even the semantics of **time_t** are different here. In particular, **tv_sec** may be used in arithmetic.

[Exs 50] Write a function `timespec_diff` that computes the difference between two **timespec** values.

LISTING 6.3. A sample program manipulating **struct tm**

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

bool leapyear(unsigned year) {
  /* All years that are divisible by 4 are leap years,
     unless they start a new century, provided they
     are not divisible by 400. */
  return !(year % 4) && ((year % 100) || !(year % 400));
}

#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;
  // Takes care of leap years
  if ((t.tm_mon > 1) && leapyear(t.tm_year+1900))
    ++t.tm_yday;
  return t;
}

int main(void) {
  struct tm today = {
    .tm_year = 2019-1900,
    .tm_mon  = 4-1,
    .tm_mday = 3,
    .tm_hour = 10,
    .tm_min  = 0,
    .tm_sec  = 47,
  };
  printf("this␣year␣is␣%d,␣next␣year␣will␣be␣%d\n",
         today.tm_year+1900, today.tm_year+1900+1);
  today = time_set_yday(today);
  printf("day␣of␣the␣year␣is␣%d\n", today.tm_yday);
}
```

Much different than for other programming languages such as C++, the visibility of declaration **struct** stardate is the same as for **struct** person. A **struct** itself (here, person) does not define a new scope for a **struct** (here, stardate) that is defined within the {} of the outermost **struct** declaration. That is, if the nested **struct** declarations appear globally, both **struct**s are subsequently visible for the whole C file. If they appear inside the body of a function, their visibility is bound to the {} compound statement in which they are found.

Takeaway 6.3.3 #1 *All **struct** declarations in a nested declaration have the same scope of visibility.*

So, a more adequate version would be as follows:

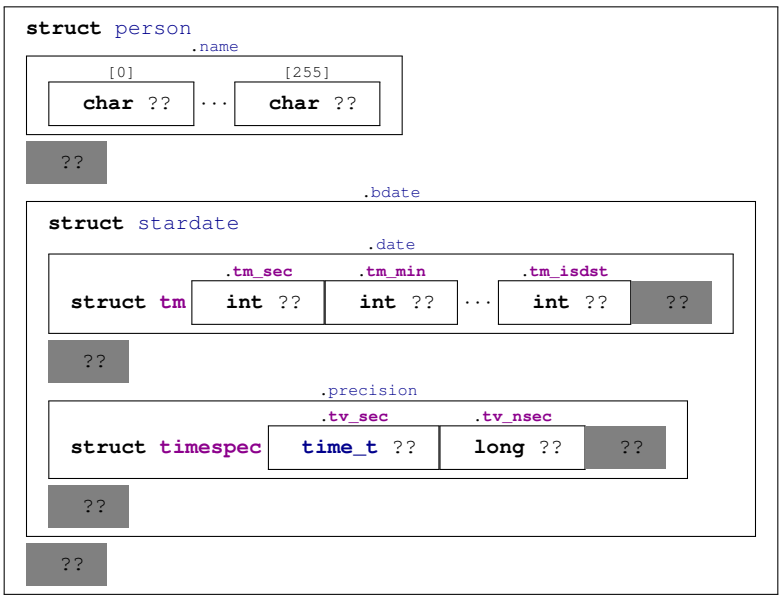


FIGURE 6.1. A structure layout

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

This version places all **struct**s on the same level because they end up there anyway. Nevertheless, it doesn't change the structural view as we have previously presented: the layout and semantics of **struct** person stay exactly the same as before.

6.3.4. *Coalescing structure fields.* We have seen that our compiler places the fields of a structure in the same order into the storage as they are defined. If the fields have different sizes, the compiler may want to put them at specific positions of the structure. The main reason to do so is the ease of access to such a field; due to the organization of storage into words that comprise several bytes, it might be better to start a new field at such a word boundary. We will discuss this feature, called *alignment* later in section 12.7. Alignment can amount to some wasted space after any field, called *byte padding*, the grey areas in our scheme for the type person in figure 6.1. If there is such padding, it will always consist of 1 or several bytes.

Takeaway 6.3.4 #1 *There can be padding after any structure member.*

Takeaway 6.3.4 #2 *There is no padding at the beginning of a structure.*

One of the possibilities to reduce the waste by padding bytes is to chose a specific ordering of the members.[Exs 51]

[Exs 51] Create six different structure types for each possibility to order three fields inside a structure: one **unsigned char**, one **unsigned**, and one **unsigned long long**. Print the sizes of these six structures; they should be significantly different. Compute the minimal size as the sum of the sizes of each member. Does any of your structure have this size? Which of your structures comes closest to that ideal size?

Another possible waste of bits and bytes in our structure can originate from an inefficient use. Remember that we used **unsigned** values to represent sets of birds. Effectively, we only needed 4 bits within such an **unsigned**; all other bits are wasted. This phenomenon is called *bit padding* because, in contrast to the previous such padding, in general, it does not fall on byte boundaries and can go down to account for single bits.

This waste is quite high in the predefined structure **tm**: indeed, a field such as **tm_sec** only has 61 possible values, so it can be stored in 6 bits instead of the at least 16 bits that an **int** member occupies. C traditionally has a mechanism called *bit-field* that can be used to reduce the bits that a member of a structure occupies:

```
// *** This has errors, don't use it! ***
struct tib {
  int tib_sec  :6; // Seconds after the minute    [0, 60]
  int tib_min  :6; // Minutes after the hour      [0, 59]
  int tib_hour :5; // Hours since midnight        [0, 23]
  int tib_mday :5; // Day of the month            [1, 31]
  int tib_mon  :4; // Months since January        [0, 11]
  int tib_year;    // Years since 1900
  int tib_wday :3; // Days since Sunday           [0, 6]
  int tib_yday :9; // Days since January          [0, 365]
  int tib_isdst:1; // Daylight Saving Time flag
};
```

That is, we put the number of bits that we need at the end of a member declaration, separated by a `:` character. So, in this case, we are indicating that we need at least 39 bit for the bit-fields (plus `sizeof(int) * CHAR_BIT` bit for the **int**) to represent all the values that interest us. It is then up to the compiler to organize the structure by coalescing successive bit-fields into larger units. A common layout here could be to group the first five fields with their 28 bits into one unit of the size of an **int**, and then have `tib_year` in a separate **int**, and then have another unit for the final 13 bit. Instead of `9*sizeof(int)` this scheme only uses `3*sizeof(int)`, three times less.

All of this still is comfortable to use. The fields can be used much as before: a member designator `x.tib_year`, for example, can be used in expressions or in assignments as the corresponding one in **struct tm**, and designated initializers in the form `.tib_mon = 3` work as expected.

But, the traditional bit-fields as previously presented have some drawbacks and therefore the code that is shown there may be erroneous. First, other than in all other declarations a specification of **int** for a bit-field may correspond to a signed or an unsigned type. On some architectures where **int** here means actually **unsigned**, the code actually is correct; all values as indicated can be stored in the corresponding field. On (most) other architectures where the field is **signed**, we lack a bit for the representation of most fields. For example, if the field `tib_mday` has 5 bit and is signed, it can hold the values −16, ..., 15. An assignment such as `x.tib_mday = 31` has a value that is out of that range; hopefully the compiler will then choose the corresponding value with the same bit pattern, −1, for the store operation. But then, when such a value is read, it is interpreted as negative, and any computations with dates go wrong.

This design flaw can be circumvented by revisiting the specification. We could augment all specifications by adding 1 sign bit. But then for our example the first five fields already would need 33 bit and on most architectures will not be stored in a single unit. The other possibility is to use **unsigned** for all our bit-fields, which is what I would recommend.

Takeaway 6.3.4 #3 *Do not use bare* ***int*** *for the specification of a bit-field.*

There is also a second drawback to bit-fields from before C23, namely that the type to which a bit-field resolves in an expression (something like `x.tib_mday`) is not sufficiently specified by the standard, and compilers currently diverge. This is nothing that we can observe at our current level, but it may bite us much later when we try to infer types for declarations or type-generic function calls in section 18.

With the introduction of the **`_BitInt`** types, we now have a new possibility:

```
struct tbi {
  unsigned _BitInt(6) tbi_sec  :6; // Seconds after the minute [0, 60]
  unsigned _BitInt(6) tbi_min  :6; // Minutes after the hour   [0, 59]
  unsigned _BitInt(5) tbi_hour :5; // Hours since midnight     [0, 23]
  unsigned _BitInt(5) tbi_mday :5; // Day of the month         [1, 31]
  unsigned _BitInt(4) tbi_mon  :4; // Months since January     [0, 11]
  signed              tbi_year;    // Years since 1900
  unsigned _BitInt(3) tbi_wday :3; // Days since Sunday        [0, 6]
  unsigned _BitInt(9) tbi_yday :9; // Days since January       [0, 365]
  bool                tbi_isdst:1; // Daylight Saving Time flag
};
```

Here, all bit-fields have exactly the type and behave exactly as specified. The types that are explicitly specified as unsigned types behave as such. For example, a test

```
  x.tbi_min < 60
```

is sufficient to know whether the field is in its valid range.

Takeaway 6.3.4 #4 *Use a* ***`_BitInt`*** *(*N*) type for a numerical bit-field of width* N.

Note, though, that the rules for using **`_BitInt`** types in arithmetic may be marginally different than for using **`signed int`**.

The field `tbi_isdst` is modeled as **`bool`**, which is how any field that acts as a flag should be specified.

Takeaway 6.3.4 #5 *Use* ***`bool`*** *as type of a flag bit-field of width* 1.

6.4. New names for types: Type aliases. As we saw in the previous section, a structure introduces not only a way to aggregate differing information into one unit but also 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 its use 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`**. Using it, a type can have several names, and we can even reuse the ***tag name***^{C} that we used in the structure declaration:

```
  typedef struct birdStruct birdStructure;
  typedef struct birdStruct birdStruct;
```

Then, **`struct`** `birdStruct`, `birdStruct`, and `birdStructure` can all be used interchangeably. My favorite use of this feature is the following idiom:

```
typedef struct birdStruct birdStruct;
struct birdStruct {
  ...
};
```

That is, we *precede* the proper **`struct`** declaration by a **`typedef`** using exactly the same name. This works because in the combination of **`struct`** with a following name, the ***tag***^{C} is always valid, a ***forward declaration***^{C} of the structure.

Takeaway 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 who come from there.

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

```
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: **`double`**`[16][64]`.

Takeaway 6.4 #2 *A* **`typedef`** *only creates an alias for a type but never a new type.*

The C standard also uses **`typedef`** a lot internally. The semantic integer types such as **`size_t`** that we saw in subsection 5.2 are declared with this mechanism. The standard often uses names that terminate with **`_t`** for **`typedef`**. This naming convention ensures that the introduction of such a name in an upgraded version of the standard will not conflict with existing code. So you shouldn't introduce such names yourself in your code.

Takeaway 6.4 #3 *Identifier names terminating with* **`_t`** *are reserved.*

Summary

- Arrays combine several values of the same base type into one object.
- Pointers either refer to other objects, are null or are invalid.
- Structures combine values of different base types into one object.
- **`typedef`**s provide new names for existing types.

7. Functions

This section covers

- An introduction to simple functions
- Working with **main**
- Understanding recursion

We have already seen the different means that C offers for *conditional execution*: execution that, based on a value, chooses one branch of the program over another to continue. The reason for a potential "jump" to another part of the program code (for example, to an **else** branch) is a run-time decision that depends on run-time data. This section starts with a discussion of *unconditional* ways to transfer control to other parts of our code: by themselves, they do not require any runtime data to decide where to go.

The code examples we have seen so far often used functions from the C library that provided features we did not want (or were not able) to implement ourselves, such as **printf** for printing and **strlen** for computing the length of a string. The idea behind this concept of functions is that they implement a certain feature once and for all and that we then can rely on that feature in the rest of our code.

A function for which we have seen several definitions is **main**, the entry point of execution into a program. In this section, we will look at how to write functions ourselves that may provide features just like the functions in the C library.

The main reasons motivating the concept of functions are *modularity* and *code factorization*:

- Functions avoid code repetition. In particular, they avoid easily introduced copy-and-paste errors and spare the effort of editing in multiple places if you modify a piece of functionality. Thereby, functions increase readability and maintainability.
- Use of functions decreases compilation times. A given code snippet that we encapsulate in a function is compiled only once, not at each point where it is used.
- Functions simplify future code reuse. Once we have extracted code into a function that provides certain functionality, it can easily be applied in other places that we did not even think of when implementing the function.
- Functions provide clear interfaces. Function arguments and return types clearly specify the origin and type of data that flows into and out of a computation. Additionally, functions allow us to specify invariants for a computation: pre- and post-conditions.
- Functions provide a natural way to formulate algorithms that use a "stack" of intermediate values.

In addition to functions, C has other means of unconditional transfer of control, which 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 subsection 8.8).
- **goto** transfers control within a function body (see subsections 13.2.2 and 15.6).
- **setjmp** and **longjmp** can be used to return unconditionally to a calling context (see subsection 19.5).
- Certain events in the execution environment or calls to the function **raise** may raise *signals* that pass control to a specialized function, a *signal handler*.

7.1. Simple functions. We have used a lot of functions and seen some of their declarations (for example, in section 6.1.5) and definitions (such as listing 6.3). In all

of these functions, parentheses `()` play an important syntactical role. They are used for function declarations and definitions to encapsulate the list of parameter declarations. For function calls, they hold the list of arguments for that concrete call. This syntactic role is similar to `[]` for arrays: in declarations and definitions, they contain the size of the corresponding dimension. In a designation like `A[i]`, they are used to indicate the position of the accessed element in the array.

All the functions we have seen so far have a ***prototype***^{C}: their declaration and definition, including a parameter type-list and a return type. To see that, let us revisit the `leapyear` function from listing 6.3:

yday.c

```
5  bool leapyear(unsigned year) {
6    /* All years that are divisible by 4 are leap years,
7       unless they start a new century, provided they
8       are not divisible by 400. */
9    return !(year % 4) && ((year % 100) || !(year % 400));
10 }
```

A declaration of that function (without a definition) could look as follows:

```
bool leapyear(unsigned year);
```

Alternatively, we could even omit the name of the parameter and/or add the *storage specifier* **`extern`**:[52]

```
extern bool leapyear(unsigned);
```

Important for such a declaration is that the compiler sees the types of the argument(s) and the return type, so here the prototype of the function is "function receiving an **`unsigned`** and returning an **`bool`**."

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`**, like **`main`** in our very first example (listing 1.1).
- If the function doesn't return a value, the return type is given as **`void`**, for example `swap_double`.

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

```
extern double fbar(double);

...
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 use of the return value in an expression: the caller knows that the return type is **`double`**, so floating-point division is applied for the result expression.

Historically, C had ways to declare functions without prototype; they have been retired with C23.

Takeaway 7.1 #1 *All functions must have prototypes.*

[52] More details on the keyword **`extern`** will be provided in subsection 13.2.

A notable exception to that rule are functions that can receive a varying number of parameters, such as **printf**. They use a mechanism for parameter handling called a ***variable argument list***[C], which comes with the header `<stdargs.h>`. <stdargs.h>

We will see later (subsection 17.4.2) how this works, but this feature is to be avoided in any case. Already from your experience with **printf**, you can imagine why such an interface poses difficulties. You, as the 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.

Takeaway 7.1 #2 *Functions have only one entry but can have several* ***return****s.*

All **return**s in a function must be consistent with the function declaration. For a function that expects a return value, all **return** statements must contain an expression; in functions that expect no such value, a **return** statement that contains an expression is erroneous.

Takeaway 7.1 #3 *A function* ***return*** *must be consistent with its type.*

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

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

Takeaway 7.1 #4 *Reaching the end of the body of a function is equivalent to a* ***return*** *statement without an expression.*

Similar to the evaluation of a variable, a function that is expected to return a value would return an uninitialized value, and this could jeopardize the execution if the call tries to evaluate it. Therefore, this construct is only allowed for functions that do not return a value.

Takeaway 7.1 #5 *Reaching the end of the body of a function is only allowed for* ***void*** *functions.*

7.2. main is special. Perhaps you 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 so pivot between the runtime system and the application, **main** 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 C platform may provide other interfaces. Two variations 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 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: **EXIT_SUCCESS** or **EXIT_FAILURE** indicates success or failure of the execution from the programmer's point of view. These are the only two values that are guaranteed to work on all platforms.

Takeaway 7.2 #1 *Use **EXIT_SUCCESS** and **EXIT_FAILURE** as return values for **main**.*

In addition, there is a special exception for **main**, as it is not required to have an explicit **return** statement.

Takeaway 7.2 #2 *Reaching the end of **main** is equivalent to a **return** with **EXIT_SUCCESS**.*

Personally, I am not much of a fan of such exceptions without tangible gain; they just make arguments about programs more complicated.

The library function **exit** has a special relationship with **main**. As the name indicates, a call to **exit** terminates the program. The prototype is as follows:

```
[[noreturn]] void exit(int status);
```

This functions terminates the program exactly as a **return** from **main** would. The status parameter has the role that the return expression in **main** would have.

Takeaway 7.2 #3 *Calling **exit**(s) is equivalent to the evaluation of **return** s in **main**.*

We also see that the prototype of **exit** is special because it has a **void** type. Just like a **return** statement, **exit** never fails.

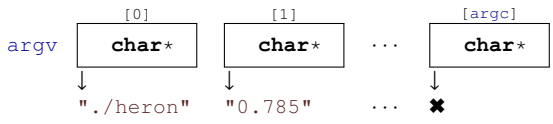
Takeaway 7.2 #4 ***exit** never fails and never returns to its caller.*

The latter is indicated by the attribute [[**noreturn**]]. This attribute should only be used for such special functions.[53]

There is another feature in the second prototype of **main**: argv, the vector of command-line arguments. We looked at some examples where we used this vector to communicate values from the command line to the program. For example, in listing 3.1,

heron.c
```
12    double const a = strtod(argv[i], nullptr);  // arg -> double
```

these command-line arguments were interpreted as **double** data for the program:



So each of the argv[i] for $i = 0, \ldots, argc$ is a pointer similar to those we encountered earlier. As an easy first approximation, we can see them as strings.

Takeaway 7.2 #5 *All command-line arguments are transferred as strings.*

[53] C versions before C23 had the keyword **_Noreturn** and the macro **noreturn**, a pretty-printed version of it, which came with the header stdnoreturn.h.

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 argv strings, two elements hold special values.

Takeaway 7.2 #6 *argv[0] points to the name of the program invocation.*

There is no strict rule about what that program name should be, but usually it is the name of the program executable.

Takeaway 7.2 #7 *argv[argc] is a null pointer.*

In the argv array, the last argument could always be identified using this property, but this feature isn't very 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 body. Their identifiers (names) don't conflict with other similar identifiers in other functions, and once we leave the function, all the mess we leave behind is cleaned up.

Even better, whenever we call a function, even one we have called before, a new set of local variables (including function parameters) is created, and these are newly initialized. This also holds 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*, and the concept is called *recursion*.

Recursive functions are crucial for understanding C functions. They demonstrate and use primary features of the function call model and are only fully functional with these features. As a first example, we will look at an implementation of Euclid's algorithm to compute the *greatest common divisor* (gcd) of two numbers:

euclid.h

```
8  inline size_t gcd2(size_t a, size_t b) [[__unsequenced__]] {
9    assert(a <= b);
10   if (!a) return b;
11   size_t r = b % a;
12   return gcd2(r, a);
13 }
```

As you can see, this function is short and seemingly nice; it makes some assumptions about its arguments and thus is not full interface for gcd.[54] But to understand how it works, we need to thoroughly understand how functions work and how we transform mathematical statements into algorithms.

Given two integers $a, b > 0$, the gcd is defined as the greatest integer $c > 0$ that divides into both a and b. Here is the formula:

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

If you are not used to such mathematical formulations, this is probably a bit hard to swallow, but be ensured that with the explanations and examples that are to come shortly, you will see much clearer what this is all about.

If we also assume that $a < b$, it can be shown that two *recursive* formulas 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 integer or if we replace the larger of the two with the modulus of the other. These formulas have been used to

[54] The [[unsequenced]] attribute will be discussed much later in section 16.3.

compute the gcd since the days of ancient Greek mathematics. They are commonly attributed to Euclid (Εὐκλείδης, around 300 B.C.) but may have been known even before him. The term *recursion* for such formulas (and derived from that for functions) refers to the fact that the value of a term (here, $\gcd(a, b)$) is explained by using the same term but with different values (here, for example, $\gcd(a, b - a)$).

Our C function `gcd2` uses equation (3). First (line 9), it checks whether a precondition for the execution of this function is satisfied: that is, whether the first argument is less than or equal to the second. It does this by using the **`assert`** macro from `<assert.h>`. This would abort the program with an informative message if the function was called with arguments that didn't satisfy that condition (we will see more explanations of **`assert`** in subsection 8.8).

<assert.h>

Takeaway 7.3 #1 *Make all preconditions for a function explicit.*

Then, line 10 checks whether `a` is `0`, in which case it returns `b`. This is an important step in a recursive algorithm.

Takeaway 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 modern systems with large amounts of memory, this may take some time, during which the system will be completely unresponsive. You'd better not try it.

Otherwise, we compute the remainder `r` of `b` modulo `a` (line 11). Then the function is called recursively with `r` and `a`, and the return value of that is directly returned.

Figure 7.1 shows an example of the different recursive calls that are issued from an initial call `gcd2(18, 30)`. Here, the recursion goes four levels deep. Each level implements its own copies of the variables `a`, `b`, and `r`.

For each recursive call, modulo arithmetic (takeaway 4.2.2 #5) 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

```
15 inline size_t gcd(size_t a, size_t b) [[__unsequenced__]] {
16   assert(a);
17   assert(b);
18   if (a < b)
19     return gcd2(a, b);
20   else
21     return gcd2(b, a);
22 }
```

Takeaway 7.3 #3 *Ensure the preconditions of a recursive function in a wrapper function.*

This avoids having to check the precondition at each recursive call: the **`assert`** macro is such that it can be disabled in the final production object file.

Another famous example of a recursive definition of an integer sequence are *Fibonnacci numbers*, a sequence of numbers that appeared as early as 200 B.C. in Indian texts. In modern terms, the sequence can be defined as

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

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

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

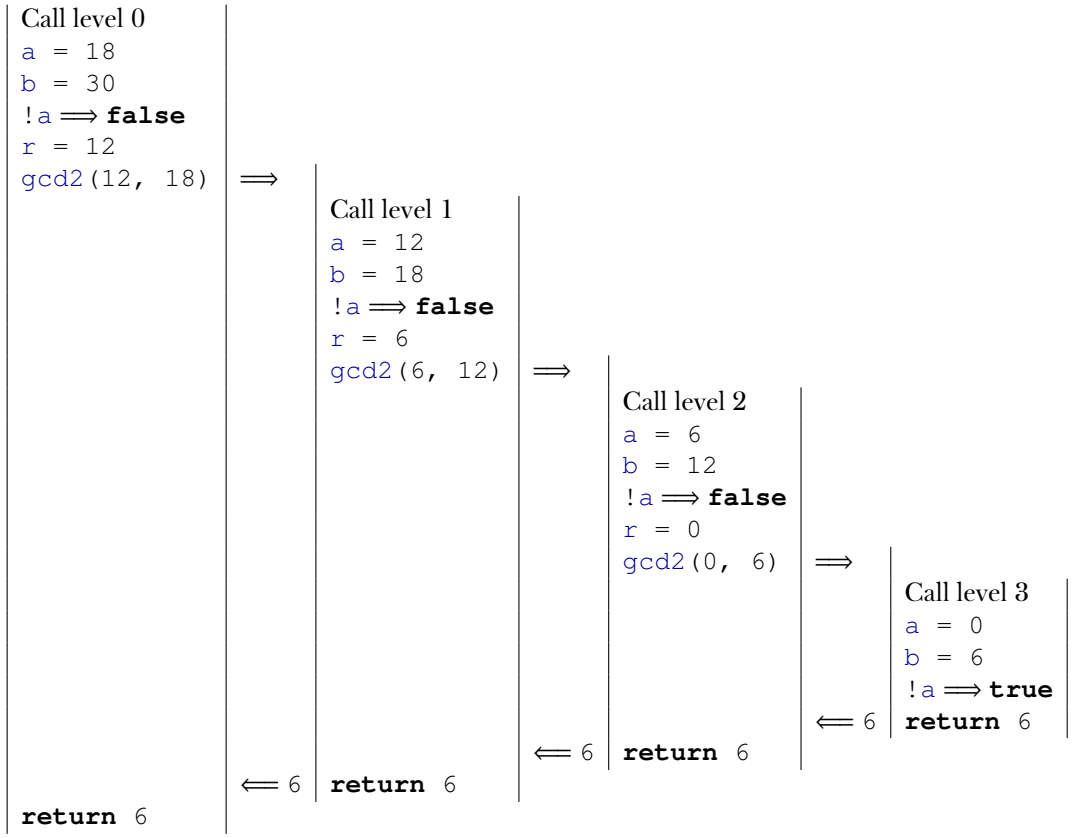


FIGURE 7.1. Recursive call `gcd2(18, 30)`

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, and 987.
With the golden ratio,

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

it can be shown that

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

and so, asymptotically, we have

$$F_n \approx \frac{\varphi^n}{\sqrt{5}} \tag{9}$$

So, the growth of F_n is exponential.

The recursive mathematical definition can be translated in a straightforward manner into a C function:

fibonacci.c

```
4  size_t fib(size_t n) [[__unsequenced__]] {
5    if (n < 3)
6      return 1;
7    else
8      return fib(n-1) + fib(n-2);
```

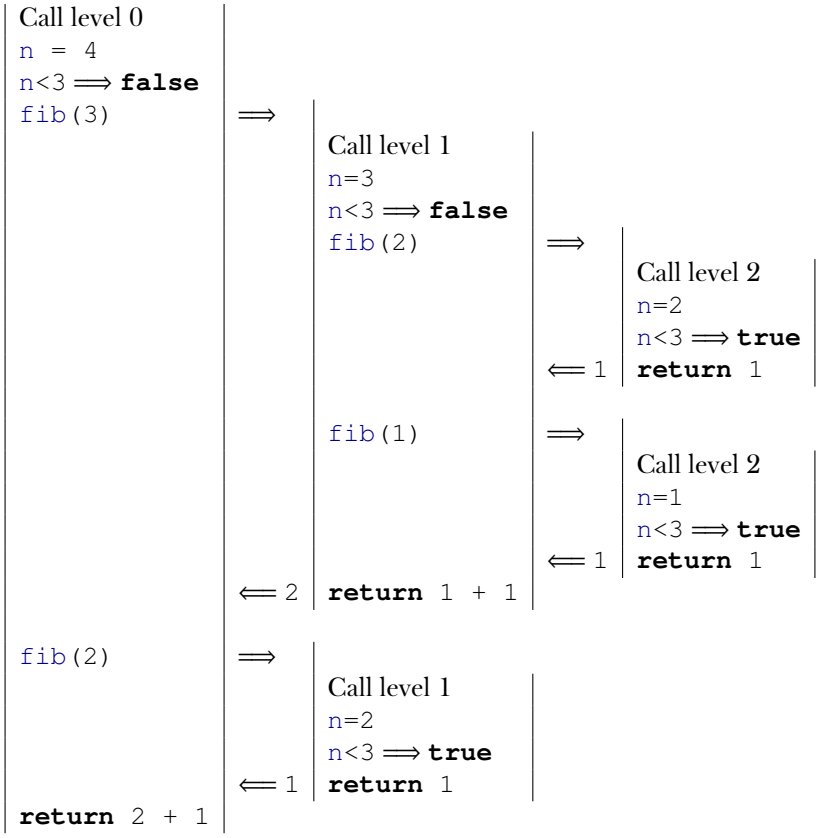


FIGURE 7.2. Recursive call `fib(4)`

```
9 }
```

Here, again, we first check for the termination condition: whether the argument to the call, `n`, is less than `3`. If it is, the return value is `1`; otherwise, we return the sum of calls with argument values `n-1` and `n-2`.

Figure 7.2 shows an example of a call to `fib` with a small argument value. We see that this leads to three 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 three *leaf calls*: calls to the function that fulfill the termination condition and, thus, by themselves, do not go into recursion. [Exs 55]

Implemented like that, the computation of the Fibonacci numbers is quite slow. [Exs 56] In fact, it is easy to see that the recursive formula for the function itself also leads to an analogous formula for the function's execution time:

$$T_{\texttt{fib(1)}} = C_0 \tag{10}$$

$$T_{\texttt{fib(2)}} = C_0 \tag{11}$$

$$T_{\texttt{fib(i)}} = T_{\texttt{fib(i-1)}} + T_{\texttt{fib(i-2)}} + C_1 \qquad \text{for all } i > 3 \tag{12}$$

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

[Exs 55] Show that a call `fib(n)` induces F_n leaf calls.

[Exs 56] Measure the times for calls to `fib(n)` with n set to different values. On POSIX systems, you can use `/bin/time` to measure the run time of a program's execution.

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

$$T_{\texttt{fib(n)}} = F_n(C_0 + C_1) \approx \varphi^n \cdot \frac{C_0 + C_1}{\sqrt{5}} = \varphi^n \cdot C_2 \tag{13}$$

with another platform-dependent constant C_2. So the execution time of `fib(n)` is exponential in `n`, which usually rules out using such a function in practice.

Takeaway 7.3 #4 *Multiple recursion may lead to exponential computation times.*

If we look at the nested calls in figure 7.2, we see that we have the call `fib(2)` twice, and thus all the effort to compute the value for `fib(2)` is repeated. The function `fibCacheRec` avoids such repetitions. It receives an additional argument, `cache`, which is an array that holds all values that have already been computed:

fibonacciCache.c

```
5  /* Compute Fibonacci number n with the help of a cache that may
6     hold previously computed values. */
7  size_t fibCacheRec(size_t n, size_t cache[static n]) {
8    if (!cache[n-1]) {
9      cache[n-1]
10       = fibCacheRec(n-1, cache) + fibCacheRec(n-2, cache);
11   }
12   return cache[n-1];
13 }
14
15 size_t fibCache(size_t n) {
16   if (n+1 <= 3) return 1;
17   /* Set up a VLA to cache the values. */
18 #if __STDC_VERSION__ > 202311L
19   /* Since C23, VLA can be default initialized. */
20   size_t cache[n] = { };
21 #else
22   size_t cache[n]; memset(cache, 0, n*sizeof(*cache));
23 #endif
24   /* Non-trivial initialization is replaced by assignment. */
25   cache[0] = 1; cache[1] = 1;
26   /* Call the recursive function. */
27   return fibCacheRec(n, cache);
28 }
```

By trading storage against computation time, the recursive calls are affected only if the value has not yet been computed. Thus, the `fibCache(i)` call has an execution time that is linear in n:

$$T_{\texttt{fibCache(n)}} = n \cdot C_3 \tag{14}$$

for a platform-dependent parameter C_3.[Exs 57] 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) discuss implementation details, nor did we perform concrete measurements of execution time. [Exs 58]

Takeaway 7.3 #5 *A bad algorithm will never lead to a performing implementation.*

Takeaway 7.3 #6 *Improving an algorithm can dramatically improve performance.*

[Exs 57] Prove equation (14).

[Exs 58] Measure times for `fibCache(n)` call with the same values as for `fib`.

For the fun of it, `fib2Rec` shows a third implemented algorithm for the Fibonacci sequence. It gets away with a constant-length array (CLA) instead of a variable-length array (VLA):

fibonacci2.c

```
4  void fib2rec(size_t n, size_t buf[static 2]) [[__unsequenced__]]
       {
5    if (n > 2) {
6      size_t res = buf[0] + buf[1];
7      buf[1] = buf[0];
8      buf[0] = res;
9      fib2rec(n-1, buf);
10   }
11 }
12
13 size_t fib2(size_t n) [[__unsequenced__]] {
14   size_t res[2] = { 1, 1, };
15   fib2rec(n, res);
16   return res[0];
17 }
```

Proving that this version is still correct is left as an exercise.[Exs 59] Also, up to now we have only had rudimentary tools to assess whether this is "faster" in any sense we might want to give the term.[Exs 60]

CHALLENGE 9 (Factorization). *Now that we've covered functions, see wether you can implement a program* `factor` *that receives a number* `N` *on the command line and prints out*

```
N: F0 F1 F2 ...
```

where `F0` *and so on are all the prime factors of* `N`.
The core of your implementation should be a function that, given a value of type `size_t`*, returns its smallest prime factor.*
Extend this program to receive a list of such numbers and output such a line for each of them.

[Exs 59] Use an iteration statement to transform `fib2rec` into a nonrecursive function `fib2iter`.

[Exs 60] Measure times for `fib2(n)` calls with the same values as `fib`.

Summary

- Functions have a prototype that determines how they can be called.
- Terminating `main` and calling `exit` are the same.
- Each function call has its own copy of local variables and functions can be called recursively.

8. C library functions

This section covers

- Doing math, handling files, and processing strings
- Manipulating time
- Managing the runtime environment
- Terminating programs

The functionality that the C standard provides is separated into two big parts. One is the proper C language, and the other is the *C library*. We have looked at several functions that come with the C library, including **`printf`**, **`puts`**, and **`strtod`**, so you should have a good idea what to expect: basic tools that implement features that we need in everyday programming and for which we need clear interfaces and semantics to ensure portability.

On many platforms, the clear specification through an *application programming interface* (*API*) also allows us to separate the compiler implementation from the library implementation. For example, on Linux systems, we have a choice of different compilers, most commonly `gcc` and `clang`, and different C library implementations, such as the GNU C library (`glibc`), `dietlibc`, and `musl`. Potentially, any of these choices can be used to produce an executable.

We will first discuss the general properties and tools of the C library and its interfaces and then describe some groups of functions: mathematical (numerical) functions, input/output functions, string processing, time handling, access to the runtime environment, and program termination.

8.1. General properties of the C library and its functions. Roughly, library functions target one or two purposes:

Platform abstraction layer. Functions that abstract from the specific properties and needs of the platform. These are functions that need platform-specific bits to implement basic operations such as IO, which could not be implemented without deep knowledge of the platform. For example, **`puts`** has to have some concept of a "terminal output" and how to address it. Implementing these functionalities would exceed the knowledge of most C programmers because doing so requires OS- or even processor-specific magic. Be glad that some people did that job for you.

Basic tools. Functions that implement a task (such as **`strtod`**) that often occurs in programming in C and for which it is important that the interface is fixed. These should be implemented relatively efficiently because they are used a lot, and they should be well tested and bug free so we can rely safely on them. Implementing such functions should, in principle, be possible for any confirmed C programmer.[Exs 61]

A function like **`printf`** can be viewed as targeting both purposes: it can effectively be separated into a formatting phase, providing a basic tool and an output phase that is platform specific. There is a function **`snprintf`** (explained much later, in subsection 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 give the same output as **`printf`** as a whole.

In the following sections, we will discuss the different header files that declare the interfaces of the C library (subsection 8.1.1), the different types of interfaces it provides (subsection 8.1.2), the various error strategies it applies (subsection 8.1.3), an optional series of interfaces intended to improve application safety (subsection 8.1.4), and tools that we can use to assert platform-specific properties at compile time (subsection 8.1.5).

[Exs 61] Write a function `my_strtod` that implements the functionality of **`strtod`** for decimal floating-point constants.

TABLE 8.1. C library headers

Name	Description	Section
`<assert.h>`	Asserting runtime conditions	8.8
`<complex.h>`	Complex numbers	5.7.8
`<ctype.h>`	Character classification and conversion	8.5
`<errno.h>`	Error codes	8.1.3
`<fenv.h>`	Floating-point environment	15.1.4
`<float.h>`	Properties of floating-point types	5.7
`<inttypes.h>`	Formatting conversion of integer types	5.7.6
`<iso646.h>`	Alternative spellings for operators	18.1.2
`<limits.h>`	Properties of integer types	5.1.3
`<locale.h>`	Internationalization	8.7
`<math.h>`	Type-specific numerical functions	8.3
`<setjmp.h>`	Non-local jumps	19.5
`<signal.h>`	Signal-handling functions	19.6
`<stdalign.h>`	Alignment of objects	12.7
`<stdarg.h>`	Functions with varying numbers of arguments	17.4.2
`<stdatomic.h>`	Atomic operations	19.6
`<stdbit.h>`	Bit operations	5.7.2
`<stdbool.h>`	Booleans	3.1
`<stdckdint.h>`	Checked integer arithmetic	8.2
`<stddef.h>`	Basic types and macros	5.2
`<stdint.h>`	Exact-width integer types	5.7.6
`<stdio.h>`	Input and output	8.4
`<stdlib.h>`	Basic functions	2
`<stdnoreturn.h>`	Non-returning functions	7
`<string.h>`	String handling	8.5
`<tgmath.h>`	Type-generic numerical functions	8.3
`<threads.h>`	Threads and control structures	20
`<time.h>`	Handling time	8.6
`<uchar.h>`	Unicode characters	14.3
`<wchar.h>`	Wide strings	14.3
`<wctype.h>`	Wide character classification and conversion	14.3

8.1.1. *Headers.* The C library has a lot of functions, far more than we can handle in this book. A ***header***^{C} file bundles interface descriptions for a number of features, mostly functions. The header files that we will discuss here provide features of the C library, but later we can create our own interfaces and collect them in headers (section 10).

On this level, we will discuss the functions from the C library that are necessary for basic programming with the elements of the language we have seen so far. We will complete this discussion on higher levels, when we discuss a range of concepts. Table 8.1 has an overview of the standard header files.

8.1.2. *Interfaces.* Most interfaces in the C library are specified as functions, but implementations are free to chose to implement them as macros, where doing so is appropriate. Compared to those we saw in subsection 5.6.3, this uses a second form of macros that are syntactically similar to functions, ***function-like 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 bad to pass any expression with

TABLE 8.2. Error return strategies for C library functions. Some functions may also indicate a specific error condition through the value of the **errno** macro.

Failure return	Test	Typical case	Example
null pointer	`!value`	Other values are valid	**fopen**
Special error code	`value == code`	Other values are valid	**puts**, **clock**, **mktime**, **strtod**, **fclose**
Nonzero value	`value`	Value otherwise unneeded	**fgetpos**, **fsetpos**
Special success code	`value != code`	Case distinction for failure condition	**thrd_create**
Negative value	`value < 0`	Positive value is a counter	**printf**

side effects to such a macro or function. Hopefully, our previous discussion about side effects (takeaway 4.3 #2) has already convinced you not to do that.

Some of the interfaces we will look at have arguments or return values that are pointers. We can't handle these completely yet, but in most cases we can get away with passing in known pointers or **nullptr** for pointer arguments. Pointers as return values will only occur in situations where they can be interpreted as an error condition.

8.1.3. *Error checking.* C library functions usually indicate failure through a special return value. What value indicates the failure can be different and depends on the function itself. Generally, you have to look up the specific convention in the manual page for the functions. Table 8.2 gives a rough overview of the 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.

Typical error-checking code looks like the following:

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

<stdio.h>

Here we see that **puts** falls into the category of functions that return a special value on error, **EOF**, ("end-of-file"). The **perror** function from `<stdio.h>` is then used to provide an additional diagnostic that depends on the specific error. **exit** ends the program execution. Don't sweep failures under the carpet.

Takeaway 8.1.3 #1 *Failure is always an option.*

Takeaway 8.1.3 #2 *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 early in development.

Takeaway 8.1.3 #3 *Fail fast, fail early, and fail often.*

C has one major state variable that tracks errors of C library functions: a dinosaur called **errno**. The **perror** function 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, the 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;
  }
}
```

8.1.4. *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 led (and still leads) to a lot of security bugs and exploits and is generally something that should be handled very carefully.

C11 addressed this sort of problems by deprecating or removing some functions from the standard and by adding an optional series of new interfaces that check consistency of the parameters at runtime. These are the *bounds-checking interfaces* of *Annex K* of the C standard. Unlike 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 whether this optional interfaces is supported, and **__STDC_WANT_LIB_EXT1__** switches it on. The latter 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. */
```

This mechanism was (and still is) open to much debate, and therefore Annex K is an optional feature. Many modern platforms have consciously chosen not to support it. There even has been an extensive study by O'Donell and Sebor [2015] that concluded that the introduction of these interfaces has created many more problems than it solved. In the following, such optional features are marked with a gray background:

Annex K

The bounds-checking functions usually use the suffix `_s` on 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.

Takeaway 8.1.4 #1 *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.

8.1.5. *Platform preconditions.* An important goal of 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 in the gaps. Unfortunately, this is not always an option, in which case we should clearly identify code preconditions.

Takeaway 8.1.5 #1 *Missed preconditions for the execution platform must abort compilation.*

The classic tools to achieve this are ***preprocessor conditionals***^{C}, as we saw earlier:

```
#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 executed only if the condition (here !**__STDC_LIB_EXT1__**) is true. It aborts the compilation process with an error message; a similar **#warning** directive allows compilation to continue but ensures that a warning message is provided. The conditions that we can place in such a construct are limited.[Exs 62]

Takeaway 8.1.5 #2 *In a preprocessor conditional, only evaluate macros and integer literals.*

As an extra feature in these conditions, identifiers that are unknown evaluate to `0`. So, in the previous example, the expression is valid even if **__STDC_LIB_EXT1__** is unknown at that point.

Takeaway 8.1.5 #3 *In a preprocessor conditional, unknown identifiers evaluate to `0`.*

There are special operators in preprocessor conditionals that can query special capacities of the compiler and if specific resources are available.

TABLE 8.3. Tests for preprocessor conditionals. The result is true if the corresponding feature named as an argument is supported, and false otherwise.

operator	argument
defined	macro name
__has_include	header name
__has_embed	binary file name
__has_c_attribute	attribute name

The **defined** test has even shortcuts that integrate directly in the **#** syntax.

TABLE 8.4. Shortcuts for preprocessor conditionals

shortcut	meaning	availability
#ifdef (X)	**#if defined** (X)	
#ifndef (X)	**#if** !**defined** (X)	
#elifdef (X)	**#elif defined** (X)	since C23
#elifndef (X)	**#elif** !**defined** (X)	since C23

If we want to test more sophisticated conditions that are not known to the preprocessor but only in subsequent compilation phases, **static_assert** can be used. Here, the guaranty is that the condition is always evaluated at compile time but after preprocessing:

```
static_assert(sizeof(double) == sizeof(long double),
  "Extra precision needed for convergence.");
```

Before C23, the keyword was written **_Static_assert**, and **static_assert** was available as a macro from `<assert.h>`.

<assert.h>

[Exs 62] Write a preprocessor condition that tests whether **int** has two's complement sign representation.

TABLE 8.5. Functions for integer arithmetic. Most of these are type-generic macros, but some also have function interfaces for specific types. x and y denote the arguments, $\overline{x}$ is a bitwise complement, w the width, and m the maximum value of the type.

Function	Description	Remark
abs, **labs**, **llabs**	$\lvert x\rvert$	type independent
div, **ldiv**, **lldiv**	x/y and $x\%y$	type independent
ckd_add	$x+y$ (LS bits) & overflow flag	target dependent
ckd_mul	$x*y$ (LS bits) & overflow flag	target dependent
ckd_sub	$x-y$ (LS bits) & overflow flag	target dependent
stdc_bit_ceil	$2^{\lceil \log_2 x\rceil}$	1 if $x \equiv 0$, 0 if $x > m/2$
stdc_bit_floor	$2^{\lfloor \log_2 x\rfloor}$	0 if $x \equiv 0$
stdc_bit_width	$1 + \lfloor \log_2 x\rfloor$	0 if $x \equiv 0$
stdc_count_ones	Number of 1-bits in x	type independent
stdc_count_zeros	Number of 1-bits in $\overline{x}$	type dependent
stdc_has_single_bit	there is n with $x \equiv 2^n$	type independent
stdc_first_leading_one	$w - \lfloor \log_2 x\rfloor$	0 if $x \equiv 0$
stdc_first_leading_zero	$w - \lfloor \log_2 \overline{x}\rfloor$	0 if x has all bits set
stdc_leading_ones	$w - \lfloor \log_2 \overline{x}\rfloor - 1$	w if $x \equiv 0$
stdc_leading_zeros	$w - \lfloor \log_2 x\rfloor - 1$	w if $x \equiv 0$
stdc_first_trailing_one	One plus LS 1-bit in x	0 if $x \equiv 0$
stdc_first_trailing_zero	One plus LS 0-bit in x	0 if $x \equiv 0$
stdc_trailing_ones	One plus LS 0-bit in x	w if $x \equiv 0$
stdc_trailing_zeros	One plus LS 1-bit in x	w if $x \equiv 0$

8.2. Integer arithmetic. Much of the functionality for integer arithmetic is already defined by operators. The C library adds some functionality to that, where there is no closed notation available or where care has to be taken when specific arguments values need special considerations. Most of these functions are new in C23. Table 8.5 has an overview

There are several functions that provide access to usual integer arithmetic. The first two families, **abs** and **div**, come with the `<stdlib.h>` header, and an `l` prefix stands for a **long** argument and `ll`, for **long long**. The **abs** functions are provided as a convenience interface because C does not have a closed notation for the absolute value of a signed integer. Also, the expression for it,


```
(x < 0) ? -x : x
```

would evaluate `x` twice.

The **div** functions have a bit more importance because they simultaneously provide the result of two operations, quotient and remainder:

```
auto res = div(x, y);
printf("%d/%d is %d, remainder %d\n", x, y, res.quot, res.rem);
```

The return types of these functions are structures, but the names of these structures should not concern you much. With C23, it is easier to capture the returned value by inferring its type through **auto** as indicated. For all three **div** functions, the specific values can then be accessed by the **quot** and **rem** members. Nowadays, optimizing compilers are generally good in collecting quotient and remainder operations that can be fused into a single instruction, so these interfaces are not of much use anymore.

Quotient and remainder operations are well-defined for most values; exceptions are only division by 0 and some combinations of **INT_MIN** and similar. For the other three usual arithmetic operations, addition, subtraction and multiplication, there are unfortunately many more situations. Writing correct C code that predicts if such an operation has a result that is out of bounds is relatively challenging. So the next three type-generic functions in the table have been introduced by C23 with the new header `<stdckdint.h>`. They provide the result of the operation through their first argument (which is a pointer) and return a Boolean that is **true** whether the operation overflowed. In any case, the operation is unconditionally valid; the result value has the least significant bits of the correct result:

<stdckdint.h>

```
unsigned result = 0;
bool overflow = ckd_add(&result, UINT_MAX, UINT_MAX);
printf("Overflow␣flag␣%s,␣result␣%x\n",
       (overflow ? "true" : "false"),
       result);
```

Here, all types are **unsigned**, so the result is just reduced modulo as for usual arithmetic for unsigned values. In the example, this is **UINT_MAX**−1. Additionally, the return of the call is **true** to reflect that the result with arbitrary precision has the value 2×**UINT_MAX**, which doesn't fit in an **unsigned**.

Now, change the example to use **signed** types and the minimum value:

```
signed result = 0;
bool overflow = ckd_add(&result, -INT_MAX, -INT_MAX);
printf("Overflow␣flag␣%s,␣result␣%x\n",
       (overflow ? "true" : "false"),
       result);
```

The operation −**INT_MAX** + −**INT_MAX** overflows, so the evaluation of such an expression has your program fail. Here, the mathematical result is 2×**INT_MIN**+2, which also does not fit into a **signed**. The call as presented is still well-defined, `result` has all bits 0 except for the bit in position 1, and the return value is **true**. So, although the mathematical result is negative, the call to **ckd_add** has a result of 2, a positive value.

The other functions concern bit operations. They were also introduced by C23 with a new header, `<stdbit.h>`, and receive any unsigned integer value of standard or extended type as an argument. Their return value is as indicated in table 8.5; the second column has the general formula, and the third shows the result for exceptional cases where the general formula is not valid. Note the following:

<stdbit.h>

- All functions have defined results for all argument values. This holds even if the compiler may realize the function for most values with a specific hardware instruction; it is their task (and not yours) to handle the special cases correctly.
- The first set of functions in the group has descriptive names that are derived directly from a tangible definition of the property of a number. Prefer these over the others to improve the readability of your code.
- The second set are all variants that deal with the magnitude of the argument or of its complement. Only use them if you are really interested in the outcome of the exceptional cases as they are given. If you have to deal with the exceptional cases yourself, use **stdc_bit_width**; your compiler knows much better than you how to optimize such conditional expressions.
- The results of the following functions are independent of the type of the argument:

  stdc_bit_floor stdc_bit_width stdc_count_ones
  stdc_has_single_bit stdc_first_trailing_one
  stdc_first_trailing_zero

In particular, the latter two make life easier for you or other readers of your code than other variants with similar functionality. For all six, it is largely preferable to use the type-generic version of the interface if you can.

- The results of the following functions depend on the width of the type of the argument:

  stdc_bit_ceil stdc_count_zeros stdc_first_leading_one stdc_first_leading_zero stdc_leading_ones stdc_leading_zeros stdc_trailing_ones stdc_trailing_zeros

  These are a bit more difficult to capture for your readers. Avoid them if you can.

8.3. Numerics. Numerical *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, it has a macro that dispatches an invocation, such as **sin**`(x)` or **pow**`(x, n)`, to the function that inspects the type of `x` in its argument and for which the return value is of that same type.

`<math.h>` `<tgmath.h>`

The type-generic macros that are defined are far too many to describe in detail here. Table 8.6 gives an overview of the functions that are provided.

Table 8.6: Numerical functions that work with floating point types. Names followed by `[f|l]` describe three functions for **double**, **float**, and **long double** arguments. The others are type-generic macros that adapt to the concrete type of their arguments.

Function	Description	
acosh	Hyperbolic arc cosine	
acos, **acospi**	Arc cosine (divided by π)	C23
asinh	Hyperbolic arc sine	
asin, **asinpi**	Arc sine (divided by π)	C23
atan2, **atan2pi**	Arc tangent, two arguments (divided by π)	C23
atanh	Hyperbolic arc tangent	
atan, **atanpi**	Arc tangent (divided by π)	C23
canonicalize`[f|l]`	Canonicalize a floating point value	
cbrt	$\sqrt[3]{x}$	
ceil	$\lceil x \rceil$	
compoundn	$(1+x)^n$	C23
copysign	Copies the sign from y to x	
cosh	Hyperbolic cosine	
cos, **cospi**	Cosine function, $\cos x$ ($\cos \pi x$)	C23
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	
fadd, **dadd**	Rounded addition to **float** or **double**	C23
fdim	Positive difference	
fdiv, **ddiv**	Rounded division to **float** or **double**	C23
floor	$\lfloor x \rfloor$	
fmaximum_mag`[_num]`	Floating-point value of maximal magnitude	C23
fmax, **fmaximum**`[_num]`	Floating-point maximum	C23

Table 8.6: Numerical functions, continued.

Function	Description	
fma	$x \cdot y + z$	
fminimum_mag[_num]	Floating-point value of minimal magnitude	C23
fmin, **fminimum**[_num]	Floating-point minimum	C23
fmod	Remainder of floating-point division	
fmul, **dmul**	Rounded multiplication to **float** or **double**	C23
fpclassify	Classifies a floating-point value	
frexp	Significand and exponent	
fromfp[f|l]	Round to signed integer value with specific width	C23
fsub, **dsub**	Rounded subtraction to **float** or **double**	C23
hypot	$\sqrt[2]{x^2 + y^2}$	
ilogb	$\lfloor \log_{\texttt{FLT\_RADIX}} x \rfloor$ as integer	
isfinite	Checks if finite	
isinf	Checks if infinite	
isnan	Checks if NaN	
isnormal	Checks if representation is 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	$\log_{\texttt{FLT\_RADIX}} x$ as floating point	
log	$\log_e x$	
modf[f|l]	Integer and fractional parts	
nan[f|l]	Not-a-number (NaN) of the corresponding type	
nearbyint	Nearest integer using the current rounding mode	
nextafter	Next representable floating-point value	
nexttoward		
nextup		
pow	x^y	
pown	x^n, n integer	C23
powr	x^y, computed as $e^{y \log_e x}$	C23
remainder	Signed remainder of division	
remquo	Signed remainder and the last bits of the division	
rint, **lrint**, **llrint**	Nearest integer using the current rounding mode	
rootn	$\sqrt[n]{x}$	C23
round, **lround**, **llround**	`sign(x)` $\cdot \lfloor \lvert x \rvert + 0.5 \rfloor$, integer	
roundeven	`sign(x)` $\cdot \lfloor \lvert x \rvert + 0.5 \rfloor$, floating point	C23
scalbn, **scalbln**	$x \cdot$ **FLT_RADIX**^{y}	
signbit	Checks if negative	
sinh	Hyperbolic sine	
sin, **sinpi**	Sine function, $\sin x$ ($\sin \pi x$)	C23
sqrt	$\sqrt[2]{x}$	
tanh	Hyperbolic tangent	
tan, **tanpi**	Tangent function, $\tan x$ ($\tan \pi x$)	C23
tgamma	Gamma function, $\Gamma(x)$	
trunc	`sign(x)` $\cdot \lfloor \lvert x \rvert \rfloor$	
ufromfp[f|l]	Round to unsigned integer value with specific width	C23

Nowadays, implementations of numerical functions should be high quality, be efficient, and have 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 them. Many of them are not just implemented as C functions but also can use processor-specific instructions. For example, processors may have fast approximations of **sqrt** and **sin** functions or implement a *floating-point multiply add*, **fma**, in a low-level instruction. In particular, there is a good chance 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**, and **trunc**. So, replacing them or re-implementing them in handcrafted code is usually a bad idea.

8.4. Input, output, and file manipulation. We have seen some of the IO functions that come with the header file `<stdio.h>`: **puts** and **printf**. Whereas the second lets you format output in a convenient fashion, the first is more basic: it just outputs a string (its argument) and an end-of-line character. <stdio.h>

8.4.1. *Unformatted text output.* There is an even more basic function than **puts**: **putchar**, which outputs a single character. The interfaces of these two functions are as follows:

```
int putchar(int c);
int puts(char const s[static 1]);
```

The type **int** as a parameter for **putchar** is a historical accident that shouldn't hurt you much. In contrast to that, having a return type of **int** is necessary so the function can return errors to its caller. In particular, it returns the argument `c` if successful and a specific negative value **EOF** (end-of-file) that is guaranteed not to correspond to any character on failure.

With this function, we could actually re-implement **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;
}
```

This is just an example; it is probably less efficient than the **puts** that your platform provides.

Up to now, we have only seen how to output to the terminal. Often, you'll want to write results to permanent storage, and the type **FILE**⋆ for ***streams***^{C} provides 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);
```

Here, the ⋆ in the **FILE**⋆ type again indicates that this is a pointer type, and we won't go into the details. The only thing we need to know for now is that a pointer can be tested to determine whether it is null (takeaway 6.2 #4), so we will be able to test whether a stream is valid.

The identifier **FILE** represents an ***opaque type***^{C}, for which we don't know more than is provided by the functional interfaces that we will see in this section. The fact 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 historical interfaces that predate standardization.

Takeaway 8.4.1 #1 *Opaque types are specified through functional interfaces.*

Takeaway 8.4.1 #2 *Don't rely on implementation details of opaque types.*

If we don't do anything special, two streams are available for output: **stdout** and **stderr**. We have already 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 and also is linked to the terminal by default, with perhaps slightly different properties. In any case, these two are closely related. The purpose of having two of them is to be able to distinguish "usual" output (**stdout**) from "urgent" output (**stderr**).

We can rewrite the previous 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,    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.

Takeaway 8.4.1 #3 ***puts*** *and* ***fputs*** *differ in their end-of-line handling.*

8.4.2. *Files and streams.* If we want to write 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 simply 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;
}
```

This ***opens a file***^{C} called `"mylog.txt"` in the filesystem and provides access to it through the variable `logfile`. The mode argument `"a"` opens the file for appending; that is, the contents of the file are preserved, if they exist, and writing begins at the current end of that file.

There are multiple reasons why opening a file might not succeed: for example, the filesystem might be full, or the process might not have permission to write at the indicated place. We check for such an error condition (takeaway 8.1.3 #2) and exit the program if necessary.

As we have seen, the **perror** function 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 does not have to) contain more information that helps the user of the program deal with the error.

TABLE 8.7. Modes and modifiers for **fopen** and **freopen**. One of the first three must start the mode string, optionally followed by one or more of the other three. See table 8.8 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	Opens file for reading and writing
`'b'`	Binary		Views as a binary file; otherwise a text file
`'x'`	Exclusive		Creates a file for writing if it does not yet exist

Annex K

There are also bounds-checking replacements **fopen_s** and **freopen_s**, which ensure that the arguments that are passed are valid pointers. Here, **errno_t** is a type that comes with `<stdlib.h>` and 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 8.7 contains an overview of the characters that may appear in that string. Three base modes regulate what happens to a pre-existing file, if any, and where the stream is positioned. In addition, three modifiers can be appended to them. Table 8.8 has a complete list of the possible combinations.

These tables show that a stream can be opened not only for writing but also for reading; we will see shortly 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. For `'r'`, file content that is preserved and positioned at the beginning should not be overwritten accidentally, so this is for reading.

The modifiers are used less commonly in everyday coding. "Update" mode with `'+'` should be used carefully. Reading and writing at the same time is not easy and needs some special care. For `'b'`, we will discuss the difference between text and binary streams in more detail in subsection 14.6.

There are three other principal interfaces to handle streams, **freopen**, **fclose**, and **fflush**:

```
FILE *freopen(const char *pathname, const char *mode, FILE *stream);
int fclose(FILE* fp);
int fflush(FILE* stream);
```

The primary uses for **freopen** and **fclose** are straightforward. **freopen** can associate a given stream to a different file and eventually change the mode. This is

TABLE 8.8. Mode strings for **fopen** and **freopen**. These are the valid combinations of the characters in table 8.7.

`"a"`	Creates an empty text file if necessary; open for writing at end-of-file
`"w"`	Creates an empty text file or wipes out content; open for writing
`"r"`	Opens an existing text file for reading
`"a+"`	Creates an empty text file if necessary; open for reading and writing at end-of-file
`"w+"`	Creates an empty text file or wipes out content; open for reading and writing
`"r+"`	Opens an 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 a binary file instead of a text file
`"wx"` `"w+x"` `"wbx"` `"w+bx"` `"wb+x"`	Same as above, but error if the file exists prior to the call

particularly useful for associating the standard streams to a file. For example, our little program from our previous discussion 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;
}
```

8.4.3. *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. 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 output immediately on the terminal or where don't want to close the file yet but want to ensure that all content we have written has properly reached its destination. Listing 8.1 shows an example that writes 10 dots to **stdout** with a delay of approximately one second between all writes.[Exs 63]

The most common form of IO buffering for text files is ***line buffering***^{C}. In that mode, output is only physically written if the end of a text line is encountered. So usually, text that is written with **puts** appears immediately on the terminal; **fputs** waits until it encounters 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.

Takeaway 8.4.3 #1 *Text input and output converts data.*

[Exs 63] Observe the behavior of the program by running it with zero, one, and two command-line arguments.

LISTING 8.1. Flushing buffered output

```
#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, [[maybe_unused]] 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);
}
```

Text input and output converts data because internal and external representations of text characters are not necessarily the same. Unfortunately, there are still many different character encodings; the C library is in charge of doing the conversions correctly, if it can. Most notoriously, the end-of-line encoding in files is platform dependent.

Takeaway 8.4.3 #2 *There are three commonly used conversion to encode end-of-line.*

C gives us a very suitable abstraction in using `'\n'` for this, regardless of the platform. Another modification you should be aware of when doing text IO is that white space that precedes the end of line may be suppressed. Therefore, the presence of ***trailing white space***^{C} such as blank or tabulator characters cannot be relied upon and should be avoided.

Takeaway 8.4.3 #3 *Text lines should not contain trailing white space.*

The C library additionally also has very limited support for manipulating files within the filesystem:

```
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.

8.4.4. *Formatted output.* We have covered how to use **`printf`** for formatted output. The function **`fprintf`** is very similar to that, but it has an additional parameter that allows us to specify the stream to which the output is written:

TABLE 8.9. Format specifications for **printf** and similar functions with the general syntax `"%[FF][WW][.PP][LL]SS"`, where `[]` surrounding a field denotes that it is optional

FF	Flags	Special form of conversion
WW	Field width	minimum width
PP	Precision	
LL	Modifier	Select width of type
SS	Specifier	Select conversion

TABLE 8.10. Format specifiers for **printf** and similar functions

'd' or 'i'	Decimal	Signed integer
'u'	Decimal	Unsigned integer
'b'	Binary	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 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 (called ***trailing arguments***[C]) that are to be printed. An important constraint is that this number must correspond exactly to the `'%'` specifiers; otherwise, the program fails.

Takeaway 8.4.4 #1 *Trailing arguments in calls to* **printf** *must exactly correspond to the format specifiers.*

With the syntax `%[FF][WW][.PP][LL]SS`, a complete format specification can be composed of five parts: flags, width, precision, modifiers, and specifier. See table 8.9 for details. The specifier is not optional and selects the type of output conversion that is performed. See table 8.10 for an overview.

As you can see, for most types of values, there is a choice of format. You should chose the one that is most appropriate for the *meaning* of the value that the output is to convey. For all numerical *values*, this should usually be a decimal format.

Takeaway 8.4.4 #2 *Use `"%d"` and `"%u"` formats to print integer values.*

If, on the other hand, you are interested in a bit pattern, use the binary (small bit sets) or hexadecimal (large ones) formats over octal. It better corresponds to modern architectures that have 8-bit character types.

Takeaway 8.4.4 #3 *Use the `"%b"` or `"%x"` formats to print bit patterns.*

Also observe that these formats receives unsigned values, which is yet another incentive to use only unsigned types for bit sets. Hexadecimal representations are better suited than binary if the sets become big, but seeing hexadecimal values and associating the corresponding bit pattern requires training. Table 8.11 has an overview of the digits, the values and the bit pattern they represent.

TABLE 8.11. Hexadecimal values and bit patterns

Digit	Value	Pattern	Digit	Value	Pattern
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	A	10	1010
3	3	0011	B	11	1011
4	4	0100	C	12	1100
5	5	0101	D	13	1101
6	6	0110	E	14	1110
7	7	0111	F	15	1111

For floating-point formats, there is even more choice. If you do not have specific needs, the generic format is the easiest to use for decimal output.

Takeaway 8.4.4 #4 *Use the `"%g"` format to print floating-point values.*

The modifier part is important to specify the exact type of the corresponding argument. Table 8.12 gives the codes for the standard types. This modifier is particularly important because interpreting a value with the wrong modifier can cause severe damage. The **`printf`** functions only have knowledge about their arguments through the format specifiers, so giving a function the wrong size may lead it to read more or fewer bytes than provided by the argument or to interpret the wrong hardware registers.

Takeaway 8.4.4 #5 *Using an inappropriate format specifier or modifier makes the behavior undefined.*

A good compiler should warn about wrong formats; please take such warnings seriously. Note also the presence of special modifiers for semantic types. In particular, the

TABLE 8.12. 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
`"w`*N*`"`	**`uint`***N*`_t`**, **`int`***N*`_t`**, **`uint_least`***N*`_t`** or **`int_least`***N*`_t`** for *N* usually 8, 16, 32, 64 or 128	Integer
`"wf`*N*`"`	**`uint_fast`***N*`_t`** or **`int_fast`***N*`_t`**	Integer
`"L"`	**`long double`**	Floating point

TABLE 8.13. Format flags for **printf** and similar functions

Character	Meaning	Conversion
"#"	Alternate form, such as prefix **0x**	"aAeEfFgGoxX"
"0"	Zero padding	Numeric
"-"	Left adjustment	Any
"␣"	' ' for positive values, '-' for negative	Signed
"+"	'+' for positive values, '-' for negative	Signed

combination "%zu" is very convenient because we don't have to know the base type to which **size_t** corresponds.

The width (WW) and precision (.PP) can be used to control the general appearance of a printed value. For example, for the generic floating-point format "%g", the precision controls the number of significant digits. A format of "%20.10g" specifies an output field of 20 characters with at most 10 significant digits. How these values are interpreted specifically differs for each format specifier.

The flag can change the output variant, such as prefixing with signs ("%+d"), **0x** for hexadecimal conversion ("%#X"), 0 for octal ("%#o"); padding with 0; or adjusting the output within its field to the left instead of the right. See table 8.13. Remember that a leading zero for integers is usually interpreted as introducing an octal number, not a decimal. So, using zero padding with left adjustment "%-0" is not a good idea because it can confuse the reader about the convention that is applied.

If we know that the numbers we write will be read back in from a file later, the forms "%+d" for signed types, "%#X" for unsigned types, and "%a" for floating point are the most appropriate. They guarantee that the string-to-number conversions will detect the correct form and that the storage in a file will not lose information.

Takeaway 8.4.4 #6 *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, the format, and any string arguments are valid pointers. They **don't** check whether 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 re-opening **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 filename 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 of the two streams if they are both connected to the same terminal.

8.4.5. *Unformatted text input.* Unformatted input is best done with **fgetc** for a single character and **fgets** for a string. The **stdin** standard stream is always defined and usually connects to terminal input:

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

Annex K

getchar and **gets_s** also read from **stdin**, but they don't add much to the previous interfaces that are more generic:

```
char* gets_s(char s[static 1], rsize_t n);
```

Historically, in the same spirit in which **puts** specializes **fputs**, the prior version of the C standard had a **gets** interface. Its buffer overflow handling was inherently unsafe, and so it has been removed from the C standard.

Takeaway 8.4.5 #1 *Don't use* ***gets***.

The following listing shows a function that has functionality equivalent to **fgets**.

LISTING 8.2. Implementing **fgets** in terms of **fgetc**

```
1  char* fgets_manually(char s[restrict], int n,
2                       FILE*restrict stream) {
3    if (!stream) return nullptr;
4    if (!n) return s;
5    /* Reads at most n-1 characters */
6    for (size_t i = 0; i < n-1; ++i) {
7      int val = fgetc(stream);
8      switch (val) {
9        /* EOF signals end-of-file or error */
10       case EOF: if (feof(stream)) {
11         s[i] = 0;
12         /* Has been a valid call */
13         return s;
14       } else {
15         /* Error */
16         return nullptr;
17       }
18       /* Stop at end-of-line. */
19       case '\n': s[i] = val; s[i+1] = 0; return s;
20       /* Otherwise just assign and continue. */
21       default: s[i] = val;
22     }
23   }
24   s[n-1] = 0;
25   return s;
26 }
```

Again, such example code is not meant to replace the function but to illustrate properties of the functions in question: here, the error-handling strategy.

Takeaway 8.4.5 #2 *`fgetc` returns `int` to be able to encode a special error status, `EOF`, in addition to all valid characters.*

Also, detecting 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 whether a stream's position has reached its end-of-file marker.

Takeaway 8.4.5 #3 *End-of-file can only be detected* after *a failed read.*

Listing 8.3 presents an example that uses both input and output functions.

LISTING 8.3. A program to dump multiple text files to `stdout`

```
#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] = { };
  for (int i = 1; i < argc; ++i) {        // Processes args
    FILE* instream = fopen(argv[i], "r"); // as filenames
    if (instream) {
      while (fgets(buffer, buf_max, instream)) {
        fputs(buffer, stdout);
      }
      fclose(instream);
      ret = EXIT_SUCCESS;
    } else {
      /* Provides some error diagnostic. */
      fprintf(stderr, "Could not open %s: ", argv[i]);
      perror(0);
      errno = 0;                          // Resets the error code
    }
  }
  return ret;
}
```

This is 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 64][Exs 65][Exs 66][Exs 67]

8.5. 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 independently of the encoding. The most important tools are given by the language itself: integer character constants, such as `'a'` and `'\\n'`, and string literals, such as `"hello:\tx"`, should always do the right thing on your platform. As you perhaps remember, there are not necessarily literals for types that are narrower than `int`. As a historical artifact, integer character literals, such as

[Exs 64] Under what circumstances will this program finish with success or failure return codes?

[Exs 65] Surprisingly, this program even works for files with lines that have more than 31 characters. Why?

[Exs 66] Have the program read from `stdin` if no command-line argument is given.

[Exs 67] Have the program precede all output lines with line numbers if the first command-line argument is `"-n"`.

`'a'`, have type **int**, not **char** as you would probably expect.[68] Handling such literals can become cumbersome if you have to deal with character classes.

Therefore, the C library provides functions and macros that deal with the most commonly used classes through the header `<ctype.h>`. It has the classifier functions <ctype.h>

isalnum isalpha isblank iscntrl isdigit isgraph
islower isprint ispunct isspace isupper isxdigit

and conversions **toupper** and **tolower**. Again, for historical reasons, all of these take their arguments as **int** and also return **int**. See table 8.14 for an overview of the classifiers. The functions **toupper** and **tolower** convert alphabetic characters to the corresponding case and leave all other characters as they are.

TABLE 8.14. Character classifiers. The third column indicates whether C implementations may extend these classes with platform-specific characters, such as `'ä'` as a lowercase character or `'€'` as punctuation.

Name	Meaning	C locale	Extended
islower	Lowercase	`'a' ··· 'z'`	Yes
isupper	Uppercase	`'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

The table has some special characters, such as `'\n'` for a newline character, which we have encountered previously. All the special encodings and their meaning are given in table 8.15.

TABLE 8.15. Special characters in character and string literals

`'\''`	Quote
`'\"'`	Double quotes
`'\?'`	Question mark
`'\\'`	Backslash
`'\a'`	Alert
`'\b'`	Backspace
`'\f'`	Form feed
`'\n'`	New line
`'\r'`	Carriage return
`'\t'`	Horizontal tab
`'\v'`	Vertical tab

[68] In addition, there are character literals `u8'a'`, `u'a'`, and `U'a'` that usually have the types **uint8_t**, **uint16_t**, and **uint32_t**, respectively. We will see their use in the following discussion.

Integer character literals can also be encoded numerically as an octal value of the form `'\037'` or as a 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 requires special care to mark the end of such a character: `"\xdeBruyn"` is not the same as `"\xde" "Bruyn"`[69] but corresponds to `"\xdeB" "ruyn"`, the character with code 3,563 followed by the four characters `'r'`, `'u'`, `'y'`, and `'n'`. Using this feature is only portable in the sense that it will compile on all platforms as long as a character with code 3,563 exists. Whether it exists and what that character actually is depends on the platform and the particular setting for program execution.

Takeaway 8.5 #1 *The interpretation of numerically encoded characters depends on the execution character set.*

So, their use is not fully portable and should be avoided.

The following function `hexatridecimal` uses some of these functions to provide a base 36 numerical value for all alphanumerical characters. This is analogous to hexadecimal literals, only all other letters have a value in base 36, too. Note that lower- and uppercase letters result in the same values; in particular, the characters `'a'` and `'A'` both map to the value 10.[Exs 70][Exs 71][Exs 72]

strtoul.c

```
/* Supposes that lowercase characters are contiguous. */
static_assert('z'-'a' == 25,
              "alphabetic characters not contiguous");
#include <ctype.h>
/* Converts 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 lowercase */
    a = toupper(a);
    /* Returns value >= 36 if not Latin uppercase */
    return (isupper(a)) ? 10 + (a - 'A') : -1;
  }
}
```

In addition to **strtod**, the C library has

strtoul strtol strtoumax strtoimax
strtoull strtoll strtold strtof

[69] But remember that consecutive string literals are concatenated (takeaway 5.3 #1).

[Exs 70] The second **return** of `hexatridecimal` makes an assumption about the relation between `a` and `'A'`. What is it?

[Exs 71] Describe an error scenario in which this assumption is not fulfilled.

[Exs 72] Fix this bug: that is, rewrite this code such that it makes no assumption about the relation between `a` and `'A'`.

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 lowercased letter L) for **long**, `d` for **double**, `f` for float, and `[i|u]max` for **intmax_t** and **uintmax_t**.

The interfaces with an integral return type all have three parameters, such as **strtoul**

```
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`, `2`, `8`, `10`, and `16`. The last four correspond to binary, octal, decimal, and hexadecimal encoding, respectively. The first, `0`, is a combination of these four, where the base is chosen according to the usual rules for the interpretation of text as numbers: `"7"` is decimal, `"0b10"` is binary, `"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 be used to obtain the position of the remaining data, but this is still too involved for us. For the moment, it suffices to pass a `0` for that parameter to ensure that everything 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 similarly, only the number of function parameters is limited to two.

Next, we will demonstrate how such functions can be implemented from more basic primitives. Let us first look at `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

```
31 unsigned long Strtoul_inner(char const s[static 1],
32                             size_t i,
33                             unsigned base) {
34   unsigned long ret = 0;
35   while (s[i]) {
36     unsigned c = hexatridecimal(s[i]);
37     if (c >= base) break;
38     /* Maximal representable value for 64 bit is
39        3w5e11264sgsf in base 36 */
40     if (ULONG_MAX/base < ret) {
41       ret = ULONG_MAX;
42       errno = ERANGE;
43       break;
44     }
45     ret *= base;
46     ret += c;
47     ++i;
48   }
49   return ret;
50 }
```

If the string represents a number that is too big for an **unsigned long**, this function returns **ULONG_MAX** and sets **errno** to **ERANGE**. If the string contains no digit at all, this function just returns 0, and such an error cannot be detected from that return value. The `endptr` parameter of **strtoul** could be used for that, if only we had already mastered pointers well enough.

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) {              /* Tests if base          */
    errno = EINVAL;              /* Extends the specification */
    return ULONG_MAX;
  }
  size_t i = strspn(s, "␣\f\n\r\t\v"); /* Skips spaces    */
  bool switchsign = false;       /* Looks for a sign       */
  switch (s[i]) {
  case '-' :
    switchsign = true;
    [[fallthrough]];
  case '+' :
    ++i;
  }
  /* Test if there is anything left in the string. */
  if (!s[i]) return 0;
  if (!base || base == 16 || base == 2) { /* use prefix */
    size_t adj = find_prefix(s, i, "0x");
    switch (adj) {
    case 2:                    // There is a 0x or a 0X prefix
      if (!base || base == 16) base = 16;
      // If we are looking for another base, the x is the end
      else return 0;
      break;
    case 1:                    // There is a 0 prefix
      adj = find_prefix(s, i, "0b");
      switch (adj) {
      case 1:                  // There is only a 0 prefix
        if (!base) base = 8;
        break;
      default:                 // There is a 0b or a 0B prefix
        if (!base || base == 2) base = 2;
        // If we are looking for another base, the b is the end
        else return 0;
        break;
      }
      break;
    default:
      if (!base) base = 10;
      break;
    }
    i += adj;
  }
  /* Test again, maybe the prefix 0 was the only digit. */
  if (!s[i]) return 0;
  /* Now, starts the real conversion */
  unsigned long ret = Strtoul_inner(s, i, base);
  return (switchsign) ? -ret : ret;
}
```

It wraps `Strtoul_inner` and does the previous adjustments that are needed: it skips white space, looks for an optional sign, adjusts the base in case the `base` parameter was `0`, and skips an eventual `0`, **`0b`**, or **`0x`** prefix. Observe also that if a minus sign has been provided, it does the correct negation of the result in terms of **`unsigned long`**

arithmetic.[Exs 73] The code that deals with the character for a + or − sign also uses the attribute `[[fallthrough]]` to indicate that running from the − case into the + case is intentional so no warning should be given.

To skip the spaces, `Strtoul` uses **strspn**, one of the string search functions provided by `<string.h>`. This function returns the length of the initial sequence in the first parameter that entirely consists of any character from the second parameter. The function **strcspn** ("c" for "complement") works similarly, but it looks for an initial sequence of characters **not** present in the second argument. <string.h>

This header provides a lot more memory and 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.

8.5.1. *Portability of string processing.* Unfortunately, the functions that convert strings to numbers have changed semantics between different versions of the C standard and are not fully consistent with string literals for these numbers themselves. This also has an effect on formatted input functions that we will see in 14.2 because they rely on these string functions.

The first problem is that the functions have changed the format they accept twice, for C99 and for C23. Take, for example, an innocent looking call:

```
double x = strtod("0x1.0P0", nullptr);
```

This is supposed to process the string literal up to its end and should return the value 1.0. But since hexadecimal floating point literals had only been introduced in C99, libraries from before would stop at the `x` character and return the value 0.0.

Similarly, introducing a prefix of **0b** for integer literals in C23 has the effect that

```
unsigned long res = strtoul("0b1", nullptr, 2);
```

results in 0 for C17 and 1 for C23 because for the first, the interpretation stops before the `b`. This semantic change concerns **strtol** and similar functions for base `0` and `2`.

Takeaway 8.5.1 #1 *Don't use the string conversion functions to determine the boundaries of numbers.*

Second, the new digit separator `'` that was introduced in C23 has no portable correspondence in the string processing functions. Here is an artificial example that shows the problem:

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

#define STRINGIFY_(X) #X
#define STRINGIFY(X) STRINGIFY_(X)

char const elements[] = STRINGIFY(ULLONG_MAX);

int main(int argc, char* argv[argc+1]) {
    char const* p = (argc > 1) ? argv[1] : elements;
    if (strtoull(p, nullptr, 0) <= 65535) {
        printf("unusual platform with %s max\n", p);
    }
}
```

Here, the initializer of `elements` (using **ULLONG_MAX**) would typically expand to a string such as

[Exs 73] Implement a function `find_prefix` as needed by `Strtoul`.

```
"0xffffffffffffffffLL"
"18446744073709551615"
```

which then would be correctly recognized as numbers and then skip the call to **printf**.

In C23, if the implementation chooses to change that to use digit separators (for example, to improve readability) the expansion now might look like

```
"0xffff'ffff'ffff'ffffLL"
"18'446'744'073'709'551'615"
```

So, the call to **strtoull** would only see the leading **0x**ffff or 18 and run into the branch with the call to **printf**.

Takeaway 8.5.1 #2 *Don't use the string conversion functions to scan numbers that originate from number literals.*

8.6. Time. The first class of "times" can be classified as calendar times, times with a granularity and range that would typically appear in a human calendar for appointments, birthdays, and so on. The following are some of the functional interfaces that deal with times and that are all provided by the <time.h> header:

<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);
int timespec_getres(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 uses the return value of **time**(0). As we have seen, two such times taken from different moments during program execution can then be used to express a time difference by means of **difftime**.

Let's see what all this is doing from the human perspective. As we know, **struct tm** structures a calendar time mainly as you would expect. It has hierarchical date members such as **tm_year** for the year, **tm_mon** for the month, and so on, down to the granularity of a second. It has one pitfall, though: how the members are counted. All but one start with 0: for example, **tm_mon** set to 0 stands for January, and **tm_wday** 0 stands for Sunday.

Unfortunately, there are exceptions:

- **tm_mday** starts counting days in the month at 1.
- **tm_year** must add 1900 to get the year in the Gregorian calendar. Years represented that way should be between Gregorian years 0 and 9999.
- **tm_sec** is in the range from 0 to 60, inclusive. The latter is for the rare occasion of leap seconds.

Three supplemental date members are used to supply additional information to a time value in a **struct tm**:

- **tm_wday** for the weekday.
- **tm_yday** for the day in the year.
- **tm_isdst** is a flag that informs us whether a date is considered to be in daylight savings time for the local time zone.

The consistency of all these members can be enforced with the function **mktime**. It operates in three steps:

(1) The hierarchical date members 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 or to 0 otherwise.

mktime also serves an extra purpose. It returns the time as a **time_t**. **time_t** represents the same calendar times as **struct tm** but is defined to be an arithmetic type more suited to compute with such types. It operates on a linear time scale. A **time_t** value of 0 at the beginning of **time_t** is called the *epoch*[C] in C jargon. Often this corresponds to the beginning of January 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.

Other traditional functions that manipulate time in C are a bit dangerous because they operate on global state:

```
[[deprecated]] char *asctime(const struct tm *timeptr);
[[deprecated]] char *ctime(const time_t *timer);
struct tm *gmtime(const time_t *timer);
struct tm *localtime(const time_t *timer);
```

We will not discuss them further because safer variants of these interfaces have been added to C23:

```
time_t timegm(struct tm *timeptr);
struct tm *gmtime_r(const time_t *timer, struct tm *buf);
struct tm *localtime_r(const time_t *timer, struct tm *buf);
```

Figure 8.1 shows how these functions interact.

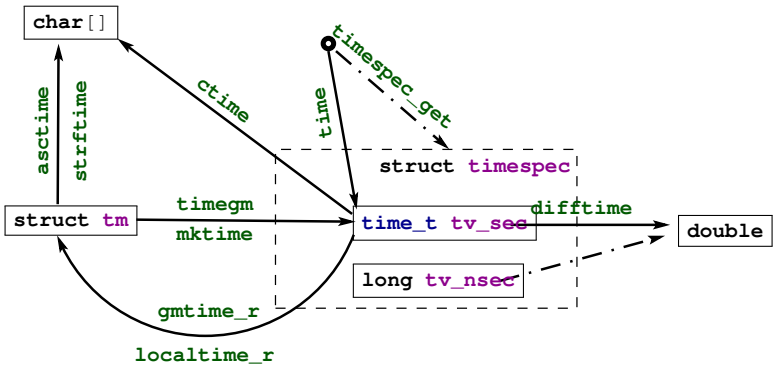


FIGURE 8.1. Time conversion functions

Two functions for the inverse operation from **time_t** into **struct tm** come into view:

- **localtime_r** stores the broken-down local time.
- **gmtime_r** stores the broken-down time, expressed as universal time, UTC.

As indicated, they differ in the time zone they assume for the conversion. Under normal circumstances, **localtime_r**/**mktime**, and **gmtime_r**/**timegm** should be inverse to each other, respectively.

Textual representations of calendar times are also available. Indicated by the attribute [[**deprecated**]], **asctime** is deprecated and should not be used in new

code. It 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"
```

The reason why this function has been deprecated is that it uses a static buffer to return the result. So subsequent or parallel invocations of the function erase the stored content of the string.

strftime is more flexible and allows us to compose a textual representation with format specifiers. This should be the interface of choice whenever you want a textual presentation of time.

Table 8.16: **strftime** format specifiers. Those selected in the Locale column may differ dynamically according to locale runtime settings; see subsection 8.7. Those selected in the ISO 8601 column 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	✓	
`"%b"`	Abbreviated month name	✓	
`"%h"`	Equivalent to `"%b"`	✓	
`"%Y"`	Year		
`"%y"`	Year (`"00"` to `"99"`)		
`"%C"`	Century number (year/100)		
`"%G"`	Week-based year; the same as `"%Y"`, except if the ISO week number belongs another year		✓
`"%g"`	Like `"%G"`, (`"00"` to `"99"`)		✓
`"%u"`	Weekday (`"1"` to `"7"`), Monday being `"1"`		
`"%w"`	Weekday (`"0"` to `"6"`, Sunday being `"0"`		
`"%A"`	Full weekday name	✓	
`"%a"`	Abbreviated weekday name	✓	
`"%j"`	Day of the 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 four days in the new year		✓
`"%Z"`	Timezone name	✓	
`"%z"`	`"+hhmm"` or `"-hhmm"`, the hour and minute offset from UTC		
`"%n"`	Newline		
`"%t"`	Horizontal tabulator		
`"%%"`	Literal `"%"`		
`"%x"`	Date	✓	
`"%D"`	Equivalent to `"%m/%d/%y"`		
`"%F"`	Equivalent to `"%Y-%m-%d"`		✓

Table 8.16: **strftime** format specifiers, continued.

Spec	Meaning	Locale	ISO 8601
"%X"	Time	✓	
"%p"	Either "AM" or "PM": noon is "PM", midnight is "AM"	✓	
"%r"	Equivalent to "%I:%M:%S␣%p"	✓	
"%R"	Equivalent to "%H:%M"		
"%T"	Equivalent to "%H:%M:%S"		✓
"%c"	Preferred date and time representation	✓	

It works similarly to the **printf** family but has special %-codes for dates and times; see table 8.16. Here, the Locale column 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 subsection 8.7. **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 as a consequence **time** itself) only has a granularity of seconds.

If we need more precision than that, **struct** **timespec** and the **timespec_get** function can be used. With that, we have an additional member **tv_nsec** that provides nanosecond precision. The second argument, base, has only one value that is required by the C standard: **TIME_UTC**. You should expect a call to **timespec_get** with that value to be consistent with calls to **time**; the resolution of that clock can be queried with **timespec_getres** (since C23). They both refer to Earth's reference time. Other clocks can be relative to the planetary or other physical system your computer system is involved with.[74] In particular, there is an additional interface that is provided by the C standard library and that collects the processing time that is attributed to the current execution:

```
clock_t clock(void);
```

For historical reasons, this introduces yet another type, **clock_t**. It is an arithmetic type 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. To deal with that situation a more easily, C23 adds optional macros:

- **TIME_ACTIVE** for a time base that is sought to be compatible with **clock**
- **TIME_THREAD_ACTIVE**, which measures processing times restricted to the current thread
- **TIME_MONOTONIC** for a time base that is independent of time adjustments

Specific platforms may provide even other macros (starting with **TIME_**) for other time bases.

[74] Be aware that objects that move fast relative to Earth, such as satellites and spacecraft, may perceive relativistic time shifts compared to UTC.

CHALLENGE 10 (Performance comparison of sorting algorithms). *Can you compare the time efficiency of your sorting programs (challenge 1) with data sizes of several orders of magnitude?*
Be careful to check that you have some randomness in the creation of the data and that the data size does not exceed the available memory of your computer.
For both algorithms, you should roughly observe a behavior that is proportional to $N \log N$*, where* N *is the number of elements that are sorted.*

8.7. Runtime environment settings. A C program can access an ***environment list***^{C}, which is 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 historical function **getenv** to access this list:

```
char* getenv(char const name[static 1]);
```

Given our current knowledge, with this function, we are only able to test whether a `name` is present in the environment list:

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

Annex K

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

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

This function copies the value that corresponds to `name` (if any) from the environment into `value`, a **char**`[maxsize]`, provided that it fits. Printing such a value can look as this:

```
void printenv(char const name[static 1]) {
  if (getenv(name)) {
    char value[256] = { };
    if (getenv_s(nullptr, 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, after detecting whether the environment variable exists, **getenv_s** can safely be called with the first argument set to **nullptr**. Additionally, it is guaranteed that the `value` target buffer will only be written if the intended result fits in it. The `len` parameter could be used to detect the real length that is needed, and dynamic

TABLE 8.17. Categories for the **setlocale** function

LC_COLLATE	String comparison through **strcoll** and **strxfrm**
LC_CTYPE	Character classification and handling functions; see subsection 8.5.
LC_MONETARY	Monetary formatting information, **localeconv**
LC_NUMERIC	Decimal-point character for formatted I/O, **localeconv**
LC_TIME	**strftime**; see subsection 8.6
LC_ALL	All of the above

buffer allocation could be used to print out even large values. We will wait until higher levels to see such usages.

Which environment variables are available to programs depends 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, and `"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: `<locale.h>`

```
char* setlocale(int category, char const locale[static 1]);
```

In addition to `"C"`, the C standard 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 8.17 gives an overview of the possible values and the part of the C library they affect. Additional platform-dependent categories may be available.

8.8. Program termination and assertions. We have looked at the simplest way to terminate a program: a regular return from **main**.

Takeaway 8.8 #1 *Regular program termination should use a* ***return*** *from* ***main****.*

Using the function **exit** from within **main** is kind of senseless because it can be done just as easily with a **return**.

Takeaway 8.8 #2 *Use* ***exit*** *from a function that may terminate the regular control flow.*

The C library has three other functions that terminate program execution. They are, in order of severity, as follows:

```
[[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 whether the program execution is considered to be a success. 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.

Takeaway 8.8 #3 *Don't use functions other than* **exit** *for program termination, 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 is even 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 we have not yet seen: ***function parameters***^{C}. For example, the first reads "function **atexit** that returns an **int** and that receives a function func as a parameter."[75]

We will not go into detail here. An example will show 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 the status of the execution. This might be a nice way to impress your co-workers if you are in need of some respect. More seriously, this is the ideal place to put all kinds of cleanup code, such as freeing memory or writing a termination timestamp to a log file. Observe that the syntax for calling is **atexit**(sayGoodBye). There are no () for sayGoodBye itself: here, sayGoodBye is not called at that point; only a reference to the function is passed to **atexit**.

Under rare circumstances, 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 an alternative list may be useful if the normal execution of the handlers is too time consuming or you need to use **quick_exit** to terminate execution from a signal handler; we will see that in section 19.6. Use with care.

The next function, **_Exit**, is even more severe: it inhibits both types of application-specific handlers to be executed. The only things that are 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 doesn't it call the application handlers, but it also inhibits the execution of some system cleanups. Use this with extreme care.

At the beginning of this section, we looked at **static_assert**, which should be used to make compile-time assertions. They can test for any form of compile-time Boolean expression. Two other identifiers come from <assert.h> and 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

<assert.h>

[75] 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 the pointer variant. For us, this distinction is not yet relevant; it is simpler to think of a function being passed by reference.

time execution passes by the call to this macro, the expression is evaluated. The functions `gcd` and `gcd2` from subsection 7.3 show typical use cases of **`assert`**: a condition that is supposed to hold in *every* execution.

If the condition doesn't hold, a diagnostic message is printed, and **`abort`** is called. So, none of this should make it through into a production executable. From the earlier discussion, 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 values of variables.

Takeaway 8.8 #4 *Use as many **`assert`**s as you can to confirm runtime properties.*

As mentioned, **`NDEBUG`** inhibits the evaluation of the expression and the call to **`abort`**. **`NDEBUG`** would typically only be set for the whole compilation of a program or library that is considered to be ready for production because, if set, **`assert`** would not trigger if an erroneous value occurs. Because it is set for the whole compilation of a project, it would usually be set from the compiler command line and not with a **`#define`** directive.

Takeaway 8.8 #5 *In production compilations, use **`NDEBUG`** to switch off all **`assert`**s.*

CHALLENGE 11 (Image segmentation). *In addition to the C standard library, there are many other support libraries out there that provide very different features. Among those are a lot that do image processing of some kind. Try to find a suitable such image-processing library that is written in or interfaced to C and allows you to treat grayscale images as two-dimensional matrices of base type **`unsigned char`**.*
The goal of this challenge is to perform a segmentation *of such an image: to group the pixels (the **`unsigned char`** elements of the matrix) into connected regions that are "similar" in some sense or another. Such a segmentation forms a partition of the set of pixels, much as we saw in challenge 4. Therefore, you should use a Union-Find structure to represent regions, one per pixel at the start.*
Can you implement a statistics function that computes a statistic for all regions? This should be another array (the third array in the game) that for each root holds the number of pixels and the sum of all values.
Can you implement a merge criterion for regions? Test whether the mean values of two regions are not too far apart: say, no more than five gray values.
Can you implement a line-by-line merge strategy that, for each pixel on a line of the image, tests whether its region should be merged to the left and/or to the top?
Can you iterate line by line until there are no more changes, such that the resulting regions/sets all test negatively with their respective neighboring regions?
Now that you have a complete function for image segmentation, try it on images with assorted subjects and sizes. Vary your merge criterion with different values for the the mean distance instead of five.

Summary

- The C library is interfaced via a bunch of header files.
- Mathematical functions are best used via the type-generic macros from `tgmath.h`.
- Input and output (IO) are interfaced via `stdio.h`. There are functions that do IO as text or as raw bytes. Text IO can be direct or structured by formats.
- String processing uses functions from `ctype.h` for character classification, from `stdlib` for numerical conversion, and from `string.h` for string manipulation.
- Time handling in `time.h` has *calendar time* that is structured for human interpretation and *physical time* that is structured in seconds and nanoseconds.
- Standard C only has rudimentary interfaces to describe the execution environment of a running program; **`getenv`** provides access to environment variables, and `locale.h` regulates the interface for human languages.

LEVEL 2

Cognition

The Eurasian jay may be solitary or found in pairs. It is known for its mimicry of other bird calls, for its alertness, and for its dispersal of seeds that contribute to forest expansion.

Now we are advanced enough to go to the heart of C. 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 (sections 9 and 10). Then it fills in the gaps for the major C constructs that we have skipped so far: it fully explains pointers (section 11), familiarizes you with C's memory model (section 12) and dynamic storage allocation (section 13), and allows you to understand most of C's library interface (section 14). We finish with a more systematic discussion of all the possible failures of C programs (section 15).

9. Style

This section covers

- Writing readable code
- Formatting code
- Naming identifiers

Programs serve a dual-purpose. 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 the people (users, customers, maintainers, lawyers, and so on) who have to deal with it. Therefore, we have a prime directive.

Takeaway 9 #1 *All C code must be readable.*

The difficulty with that directive is knowing what constitutes "readable." Not all experienced C programmers agree, so we will begin by trying to establish a minimal list of necessities. The first thing we must have in mind when discussing the human condition is that it is constrained by two major factors: physical ability and cultural baggage.

Takeaway 9 #2 *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 consists of roughly 30 lines of 80 columns, making a "surface" of 2,400 characters. Everything that doesn't fit has to be memorized. For example, our very first program in listing 1.1 fits into these constraints.

By its humorous reference to Kernighan and Ritchie [1978], the Linux coding style also refers to another fundamental fact.

Takeaway 9 #3 *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.

Takeaway 9 #4 *Each established project constitutes its own 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 them; 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 of 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.

Takeaway 9.1 #1 *Choose a consistent strategy for white space and other text formatting.*

Formatting concerns indentation, placement of parentheses and all kinds of brackets (`{}`, `[]`, and `()`), spaces before and after operators, trailing spaces, and multiple new lines. The human eye and brain are quite peculiar in their habits, and to ensure that they work properly and efficiently, everything must be in sync.

In the introduction for level 1, you saw a lot of the coding style rules applied to the code in this book. Take them as an example of one style; you will most likely encounter other styles as you go along. Let us recall some of the rules and introduce some others that have not yet been presented:

- We use prefix notation for code blocks: that is, an opening `{` is at 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`**) with a space.
- A ternary expression has spaces around the `?` and the `:`.
- Punctuation marks (`:`, `;`, 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 understand new code in 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 can help you with them.

Takeaway 9.1 #2 *Have your text editor automatically format your code correctly.*

I personally use Emacs (https://www.gnu.org/software/emacs/) 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 life that gives you less. Text editors, integrated development environments (IDEs), and code generators are there for us, not the other way around.

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 command-line tools that do the formatting. Here, I have a long-time preference for `astyle` (artistic style `http://sourceforge.net/projects/astyle/`). Again, your mileage may vary; please choose any tool that ensures the task for you.

9.2. Naming. The limit of such automatic formatting tools is reached when it comes to naming.

Takeaway 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 the subject of endless ideological debate.

For C, various technical restrictions apply; they are meant to help you, so take them seriously. First, we target *all identifiers*: types (**`struct`** or not), **`struct`** and **`union`** members, variables, enumerations, macros, functions, function-like macros. There are so many tangled ***name spaces***[C] that you have to be careful.

In particular, the interaction between header files and macro definitions can have surprising effects. Here is a seemingly innocent example:

```
double memory_sum(size_t N, size_t I, double strip[N][I]);
```

- `N` is a capitalized identifier, and thus your collaborator could be tempted to define a macro `N` as a big number.
- **`I`** is used for the root of -1 as soon as someone includes `<complex.h>`. `<complex.h>`
- The identifier `strip` might be used by a 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.

Takeaway 9.2 #2 *Any identifier that is visible in a header file must be conforming.*

Here, *conforming* is a wide field. In C jargon, an identifier is ***reserved***^{C} if its meaning is fixed by the C standard, and you may not redefine it otherwise:

- Names starting with an underscore and a second underscore or a capital letter are reserved for language extensions and other internal use.
- Names starting with an underscore are reserved for file scope identifiers 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. This includes a lot of identifiers, such as all functions in the C library, all identifiers starting with `str` (like our `strip`, earlier), all identifiers starting with `E`, all identifiers ending in `_t`, and many more.

What makes all of these rules relatively difficult is that you might not detect any violation for years, and then, suddenly, 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.

Takeaway 9.2 #3 *Don't pollute the global space of identifiers.*

Expose only types and functions as interfaces that are part of the ***application programming interface***^{C} (***API***^{C})— that is, those that are supposed to be used by users of your code.

A good strategy for a library used by others or in other projects is to use naming prefixes that are unlikely to create conflicts. For example, many functions and types in the POSIX thread API are prefixed with `pthread_`. For my tool box P99, I use the prefixes `p99_` and `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 and which you might not think of immediately:

- Member names of **`struct`** and **`union`**
- Parameter names in function interfaces

The first point is the reason why the members in standard structures usually have a prefix to their names: **`struct timespec`** has **`tv_sec`** as a member name because an uneducated user might declare a macro `sec` that would interfere in an unpredictable way when including `<time.h>`. For the second point, we saw an example earlier. In P99, I would specify such a function something like this:

<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:

- So-called **`inline`** functions, which are functions whose definition (not only declaration) is visible in a header file
- Function-like macros

We will discuss these features much later in sections 16.1 and 17.

Now that we have clarified the technical points of naming, we will look at the semantic aspect.

Takeaway 9.2 #4 *Names must be recognizable and quickly distinguishable.*

TABLE 9.1. Some examples of well and badly distinguishable identifiers.

		Recognizable	Distinguishable	Quickly
`llllllllOll`	`lllllllllOll`	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

That has two parts: distinguishable *and* quickly. Compare the identifiers in table 9.1.

For your personal taste, the answers on the right side of this table may be different. This reflects *my* taste: 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 an implicit semantic.

For me, because I have 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.

The 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 *find first (bit) set*, but this is not immediately obvious to me. What it means is even less clear: which bit is first, the most significant or the least significant? In C23, which now integrates these functionalities via the `<stdbit.h>` header, more significant names have been chosen, **`stdc_bit_width`** and **`stdc_trailing_zeros`**. `<stdbit.h>`

There are several conventions that combine multiple 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},[1] which encodes type information in the prefix of the identifiers, such as `szName`, where `sz` stands for *string* and *zero terminated*

As you might imagine, none of these is ideal. The first two tend to obscure our view: they easily clog up a whole precious line of programming text with an unreadable expression:

```
1  return theVerySeldomlyUsedConstant*theVerySeldomlyUsedConstant/
       number_of_elements;
```

Hungarian notation, in turn, tends to use obscure abbreviations for types or concepts, produces unpronounceable identifiers, and completely breaks down if you have an API change.

So, in my opinion, none of these rules or strategies have absolute values. I encourage you to take a pragmatic approach to the question.

Takeaway 9.2 #5 *Naming is a creative act.*

It is not easily subsumed by simple technical rules. Obviously, good naming is more important the more widely an identifier is used. So, it is particularly important

[1] Invented in Simonyi [1976], the PhD thesis of Simonyi Károly.

for identifiers for which the declaration is generally out of view of the programmer: global names that constitute the API.

Takeaway 9.2 #6 *File scope identifiers must be comprehensive.*

What constitutes *comprehensive* here should be derived from the type of the identifier. Type names, constants, variables, and functions generally serve different purposes, so different strategies apply.

Takeaway 9.2 #7 *A type name identifies a concept.*

Examples of such concepts are *time* for **struct timespec**, *size* for **size_t**, a collection of corvidae for **enum** `corvid`, *person* for a data structure that collects data about people, *list* for a chained list of items, *dictionary* for a query data structure, and so on. If you have difficulty coming up with a concept for a data structure, an enumeration, or an arithmetic type, you should probably revisit your design.

Takeaway 9.2 #8 *A global constant identifies an artifact.*

That is, a constant *stands out* for some reason from the other possible constants of 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**), to be factual (`corvid_num`), for a reason that is culturally motivated (`fortytwo`), or as a design decision.

Generally, we will see shortly 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.

Takeaway 9.2 #9 *A global variable identifies state.*

Typical names for such variables are `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, and `visited_entries` for a hash table that collects shared data.

Takeaway 9.2 #A *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. Here are 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 member could be called `toto_getFlag`
- The corresponding one that sets such a member would be `toto_setFlag`
- A function that multiples two matrices is `matrixMult`

9.3. Internationalization, so to speak. Generally, in the English-speaking world, the term "internationalization" refers to the ability of a platform or feature to adapt to other language conventions than English. I put this term into quotes because I think that it already contains quite an amount of hybris, namely that English would be the center of the world's culture and everybody else just spins around it as of secondary importance.

Then, the term also goes too short because the features to which it usually refers are not only to different national cultures or languages; they also concern subcultures (such as adepts of the invented Klingon language) and specific technical requirements

in specialized contexts, such as mathematical notation, IPA phonetics, and graphical characters.

So, let's first come back to using other languages and scripts in coding. Note that this question is different (and should be dissociated) from the question of whether a program supports one or multiple languages for its users. In section 8.7 we have already seen some features that help adapt a program to the environment in which it is executed.

Note that English is definitively of high importance in the computer science industry and often serves as a *lingua franca*. As a German living in France writing an English book about C, I am confronted with that every day.[2]

But outside of my bubble, it may be completely sensible to use different language conventions for variable and function names and code comments. In particular, programming communities in cultures that use scripts that are not based on Latin should (and do) use other scripts and languages to a large extent. As it had been developed in English, C has a historical baggage of implicitly forcing that language in coding. Whenever these things are discussed publicly (for example, on social networks), often you would hear quite pronounced (perceivably arrogant) opinions on why one would even think of using another language or, even more qualified, that the person would not know how to enter a non-English character on their keyboard.

Luckily, affairs have slowly evolved such that communities now have a real choice about the language features they want to use. Unicode support (see section 14) is usually good enough, and C has embraced at least some level of language support, even for the coding itself. Code as in the following should easily be accepted and maintainable without difficulties:

```
long année = 1990L; // Année de l'écriture de l'œuvre
```

If it isn't, it is not the fault of the person who wrote that code, but it is your environment, implementation, system, or institution that is to blame. If you are surprised by such code, you should first question your own compass and your own prejudice before suggesting a change.

Takeaway 9.3 #1 *The natural language of a project should be chosen to accommodate the majority of the participants.*

That is, again, what language to choose depends on a lot of factors. The most important is that everybody in the project is mostly at ease. In Western societies, that will often mean that projects use English, but don't take this as a given just because you were raised with that assumption.

The other aspect that should be more focused on is the adequate technical language that is used for domain-specific problems. As an example domain, consider mathematics. If you look into usual mathematical texts, you will see a lot of different scripts all mixed together with a lot of conventions about proper names (for example, π, $\mathbb{C}$, $\mathbb{N}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{Z}$), classes of symbols (α, β, and γ for angles, δ, and ε for small quantities, $\aleph$ for infinities) that all contribute so that somebody from the corresponding community, such a text is readable with the blink of an eye. There should be no restriction to using such characters and conventions anywhere in the documentation to describe what the code does. Modern systems *have to be able* to use these and work with them smoothly.

When it comes to using other scripts and mathematical symbols at the core of the language (namely, as identifiers), things are much more disputed and have been much more difficult technically in the past. Previous versions of the C standard already allowed, in principle, the addition of a large panel of characters, but the specification for the allowed set of identifiers had been not easy to comprehend, and the platforms

[2] These are my four main languages. What are yours?

had not exactly been able to ease their integration into your code. The only portable way to do so has been with a crude syntax such as here:

```
long ann\u00E9e = 1990L; // Ann\u00E9e de l'\u00E9criture de l'\u0153uvre
```

Here, the aberrations of `\u00E9` and `\u0153` stand for the characters "é" and "œ," respectively. Having to encode them in that way somehow defeats the purpose, and this has only a clear application as an intermediate format. Tooling was needed to transform your native encoding before feeding it into the compiler, and not very surprisingly, nobody used this kind of feature.

With C23, the tide may have turned. Language support is now more straightly referring to Unicode, and the rules for allowed identifiers are now clear and refer to a substandard of Unicode, namely UAX #31 ("Unicode Identifier and Pattern Syntax"). The problem that part of Unicode addresses is the fact that many characters in non-English languages are composed of different components. For example, the "é" that we used previously is obtained from a plain Latin "e" with code `\u0065`) and an acute accent "´" with code `\u0301`. So, in general, there are several inputs (here, `\u00E9` and `\u0065\u0301`) that, depending on the context, are perceived as representing the same character.

The method from UAX #31 chosen for the C programming language to deal with kind of ambiguity is called *Normalization Form C*[3] To know to which unique character sequence a given string (or, here, identifier) maps, it is first decomposed into all the base characters, accents, and so on, and then it is recomposed. For our example of "é" or something like the *Hangul Syllable Gag* "각" (`\uAC01`), this is not very interesting. The resulting character of this procedure is the one with which we started:

```
\u00E9 → \u0065\u0301       → \u00E9
\uAC01 → \u1100\u1161\u11A8 → \uAC01
```

It becomes more interesting when several characters have the same decomposition, namely "Å" (Latin Capital Letter A with Ring Above) and "Å" (Angstrom sign):[4]

```
\u00C5 → \u0041\u030A → \u00C5
\u21B2 → \u0041\u030A → \u00C5
```

Here, the code point `\u21B2` is seen as having the same decomposition as the direct alphabetic composition and the Form C projects it to that code point `\u00C5`.

Another ambiguity occurs for signs that are themselves considered letters but based on Greek letters, such as the *Ohm sign*[5] (`\u2126`), which stems from *Greek capital letter Omega* "Ω" (`\u03A9`):

```
\u03A9 → \u03A9 → \u03A9
\u2126 → \u03A9 → \u03A9
```

On the other hand, technical symbols identified as alphabetic letters but with a glyph that is clearly distinct from all other alphabetic letters such as "ℜ" and "ℑ" (for real and imaginary parts of a complex number) map to themselves.

Takeaway 9.3 #2 *Alphabetic letters are only allowed in identifiers if they map to themselves for Normalization Form C.*

Such a rule is perhaps not explicit enough, so let's reformulate it.

[3] Choosing the same name "C" for the language and the Normalization Form is just a coincidence.

[4] A metric unit named after the Swedish physicist Anders Jonas Ångström (1814–1874).

[5] The unit for electrical resistance is named after the German physicist Georg Simon Ohm (1789–1854).

Takeaway 9.3 #3 *Only use alphabetic letters in identifiers if they originate directly from natural languages or they are clearly distinctive from all natural languages.*

But be aware that Normalization Form C does not solve all the possible problems. In particular, glyphs from different languages might be indistinguishable, such as a *Greek capital letter Alpha* and *Latin capital letter A*, which have the same glyph "A."

In other positions than the first, identifiers may also contain numbers. You may use the wide spectrum of Unicode's *Decimal Digit* category, but be careful because these may also have glyphs that are not easily distinguishable. For example, a name `a`**0** (using *mathematical bold digit zero*, **0**) may look very simlar to `a0` (using a usual *digit zero* 0 character).

Takeaway 9.3 #4 *Only use letters from different scripts or variations of decimal digits in identifiers if they are clearly distinctive from one another.*

One feature that is still missed by the rules that C23 introduced is subscript and superscript numbers. For my taste, it would be nice if we could distinguish variable names, such as, for example, $\texttt{a}_1$ and $\texttt{a}_2$. The compilers on my platform already allow this as an extension, but this might not (yet) be fully portable:

Takeaway 9.3 #5 *Using subscript or superscript letters in identifiers is not portable.*

Summary

- Coding style is a matter of culture. Be tolerant and patient.
- The choice of a project's natural language is important and should be consensual.
- Code formatting is a matter of visual habits. It should be automatically provided by your environment so you and your co-workers can read and write code effortlessly.
- Naming variables, functions, and types is an art and plays a central role in the comprehensiveness of your code.
- Identifiers used for names may be written with non-Latin characters to express ideas in the project's natural language or in the accepted terminology of the specific domain.

10. Organization and documentation

This section covers

- How to document interfaces
- How to explain implementations

Being an important societal, cultural, and economic activity, programming needs a certain form of organization to be successful. As with 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 through the experience of reading our own code sometime after we have written it and not having any clue what it was all about.

Documenting or, more generally, explaining program code is not an easy task. We have to find the right balance between providing context and necessary information and boringly stating the obvious. Let's have a look at the two following lines:

```
1   u = fun4you(u, i, 33, 28);  // ;)
2   ++i;                        // incrementing i
```

The first line isn't good because it uses magic constants, a function name that doesn't tell what is going on, and a variable name that does not have much meaning, at least to me. The smiley comment indicates that the programmer had fun when writing this, but it is not very helpful to the casual reader or maintainer.

In the second line, the comment is superfluous and states what any even not-so-experienced programmer knows about the ++ operator.

Compare that to the following:

```
1 /* 33 and 28 are suitable because they are coprime. */
2 u = nextApprox(u, i, 33, 28);
3 /* Theorem 3 ensures that we may move to the next step. */
4 ++i;
```

Here, we may deduce a lot more. 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 *in which manner* rules, in order of their importance:

Takeaway 10 #1 (what) *Function interfaces describe* what *is done.*

Takeaway 10 #2 (what for) *Interface comments document the purpose of a function.*

Takeaway 10 #3 (how) *Function code shows* how *the function is organized.*

Takeaway 10 #4 (in which manner) *Code comments explain the manner in which function details are implemented.*

In fact, if you think of a larger library project used by others, you'd expect that all users will read the interface specification (such as in the synopsis part of a `man` page), and most of them will read the explanation about these interfaces (the rest of the `man` page). Considerably fewer users will look at the source code and read about *how* or *in which manner* a particular interface implementation does things the way it does.

A first consequence of these rules is that code structure and documentation go hand in hand. The distinction between interface specification and implementation is especially important.

Takeaway 10 #5 *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} (*TU*), ending with `".c"`.

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

Takeaway 10 #6 *Document the interface; explain the implementation.*

10.1. Interface documentation. In contrast to more recent languages such as Java and Perl, C has no "built-in" documentation standard. But in recent years, a cross-platform public domain tool has been widely adopted in many projects: doxygen (`https://www.doxygen.nl/`). 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.

Takeaway 10.1 #1 *Document interfaces thoroughly.*

Doxygen has a lot of categories that help with that, but an extended discussion goes far beyond the scope of this book. Just consider the following example:

heron_k.h

```
116 /**
117  ** @brief use the Heron process to approximate @a a to the
118  ** power of `1/k`
119  **
120  ** Or in other words this computes the @f$k^{th}@f$ root of @a a
     .
121  ** As a special feature, if @a k is `-1` it computes the
122  ** multiplicative inverse of @a a.
123  **
124  ** @param a must be greater than `0.0`
125  ** @param k should not be `0` and otherwise be between
126  ** `DBL_MIN_EXP*FLT_RDXRDX` and
127  ** `DBL_MAX_EXP*FLT_RDXRDX`.
128  **
129  ** @see FLT_RDXRDX
130  **/
131 double heron(double a, signed k) [[__unsequenced__]];
```

Doxygen produces online documentation for that function that looks similar to figure 10.1 and is also 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

```
double heron(double a, signed k) [[__unsequenced__]];
```

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 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 of the function.

Additionally, we see that there is some markup capacity inside comments and that doxygen was able to identify the place in translation unit `"heron_k.c"` that defines the function and the call graph of the different functions involved in the implementation.

To provide good project organization, it is important that users of your code be able to easily find connected pieces and not have to search all over the place.

Takeaway 10.1 #2 *Structure your code in units that have strong semantic connections.*

Most often, this 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 this:

```
1  #ifndef BRIAN_H
2  #define BRIAN_H 1
3  #include <time.h>
4
5  /** @file
6   ** @brief Following Brian the Jay
7   **/
8
9  typedef struct brian brian;
10 enum chap { sct, en, };
11 typedef enum chap chap;
12
13 struct brian {
```

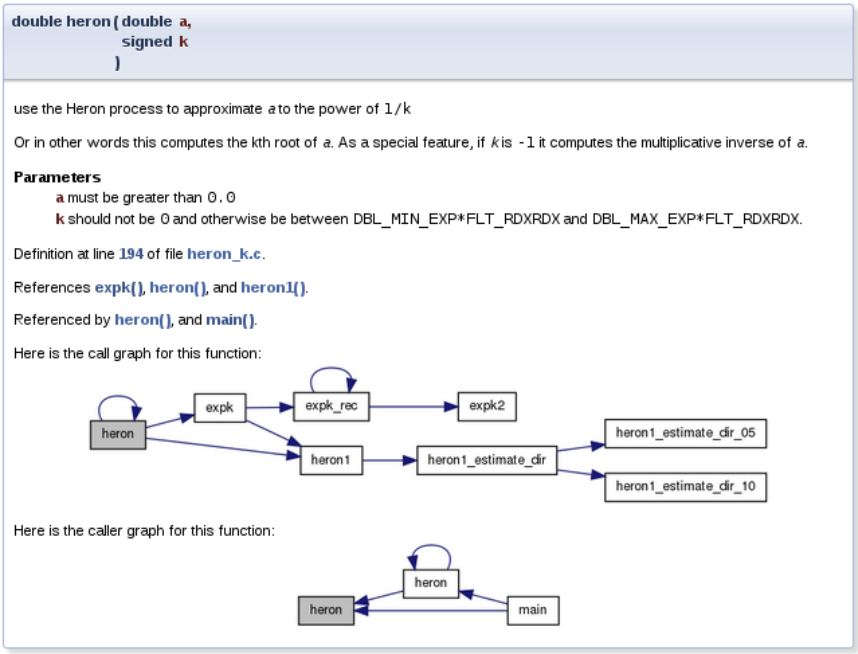


FIGURE 10.1. Documentation produced by doxygen

```
14    struct timespec ts; /**< point in time */
15    unsigned counter;   /**< wealth        */
16    chap masterof;      /**< occupation    */
17  };
18
19  /**
20   ** @brief get the data for the next point in time
21   **/
22  brian brian_next(brian);
23
24  ...
25  #endif
```

That file comprises all the interfaces that are necessary to use the **`struct`**. It also includes other header files that might be needed to compile these interfaces and protect against multiple inclusions 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 be quite readable, provided 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). The structure of the code shows what it does and how.

Takeaway 10.2 #1 *Implement literally.*

A C program is a descriptive text about what is to be done. The rules for naming entities that we introduced earlier play a crucial role in making that descriptive text readable and clear. Another requirement is to have an obvious flow of control through visually clearly distinctive structuring in compound statements (grouping with `{}`) that are linked together with comprehensive control statements.

Takeaway 10.2 #2 *Control flow must be obvious.*

There are many possible ways to obfuscate control flow. The most important are as follows:

Buried jumps: **`break`**, **`continue`**, **`return`**, and **`goto`**[6] statements that are buried in a complicated nested structure of **`if`** or **`switch`** statements, eventually combined with loop structures.

Flyspeck expressions: Controlling expressions that combine a lot of operators in an unusual way (for example, `!!++*p--` or `a -~> 0`) such that they must be examined with a magnifying glass to understand where the control flow goes from here.

In the following subsections, we will focus on several concepts that can be crucial for the readability and performance of C code:

- A *macro* can be a convenient tool to abbreviate a certain feature but, if used carelessly, may also obfuscate code that uses it and trigger subtle bugs (subsection 10.2.1).
- As we saw previously, functions are the primary choice in C for modularization. Here, a particular property of some functions is especially important; a function that is *pure* only interacts with the rest of the program via its interface. Therefore, pure functions are easily understandable by humans and compilers and generally lead to quite efficient implementations (subsection 10.2.2).
- We already have seen that attributes can be used to attach more information to our code. Section 10.2.3 discusses them in more detail.

10.2.1. *Macros.* We already know one tool that can be abused to obfuscate control flow: macros. As you hopefully remember from subsections 5.6.3 and 8.1.2, macros define textual replacements that can contain almost any C text. Because of the problems we will illustrate here, many projects ban macros completely. This is not the direction the evolution of the C standard goes, though. As we have seen, for example, type-generic macros are *the* modern interface to numerical functions (see 8.3). Macros should be used for initialization constants (5.6.3) or to implement compiler magic (**`errno`**; section 8.1.3).

So, instead of denying it, we should try to tame the beast and set up some simple rules that confine the possible damage.

Takeaway 10.2.1 #1 *Macros should not change the control flow in a surprising way.*

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

```
1  #define begin {
2  #define end }
3  #define forever for (;;)
4  #define ERRORCHECK(CODE) if (CODE) return -1
5
6  forever
7    begin
8    // do something
9    ERRORCHECK(x);
10   end
```

[6] These will be discussed in subsections 13.2.2 and 15.6.

Don't do that. The visual habits of C programmers and our tools don't easily work with something like that, and if you use such things in complicated code, they will almost certainly go wrong.

Here, the ERRORCHECK macro is particularly dangerous. Its name doesn't suggest that a nonlocal jump such as a **return** might be hidden in there. And its implementation is even more dangerous. Consider the following two lines:

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

These lines are rewritten as

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

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

```
1 if (a) {
2   if (x) return -1;
3   else puts("a␣is␣0!");
4 }
```

which is probably quite surprising to the casual reader.

This doesn't mean control structures shouldn't be used in macros at all. They just should not be hidden and should have no surprising effects. This macro by itself is probably not as obvious, but its *use* has no surprises:

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

The name of the following macro makes it explicit that there might be a **return**. The dangling **else** problem is handled by the replaced text:

```
1 if (a) ERROR_RETURN(x);
2 else puts("a␣is␣0!");
```

In contrast to all of the previous dangling **else**, the next example structures the code as expected, with the **else** associated with the first **if**:

```
1 if (a) do {
2   if (CODE) return -1;
3 } while (false);
4 else puts("a␣is␣0!");
```

The **do**–**while**(**false**)-trick is obviously ugly, and you shouldn't abuse it. But it is a standard trick to surround one or several statements with a { } compound statement without changing the program structure that is visible to the naked eye and without incurring the dangling **else** and similar problems.

Takeaway 10.2.1 #2 *Function-like macros should syntactically behave like function calls.*

Possible pitfalls are

if without else: Already demonstrated.

***Trailing semicolons*:** These can terminate an external control structure in a surprising way.

***Comma operators*:** The comma is an ambiguous character in C. In most contexts, it is used as a list separator, such as 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 nontrivial context.[Exs 7] 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 8]

10.2.2. *Pure functions.* Functions in C such as `size_min` (subsection 4.5) and `gcd` (subsection 7.3) that we declared ourselves are limited in terms of what we can express; they don't operate on objects but rather on values. In a sense, they are extensions of the value operators in table 4.1 and not of the object operators in table 4.2.

Takeaway 10.2.2 #1 *Function parameters are passed by value.*

That is, when we call a function, all parameters are evaluated, and the parameters (variables local to the function) receive the resulting values as initialization. The function then does whatever it has to do and 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) and are difficult to predict (the places of modification are scattered all over) and maintain.

Takeaway 10.2.2 #2 *Global variables are frowned upon.*

A function with the following two properties is called ***pure***^{C}:

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

The only interest in 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 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. For example,

- The function reads part of the program's changeable state by means other than through its arguments.
- The function modifies a global object.
- The function keeps a persistent internal state between calls.[9]
- The function does IO.[10]

[Exs 7] Consider a macro `sum(a, b)` that is implemented as `a+b`. What is the result of `sum(5, 2)*7`?

[Exs 8] Let `max(a, b)` be implemented as `((a) < (b) ? (b) : (a))`. What happens for `max(i++, 5)`?

[9] Persistent state between calls to the same function can be established with local `static` variables. We will see this concept in subsection 13.2.

[10] Such an IO would occur, for example, by using `printf`.

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 effect 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.

Takeaway 10.2.2 #3 *Express small tasks as pure functions whenever possible.*

With pure functions, we can go surprisingly far, even for an object-oriented programming style if, as a first approach, we are willing to accept a little bit of copying data around. Consider the following structure type `rat` that is supposed to be used for rational arithmetic:

rationals.h

```
8  struct rat {
9    bool sign;
10   size_t num;
11   size_t denom;
12 };
```

This is a direct implementation of such a type and nothing you should use as a library outside the scope of this learning experience. For simplicity, it has a numerator and denominator of identical type (`size_t`) and keeps track of the sign of the number in member `.sign`. A first (pure) function is `rat_get`, which takes two numbers and returns a rational number that represents their quotient.

rationals.c

```
5  rat rat_get(signed sign, size_t num, size_t denom) [[
       __unsequenced__]] {
6    rat ret = {
7      .sign = (sign < 0),
8      .num = num,
9      .denom = denom,
10   };
11   return ret;
12 }
```

As you can see, the function is quite simple. It just initializes a compound literal with the correct sign and numerator and denominator values. Notice that if we define a rational number this way, several representations will represent the same rational number. For example, the number $\frac{6}{15}$ is the same as $\frac{2}{5}$.

To deal with this equivalence in the representations, we need functions that do maintenance. The main idea is that such rational numbers should always be normalized; that is, use the representation such that the numerator and denominator have the fewest factors. Not only is this easier for humans to capture, but it also may avoid overflows while doing arithmetic. Here, the `gcd` function is as we described earlier.

rationals.c

```
14 rat rat_get_normal(rat x) [[__unsequenced__]] {
15   size_t c = gcd(x.num, x.denom);
16   x.num /= c;
17   x.denom /= c;
18   return x;
19 }
```

Another function does the inverse of normalization; it multiplies the numerator and denominator by a redundant factor:

. rationals.c

```
21 rat rat_get_extended(rat x, size_t f) [[__unsequenced__]] {
22   x.num *= f;
23   x.denom *= f;
24   return x;
25 }
```

This way, we may define functions that are supposed to be used by others: `rat_get_prod` and `rat_get_sum`. `rat_get_prod` first computes a representation of the result in a simple way— by just multiplying numerators and denominators, respectively. Then, the resulting representation might not be normalized, so we call `rat_get_normal` when we return the result.

. rationals.c

```
27 rat rat_get_prod(rat x, rat y) [[__unsequenced__]] {
28   rat ret = {
29     .sign = (x.sign != y.sign),
30     .num = x.num * y.num,
31     .denom = x.denom * y.denom,
32   };
33   return rat_get_normal(ret);
34 }
```

`rat_get_sum` is a bit more complicated. We have to find the common denominator before we can compute the numerator of the result and we have to keep track of the signs of the two rational numbers to see how we should add up the numerators.

. rationals.c

```
36 rat rat_get_sum(rat x, rat y) [[__unsequenced__]] {
37   size_t c = gcd(x.denom, y.denom);
38   size_t ax = y.denom/c;
39   size_t bx = x.denom/c;
40   x = rat_get_extended(x, ax);
41   y = rat_get_extended(y, bx);
42   assert(x.denom == y.denom);
43
44   if (x.sign == y.sign) {
45     x.num += y.num;
46   } else if (x.num > y.num) {
47     x.num -= y.num;
48   } else {
49     x.num = y.num - x.num;
50     x.sign = !x.sign;
51   }
52   return rat_get_normal(x);
53 }
```

As you can see, the fact that these are all pure functions ensures that they can be easily used, even in our own implementation here.[Exs 11][Exs 12]The only thing we have to watch is to always assign the return values of our functions to a variable, such as the assignment to `x` on line 40. Otherwise, since we don't operate on the object `x` but only on its value, changes during the function would be lost.[13]

As mentioned earlier, because of the repeated copies, this may result in compiled code that is not as efficient as it could be. But this is not dramatic at all: the overhead from the copy operation can be kept relatively low by good compilers. With optimization switched on, they usually can operate directly on the structure in place, as it is returned from such a function. Then such worries might be completely premature because your program is short and sweet anyway or because its real performance problems lay elsewhere. Usually, this should be completely sufficient for the level of programming skills that we have reached so far. Later, we will learn how to use that strategy efficiently by using the **`inline`** functions (subsection 16.1) and *link-time optimization* that many modern tool chains provide.

Listing 10.1 lists all the interfaces of the `rat` type that we have seen so far (first group). We have already looked at the interfaces to other functions that work on *pointers* to `rat`. These will be explained in more detail in subsection 11.2.

Listing 10.1. A type for computation with rational numbers.

```
1  #ifndef RATIONALS_H
2  # define RATIONALS_H 1
3  # include <stdbool.h>
4  # include "euclid.h"
5
6  typedef struct rat rat;
7
8  struct rat {
9    bool sign;
10   size_t num;
11   size_t denom;
12 };
13
14 /* Functions that return a value of type rat. */
15 rat rat_get(signed sign, size_t num, size_t denom) [[
       __unsequenced__]];
16 rat rat_get_normal(rat x) [[__unsequenced__]];
17 rat rat_get_extended(rat x, size_t f) [[__unsequenced__]];
18 rat rat_get_prod(rat x, rat y) [[__unsequenced__]];
19 rat rat_get_sum(rat x, rat y) [[__unsequenced__]];
20
21
22 /* Functions that operate on pointers to rat. */
23 void rat_destroy(rat* rp) [[__unsequenced__]];
24 rat* rat_init(rat* rp,
25               signed sign,
26               size_t num, size_t denom) [[__unsequenced__]];
27 rat* rat_normalize(rat* rp) [[__unsequenced__]];
```

[Exs 11]The function `rat_get_prod` can produce intermediate values that may cause it to produce wrong results, even if the mathematical result of the multiplication is representable in `rat`. How is that possible?

[Exs 12]Reimplement the `rat_get_prod` function so that it produces a correct result every time the mathematical result value is representable in a `rat`. This can be done with two calls to `rat_get_normal` instead of one.

[13]C23 now provides the `[[nodiscard]]` attribute that suggests that the compiler should issue a warning if we do not use the return value.

```
28 rat* rat_extend(rat* rp, size_t f) [[__unsequenced__]];
29 rat* rat_sumup(rat* rp, rat y) [[__unsequenced__]];
30 rat* rat_rma(rat* rp, rat x, rat y) [[__unsequenced__]];
31
32 /* Functions that are implemented as exercises. */
33 /** @brief Print @a x into @a tmp and return tmp. **/
34 char const* rat_print(size_t len, char tmp[len], rat const* x);
35 /** @brief Print @a x normalize and print. **/
36 char const* rat_normalize_print(size_t len, char tmp[len],
37                                 rat const* x);
38 rat* rat_dotproduct(rat rp[static 1], size_t n,
39                     rat const A[n], rat const B[n]);
40
41 #endif
```

10.2.3. *Attributes.* Attributes recently appeared in C23. They are designed to play an important role as an annotation tool and are meant to facilitate the development of new features by compiler and tool implementors. For example, the `[[__unsequenced__]]` attributes in listing 10.1 indicates that most functions there have the unsequenced property, which is a generalization of the property of a function to be pure. Now when application code is compiled that only sees this header file, the compiler still can make strong assumptions about the function. If it has no pointer parameters or pointer return, the function is pure and the return value only depends on the argument values that are passed into a call. If it has pointer parameters or pointer returns (such as `rat_normalize`; see the following discussion), the possible interference with a function call is limited to the objects that are visible through these pointers.
C23 introduces a handful of *standard attributes* together with macro-safer variants that have double underscores such as `__unsequenced__`. They are

deprecated fallthrough maybe_unused nodiscard
noreturn unsequenced reproducible

We have already seen most of them in action. Currently, only two, `[[deprecated]]` and `[[nodiscard]]`, may receive an argument in the form of a string:

```
[[deprecated("tell␣me␣all␣about␣it")]].
```

The meanings of these attributes are as follows:

- The `[[deprecated]]` attribute indicates that the feature to which it is attached is not meant to be used directly. Reasons for that can be multiple, but the main reason is probably that the feature is obsolete (and might be removed) or that it is not part of a public interface. The effect of using a feature with such an attribute is usually that the compiler issues a diagnostic. But contrary to similar implementation-defined attributes, a deprecated feature should not be diagnosed when it is used within another feature that is also deprecated. Section 13.1.1 has a complete example that highlights that property of `[[deprecated]]`.
- As we have seen in section 3.3, **`switch`** statements may have a very complicated control flow. To avoid errors, many coding styles try to restrict that control flow so falling from one **`case`** into the other without a **`break`** is considered erroneous. The `[[fallthrough]]` attribute indicates that such a possible control flow is intentional and that no diagnosis should be issued.

- At some places we may have declarations or definitions of identifiers that will not necessarily be used in the sequel. Examples include names of unused parameters that are given for documentation purposes or **static** functions in a header file that are not always used by code that includes the header. The [[**maybe_unused**]] avoids diagnostics in such cases.
- The main use of the [[**nodiscard**]] attribute is to indicate that a function has a return value that is important and that should always be taken into account. This is particularly important for the allocation of storage; section 13.1.1 also shows an example of this usage.
- As we have seen in section 8.8, the [[**noreturn**]] attribute is also associated to a function. It indicates that the function will *never* return to the caller and, consequently, the compiler may optimize the surrounding code.
- The [[**unsequenced**]] and [[**reproducible**]] attributes will be discussed in more detail in section 16.3.

In addition to the standard attributes, there are *prefixed attributes*. Their names have the form

```
prefix::suffix
```

where prefix is an identifier that is usually chosen by a compiler or tool implementor, :: is a new syntax token that in C appears nowhere else, and suffix is an identifier that indicates the particular feature or property. Current prefixes I know of are for the three compiler families clang, gnu and msvc. But maybe surprisingly, this is not a strict boundary; for example, the clang compiler implements many attributes from the gnu compilers and keeps the gnu prefix there.

In general, such prefixed attributes may also receive arguments; the only syntactic constraint is that the argument has to contain correctly nested balanced pairs of (), [], and {} parenthesis. Obviously, the implementation is free to reject anything it does not understand, but that is the only restriction from the point of view of the standard. For example, gcc and clang support the format attribute.

```
#if __has_c_attribute(__gnu__::__format__)
  [[__gnu__::__format__(__printf__, 3, 4)]]
#endif
  int snprintf(char *buf, size_t size, const char *frmt, ...);
```

Here, this attribute indicates that **snprintf** processes a format specification à la **printf** in position 3 and that the variable argument list starts at position 4. With such information, the compiler then can issue a warning on formats that are not string literals or on format specifiers that receive the wrong types of arguments.

With the attribute feature, C23 also provides a test feature **__has_c_attribute** for attributes. This feature can be used in preprocessor conditionals, similar to **defined** for macros. In the previous example, the platform-specific attribute __gnu__::__format__ is only used if it is supported.

To be sure that the test feature even exists, it can also be queried. Our fallback header <c23-fallback.h> has the following to accommodate platforms that might not yet implement this feature. If a compiler does not have the feature **__has_c_attribute**, it is very unlikely it will implement attributes at all.

Beware that neither the standard attributes nor identifiers in prefixed attributes are keywords. Therefore, they can and will interact with application macros, possibly with disastrous consequences.

Takeaway 10.2.3 #1 *Identifiers in attributes can be replaced by preprocessing.*

. c23-fallback.h

```
#ifndef __has_c_attribute
# define __has_c_attribute(X) 0
#endif
```

Therefore, C23 foresees that all its standard attributes additionally have a form that is prefixed and postfixed with double underscores and it recommends that all prefixed attributes additionally have such a modified form, too. Because identifiers with double underscores are reserved, application-defined macros are guaranteed not to interact with these.

Takeaway 10.2.3 #2 *Use the double underscore forms of attributes in header files.*

Summary

- For each part of a program, we have to distinguish the object (what are we doing?), the purpose (what are we doing it for?), the method (how are we doing it?), and the implementation (in which manner are we doing it?).
- The function and type interfaces are the essence of software design. Changing them later is expensive.
- An implementation should be as literal and obvious in its control flow as possible. Complicated reasoning should be avoided and made explicit where necessary.
- Attributes can add valuable information to the interface and the implementation that may improve diagnostic, safety, security, and performance.

11. Pointers

This section covers

- Introduction to pointer operations
- Using pointers with structs, arrays, and functions

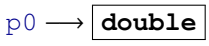
Pointers are the first real hurdle to a deeper understanding of C. They are used in contexts where we have to be able to access objects from different points in the code or where data is structured dynamically on the fly.

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 abstract 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 because it is crucial for the understanding of most of the rest of this book.

The term ***pointer***^{C} stands for a specially derived type construct that "points" or "refers" to something. We have seen the syntax for this construct, a type (the ***referenced type***^{C}) that is followed by a `*` character. For example, `p0` is a pointer to a **`double`**:

```
double* p0;
```

The idea is that we have one variable (the pointer) that points to the memory of another object:



An important distinction that we will have to make throughout this section is between the pointer (to the left of the arrow) and the unnamed object that is pointed to (to the right).

Our first usage of a pointer will be to break the barrier between the code of the caller of a function and the code inside a function, allowing us to write functions that are *not* pure. This example will be a function with this prototype:

```
void double_swap(double* p0, double* p1);
```

Here, we see two function arguments that "point" to objects of type **`double`**. In the example, the function `double_swap` is supposed to interchange (*swap*) the contents of these two objects. For example, when the function is called, `p0` and `p1` could be pointing to **`double`** variables `d0` and `d1`, respectively, that are defined by the caller:

By receiving information about two such objects, the function `double_swap` can effectively change the contents of the two **`double`** objects without changing the pointers themselves:

Using pointers, the function will be able to apply the change directly to the variables of the calling function; a pure function without pointers or arrays would not be able to do this.

In this section, we will go into the details of different operations with pointers (subsection 11.1) and specific types for which pointers have particular properties: structures (subsection 11.2), arrays (subsection 11.3), and functions (subsection 11.4).

11.1. Pointer operations. Pointers are an important concept, so there are several C language operations and features just for them. Most importantly, specific operators allow us to deal with the "pointing-to" and "pointed-to" relation between pointers and the objects to which they point (subsection 11.1.1). Also, pointers are considered ***scalars***^{C}: arithmetic operations are defined for them, offset additions (subsection 11.1.2) and subtractions (subsection 11.1.3). They have state (subsection 11.1.4) and a dedicated "null" state (subsection 11.1.5).

11.1.1. *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 variables of the function. Pointers are a suitable abstraction to do this.

So, let us use the function `double_swap` from earlier to swap the contents of two **`double`** objects `d0` and `d1`. For the call, we use the unary ***address-of***^{C} operator *"`&`"*. It allows us to refer to an object through its ***address***^{C}. A call to our function could look like this:

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

The type that the address-of operator returns is a ***pointer type***^{C} and can be specified with the `*` notation that we have seen. An implementation of the function could look like this: Inside the function, pointers `p0` and `p1` hold the addresses of the objects on

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

which the function is supposed to operate (in our example, the addresses of `d0` and `d1`). But the function knows nothing about the names of the two variables `d0` and `d1`; it only knows `p0` and `p1`.

To access them, another construct that is the inverse of the address-of operator is used: the unary ***object-of***^{C} operator *"`*`"*: `*p0` then is the object corresponding to the first argument. With the previous call, that would be `d0`, and similarly `*p1` is the object `d1`.[Exs 14]

Please note that the `*` character plays two different roles in the definition of `double_swap`. In a declaration, it creates a new type (a pointer type), whereas in an expression, it ***dereferences***^{C} the object to which a pointer ***refers***^{C}. To help distinguish these two usages of the same symbol, we usually flush the `*` to the left with no blanks in between if it modifies a type, such as in

```
double* p0;
```

and to the right if it dereferences a pointer in an expression, such as in

```
*p0 = *p1;
```

[Exs 14] Write a function that receives pointers to three objects and that shifts the values of these objects cyclically.

Remember from subsection 6.2 that in addition to holding a valid address, pointers may also be null or invalid.

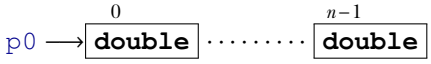
Takeaway 11.1.1 #1 *A program execution that uses $\star$ with an invalid or null pointer fails.*

In practice, though, both cases will usually behave differently. The first might access a random object in memory and modify it. Often this leads to bugs that are difficult to trace because it will poke into objects it is not supposed to. The second, if the pointer is null, will manifest early during development and nicely crash our program. Consider this to be a feature.

11.1.2. *Pointer addition.* We already have seen that a valid pointer holds the address of an object of its reference type, but actually C assumes more than that

Takeaway 11.1.2 #1 *A valid pointer refers to the first element of an array of the reference type.*

In other words, a pointer may be used to refer not only to one instance of the reference type but also to an array of an unknown length n.



This entanglement between the concept of pointers and arrays is taken an important step further in the syntax. In fact, for the specification of the function `double_swap`, we wouldn't even need the pointer notation. In the notation we have used so far, it can equally be written as

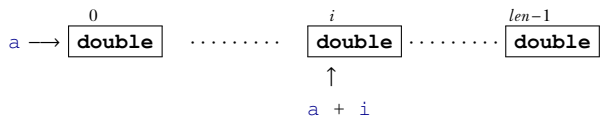
```
void double_swap(double p0[static 1], double p1[static 1]) {
  auto tmp = p0[0];
  p0[0] = p1[0];
  p1[0] = tmp;
}
```

Both the use of array notation for the interface and the use of `[0]` to access the first element are simple ***rewrite operations***^{C} that are built into the C language. We will see more of this later.

Simple additive arithmetic allows us to access the following elements of this array. This function sums all elements of an array:

```
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;
}
```

Here, the expression `a+i` is a pointer that points to the i^{th} element in the array:

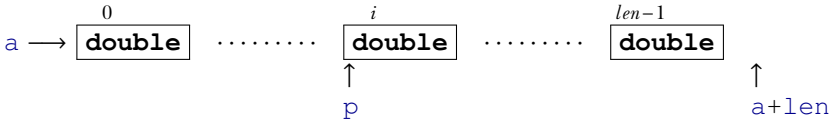


Pointer addition can be done in different ways, so the following functions sum up the array in exactly the same order:

```
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;
}
```

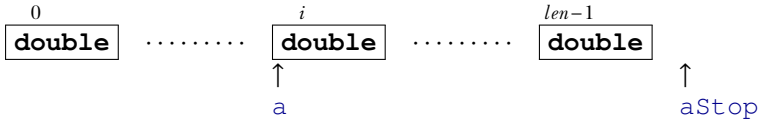
```
double sum2(size_t len, double const* a) {
  double ret = 0.0;
  for (double const*const aStop = a+len; a < aStop; ++a) {
    ret += *a;
  }
  return ret;
}
```

In iteration i of function `sum1`, we have the following picture:



The pointer `p` walks through the elements of the array until it is greater than or equal to `a+len`, the first pointer value that lies beyond the array.

For function `sum2`, we have the following picture:



Here, `a` refers to the i^{th} element of the array. The 0^{th} element is not referenced again inside the function, but the information about the end of the array is kept in the variable `aStop`.

These functions can then be called analogously to the following:

```
double s0_7 = sum0(7, &A[0]);    // For the whole
double s1_6 = sum0(6, &A[1]);    // For the last 6
double s2_3 = sum0(3, &A[2]);    // For the 3 in the middle
```

Unfortunately, there is no way to know the length of the array hidden behind a pointer, so we have to pass it as a parameter into the function. The trick with **`sizeof`**, which we saw in subsection 6.1.3, doesn't work.

Takeaway 11.1.2 #2 *The length of an array object cannot be reconstructed from a pointer.*

So, here, we see a first important difference from arrays.

Takeaway 11.1.2 #3 *Pointers are not arrays.*

If we pass arrays through pointers to a function, it is important to retain the real length of the array. This is why we prefer the array notation for pointer interfaces throughout this book:

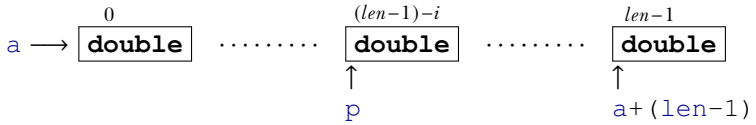
```
double sum0(size_t len, double const a[len]);
double sum1(size_t len, double const a[len]);
double sum2(size_t len, double const a[len]);
```

These specify exactly the same interfaces as shown earlier, but to the casual reader of the code, they clarify that `a` is expected to have `len` elements.

11.1.3. *Pointer subtraction and difference.* The pointer arithmetic we have discussed so far concerned addition of an integer and a pointer. There is also an inverse operation that can subtract an integer from a pointer. If we wanted to visit the elements of the array downward, we could use this:

```
double sum3(size_t len, double const* a) {
  double ret = 0.0;
  double const* p = a+len-1;
  do {
    ret += *p;
    --p;
  } while (p > a);
  return ret;
}
```

Here, `p` starts out at `a+(len-1)`, and in the i^{th} iteration, the picture is



Note that the summation order in this function is inverted.[15]

There is also an operation, ***pointer difference***^{C}, that takes two pointers and computes their distance counted in number of elements. To see that, we extend `sum3` to a new version that checks for an error condition (one of the array elements being an infinity). In that case, we want to print a comprehensive error message and return the culprit to the caller:[16]

```
double sum4(size_t len, double const* a) {
  double ret = 0.0;
  double const* p = a+len-1;
  do {
    if (isinf(*p)) {
      fprintf(stderr,
              "element␣\%tu␣of␣array␣at␣\%p␣is␣infinite\n",
              p-a,           // Pointer difference!
              (void*)a);     // Prints the pointer value
      return *p;
    }
    ret += *p;
    --p;
  } while (p > a);
  return ret;
}
```

Here, we use the expression `p-a` to compute the position of the actual element in the array.

This is only defined if the two pointers refer to elements of the same array object. If they are not, the program is erroneous and bad things may happen.

Takeaway 11.1.3 #1 *Only subtract pointers to elements of the same array object.*

The value of such a difference then is simply the difference of the indices of the corresponding array elements:

[15] Because of differences in rounding, the result might be slightly different than for the first three functions in this series.

[16] **isinf** comes from the `<math.h>` header.

```
double A[4] = { 0.0, 1.0, 2.0, -3.0, };
[[maybe_unused]] double* p = &A[1];
[[maybe_unused]] double* q = &A[3];
assert(p-q == -2);
```

We 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**.[17] 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 is just the signed integer type that corresponds to **size_t**, but we shouldn't care much.

<stddef.h>

Takeaway 11.1.3 #2 *All pointer differences have type* **ptrdiff_t**.

Takeaway 11.1.3 #3 *Use* **ptrdiff_t** *to encode signed differences of positions or sizes.*

Function sum4 also shows a recipe to print a pointer value for debugging purposes. We use the format character %p, and the pointer argument is *cast* by (**void***) a to the obscure type **void***. For the moment, take this recipe as a given; we do not yet have all the baggage to understand it in full (more details will follow in subsection 12.4).

Takeaway 11.1.3 #4 *For printing, cast pointer values to* **void*** *and use the format* %p.

11.1.4. *Pointer validity.* Earlier (takeaway 11.1.1 #1), we saw that we must be careful about the address that a pointer contains (or does not contain). Pointers have a value, the address they contain, and that value can change.

Setting a pointer to null if it does not have a valid address is very important and should not be forgotten. It helps to check and keep track of whether a pointer has been set.

Takeaway 11.1.4 #1 *Pointers have a truth value.*

To avoid clunky comparisons (takeaway 3.1 #3), in C programs, you often will see code like this:

```
char const* name = nullptr;

// Do something that eventually sets name

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

Therefore, it is important to control the state of all pointer variables. We have to ensure that pointer variables are always null, unless they point to a valid object that we want to manipulate.

Takeaway 11.1.4 #2 *Set pointer variables to null as soon as you can.*

[17] The attribute [[maybe_unused]] avoids spurious warnings if the compilation has **NDEBUG** defined and p and q are otherwise unused.

In most cases, the simplest way to ensure this is to initialize pointer variables explicitly (takeaway 6.2 #6).

We have seen some examples of *representations* of different types—that is, the way the platform stores the value of a particular type in an object. The representation of one type (say, **size_t**) could be completely senseless to another type (for example, **double**). As long as we only use variables directly, C's type system will protect us from any mixup of these representations; a **size_t** object will always be accessed as such and never be interpreted as a (senseless) **double**.

If we do not use them carefully, pointers can break that barrier and lead us to code that tries to interpret the representation of a **size_t** as **double**. More generally, C even has coined a term for bit patterns that are nonsense when they are interpreted as a specific type: a ***non-value representation***^{C} for that type.[18]

Takeaway 11.1.4 #3 *A program execution that accesses an object that has a non-value representation for its type fails.*

Ugly things can happen if you do, so please don't try.

Thus, not only must a pointer be set to an object (or null), but such an object also must have the correct type.

Takeaway 11.1.4 #4 *When dereferenced, a pointed-to object must be of the designated 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
                            // Program failure
```

On 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 even if p is valid at that point, accessing the contents as a type of **double** makes no sense, and C generally forbids such access.

In the previous example, the pointer addition itself is correct, as long as we don't access the object on 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 addition as in the example wouldn't work reliably.

Takeaway 11.1.4 #5 *A pointer must point to a valid object, one position beyond, 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:

```
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
                            // Program failure
```

[18] Prior to C23, the term that was used was ***trap representation***^{C}.

However, this last example may crash at the increment operation:

```
double A[2] = { 0.0, 1.0, };
double* p = &A[0];
printf("element␣%g\n", *p); // Referencing object
p += 3;                     // Invalid pointer addition
                            // Program failure
```

Takeaway 11.1.4 #6 *A program execution that computes a pointer value outside the bounds of an array object (or one element beyond) fails.*

11.1.5. *Null pointers.* You may have wondered why, in all this discussion about pointers, the macro **NULL** has not yet been used. The reason is that, unfortunately, the original concept in pre-C23 of *null pointer constants*, given as a "generic pointer of value 0," didn't succeed very well.

C has the concept of a ***null pointer***^{C} that corresponds to a null value of any pointer type.[19] Here,

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

`nix` and `nax` would be pointer objects of value null. But unfortunately, a ***null pointer constant***^{C} is then not what you'd expect.

First, here, the term *constant* refers to a compile-time constant, not to a ***const***-*qualified* object. For that reason, both pointer objects *are not* null pointer constants. Second, the permissible type for these constants in pre-C23 was quite peculiar: it may be any constant expression of integer type and value `0` or of type **void***. Other pointer types are not permitted, and we will learn about pointers of that "type" in subsection 12.4.

The definition in the C standard of a possible expansion of the macro **NULL** is quite loose; it just has to be a null pointer constant. Therefore, a pre-C23 compiler could choose 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`.[20]

It is important to note that the type behind **NULL** is not prescribed by the C standard. Often, people use it to emphasize that they are talking about a pointer constant, which it simply isn't on many platforms. Using **NULL** in a context that we have not mastered completely is even dangerous. This will in particular appear in the context of functions with a variable number of arguments, which will be discussed in subsection 17.4.2. For the moment, we will go for the simplest solution:

Takeaway 11.1.5 #1 *Use* ***nullptr*** *instead of* ***NULL***.

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

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

NULL hides more than it clarifies, and because C compilers are always trying to be backward compatible, you will never be sure what you get.

11.2. Pointers and structures. 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. For example, let us 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 objects directly:

timespec.c

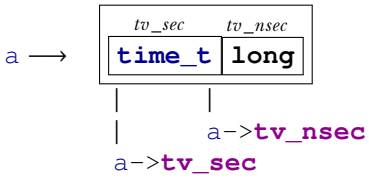
`timespec_diff`:

compute a time difference

This uses a **double** to compute the time. If we want to be able to track times without further loss of precision and have **double** with 52 bit mantissa, this corresponds to a maximal time difference of about 4.5E6 seconds, or 52 days.

```
double timespec_diff(struct timespec const* later,
                     struct timespec const* sooner){
  /* Be careful: tv_sec could be an unsigned type */
  if (later->tv_sec < sooner->tv_sec)
    return -timespec_diff(sooner, later);
  else
    return
      (later->tv_sec - sooner->tv_sec)
      /* tv_nsec is known to be a signed type. */
      + (later->tv_nsec - sooner->tv_nsec) * 1E-9;
}
```

For convenience, we will use a new operator, `->`. Its arrow-like symbol is meant to represent a pointer as the left operand that "points" to a member of the underlying **struct** as the right operand. It is equivalent to a combination of `*` and `.`. To have the same effect, we would have to use parentheses and write `(*a).tv_sec` instead of `a->tv_sec`. This could quickly become a bit clumsy, so the `->` operator is what everybody uses.



Observe that a construct like `a->tv_nsec` is *not* a pointer, but an object of type **long**, the number itself.

As another example, let us again consider the type `rat` for rational numbers that we introduced in subsection 10.2.2. The functions operating on pointers to that type in listing 10.1 could be written as follows:

.

rationals.c

```
97 void rat_destroy(rat* rp) [[__unsequenced__]] {
98   if (rp) *rp = (rat){ };
99 }
```

The function `rat_destroy` ensures that all data that might be present in the object is erased and set to all-bits `0`:

. rationals.c

```
rat* rat_init(rat* rp,
              signed sign,
              size_t num,
              size_t denom) [[__unsequenced__]] {
  if (rp) *rp = rat_get(sign, num, denom);
  return rp;
}
```

. rationals.c

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

. rationals.c

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

The other three functions are simple ***wrappers***^{C} around the pure functions that we already know. We use two pointer operations 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.[Exs 21][Exs 22]

All four functions check and return their pointer argument. This is a convenient strategy to compose such functions, as we can see in the definitions of the following two arithmetic functions:

. rationals.c

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

The function `rat_rma` ("rational multiply add") comprehensively shows its purpose: to add the product of the two other function arguments to the object referred to by `rp`. It uses the following function for the addition:

. rationals.c

```
rat* rat_sumup(rat* rp, rat y) [[__unsequenced__]] {
  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);
```

[Exs 21] Implement function `rat_print` as declared in listing 10.1. This function should use `->` to access the members of its `rat*` argument. The printout should have the form $\pm nom/denum$.

[Exs 22] Implement `rat_print_normalized` by combining `rat_normalize` and `rat_print`.

```
126
127   if (rp->sign == y.sign) {
128     rp->num += y.num;
129   } else if (rp->num > y.num) {
130     rp->num -= y.num;
131   } else {
132     rp->num = y.num - rp->num;
133     rp->sign = !rp->sign;
134   }
135   return rat_normalize(rp);
136 }
```

The function `rat_sumup` is a more complicated example, where we apply two maintenance functions to the pointer arguments.[Exs 23]

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 a library and its implementation. For example, a fictive type `toto` could 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);
```

Neither the programmer nor the compiler would 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 translation unit that defines the functions. And the compiler gets away with it because it 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 previous example, `toto_doit(nullptr, 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 type toto hiding a pointer */
typedef struct toto_s* toto;
toto toto_get(void);
void toto_destroy(toto);
void toto_doit(toto, unsigned);
```

This is valid C, but it hides the fact that **`nullptr`** is a special value that `toto_doit` may receive.

Takeaway 11.2 #1 *Don't hide pointer types inside 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);
```

[Exs 23] Implement the function `rat_dotproduct` from listing 10.1 such that it computes $\sum_{i=0}^{n-1} A[i] * B[i]$ and returns that value in `*rp`.

Here, the fact that the interface receive a pointer is still sufficiently visible.

CHALLENGE 12 (text processor). *For a text processor, can you use a doubly linked list to store text? The idea is to represent a "blob" of text through a* `struct` *that contains a string (for the text) and pointers to preceding and following blobs.*
Can you build a function that splits a text blob in two at a given point?
One that joins two consecutive text blobs?
One that runs through the entire text and puts it in the form of one blob per line?
Can you create a function that prints the entire text or prints until the text is cut off due to the screen size?

11.3. Pointers and arrays. We are now able to attack the major hurdles to understanding the relationship between arrays and pointers: the fact that C uses the same syntax for pointer and array element access *and* that it rewrites array parameters of functions to pointers. Both features provide convenient shortcuts for the experienced C programmer but also are a bit difficult for novices to digest.

11.3.1. *Array and pointer access are the same.* The following statement holds regardless of whether `A` is an array or a pointer.

Takeaway 11.3.1 #1 *The two expressions `A[i]` and `*(A+i)` are equivalent.*

If it is a pointer, we understand the second expression. Here, it just says that we may write the same expression as `A[i]`. Applying this notion of array access to pointers should improve the readability of your code. The equivalence does not mean that, suddenly, an array object appears where there was none. If `A` is null, `A[i]` should crash 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].

Takeaway 11.3.1 #2 (array decay) *Evaluation of an array `A` returns `&A[0]`.*

In fact, this is the reason there are no "array values" and all the difficulties they entail (takeaway 6.1.2 #2). Whenever an array occurs that requires a value, it decays to a pointer, and we lose all additional information.

11.3.2. *Array and pointer parameters are the same.* Because of the decay, arrays cannot 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 they work? The trick C gets away with is to rewrite array parameters to pointers.

Takeaway 11.3.2 #1 *In a function declaration, any array parameter rewrites to a pointer.*

Think of this and what it means for a while. Understanding this "chief feature" (or character flaw) is central for coding easily in C.

To come back to our examples from subsection 6.1.5, the functions that were written with array parameters could be declared as follows:

```
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 habit, culture, or other social contexts. The rule that we follow in this book is to use array notation if we suppose it can't be null and pointer notation if it corresponds to a single item of the base type, which can also 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 possible. And to make it possible, it is usually better to specify the length before the arrays/pointers. An interface, such as

tells a whole story. This becomes even more interesting if we handle two-dimensional arrays. A typical matrix multiplication could look as follows:

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

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

and

```
void matrix_mult(size_t n, size_t k, size_t m,
                 double       (*C)[m],
                 double const (*A)[k],
                 double const (*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 an array*. So, there is no need to rewrite the subsequent dimensions.

Takeaway 11.3.2 #2 *Only the innermost dimension of an array parameter is rewritten.*

Finally, we have gained a lot by using array notation. We have, without any problems, passed pointers to VLAs into the function. Inside the function, we can use conventional indexing to access the elements of the matrices. Not much in the way of acrobatics is required to keep track of the array lengths.

Takeaway 11.3.2 #3 *Declare length parameters before array parameters.*

They simply have to be known at the point where you first use them.

Unfortunately, C generally gives no guarantee that a function with array-length parameters is always called correctly.

Takeaway 11.3.2 #4 *The validity of array arguments to functions must be guaranteed by the programmer.*

If the array lengths are known at compile time, compilers may be able to issue warnings, though. But when array lengths are dynamic, you are mostly on your own: be careful.

Note also that in the previously discussed prototypes, we have matrices `A` and `B` where the base type is **`const`**-qualified. Doing this consistently is only possible since

C23, which changed the rules for this for the benefit of usability. Before C23, the following code would produce an error because the target types of the parameters `A` and `B` are qualified differently than the arguments:

```
  double matA[2][2] = { { 0, 0,}, { 0, 0,}, };
  double matB[2][2] = { { 0, 0,}, { 0, 0,}, };
  double matC[2][2] = { { 0, 0,}, { 0, 0,}, };
  matrix_mult(2, 2, 2, matC, matA, matB);
```

Previously, the rules had been the same as if we had used a function that uses a pointer-to-pointer instead of a pointer-to-array:

```
void fake_mult(size_t n, size_t k, size_t m,
               double      ** C,
               double const** A,
               double const** B);
double** fakeA = (double*[2]){
  (double[2]){ 0, 0, },
  (double[2]){ 0, 0, }, };
double** fakeB = (double*[2]){
  (double[2]){ 0, 0, },
  (double[2]){ 0, 0, }, };
double** fakeD = (double*[2]){
  (double[2]){ 0, 0, },
  (double[2]){ 0, 0, }, };
// error, *fakeA and *A not compatible
fake_mult(2, 2, 2, fakeC, fakeA, fakeB);
```

11.4. Function pointers. There is yet another construct for which the address-of operator `&` can be used: functions. We saw this concept pop up when discussing the **`atexit`** function (section 8.8), which is a function that receives a function argument. The rule is similar to that for array decay, which we described earlier:

Takeaway 11.4 #1 (function decay) *A function name without following parenthesis decays to a pointer to its start.*

Syntactically, functions and function pointers are also similar to arrays in type declarations and as function parameters:

```
  typedef void atexit_function(void);
  // Two equivalent definitions of the same type, which hides a pointer
  typedef atexit_function* atexit_function_pointer;
  typedef void (*atexit_function_pointer)(void);
  // Five equivalent declarations for the same function
  void atexit(void f(void));
  void atexit(void (*f)(void));
  void atexit(atexit_function f);
  void atexit(atexit_function* f);
  void atexit(atexit_function_pointer f);
```

Which of the semantically equivalent ways of writing the function declaration is more readable could certainly be the subject of much debate. The second version, with the `(*f)` parentheses, quickly gets difficult to read, and the fifth is frowned upon because it hides a pointer in a type. Among the others, I personally slightly prefer the fourth over the first.

The C library has several functions that receive function parameters. We have seen **`atexit`** and **`at_quick_exit`**. Another pair of functions in `<stdlib.h>` provides generic interfaces for searching (**`bsearch`**) and sorting (**`qsort`**): `<stdlib.h>`

```
typedef int compare_function(void const*, void const*);

void* bsearch(void const* key, void const* base,
              size_t n, size_t size,
              compare_function* compar);

void qsort(void* base,
           size_t n, size_t size,
           compare_function* compar);
```

Both receive an array `base` as an argument on which they perform their task. The address to the first element is passed as a **`void`** pointer, so all type information is lost. To be able to handle the array properly, the functions have to know the size of the individual elements (`size`) and the number of elements (`n`).

In addition, they receive a comparison function as a parameter that provides the information about the sort order between the elements. By using such a function pointer, the **`bsearch`** and **`qsort`** functions are very generic and can be used with any data model that allows for an ordering of values. The elements referred by the `base` parameter can be of any type `T` (**`int`**, **`double`**, string, or application defined) as long as the `size` parameter correctly describes the size of `T` and as long as the function pointed to by `compar` knows how to compare values of type `T` consistently.

A simple version of such a function would look like this:

```
int compare_unsigned(void const* a, void const* b){
  unsigned const* A = a;
  unsigned const* B = b;
  if (*A < *B) return -1;
  else if (*A > *B) return +1;
  else return 0;
}
```

The convention is that the two arguments point to elements that are to be compared, and the return value is strictly negative if `a` is considered less than `b`, `0` if they are equal, and strictly positive otherwise.

The return type of **`int`** seems to suggest that **`int`** comparison could be done more simply:

```
/* An invalid example for integer comparison */
int compare_int(void const* a, void const* b){
  int const* A = a;
  int const* B = b;
  return *A - *B;    // may overflow!
}
```

But this is not correct. For example, if `*A` is big, say **`INT_MAX`**, and `*B` is negative, the mathematical value of the difference can be larger than **`INT_MAX`**.

Because of the **`void`** pointers, a usage of this mechanism should always take care that the type conversions are encapsulated similar to the following:

```
/* A header that provides searching and sorting for unsigned. */

/* No use of inline here; we always use the function pointer. */
extern int compare_unsigned(void const*, void const*);

inline
unsigned const* bsearch_unsigned(unsigned const key[static 1],
                                 size_t nmeb, unsigned const base[nmeb]) {
   return bsearch(key, base, nmeb, sizeof base[0], compare_unsigned);
}
```

```
inline
void qsort_unsigned(size_t nmeb, unsigned base[nmeb]) {
   qsort(base, nmeb, sizeof base[0], compare_unsigned);
}
```

Here, **bsearch** (binary search) searches for an element that compares equal to `key[0]` and returns it or returns a null pointer if no such element is found. It supposes that array `base` is already sorted consistently to the ordering that is given by the comparison function. This assumption helps speed up the search. Although this is not explicitly specified in the C standard, you can expect that a call to **bsearch** will never make more than $\lceil \log_2(n) \rceil$ calls to `compar`.

If **bsearch** finds an array element that is equal to `*key`, it returns the pointer to this element. Note that this, if used like that, drills a hole in C's type system, since this returns an unqualified pointer to an element whose effective type might be **const**-qualified. Use with care. In our example, we simply convert the return value to **unsigned const***, such that we will never even see an unqualified pointer at the call side of `bsearch_unsigned`. Since C23 **bsearch** has in fact been upgraded to a type-generic macro **bsearch** of the same name, see 18.1.7, which circumvents this flaw.

The name **qsort** is derived from the *quick sort* algorithm. The standard doesn't impose the choice of the sorting algorithm, but the expected number of comparison calls should be of the magnitude of $n \log_2(n)$, just like quick sort. There are no guarantees for upper bounds; you may assume that its worst-case complexity is at most quadratic, $O(n^2)$.

Whereas there is a catch-all pointer type, **void*** that can be used as a generic pointer to object types, no such generic type or implicit conversion exists for function pointers.

Takeaway 11.4 #2 *Function pointers must be used with their exact type.*

Such a strict rule is necessary because the calling conventions for functions with different prototypes may be quite different[24] and the pointer itself does not keep track of any of this.

The following function has a subtle problem because the types of the parameters are different than what we expect from a comparison function:

```
/* Another invalid example for an int comparison function */
int compare_int(int const* a, int const* b){
  if (*a < *b) return -1;
  else if (*a > *b) return +1;
  else return 0;
}
```

When you try to use this function with **qsort**, your compiler should complain that the function has the wrong type. The variant that we gave earlier using intermediate **void const*** parameters should be almost as efficient as this invalid example, but it also can be guaranteed to be correct on all C platforms.

Calling functions and function pointers with the `(...)` operator has rules similar to those for arrays and pointers and the `[...]` operator:

```
  double f(double a);

  // Equivalent calls to f, steps in the abstract state machine
```

[24] The platform application binary interface (ABI) may, for example, pass floating points in special hardware registers.

```
  f(3);         // Decay → call
  (&f)(3);      // Address of → call
  (*f)(3);      // Decay → dereference → decay → call
  (*&f)(3);     // Address of → dereference → decay → call
  (&*f)(3);     // Decay → dereference → address of → call
```

Takeaway 11.4 #3 *The function call operator* $(\cdots)$ *applies to function pointers.*

So, technically, in terms of the abstract state machine, the pointer decay is always performed, and the function is called via a function pointer. The first, "natural" call has a hidden evaluation of the `f` identifier that results in the function pointer.

Given all this, we can use function pointers almost like functions:

```
// In a header
typedef int logger_function(char const*, ...);
extern logger_function* logger;
enum logs { log_pri, log_ign, log_ver, log_num };
```

This declares a global variable `logger` that will point to a function that prints out logging information. Using a function pointer will allow the user of this module to choose a particular function dynamically:

```
// In a .c file (TU)
extern int logger_verbose(char const*, ...);
static
int logger_ignore(char const*, ...) {
  return 0;
}
logger_function* logger = logger_ignore;

static
logger_function* loggers = {
  [log_pri] = printf,
  [log_ign] = logger_ignore,
  [log_ver] = logger_verbose,
};
```

Here, we are defining tools that implement this approach. In particular, function pointers can be used as a base type for arrays (here, `loggers`). Observe that we use two external functions (**`printf`** and `logger_verbose`) and one **`static`** function (`logger_ignore`) for the array initialization; the storage class is not part of the function interface.

The `logger` variable can be assigned just like any other pointer type. Somewhere at startup, we can have

```
if (LOGGER < log_num) logger = loggers[LOGGER];
```

Then this function pointer can be used anywhere to call the corresponding function:

```
logger("Do␣we␣ever␣see␣line␣\%lu␣of␣file␣\%s?", __LINE__+0UL, __FILE__);
```

This call uses the special macros **`__LINE__`** and **`__FILE__`** for the line number and the name of the source file. We will discuss these in more detail in subsection 17.3.

When using pointers to functions, you should always be aware that doing so introduces an indirection to the function call. The compiler first has to fetch the contents of `logger` and can only then call the function at the address it found there. This has a certain overhead and should be avoided in time-critical code.

CHALLENGE 13 (Generic derivative). *Can you extend the real and complex derivatives (challenges 2 and 5) such that they receive the function* `F` *and the value* `x` *as a parameter?*
Can you use the generic real derivatives to implement Newton's method for finding roots?
Can you find the real zeros of polynomials?
Can you find the complex zeros of polynomials?

CHALLENGE 14 (Generic sorting). *Can you extend your sorting algorithms (challenge 1) to other sort keys?*
Can you condense your functions for different sort keys to functions that have the same signature as **`qsort`***—that is, receive generic pointers to data, size information, and a comparison function as parameters?*
Can you extend the performance comparison of your sorting algorithms (challenge 10) to the C library function **`qsort`***?*

Summary

- Pointers can refer to objects and functions.
- Pointers are not arrays but refer to arrays.
- Array parameters of functions are automatically rewritten as object pointers.
- Function parameters of functions are automatically rewritten as function pointers.
- Function pointer types must match exactly when they are assigned or called.

12. The C memory model

This section covers

- Understanding object representations
- Working with untyped pointers and casts
- Restricting object access with effective types and alignment

Pointers present us with a certain abstraction of the environment and state in which our program is executed, the *C memory model*. We may apply the unary operator `&` to (almost) all objects[25] to retrieve their address and use it to inspect and change the state of our execution.

This access to objects via pointers is still an abstraction because, in C, no distinction of the "real" location of an object is made. It could reside in your computer's RAM, on a disk file, or correspond to an IO port of a temperature sensor on the moon; you shouldn't care. C is supposed to do the right thing, regardless.

And, indeed, on modern operating systems, all you get via pointers is something called *virtual memory*, basically a fiction that maps the *address space* of your process to physical memory addresses of your machine. All this was invented to ensure certain properties of your program executions:

portable: You do not have to care about physical memory addresses on a specific machine.

safe: Reading or writing virtual memory that your process does not own will affect neither your operating system nor any other process.

The only thing C must care about is the *type* of the object a pointer addresses. Each pointer type is derived from another type, its base type, and each such derived type is a distinct new type.

Takeaway 12 #1 *Pointer types with distinct base types are distinct.*

In addition to providing a virtual view of physical memory, the memory model also simplifies the view of objects themselves. It makes the assumption that each object is a collection of bytes, the *object representation* (subsection 12.1);[26] see figure 12 for a schematic view. A convenient tool to inspect that object representation is *unions* (subsection 12.2). Giving direct access to the object representation (subsection 12.3) allows us to do some fine-tuning. On the other hand, it also opens the door to unwanted or conscious manipulations of the state of the abstract machine: tools for that are untyped pointers (subsection 12.4) and casts (subsection 12.5). Effective types (subsection 12.6) and alignment (subsection 12.7) describe formal limits and platform constraints for such manipulations.

12.1. A uniform memory model. Even though generally all objects are typed, the memory model makes another simplification: all objects are an assemblage of ***bytes***^{C}. The **`sizeof`** operator that we 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 1 byte of memory: the character types **`char`**, **`unsigned char`**, and **`signed char`**.

Takeaway 12.1 #1 *`sizeof(char)` is `1` by definition.*

[25] Only objects that are declared with keyword **`register`** don't have an address; see subsection 13.2.2 in level 2

[26] The object representation is related to but not the same thing as the *binary representation* that we saw in subsection 5.1.3.

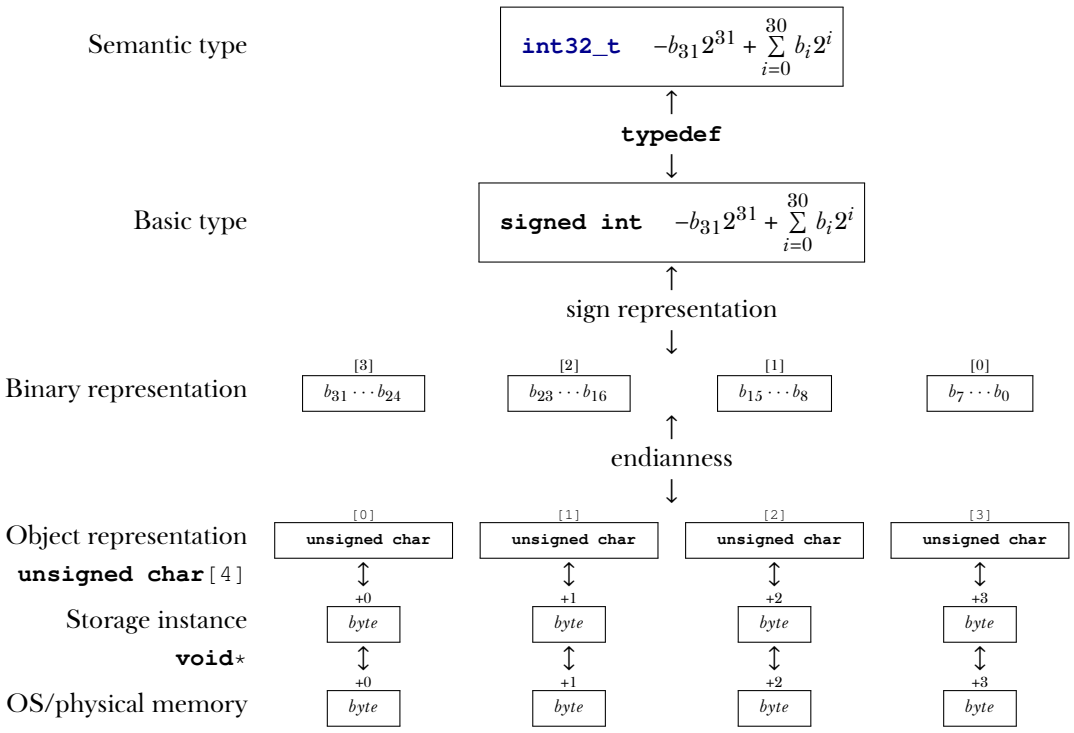


FIGURE 12.1. The different levels of the value-memory model for an **int32_t**. Example of a platform that maps this type to a 32-bit **signed int** and uses a little-endian object representation.

Not only can all objects be "accounted" in size as character types on a lower level, they can even be inspected and manipulated as if they were arrays of such character types. A little later, we will see how this can be achieved, but for the moment we will just note the following:

Takeaway 12.1 #2 *Every object* `A` *can be viewed as* **unsigned char**`[`**sizeof** `A]`.

Takeaway 12.1 #3 *Pointers to character types are special.*

Unfortunately, the types that are used to compose all other object types are derived from **char**, the type we looked at for the characters of strings. This is merely a historical accident, and you shouldn't read too much into it. In particular, you should clearly distinguish the two different use cases.

Takeaway 12.1 #4 *Use the type* **char** *for character and string data.*

Takeaway 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.

As we have seen, the **sizeof** operator counts the size of an object in terms of how many **unsigned char**s it occupies.

Takeaway 12.1 #6 *The* **sizeof** *operator can be applied to objects and object types.*

From the previous discussion, we can also distinguish two syntactic variants for **sizeof**: with and without parentheses. Whereas the syntax for an application to objects can have both forms, the syntax for types needs parentheses.

Takeaway 12.1 #7 *The size of all objects of type* `T` *is given by* **sizeof**`(T)`.

12.2. Unions. Let us now look at a way to examine the individual bytes of objects. Our preferred tool for this is the **union**. These are similar in declaration to **struct** but have different semantics:

endianness.c
```
4  typedef union unsignedInspect unsignedInspect;
5  union unsignedInspect {
6    unsigned val;
7    unsigned char bytes[sizeof(unsigned)];
8  };
9  unsignedInspect twofold = { .val = 0xAABBCCDD, };
```

The difference here is that such a **union** doesn't collect objects of different types into one bigger object, but rather *overlays* an object with several different type interpretations. That way, 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**, inclusive, and thus we interpret the number as written with 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 **0x**`AA`, **0x**`BB`, **0x**`CC`, and **0x**`DD` for the highest- to lowest-order representation digit. The complete **unsigned** value can be computed using the following expression, where **CHAR_BIT** is the number of bits in a character type:

```
1  ((0xAA << (CHAR_BIT*3))
2      |(0xBB << (CHAR_BIT*2))
3      |(0xCC << CHAR_BIT)
4      |0xDD)
```

With the **union** defined earlier, we have two different facets to look at the same `twofold` object: `twofold.val` presents it as being an **unsigned**, and `twofold.bytes` presents it as an 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 ***object representation***^{C} of an **unsigned** value using all its representation digits:

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

On my computer, I receive a result as shown here:[27]

Terminal
```
0  ~/build/modernC% code/endianness
1  value is 0xAABBCCDD
2  byte[0]: 0xDD
3  byte[1]: 0xCC
4  byte[2]: 0xBB
5  byte[3]: 0xAA
```

[27] Test the code on your own machine.

For my machine, we see that the output has 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 has the low-order representation digits before the high-order ones.

This is *not* normalized by the standard but is an implementation-defined behavior.

Takeaway 12.2 #1 *The in-memory order of the representation digits of an arithmetic type is implementation defined.*

That is, a platform provider might decide to provide a storage order that has the highest-order digits first and then print lower-order digits one by one. The storage order, the ***endianness***^{C}, as given for my machine, is called ***little-endian***^{C}. A system that has high-order representation digits first is called ***big-endian***^{C}.[28] Both orders are commonly used by modern processor types. Some processors are even able to switch between the two orders on the fly.

<stdbit.h>

Since C23, the header `<stdbit.h>` has macros that provide the endianess that the compiler implements:

platform endianess	**__STDC_ENDIAN_NATIVE__**
little	**__STDC_ENDIAN_LITTLE__**
big	**__STDC_ENDIAN_BIG__**
other	different from above

The previous output also shows another implementation-defined behavior: I used the feature of my platform that one representation digit can be printed nicely by using two hexadecimal digits. In other words, I assumed that **UCHAR_MAX**`+1` is `256` and that the number of value bits in an **unsigned char**, **CHAR_BIT**, is `8`. Again, this is implementation-defined behavior. Although the vast majority of platforms have these properties,[29] there are still some around that have wider character types.

Takeaway 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 the sign, mantissa, and exponent; and pointer types may follow any internal convention that fits the underlying architecture.[Exs 30][Exs 31][Exs 32]

12.3. Memory and state. The value of all objects constitutes the state of the abstract state machine and thus the state of a particular execution. C's memory model provides something like a unique location for (almost) all objects through the `&` operator, and that location can be accessed and modified from different parts of the program through pointers.

Doing so makes the determination of the abstract state of an execution much more difficult, if not impossible in many cases:

```
1 double blub(double const* a, double* b);
2
3 int main(void) {
4   double c = 35;
5   double d = 3.5;
```

[28] The names are derived from the fact that the big or small "end" of a number is stored first.

[29] In particular, all POSIX systems.

[Exs 30] Design a similar **union** type to investigate the bytes of a pointer type, such as **double***.

[Exs 31] With such a **union**, investigate the addresses of two consecutive elements of an array.

[Exs 32] Compare the addresses of the same variable between different executions.

```
6   printf("blub␣is␣%g\n", blub(&c, &d));
7   printf("after␣blub␣the␣sum␣is␣%g\n", c + d);
8 }
```

Here, we (as well as the compiler) only see a declaration of function `blub`, with no definition. So we cannot conclude much about what that function does to the objects its arguments point to. In particular, we don't know if the variable `d` is modified, 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 at such a function that receives two pointer arguments:

```
1 double blub(double const* a, double* b) {
2   double myA = *a;
3   *b = 2*myA;
4   return *a;       // May be myA or 2*myA
5 }
```

Such a function can operate under two different assumptions. First, if called with two distinct addresses as arguments, `*a` will be unchanged, and the return value will be the same as `myA`. But if both argument are the same, such as if the call is `blub(&c, &c)`, the assignment to `*b` will change `*a`, too.

The phenomenon of accessing the same object through different pointers is called ***aliasing***^{C}; it is a common cause for missed optimization. In both cases, either that two pointers always alias or that they never alias, the abstract state of an execution is much reduced, and the optimizer often can take much advantage of that knowledge. Therefore, C forcibly restricts the possible aliasing to pointers of the same type.

Takeaway 12.3 #1 (Aliasing) *With the exclusion of character types, only pointers of the same base type may alias.*

To see this rule in effect, consider a slight modification of our previous example:

```
1 size_t blob(size_t const* a, double* b) {
2   size_t myA = *a;
3   *b = 2*myA;
4   return *a;       // Must be myA
5 }
```

Because here the two parameters have different types, C *assumes* that they don't address the same object. In fact, it would be an error to call that function as `blob(&e, &e)` (for some variable `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; it is a road to much grief and despair. *If* you do so, the behavior of the program becomes undefined, so you have to guarantee (prove!) that no aliasing takes place.

On the contrary, we should try to write our programs so they protect our variables from ever being aliased, and there is an easy way to achieve that.

Takeaway 12.3 #2 *Avoid the `&` operator.*

Depending on the properties of a given variable, the compiler may see that the address of the variable is never taken, and thus the variable can't alias at all. In subsection 13.2, we will see which properties of a variable or object may influence such

decisions and how the **register** keyword can protect us from taking addresses inadvertently. Later, in subsection 16.2, we will see how the **restrict** keyword allows us to specify aliasing properties of pointer arguments, even if they have the same base type.

12.4. Pointers to unspecific objects. As we have seen, the object representation provides a view of an object `X` as an array **unsigned char**`[`**sizeof** `X]`. The starting address of that array (of type **unsigned char***) provides access to memory that is stripped of the original type information.

C has invented a powerful tool to handle such pointers more generically. These are pointers to a sort of *non-type*, **void**.

Takeaway 12.4 #1 *Any object pointer converts to and from* ***void****.*

Note that this only talks about object pointers, not function pointers. Think of a **void*** pointer that holds the address of an existing object as a pointer into a *storage instance* that holds the object; see figure 12 at page 184. As an analogy for such a hierarchy, you could think of entries in a phone book: a person's name corresponds to the identifier that refers to an object; their categorization with a "mobile," "home," or "work" entry corresponds to a type; and their phone number itself is some sort of address (in which, by itself, you typically are not interested). But then, even the phone number abstracts away from the specific information of where the other phone is located (which would be the storage instance underneath the object) or specific information about the other phone itself (for example, if it is on a landline or the mobile network) and what the network has to do to actually connect you to the person at the other end.

Takeaway 12.4 #2 *An object has storage, type, and value.*

Not only is the conversion to **void*** well defined, but it also is guaranteed to behave well with respect to the pointer value.

Takeaway 12.4 #3 *Converting an object pointer to* ***void***** and then back to the same type is the identity operation.*

So, the only thing that we lose when converting to **void*** is the type information; the value remains intact.

Takeaway 12.4 #4 (*avoid*^{2}*****) *A*void ***void*****.*

It completely removes any type information that is associated with 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 won't lead to an object with which we can do anything.

12.5. Explicit conversions. A convenient way to look at the object representation of object `X` would be to somehow convert a pointer to `X` to a pointer of type **unsigned char***:

```
double X;
unsigned char* Xp = &X; // error: implicit conversion not allowed
```

Fortunately, such an implicit conversion of a **double*** to **unsigned char*** is not allowed. We would have to make this conversion somehow explicit.

We already have seen that in many places, a value of a certain type is implicitly converted to a value of a different type (subsection 5.4), and that narrow integer types are first converted to **int** before any operation. 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.

Conversions of pointer types are more delicate because they can change the type interpretation of an object. Only two forms of implicit conversions are permitted for data pointers: conversions from and to **void*** and conversions that add a qualifier to the target type. Let's look at some examples:

```
1 float f = 37.0;         // Conversion: to float
2 double a = f;           // Conversion: back to double
3 float* pf = &f;         // Exact type
4 float const* pdc = &f; // Conversion: adding a qualifier
5 void* pv = &f;          // Conversion: pointer to void*
6 float* pfv = pv;        // Conversion: pointer from void*
7 float* pd = &a;         // Error: incompatible pointer type
8 double* pdv = pv;       // Error if used
```

The first two conversions that use **void*** (`pv` and `pfv`) 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. By now, you should understand well that your code should not produce compiler warnings (takeaway 1.2 #3), and 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 this error might go undetected is that our first conversion for `pv` has stripped the pointer from all type information. So, in general, the compiler can't know what type of object is behind the pointer.

In addition to the implicit conversions that we have seen until now, C also allows us to convert explicitly using ***casts***^{C}.[33] With a cast, you are telling the compiler that you know better than it does and that the type of the object behind the pointer is not what it thinks, so it should 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.

Takeaway 12.5 #1 *Don't use casts.*

They deprive you of precious information, and if you choose your types carefully, you will only need them for very special occasions.

One such occasion is when you want to inspect the contents of an object on the byte level. Constructing a **union** around an object, as we saw in subsection 12.2, might not always be possible (or may be too complicated), so here we can go for a cast:

endianness.c

```
15 unsigned val = 0xAABBCCDD;
16 unsigned char* valp = (unsigned char*)&val;
17 for (size_t i = 0; i < sizeof val; ++i)
```

[33] A cast of an expression `X` to type `T` has the form `(T)X`. Think of it like "*to cast a spell.*"

```
18      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.[34]

12.6. Effective types. To cope with different views of the same object that pointers may provide, C has introduced the concept of *effective types*. It heavily restricts how an object can be accessed.

Takeaway 12.6 #1 (Effective type) *Objects must be accessed through their effective type or through a pointer to a character type.*

Because the effective type of a **union** variable is the **union** type and none of the member types, the rules for **union** members can be relaxed.

Takeaway 12.6 #2 *Any member of an object that has an effective* **union** *type can be accessed at any time, provided the byte representation amounts to a valid value of the access type.*

For all objects we have seen so far, it is easy to determine the effective type.

Takeaway 12.6 #3 *The effective type of a variable or compound literal is the type of its declaration.*

Later, we will see another category of objects that are a bit more involved.

Note that this rule has no exceptions, and we can't change the type of such a variable or compound literal.

Takeaway 12.6 #4 *Variables and compound literals must be accessed through their declared type or through a pointer to a character type.*

Also, observe the asymmetry in all of this for character types. Any object can be seen as being composed of **unsigned char**, but no array of **unsigned char**s can be used through another type:

```
unsigned char A[sizeof(unsigned)] = { 9 };
// Valid but useless, as most casts are
unsigned* p = (unsigned*)A;
// Error: access with a type that is neither the effective type nor a
// character type
printf("value \%u\n", *p);
```

Here, the access `*p` is an error, and the program state is undefined afterward. This is in strong contrast to our dealings with **union** earlier: see subsection 12.2, where we actually could view a byte sequences as an array of **unsigned char** or **unsigned**.

The reasons for such a strict rule are multiple. The very first motivation for introducing effective types in the C standard was to deal with aliasing, as we saw in subsection 12.3. In fact, the aliasing rule (takeaway 12.3 #1) is derived from the effective type rule (takeaway 12.6 #1). As long as there is no **union** involved, the compiler knows that we cannot access a **double** through a **size_t***, and so it may *assume* that the objects are different.

[34] Note, though, that even a recent compiler at the time of this writing gets that particular code snippet wrong and is convinced that the byte-wise access goes to uninitialized data. Avoid casts as far as you may.

12.7. Alignment. The inverse direction of pointer conversions (from "pointer to character type" to "pointer to object") is not harmless at all and not only because of possible aliasing. This has to do with another property of C's memory model: ***alignment***^{C}. Objects of most noncharacter 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:

crash.c

```
11 int main(void) {
12   enable_alignment_check();
13   /* An overlay of complex values and bytes. */
14   union {
15     cdbl val[2];
16     unsigned char buf[sizeof(cdbl[2])];
17   } toocomplex = {
18     .val = { 0.5 + 0.5*I, 0.75 + 0.75*I, },
19   };
20   printf("size/alignment:␣%zu/%zu\n",
21          sizeof(cdbl), alignof(cdbl));
22   /* Run over all offsets, and crash on misalignment. */
23   for (size_t offset = sizeof(cdbl); offset; offset /=2) {
24     printf("offset\t%zu:\t", offset);
25     fflush(stdout);
26     cdbl* bp = (cdbl*)(&toocomplex.buf[offset]); // align!
27     printf("%g\t+%gI\t", creal(*bp), cimag(*bp));
28     fflush(stdout);
29     *bp *= *bp;
30     printf("%g\t+%gI", creal(*bp), cimag(*bp));
31     fputc('\n', stdout);
32   }
33 }
```

This starts with a declaration of a **union** similar to what we saw earlier. Again, we have a data object (of type **complex double**[2] in this case) that we overlay with an array of **unsigned char**. The obvious intent of this program is to print one output line per loop execution, each prefixed with the value of `offset`. Other than the fact that this part is a bit more complex, at first glance there is no major problem with it. But if I execute this program on my machine, I get

Terminal

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

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

crash.c

```
26     cdbl* bp = (cdbl*)(&toocomplex.buf[offset]); // align!
```

On the right, we see a pointer cast: 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 previous output, it seems that **`complex double`** still works well for alignments of half of its size, but with an alignment of one fourth, the program crashes.

Some architectures are more tolerant of misalignment than others, and we might have to force the system to error out on such a condition. We use the following function at the beginning to force crashing:

crash.c

`enable_alignment_check`: enable alignment check for i386 processors

Intel's i386 processor family is quite tolerant in accepting misalignment of data. This can lead to irritating bugs when ported to other architectures that are not as tolerant.

This function enables a check for this problem also for this family or processors, such that you can be sure to detect this problem early.

I found that code on Ygdrasil's blog: `http://orchistro.tistory.com/206`

```
void enable_alignment_check(void);
```

If you are interested in portable code (and if you are still here, you probably are), early errors in the development phase are really helpful.[35] So, consider crashing a feature. See the blog entry mentioned in `crash.h` for an interesting discussion of this topic.

In the previous code example, we also see a new operator, **`alignof`**, that provides us with the alignment of a specific type. You will rarely find the occasion to use it in real live code. Prior to C23, this operator was spelled **`_Alignof`**; if you are concerned about legacy code or platforms, you should include `<stdalign.h>` to do the replacement.

`<stdalign.h>`

Another keyword can be used to force allocation at a specified alignment: **`alignas`** (since C23; previously **`_Alignas`**). Its argument can be either a type or expression. It can be useful when you know that your platform can perform certain operations more efficiently if the data is aligned in a certain way.

For example, to force alignment of a **`complex`** variable to its size and not half the size, as we saw earlier, you could use

```
alignas(sizeof(complex double)) complex double z;
```

Or, if you know that your platform has efficient vector instructions for **`float`**`[4]` arrays:

```
alignas(sizeof(float[4])) float fvec[4];
```

These operators don't help against the effective type rule (takeaway 12.6 #1). Even with

```
alignas(unsigned) unsigned char A[sizeof(unsigned)] = { 9 };
```

the example at the end of section 12.6 remains invalid.

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

Summary

- The memory and object model have several layers of abstraction: physical memory, virtual memory, storage instances, object representation, and binary representation.
- Each object can be seen as an array of **`unsigned char`**.
- **`union`**s serve to overlay different object types over the same object representation.
- Memory can be aligned differently according to the need for a specific data type. In particular, not all arrays of **`unsigned char`** can be used to represent any object type.

13. Storage

This section covers

- Creating objects with dynamic allocation
- The rules of storage and initialization
- Understanding object lifetime
- Handling automatic storage

So far, most objects we have 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 the object. Sometimes they were defined at a different place in the code than they were declared, but even such a definition referred to them with a type and identifier. Another category of objects that we have seen less often is specified with a type but not with an identifier: *compound literals*, as introduced in subsection 5.6.4.

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 declared with **`static`**) or are bound to a block of statements inside a function.[36]

We also have seen that for certain objects, it is important to distinguish different instances when we declare a variable in a recursive function. Each call in a hierarchy of recursive calls has its own instance of such a variable. Therefore, it is convenient to distinguish another entity that is not exactly the same as an object, the storage instance.

In this section, we will handle another mechanism to create objects called *dynamic allocation* (subsection 13.1). In fact, this mechanism creates storage instances that are only seen as byte arrays and do not have any interpretation as objects. They only acquire a type once we store something.

With this, we have an almost complete picture of the different possibilities, and thus we can discuss the different rules for storage *duration*, object *lifetime*, and identifier *visibility* (subsection 13.2). We will also take a full dive into the rules for initialization (subsection 13.4), as these differ significantly for differently created objects.

Additionally, we propose two digressions. The first is a more detailed view of object lifetime, which allows us to access objects at surprising points in the C code (subsection 13.3). The second provides a glimpse into a realization of the memory model for a concrete architecture (subsection 13.5) and, in particular, how automatic storage may be handled on your particular machine.

13.1. malloc and friends. For programs that have to handle growing collections of data, the types of objects that we have seen so far are too restrictive. To handle varying user input, web queries, large interaction graphs and other irregular data, big matrices, and audio streams, it is convenient to reclaim storage instances for objects on the fly and then release them once they are not needed anymore. Such a scheme is called ***dynamic allocation***^{C} or sometimes just *allocation* for short.

<stdlib.h>

The following set of functions, available with `<stdlib.h>`, has been designed to provide such an interface to allocated storage:

```
#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);
```

[36] In fact, this is a bit of a simplification; we will see the gory details shortly.

The first two, **malloc** (memory allocate) and **free**, are by far the most prominent. As the names indicate, **malloc** creates a storage instance 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 storage to `0`, **realloc** grows or shrinks storage, and **aligned_alloc** ensures nondefault alignment.

Takeaway 13.1 #1 *Only use the allocation functions with a size strictly greater than zero.*

Before discussing these functions in more detail, let us look into two functions that C23 imported from POSIX, which we already briefly discussed:

```
#include <string.h>
char* strdup( char const s[static 1]);
char* strndup(char const s[static 1], size_t n);
```

They are specialized to handle character strings by conveniently combining allocation and copy operations. **strdup** can be seen as if it was defined as follows:

```
char *strdup(char const s[static 1]) {
  // +1 for the 0-termination
  char* ret = malloc(strlen(s)+1);
  return ret ? strcpy(ret, s) : nullptr;
}
```

Note that this supposes that `s` is not null and the input given by it is, in effect, a string. Otherwise, the scan using **strlen** would be erroneous.

Here, the call to the function **malloc** either allocates the requested storage space and returns a pointer to it or, if no storage is available, returns a null pointer. This property is taken into account so that the copy operation is only performed if the allocation succeeds.

Takeaway 13.1 #2 *Failed allocations result in a null pointer.*

strndup is a bit less constrained than **strdup**. It only assumes that the buffer pointed to by `s` either has a 0 value among the first `n` bytes or, otherwise, has at least that size:

```
char* strndup(char const s[static 1], size_t n) {
  char const* pos = memchr(s, 0, n);
  n = pos ? (pos-s)+1 : n;
  char* ret = malloc(n);
  if (ret) {
    memcpy(ret, s, n-1);
    ret[n-1] = 0;
  }
  return ret;
}
```

The buffer that is returned (if any) is always a string because a write of a 0 value to the last byte is assured. So, whenever we know of a bound for the size of the string, we should prefer that interface.

Takeaway 13.1 #3 *Prefer the use of **strndup** over **strdup**.*

The `<stdlib.h>` 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. Using that, they become universally applicable to all types. The following example allocates a large storage for a vector of **double**s, one element for each living person[Exs 37]: <stdlib.h>

[Exs 37] Don't try this allocation; instead, compute the size that would be needed on your platform. Is allocating such a vector feasible on your platform?

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

free(largeVec);
```

Because **malloc** knows nothing about the later use or type of the object to be stored, the size of the storage is specified in bytes. In the idiom given here, 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 number of bytes. Even if we change `largeVec` later to have type **size_t***, the allocation will adapt.

Another idiom that we will often encounter strictly takes the size of the type of the object that we want to create—an array of `length` elements of type **double**:

```
double* largeVec = malloc(sizeof(double[length]));
```

We already have been haunted by the introduction of casts, which are 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.

Takeaway 13.1 #4 *Don't cast the return of* ***malloc*** *and friends.*

<stdlib.h>

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

```
/* 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* <--
```

compilers 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, particularly in beginners' code whose authors have been following bad advice.

In the previous code, as a next step, we initialize the storage we just allocated through assignment (here, all `0.0`). It is only with these assignments that the individual elements of `largeVec` become "objects." Such an assignment provides an effective type *and* a value.

Takeaway 13.1 #5 *Storage allocated through* ***malloc*** *is uninitialized and has no type.*

13.1.1. *A complete example with varying array size.* Let us now look at an example where using a dynamic array 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`: a 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, null 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 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 element *pos* of the buffer, null otherwise.

```
double* circular_element(circular const* 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 of functions. The first pair is applied to existing storage. The pair receives a pointer to the structure and ensures that space for the data member is allocated or freed:

circular.h

`circular_init`:

Initialize a circular buffer *c* with maximally *cap* 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 cap);
```

circular.h

`circular_destroy`:

Destroy circular buffer *c*.

c must have been initialized with a call to circular_init

```
void circular_destroy(circular* c);
```

The second pair additionally allocates or deallocates the storage for the structure itself. Since these two functions do not access any member of the structure directly, they can be specified as **`inline`**:

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.

```
[[nodiscard("pointer to allocated data dropped")]]
[[__gnu__::__malloc__, __gnu_free__(circular_delete)]]
inline
circular* circular_new(size_t len) {
  return circular_init(malloc(sizeof(circular)), len);
}
```

circular.h

`circular_delete`:

Delete circular buffer *c*.

c must have been allocated with a call to circular_new

```
inline
void circular_delete(circular* c) {
  circular_destroy(c);
  free(c);
}
```

If we used regular array variables, the maximum number of elements we could store in a `circular` would be fixed once we created such an object. We want to be more flexible so this limit can be raised or lowered by means of the `circular_resize` function and the number of elements can be queried with `circular_getlength`:

circular.h

`circular_resize`:

Resize to capacity *cap*.

```
[[nodiscard("returned pointer replaces function argument")]]
circular* circular_resize(circular* c, size_t cap);
```

circular.h

`circular_getlength`:

Return the number of elements stored.

```
size_t circular_getlength(circular const* c);
```

The functions `circular_new` and `circular_resize` use an attribute that we have not seen before, `[[nodiscard]]`. It indicates, with an optional additional message, that the return value for the function should not be ignored; if we do so, the compiler will issue a warning. This is particularly important for our use case: the pointer that we pass to a function call will, in general, be invalid when the function returns, and we'd have to use the possibly new pointer value that we receive in return.

Additionally, `circular_new` uses `[[__gnu__::__malloc__]]`, a gnu specific attribute which indicates that the return value of this function provides a pointer to data that has not been seen before (the first variant) and that the function used to delete the data should be `circular_delete` (second variant).

With the function `circular_element`, the type behaves like an array of **`double`**s. Calling it with a position within the current length, we obtain the address of the element stored in that position.

Before C23 and in the previous revision of this book, we had the definition of the structure hidden inside the `.c` file, so users could only use the functions we provide as an access. Nowadays, we have the `[[deprecated]]` attribute to mark all members as deprecated so we can place even the definition of the structure in a header; we will see in the following discussion how that aspect of the structure works.

circular.h

```
20 struct circular {
21   size_t  start [[deprecated("privat")]]; /* First element     */
22   size_t  len   [[deprecated("privat")]]; /* Number of elements*/
23   size_t  cap   [[deprecated("privat")]]; /* Maximum capacity  */
24   double* tab   [[deprecated("privat")]]; /* Data array        */
25 };
```

The idea is that the pointer member `tab` will always point to an array object of length `cap`. 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 member `len`. The position inside the table `tab` is computed modulo `cap`.

The following table symbolizes one instance of this `circular` data structure, with `cap=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 contents (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 a data structure needs to call **malloc** to provide memory for the `tab` member:

circular.c

```
12 [[deprecated("implementation")]]
13 circular* circular_init(circular* c, size_t cap) {
14   if (c) {
15     if (cap) {
16       *c = (circular){
17         .cap = cap,
18         .tab = malloc(sizeof(double[cap])),
19       };
20         // Allocation failed.
21       if (!c->tab) c->cap = 0;
22     } else {
23       *c = (circular){ };
24     }
25   }
26   return c;
27 }
```

Observe that this function always checks the pointer parameter `c` for validity. Also, it guarantees to initialize all other members to `0` by assigning compound literals in both branches of the conditional.

The library function **malloc** can fail for different reasons. For example, the memory system might be exhausted from previous calls to it, or the reclaimed size for allocation might just be too large. In general-purpose systems like the one you are probably using for your learning experience, such a failure will be rare (unless voluntarily provoked), but it still is a good habit to check for it.

Takeaway 13.1.1 #1 ***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` member unconditionally:

circular.c

```
29 [[deprecated("implementation")]]
30 void circular_destroy(circular* c) {
31   if (c) {
32     free(c->tab);
33     circular_init(c, 0);
34   }
35 }
```

The library function **free** has the friendly property that it accepts a null parameter and 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**, so it is only visible for those functions and doesn't pollute the identifier name space (takeaway 9.2 #3):

. circular.c

```
[[deprecated("implementation")]]
static size_t circular_getpos(circular const c[static 1], size_t
    pos) {
  if (c->cap) {
    pos += c->start;
    pos %= c->cap;
  }
  return pos;
}
```

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

. circular.c

```
[[deprecated("implementation")]]
double* circular_element(circular const* c, size_t pos) {
  double* ret = nullptr;
  if (c) {
    if (pos < c->cap) {
      pos = circular_getpos(c, pos);
      ret = &c->tab[pos];
    }
  }
  return ret;
}
```

With all that information, you should now be able to implement all but one of the function interfaces nicely.[Exs 38] The more difficult one is `circular_resize`. It starts with some length calculations and then treats the cases in which 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) to its value afterward. The end of the function then uses a compound literal to compose the new structure by using the values found during the case analysis:

circular.c

```
[[nodiscard("returned␣pointer␣replaces␣function␣argument")]]
[[deprecated("implementation")]]
circular* circular_resize(circular* c, size_t nlen) {
  if (c) {
    size_t len = c->len;
    if (len > nlen) return nullptr;
    size_t olen = c->cap;
    if (nlen != olen) {
      size_t ostart = circular_getpos(c, 0);
      size_t nstart = ostart;
      double* otab = c->tab;
      double* ntab;
      if (nlen > olen) {
```

[Exs 38] Write implementations of the missing functions.

. circular.c

```
144         }
145         *c = (circular){
146           .cap = nlen,
147           .start = nstart,
148           .len = len,
149           .tab = ntab,
150         };
151       }
152     }
153     return c;
154   }
```

Let us now try to fill the gap in the previous code and look at the first case of enlarging an object. The essential part is a call to **realloc**:

. circular.c

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

For this call, **realloc** receives the pointer to the existing object and the new size the relocation should have. It returns either a pointer to the new object with the desired size or null. In the line immediately after, we check the latter case and terminate the function if it is not possible to relocate the object.

The function **realloc** has interesting properties:

- If the original pointer is null, it allocates a buffer like **malloc**.
- The returned pointer may or may not be the same as the argument. It is left to the discretion of the runtime system to determine whether the resizing can be performed in place (for example, if there is space available behind the object or whether a new object must be provided). Regardless, even if the returned pointer is the same, the object is considered to be a new one (with the same data). In particular, that means all pointers derived from the original become invalid.
- 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 be) with the previous pointer. The old object is taken care of.
- As far as possible, the 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 content that corresponds to the initial part before the call.
- If null is returned (that is, the relocation request could not be fulfilled by the runtime system), the old object is unchanged. So, nothing is lost.

Now that we know the newly received object has the size we want, we have to ensure that `tab` still represents a circular buffer. If previously the situation was, as in the first table, earlier (the part that corresponds to the buffer elements is contiguous), we have nothing to do. All data is nicely preserved.

If our circular buffer wraps around, we have to make some adjustments:

.

circular.c

```
103         // non-empty, and there were already two separate chunks
104         if (ostart+len > olen) {
105           size_t ulen = olen - ostart;
106           size_t llen = len - ulen;
107           if (llen <= (nlen - olen)) {
108             /* Copy the lower one up after the old end. */
109             memcpy(ntab + olen, ntab,
110                    llen*sizeof(double));
111           } else {
112             /* Move the upper one up to the new end. */
113             nstart = nlen - ulen;
114             memmove(ntab + nstart, ntab + ostart,
115                     ulen*sizeof(double));
116           }
117         }
```

The following table illustrates the difference in the contents between before and after the changes for the first subcase. The lower part finds enough space inside the part that was 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 toward 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**; the source and target elements of the copy operation can't overlap, so using **memcpy** is safe. In 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 39]

You may have noticed that the previous function definitions (but not the declarations!) are also marked with a `[[deprecated]]` attribute. This has the effect that the use of the members of the `ciruclar` structure within these functions is not diagnosed as being deprecated; otherwise, the compilation of the file `circular.c` would issue a lot of useless diagnostics. Code that uses just the header does not see the annotations with `[[deprecated]]`, so it will not diagnose the use of any of the functions but only

[Exs 39] Implement shrinking of the table; it is important to reorganize the table contents before calling **realloc**.

if the members of the structure are accessed directly. By that, user code may have local variables of the `circular` type:

```
#include "circular.h"
  ...
  circular circ;
  circular_init(&circ, 100);
  ...
  circular_destroy(&circ);
```

This would not be possible if we had hidden the structure definition completely in the `.c` source file.

As of this writing (January 2024), not all compilers completely stick to the intent of the C standard with respect to this attribute. Therefore, our source has a compiler-specific **`#pragma`** at the beginning. This could be removed once the compilers in question are fixed to follow the C standard more closely:

. circular.c

```
8  #if __GNUC__ > 4 && __GNUC__ <= 14
9  # pragma GCC diagnostic ignored "-Wdeprecated-declarations"
10 #endif
```

13.1.2. *Ensuring consistency of dynamic allocations.* As in both our code examples, calls to allocation functions, such as **`malloc`**, **`realloc`**, and **`free`**, should always come in pairs. This mustn't necessarily be inside the same function, but in most cases, simple counting of the occurrence of both should give the same number.

Takeaway 13.1.2 #1 *For every allocation, 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.

Takeaway 13.1.2 #2 *For every* ***`free`****, there must be a* ***`malloc`****,* ***`calloc`****,* ***`aligned_alloc`****, or* ***`realloc`****.*

But be aware that **`realloc`** can easily obfuscate simple counting of allocations because if it is called with an existing object, it serves as deallocation (for the old object) and allocation (for the new one) at the same time.

The memory-allocation system is meant to be simple, and thus **`free`** is only allowed for pointers that have been allocated with **`malloc`** or that are null.

Takeaway 13.1.2 #3 *Only call* ***`free`*** *with pointers as they are returned by* ***`malloc`****,* ***`calloc`****,* ***`aligned_alloc`****, or* ***`realloc`****.*

They *must not*

- Point to an object that has been allocated by other means (that is, a variable or a compound literal)
- Have been freed yet
- 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, which is one of the worst types of crashes you can have. Be careful.

13.1.3. *Flexible array members.* In the previous discussion, the type `circular` showed a way to combine a dynamically allocated array (the `tab` member) with metadata (for example, the `len` member). C has another way to couple the information about an array more directly to the array itself, called a *flexible array member* (*FLA*). Such an array must be the last member of a structure and be defined as an incomplete array:

```
typedef struct ua32 ua32;
struct ua32 {
  size_t length;
  uint32_t data[];     // flexible array member
};
```

The idea is that the member `data` and `length` are held consistent so for a pointer value `ap` pointing to an object of that type, `ap->data` always represents an array of type `uint32_t[ap->length]`:

```
size_t len = 32;
size_t size = offsetof(ua32, data) + sizeof(uint32[len]);
// Adjust if the size is too small
if (size < sizeof(ua32)) {
  size = sizeof(ua32);
}
ua32* ap = calloc(size, 1);
// Ensure that the length member is consistent with the object size.
ap->length = len;
...
// Use ap->data mostly as an array
for (size_t i = 0; i < ap->length; i++) {
  printf("ap->data[%zu] is %w32u\n", i, ap->data[i]);
}
```

Observe that the flexible array member `data` may not sit at the very end after the structure itself but may have an offset that locates it inside the structure. Therefore, we always have to watch that we allocate enough storage so we can access the structure itself.

Takeaway 13.1.3 #1 *A structure object with a flexible array member must have enough storage to access the structure as a whole.*

Unfortunately, there are no standard tools that would allocate such a structure and then initialize the value of a member, such as `length`, automatically.

Takeaway 13.1.3 #2 *Consistency between a length member and a flexible array member must be maintained manually.*

13.2. Storage duration, lifetime, and visibility. We have seen in different places that the 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 13.1.

LISTING 13.1. An example of shadowing with local variables

```
1 #include "c23-fallback.h"
2
3 void squareIt(double* p) [[__unsequenced__]] {
4   *p *= *p;
5 }
6 int main(void) {
7   double x = 35.0;
```

LISTING 13.2. An example of shadowing with 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);     /* prints 1 */
10   } else {
11     printf("%u\n", i);     /* prints 2 */
12   }
13 }
```

```
8    double* xp = &x;
9    {
10     squareIt(&x);                /* Refers to double x */
11     ...
12     [[maybe_unused]] int x = 0; /* Shadow double x */
13     ...
14     squareIt(xp);                /* Valid use of double x */
15     ...
16   }
17   ...
18   squareIt(&x);                  /* Refers to double x */
19   ...
20 }
```

Here, the visibility scope of the identifier `x` declared in line 7 starts from that line and goes to the end of the function **main** but with a noticeable interruption: from line 12 to 16, this visibility is ***shadowed***^{C} by another variable, also named `x`.

Takeaway 13.2 #1 *Identifiers only have visibility inside their scope, starting at their declaration.*

Takeaway 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 the visibility of an identifier and the usability of the object 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 where the function is defined. Then, on line 14, 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;[40] see listing 13.2.

This program has three declarations for variables named `i` but only two definitions: the declaration and definition on line 6 shadows the one on line 3. In turn, declaration line 8 shadows line 6, but it refers to the same object as the object defined on line 3.[Exs 41]

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

[Exs 41] Which value is printed by this program?

Takeaway 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:

```
1 char const A[] = { 'e', 'n', 'd', '\0', };
2 char const B[] = { 'e', 'n', 'd', '\0', };
3 char const* c = "end";
4 char const* d = "end";
5 char const* e = "friend";
6 char const* f = (char const[]){ 'e', 'n', 'd', '\0', };
7 char const* g = (char const[]){ 'e', 'n', 'd', '\0', };
```

But how many distinct array objects are there in total? It depends. The compiler has a lot of choices.

Takeaway 13.2 #4 *Read-only object literals may overlap.*

In the previous example, we have three string literals and two compound literals. These are all object literals, and they are read-only: string literals are read-only by definition, and the two compound literals are **const**-qualified. Four of them have exactly the same base type and content (`'e', 'n', 'd', '\0'`), 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 these examples, the usability of an object not only is a lexical property of an identifier or 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 start point and an end point.

Takeaway 13.2 #5 *Objects have a lifetime outside of which they can't be accessed.*

Takeaway 13.2 #6 *A program execution that refers to an object outside of its lifetime fails.*

How the start and end points of an object are defined depends on the tools 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 runtime, ***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.

Table 13.1 gives an overview of the complicated relationship between declarations and their *storage classes*, initialization, linkage, *storage duration*, and lifetime. Without going into too much detail for the moment, it shows that the usage of keywords and the underlying terminology are quite confusing.

First, unlike what the name suggests, the *storage class* **extern** may refer to identifiers with external or internal *linkage*.[42] Here, in addition to the compiler, an identifier with linkage is usually managed by another external program, the ***linker***^{C}. Such an identifier is initialized at the startup of the program, even before it enters **main**, and the linker ensures that. Identifiers accessed from different object files need *external* linkage so they all access the same object or function and the linker is able to establish the correspondence.

Important identifiers with external linkage we have seen are the functions of the C library. They reside in a system ***library***^{C}, usually called something like `libc.so`, and not in the object file you created. Otherwise, a global, file scope, object, or function

[42] Note that linkage is a property of identifiers, not of the objects they represent.

without connection to other object files should have *internal* linkage. All other identifiers have *no* linkage.[43]

Then, static *storage duration* is not the same as declaring a variable with the *storage class* **static**. The latter is merely enforcing that a variable or function has internal linkage. Such a variable may be declared in file scope (global) or a block scope (local).
[44]You probably have not yet called the linker of your platform explicitly. Usually, its execution is hidden behind the compiler frontend that you are calling, and a dynamic linker may only kick in as late as program startup without being noticed.

For the first three types of storage duration, we have seen a lot of examples. Thread storage duration (**thread_local** since C23; previously **_Thread_local**) is related to C's thread API, which we will see later, in section 20.

Allocated storage duration is straightforward: the lifetime of such an object starts from the corresponding call to **malloc**, **calloc**, **realloc**, or **aligned_alloc** that creates it. It ends with a call to **free** or **realloc** that destroys it or, if no such call is issued, at the end of the program execution.

[43]A better keyword for **extern** would perhaps be **linkage**.

[44]A better keyword for **static** in this context would perhaps be **internal**, with the understanding that any form of linkage implies static storage duration.

TABLE 13.1. Storage classes, scope, linkage of identifiers, and storage duration of the associated objects. *Tentative* indicates that a definition is implied only if there is no other definition with an initializer. *Induced* indicates that the linkage is internal if another declaration with internal linkage has been met prior to that declaration; otherwise, it is external. Compound literals follow similar rules with respect to the block in which they occur, if any.

Class	Scope	Definition	Linkage	Duration	Lifetime
Initialized	File	Yes	External	Static	Whole execution
extern, initialized	File	Yes	External	Static	Whole execution
String literal	Any	Yes	N/A	Static	Whole execution
static, initialized	Any	Yes	Internal	Static	Whole execution
constexpr	Any	Yes	Internal	Static	Whole execution
Uninitialized	File	Tentative	External	Static	Whole execution
extern, uninitialized	Any	No	Induced	Static	Whole execution
static, uninitialized	Any	Tentative	Internal	Static	Whole execution
thread_local	File	Yes	External	Thread	Whole thread
extern thread_local	Any	No	External	Thread	Whole thread
static thread_local	Any	Yes	internal	Thread	Whole thread
Non-VLA			None		
Non-VLA, **auto**	Block	Yes	None	Automatic	Block of definition
register			None		
VLA	Block	Yes	None	Automatic	From definition to end of block
Function **return** with array	Block	Yes	None	Automatic	To the end of expression

The two other cases of storage duration need additional explanation, so we will discuss them in more length next. Since C23, compound literals can also have storage classes in their type. We have a relatively simple rule.

Takeaway 13.2 #7 *A compound literal has the same lifetime as a variable that would be declared with the same storage class within the same context.*

The only possible exception is compound literals that are **const**-qualified and appear without any storage class specification in block scope. These need not even correspond to a unique object and can indeed refer to storage that is available during the whole program execution.

13.2.1. *Static storage duration.* Objects with static storage duration can be defined in different ways:

- Objects *defined* in file scope and not declared with **thread_local**. Variables and compound literals[45] can have that property.
- Variables and compound literals defined inside a block and have the storage class specifier **static** and no additional **thread_local**.
- String literals, which are arrays of **char** or a wide character type and always have static storage duration.

Such objects have a lifetime that spans the entire 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. Here's an example:

```
1 double A = 37;
2 double* p = &(static double){ 1.0, };
3 int main(void) {
4   static double 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**, and one has type **double***.

All four objects are properly initialized from the start; three are initialized explicitly, and B is initialized implicitly with 0.

Takeaway 13.2.1 #1 *Objects with static storage duration are always initialized.*

The initialization of p is an example that needs a bit more magic than the compiler itself can offer. It uses the address of another object. Such an address can usually only be computed when the execution starts. This is why most C implementations need the concept of a linker, as we discussed earlier.

The example of B shows that the name of an object with a lifetime that spans the entire program execution isn't necessarily visible in the entire program. The **extern** example 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: rules for automatic storage duration are implicit and, therefore, need the most explanation. Several cases of objects can be defined explicitly or implicitly and fall into this category:

- Any block scope variables and compound literals that are not declared **static**, declared without storage class, with the legacy storage class **auto**[46]or with **register** or **constexpr**.

[45] Since C23.

[46] Beware: this keyword also plays a role in type inference, which we will see in full in section 18.

- Some temporary objects that are returned by function calls

The simplest and most common case for the lifetime of automatic objects is when the object is not a variable-length array (VLA).

Takeaway 13.2.2 #1 *Unless automatic objects are VLA or temporary objects, they have a lifetime corresponding to the execution of their block of definition.*

That is, most local variables are created when program execution enters the scope in which they are defined, and they are destroyed when the program leaves that scope. But, because of recursion, several ***instances***^{C} of the same object may exist at the same time.

Takeaway 13.2.2 #2 *Each recursive call creates a new local instance of an automatic object.*

Objects with automatic storage duration have a big advantage for optimization: the compiler usually sees the full usage of such a variable and, with this information, is able to decide if it may be an alias. This is where the difference between the legacy **`auto`**, **`constexpr`**, and **`register`** variables come into play.

Takeaway 13.2.2 #3 *The `&` operator is not allowed for objects declared with **`register`**.*

With that, we can't inadvertently take the address of a **`register`** object (takeaway 12.3 #2). As a simple consequence, we get the following rule.

Takeaway 13.2.2 #4 *Objects declared with **`register`** can't alias.*

So, with **`register`** declarations, the compiler can be forced to tell us where we are taking the address, and we may identify spots with some optimization potential. This works well for all objects that are not arrays and that contain no arrays.

Takeaway 13.2.2 #5 *Declare local variables that are not arrays in performance-critical code as **`register`**.*

Arrays play a particular role here because they decay to the address of their first element in almost all contexts. So, for arrays, we need to be able to take addresses.

Takeaway 13.2.2 #6 *Arrays with storage class **`register`** are useless.*

There is another case where the presence of arrays needs special treatment. Some return values of functions can really be chimeras: objects with *temporary lifetime*. As you know now, functions normally return values, and as such, values are not addressable. But if the return type *contains* an array type, we must be able to take the address implicitly, so the `[]` operator is well defined. Therefore, the following function return is a temporary object, of which we may implicitly take an address by using the member designator `.ory[0]`:

```
1 struct demo { unsigned ory[1]; };
2 struct demo mem(void);
3
4 printf("mem().ory[0] is %u\n", mem().ory[0]);
```

The only reason objects with temporary lifetime exist in C is to access members of such a function return value. Don't use them for anything else.

Takeaway 13.2.2 #7 *Objects of temporary lifetime are read-only.*

Takeaway 13.2.2 #8 *Temporary lifetime ends at the end of the enclosing full expression.*

That is, their life ends as soon as the evaluation of the expression in which they occur is terminated. For example, in the previous example, the temporary object ceases to exist as soon as the argument for **printf** is constructed. Compare this to the definition of a compound literal: a compound literal would live on until the enclosing scope of the **printf** terminates.

13.3. Digression: using objects before their definition. The following section goes into more detail about how automatic objects spring to life (or not). It is a bit tough, so if you are not up to it right now, you might skip it and come back to it later. It will be needed to understand subsection 13.5 about concrete machine models, but that subsection is a digression too. Also, it introduces the new features **goto** and labels, which we need later, in subsection 15.6 for handling errors.

Let us get back to the rule for the lifetime of ordinary automatic objects (takeaway 13.2.2 #1). It is quite particular if you think about it: the lifetime of such an object starts when its scope of definition is entered, not, as one would perhaps expect, later, when its definition is first encountered during execution.

To note the difference, let us look at listing 13.3, which is a variant of an example in the C standard document.

Listing 13.3. A contrived example for the use of a compound literal

```
4  void fgoto(unsigned n) {
5    unsigned j = 0;
6    unsigned* p = nullptr;
7    unsigned* q;
8   AGAIN:
9    if (p) printf("%u: p and q are %s, *p is %u\n",
10                 j,
11                 (q == p) ? "equal" : "unequal",
12                 *p);
13   q = p;
14   p = &((unsigned){ j, });
15   ++j;
16   if (j <= n) goto AGAIN;
17 }
```

We are particularly interested in the lines printed if this function is called as `fgoto(2)`. On my computer, the output looks like this:

Terminal

```
0  1: p and q are unequal, *p is 0
1  2: p and q are equal, *p is 1
```

Admittedly, this code is a bit contrived. It uses a new construct we haven't yet seen in action, **goto**. As the name indicates, this is a ***jump statement***^{C}. In this case, it instructs the computer to continue execution at ***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 in line 14.

So, let us look at what happens with the **printf** call during execution. For `n == 2`, execution meets the corresponding line three times; but because `p` is initially null, at the

j	p	q	printf
0	0	Undetermined	Skipped
1	Addr of literal of $j = 0$	0	printed
2	Addr of literal of $j = 1$	Addr of literal of $j = 0$	printed

first passage, the **printf** call itself is skipped. The values of our three variables in that line are

Here, we see that for `j==2` pointers, `p` and `q` hold addresses obtained at different iterations. So why, then, does my printout say that both addresses are equal? Is this just a coincidence? Or is there undefined behavior because I am using the compound literal lexically at a place before it is defined?

The C standard prescribes that the output shown here *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 another way, in this example, 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`.

Takeaway 13.3 #1 *For an object that is not a VLA, lifetime starts when the scope of the definition is entered, and it ends when that scope is left.*

Takeaway 13.3 #2 *Initializers of automatic variables and compound literals are evaluated each time the definition is met.*

In this example, the compound literal is visited three times and set to the values `0`, `1`, and `2` in turn.

For a VLA, the lifetime is given by a different rule.

Takeaway 13.3 #3 *For a VLA, lifetime starts when the definition is encountered and ends when the visibility scope is left.*

So, for a VLA, our strange trick of 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 are still inside the same block. The reason for this special treatment of VLAs 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.

13.4. Initialization. In subsection 5.5, we discussed the importance of initialization. It is crucial to guarantee that a program starts in a well-defined state and stays so throughout execution. The storage duration of an object determines how it is initialized.

Takeaway 13.4 #1 *Objects of static or thread-storage duration are initialized by default.*

As you probably recall, such a default initialization is the same as initializing all members of an object by `0` (for arithmetic types) or **nullptr** (for pointer types). In particular, default initialization works well for base types that might have a nontrivial representation for their `{ }` value—namely, pointers and floating point types.

For other objects, automatic or allocated, we must do something.

Takeaway 13.4 #2 *Objects of automatic or allocated storage duration must be initialized explicitly.*

The simplest way to achieve initialization is by using initializers, which put variables and compound literals in a well-defined state as soon as they become visible. Since C23, this is even possible for arrays that we allocate as VLA if we use the default initialization of the array as a whole. For arrays that we allocate dynamically and for which we know that setting all bits to zero is a valid initialization (basically all integer types), we should use **calloc** instead of **malloc** to do the allocation. Where this is not possible (types that contain floating point data or pointers), 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 maintain because the initialization parts may visually separate definition and use. The easiest way to avoid this is to encapsulate initialization into functions.

Takeaway 13.4 #3 *Systematically provide an initialization function for each of your data types.*

Here, the emphasis is on *systematically*: you should have a consistent convention for how such initializing functions should work and how they should be named. To see that, let us go back to `rat_init`, the initialization function for our `rat` data type. It implements a specific API for such functions:

- For a type `toto`, the initialization function is named `toto_init`.
- The first argument to such a `_init` function is the pointer to the object to be initialized.
- If that pointer to an object is null, the function does nothing.
- Other arguments can be provided to pass initial values for certain members.
- The function returns the pointer to the object it received or a null pointer if an error occurred.

With these properties, such a function can be used easily in an initializer for a pointer:

```
rat const* myRat = rat_init(malloc(sizeof(rat)), 13, 7);
```

This has several advantages:

- If the call to **malloc** fails by returning a null pointer, the only effect is that `myRat` is initialized to null. Thus `myRat` is always in a well-defined state.
- If we don't want the object to be changed afterward, we can qualify the pointer target as **const** from the start. All modification of the new object happens inside the initialization expression on the right side.

Since such initialization can then appear in many places, we can also encapsulate this into another function:

```
1 rat* rat_new(long long numerator,
2              unsigned long long denominator) {
3   return rat_init(malloc(sizeof(rat)),
4                   numerator,
5                   denominator);
6 }
```

The initialization using that function becomes

```
rat const* myRat = rat_new(13, 7);
```

Macro addicts like myself 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 earlier initialization as

```
rat const* myRat = P99_NEW(rat, 13, 7);
```

This has the advantage of being at least as readable as the `rat_new` variant, but it avoids the additional declaration of such a function for all types that we define.

Such macro definitions are frowned upon by many, so some projects probably will not accept this as a general strategy, but you should at least be aware that the 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` melds 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 passed after the first is accessible inside the macro expansion as **__VA_ARGS__**. That way, we can pass any number of arguments as required by the corresponding `_init` function to `P99_NEW`.

If we have to initialize arrays using a **for** loop, things get even uglier. Here also, it is easy to encapsulate with a function:

```
1 rat* rat_vinit(size_t n, rat p[n]) {
2   if (p)
3     for (size_t i = 0; i < n; ++i)
4       rat_init(p+i, 0, 1);
5   return p;
6 }
```

With such a function, again, initialization becomes straightforward:

```
rat* myRatVec = rat_vinit(44, malloc(sizeof(rat[44])));
```

Here, encapsulation into a function is better since repeating the size can easily introduce errors:

```
1 rat* rat_vnew(size_t size) {
2   return rat_vinit(size, malloc(sizeof(rat[size])));
3 }
```

13.5. 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 is an optional digression that deviates from that: it is a glimpse into the machine model of a concrete architecture. We will see in more detail how a simple function is translated into this model and, in particular, how automatic storage duration is realized. If you really can't bear it yet, you may skip it for now. Otherwise, remember not to panic and dive in.

Traditionally, computer architectures were described with the von Neumann model.[47] 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.

[47] Invented around 1945 by J. Presper Eckert and John William Mauchly for the ENIAC project; first described by John von Neumann (1903 –1957, also known as Neumann János Lajos and Johann Neumann von Margitta), one of the pioneers of modern science, in von Neumann [1945].

The intermediate programming languages usually used to describe machine instructions as they are understood by your CPU are collectively called ***assembler***^{C} or ***assembly***^{C} languages, and they still pretty much build upon the von Neumann model. There is not one unique assembly language (like C, which is valid for all platforms) but an entire set of *dialects* that take different particularities into account–including the CPU, the compiler, and the operating system. The one we use here is the one used by the `gcc` compiler for the `x86_64` processor architecture.[Exs 48] If you don't know what that means, don't worry; this is just an example of one such architecture.

Listing 13.4 shows an assembler printout for the function `fgoto` from listing 13.3. Such assembler code operates with ***instructions***^{C} on hardware registers and memory locations. For example, the line `movl $0, -16(%rbp)` stores (moves) the value `0` to the location in memory `16` bytes below the one indicated by register `%rbp`. The assembler program also contains ***labels***^{C} that identify certain points in the program. For example, `fgoto` is the ***entry point***^{C} of the function, and `.L_AGAIN` is the counterpart in assembler to 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 the following hardware registers:

`%eax` `%ecx` `%edi` `%edx` `%esi` `%rax` `%rbp` `%rcx` `%rdx` `%rsp`

This list includes 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 of the state of a program's execution. Two others have very special roles: `%rbp` (base pointer) and `%rsp` (stack pointer).

The function disposes of a reserved area in memory, often called ***the stack***^{C}, which 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. For example, 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 stored there at three different points of the execution of the function.

...`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 for learning about automatic variables and how they 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 previous assembler code (listing 13.4) 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 it decrements `%rsp` by `48`. Here, `48` is the number of bytes the compiler has computed for all automatic objects that the `fgoto` needs. Because of this simple type of setup, the space 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.

[Exs 48] Find out which compiler arguments produce assembler output for your platform.

LISTING 13.4. An assembler version of the **fgoto** function

```
10          .type   fgoto, @function
11  fgoto:
12          pushq   %rbp                # Save base pointer
13          movq    %rsp, %rbp          # Load stack pointer
14          subq    $48, %rsp           # Adjust stack pointer
15          movl    %edi, -36(%rbp)     # fgoto#0 => n
16          movl    $0, -4(%rbp)        # init j
17          movq    $0, -16(%rbp)       # init p
18  .L_AGAIN:
19          cmpq    $0, -16(%rbp)       # if (p)
20          je      .L_ELSE
21          movq    -16(%rbp), %rax     #  p ==> rax
22          movl    (%rax), %edx        # *p ==> edx
23          movq    -24(%rbp), %rax     # (   == q)?
24          cmpq    -16(%rbp), %rax     # (p  ==  )?
25          jne     .L_YES
26          movl    $.L_STR_EQ, %eax    # Yes
27          jmp     .L_NO
28  .L_YES:
29          movl    $.L_STR_NE, %eax    # No
30  .L_NO:
31          movl    -4(%rbp), %esi      # j     ==> printf#1
32          movl    %edx, %ecx          # *p    ==> printf#3
33          movq    %rax, %rdx          # eq/ne ==> printf#2
34          movl    $.L_STR_FRMT, %edi  # frmt  ==> printf#0
35          movl    $0, %eax            # clear eax
36          call    printf
37  .L_ELSE:
38          movq    -16(%rbp), %rax     # p ==|
39          movq    %rax, -24(%rbp)     #      ==> q
40          movl    -4(%rbp), %eax      # j ==|
41          movl    %eax, -28(%rbp)     #      ==> cmp_lit
42          leaq    -28(%rbp), %rax     # &cmp_lit ==|
43          movq    %rax, -16(%rbp)     #             ==> p
44          addl    $1, -4(%rbp)        # ++j
45          movl    -4(%rbp), %eax      # if (j
46          cmpl    -36(%rbp), %eax     #       <= n)
47          jbe     .L_AGAIN            # goto AGAIN
48          leave                       # Rearange stack
49          ret                         # return statement
```

After this setup, the function is ready to go. In particular, it can easily call another function: **%rsp** now points to the top of a new memory area that 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 the function is supposed to receive in registers **%edi**, **%esi**, **%ecx**, **%rdx**, in that order. It then clears **%eax** and calls the function.

To summarize, the setup of a memory area for the automatic objects (without VLA) of a function only needs a few instructions, regardless of how many automatic objects are effectively used by the function. If the function had more, the magic number `48` would need to be modified to the new size of the area.

As a consequence of the way this is done,

Listing 13.5. An optimized assembler version of the `fgoto` function

```
12          .type   fgoto, @function
13  fgoto:
14          pushq   %rbp                # Save base pointer
15          pushq   %rbx                # Save rbx register
16          subq    $8, %rsp            # Adjust stack pointer
17          movl    %edi, %ebp          # fgoto#0 => n
18          movl    $1, %ebx            # init j, start with 1
19          xorl    %ecx, %ecx          # 0    ==> printf#3
20          movl    $.L_STR_NE, %edx    # "ne" ==> printf#2
21          testl   %edi, %edi          # if (n > 0)
22          jne     .L_N_GT_0
23          jmp     .L_END
24  .L_AGAIN:
25          movl    %eax, %ebx          # j+1  ==> j
26  .L_N_GT_0:
27          movl    %ebx, %esi          # j    ==> printf#1
28          movl    $.L_STR_FRMT, %edi  # frmt ==> printf#0
29          xorl    %eax, %eax          # Clear eax
30          call    printf
31          leal    1(%rbx), %eax       # j+1  ==> eax
32          movl    $.L_STR_EQ, %edx    # "eq" ==> printf#2
33          movl    %ebx, %ecx          # j    ==> printf#3
34          cmpl    %ebp, %eax          # if (j <= n)
35          jbe     .L_AGAIN            # goto AGAIN
36  .L_END:
37          addq    $8, %rsp            # Rewind stack
38          popq    %rbx                # Restore rbx
39          popq    %rbp                # Restore rbp
40          ret                         # return statement
```

- Automatic objects are usually available from the start of a function or scope.
- Initialization of automatic *variables* is not enforced.

This does a good job of mapping the rules for the lifetime and initialization of automatic objects in C.

The earlier assembler output is only half the story, at most. It was produced without optimization to show the principle assumptions that can be made for such code generation. When using optimization, the as-if rule (takeaway 5.1.3 #3) allows us to reorganize the code substantially. With full optimization, my compiler produces something like listing 13.5.

As you can see, the compiler has completely restructured the code. This code reproduces the *effects* that the original code had: its output is the same as before. But it doesn't use objects in memory, doesn't compare pointers for equality, and has no trace of the compound literal. For example, it doesn't implement the iteration for `j=0` at all. This iteration has no effect, 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 has to increment `j` and set up the arguments for `printf` accordingly.[Exs 49][Exs 50]

[Exs 49] Using the fact that `p` is assigned the same value over and over again, write a C program that gets closer to what the optimized assembler version looks like.

[Exs 50] Even the optimized version leaves room for improvement: the inner part of the loop can still be shortened. Write a C program that explores this potential when compiled with full optimization.

All we have seen here is code that doesn't use VLA. These change the picture. The trick that simply modifies `%rsp` with a constant doesn't work if the needed memory is not a constant size. For a VLA, the program has to compute the size during execution from the actual values of the bounds of the VLA, it has to adjust `%rsp` accordingly there and then undo the modification of `%rsp` once execution leaves the scope of the definition of the VLA. So here, the value of adjustment for `%rsp` cannot be computed at compile time but must be determined during the execution of the program.

Summary

- Storage for a large number of objects or for objects that are large in size can be allocated and freed dynamically. We have to keep track of this storage carefully.
- Identifier visibility and storage duration are different things.
- Initialization must be done systematically with a coherent strategy for each type.
- C's allocation strategy for local variables maps well to low-level handling of function stacks.

14. More involved processing and IO

This section covers

- Working with pointers
- Formatting input
- Handling extended character sets
- Input and output with binary streams
- Checking errors and cleaning up

Now that we know about pointers and how they work, we will shed new light on some C library features. C's text processing is incomplete without pointers, so we will start this section with an elaborated example in section 14.1. Then we will look at functions for formatted input (section 14.1); these require pointers as arguments, so we necessarily delayed their presentation until now. A whole new series of functions is then presented to handle extended character sets (section 14.3) and binary streams (section 14.6). We round out this section and the entire level with a discussion of clean error handling (section 14.6)).

s

14.1. Text processing. As a first example, consider the following program, which reads a series of lines, each with several numbers from **stdin**, and writes these same numbers in a normalized way to **stdout** as comma-separated hexadecimal numbers:

. numberline.c

```
int main(void) {
  char lbuf[256];
  for (;;) {
    if (fgetline(sizeof lbuf, lbuf, stdin)) {
      size_t n;
      size_t* nums = numberline(strlen(lbuf)+1, lbuf, &n, 0);
      if (nums) {
        int ret = fprintnumbers(stdout, "%#zX", ",\t", n, nums);
        if (ret < 0) return EXIT_FAILURE;
        free(nums);
      }
    } else {
      if (lbuf[0]) {  /* a partial line has been read */
        for (;;) {
          int c = getc(stdin);
          if (c == EOF) return EXIT_FAILURE;
          if (c == '\n') {
            fprintf(stderr, "line too long: %s\n", lbuf);
            break;
          }
        }
      } else break;   /* regular end of input */
    }
  }
}
```

This program splits the job into three different tasks:

- `fgetline` to read a line of text
- `numberline` to split such a line in a series of numbers of type **size_t**

- `fprintnumbers` to print them

At the heart is the function `numberline`. It splits the `lbuf` string that it receives into numbers, allocates an array to store them, and returns the count of these numbers through the pointer parameter `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** and similar functions

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 static size],
                   size_t*restrict np, int base);
```

That function itself is split into two parts, which perform quite different tasks. One performs the task of interpreting the line, `numberline_inner`. The 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.

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**:

. numberline.c

```
100 static
101 size_t numberline_inner(char const*restrict act,
102                          size_t numb[restrict static 1],
103                          int base){
104   size_t n = 0;
105   for (char* next = nullptr; act[0]; act = next) {
106     numb[n] = strtoull(act, &next, base);
107     if (act == next) break;
108     ++n;
109   }
110   return n;
111 }
```

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 the 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	30,874	'\0'
0	2	7	'8'

Remember the special rules for base 0. The effective base is deduced from the first (or first two) characters in the string. Here, the first character is a '0', so the string is interpreted as being octal, and parsing stops at the first non-digit for that base: '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 the **if-break** condition inside.

Note that the C library function **strtoull** has a historical 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.

Takeaway 14.1 #1 *The string strto... conversion functions are not **const**-safe.*

Now, the function numberline itself provides the glue around numberline_...inner:

- If np is null, it is set to point to an auxiliary.
- The input string is checked for validity.
- An array with enough elements to store the values is allocated and tailored to the appropriate size once the correct length is known.

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 a null pointer if there is none. Here, this is just used to check that within the first size bytes, there effectively is a 0 character. That way, it guarantees that all the string functions used underneath (in particular, **strtoull**) operate on a 0-terminated string.

Before C23 **memchr** was a problematic interface. There only was a function with that name that returns a **void*** that potentially points to a read-only object.

Takeaway 14.1 #2 *The function interfaces for **memchr** and **strchr** search functions are not **const**-safe.*

C23 added type-generic macro interfaces that solve this defect. We will discuss these type-generic interfaces in more detail in section 18.1.7.

Takeaway 14.1 #3 *The type-generic interfaces for **memchr** and **strchr** search functions are **const**-safe.*

In contrast, functions that return an index position within the string would be safe.

Takeaway 14.1 #4 *The **strspn** and **strcspn** search functions are **const**-safe.*

. numberline.c

```
113 size_t* numberline(size_t size,
114                    char const lbuf[restrict static size],
115                    size_t*restrict np, int base){
116   size_t* ret = nullptr;
117   size_t n = 0;
118   /* Check for validity of the string, first. */
119   if (memchr(lbuf, 0, size)) {
120     /* The maximum number of integers encoded.
121        To see that this may be as much look at
122        the sequence 08 08 08 08 ... and suppose
123        that base is 0. */
124     ret = malloc(sizeof(size_t[1+(2*size)/3]));
125     if (!ret) return nullptr;
126     n = numberline_inner(lbuf, ret, base);
127
128     size_t len = n ? n : 1;
129     size_t* ret2 = realloc(ret, sizeof(size_t[len]));
130     if (ret2) ret = ret2;
131   }
132   if (np) *np = n;
133   return ret;
134 }
```

Unfortunately, they have the disadvantage that they can't be used to check whether a **char** array is, in fact, a string. So, they can't be used here.

Now, let us look at the second function in our example:

numberline.c

`fgetline`:

read one text line of at most `size-1` bytes.

The `'\n'` character is replaced by `0`.

Returns: s if an entire line was read successfully. Otherwise, `0` is returned and *s* contains a maximal partial line that could be read. *s* is null terminated.

```
char* fgetline(size_t size,
               char s[restrict size],
               FILE*restrict stream);
```

This is quite similar to the C library function **fgets**. The first difference is the interface: the parameter order is different, and the `size` parameter is a **size_t** instead of an **int**. Like **fgets**, it returns a null pointer if the read from the stream fails. Thus, the end-of-file condition is easily detected on `stream`.

More important is that `fgetline` handles another critical case more gracefully. It detects whether the next input line is too long or whether the last line of the stream ends without a `'\n'` character:

. numberline.c

```
136 char* fgetline(size_t size,
137                char s[restrict size],
138                FILE*restrict stream){
139   s[0] = 0;
140   char* ret = fgets(s, size, stream);
141   if (ret) {
142     /* Uses non-const variant of strchr */
143     char* pos = strchr(s, '\n');
144     if (pos) *pos = 0;
145     else ret = nullptr;
146   }
147   return ret;
148 }
```

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 is read, the first `'\n'` character found in `s` is replaced with `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 51]

In addition to **fgets**, this uses **strchr** from the C library. Since C23, the lack of **const** safeness of the function interface is not an issue; the type-generic interface gives the appropriate guarantees.

Since it involves a lot of detailed error handling, we will go into more depth about the function `fprintnumbers` in subsection 15.6. For our purpose here, we restrict ourselves to the discussion of function `sprintnumbers`, which 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 static len
    ]);
```

The 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** and **fprintf**, only it doesn't print to a stream but rather to a **char** array:

The function **sprintf** always ensures that a `0` character is placed at the end of the string. It also returns the length of that string, which is the number of characters before

[Exs 51] Improve the **main** of the example such that it is able to cope with arbitrarily long input lines.

. numberline.c

```
155 int sprintnumbers(size_t tot, char buf[restrict tot],
156                  char const form[restrict static 1],
157                  char const sep[restrict static 1],
158                  size_t len, size_t nums[const restrict static
    len]) {
159   char* p = buf;   /* next position in buf */
160   size_t const seplen = strlen(sep);
161   if (len) {
162     size_t i = 0;
163     for (; i < len;) {
164       p += sprintf(p, form, nums[i]);
165       ++i;
166       if (i >= len) break;
167       memcpy(p, sep, seplen);
168       p += seplen;
169     }
170   }
171   memcpy(p, "\n", 2);
172   return (p-buf)+1;
173 }
```

the 0 character that has been written. This is used in the example to update the pointer to the current position in the buffer. **sprintf** still has an important vulnerability.

Takeaway 14.1 #5 ***sprintf** makes no provision against buffer overflow.*

That is, if we pass an insufficient buffer as a first argument, bad things will happen. Here, inside sprintnumbers, much like **sprintf** itself, we *suppose* the buffer is large enough to hold the result. If we aren't sure the buffer can hold the result, we can use the C library function **snprintf**, instead:

```
int snprintf(char*restrict s, size_t n, char const*restrict form,
     ...);
```

This function ensures additionally that no more than n bytes are ever written to s. If the return value is greater than or equal to n, the string is truncated to fit. In particular, if n is 0, nothing is written into s.

Takeaway 14.1 #6 *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 set to 0 and s set to a null pointer, **snprintf** 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 like this:

numberline.c

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

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

CHALLENGE 15 (Text processing in strings). *We've covered quite a bit about text processing, so let's see if we can actually use it.*
Can you search for a given word in a string?
Can you replace a word in a string and return a copy with the new contents?
Can you implement some regular expression-matching functions for strings? For example, find a character class such as `[A-Q]` or `[^0-9]` and match with `*` (meaning "anything") or `?` (meaning "any character").
Or can you implement a regular expression-matching function for POSIX character classes such as `[[:alpha:]]`, `[[:digit:]]`, and so on?
Can you stitch all these functionalities together to search for a regexp in a string?
Do query-replace with regexp against a specific word?
Extend a regexp with grouping?
Extend query-replace with grouping?

14.2. Formatted input. Similar to the **printf** family of functions for formatted output, the C library has a series of functions for formatted input: **fscanf** for input from an arbitrary stream, **scanf** for **stdin**, and **sscanf** from a string. For example, the following would read a line of three **double** values from **stdin**:

```
1 double a[3];
2 /* Read and process an entire line with three double values. */
3 if (scanf("␣%lg␣%lg␣%lg␣", &a[0], &a[1], &a[2]) < 3) {
4    printf("not␣enough␣input␣values!\n");
5 }
```

Tables 14.1 to 14.3 give an overview of the format for specifiers. Unfortunately, these functions are more difficult to use than **printf** and also have conventions that diverge from **printf** in subtle ways.

- To be able to return values for all formats, the arguments are pointers to the type that is scanned.

TABLE 14.1. Format specifications for **scanf** and similar functions, with the general syntax `[XX][WW][LL]SS`

	Name	Description
XX	*	Assignment suppression
WW	Field width	Maximum number of input characters
LL	Modifier	Select width of target type
SS	Specifier	Select conversion

TABLE 14.2. Format specifiers for **scanf** and similar functions. Binary support was introduced with C23. 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 subection 14.3.

SS	Conversion	Pointer to	Skip space	Analogous to function
`'d'`	Decimal	Signed type	Yes	**strtol**, base `10`
`'i'`	Binary, octal, decimal, or hex	Signed type	Yes	**strtol**, base `0`
`'b'`	Binary	Unsigned type	Yes	**strtoul**, base `2`
`'o'`	Octal	Unsigned type	Yes	**strtoul**, base `8`
`'u'`	Decimal	Unsigned type	Yes	**strtoul**, base `10`
`'x'`	Hexadecimal	Unsigned type	Yes	**strtoul**, base `16`
`'aefg'`	Floating point	Floating point	Yes	**strtod**
`'%'`	`'%'` character	No assignment	No	
`'c'`	Characters	**char**	No	**memcpy**
`'s'`	Non-whitespace	**char**	Yes	**strcspn** with `"␣\f\n\r\t\v"`
`'['`	Scan set	String	No	**strspn** or **strcspn**
`'p'`	Address	**void**	Yes	
`'n'`	Character count	Signed type	No	

- Whitespace handling is subtle and sometimes unexpected. A space character, `' '`, in the format matches any sequence of whitespace: spaces, tabs, and newline characters. Such a sequence may for example be empty or even contain several newline characters.
- String handling is different. Because the arguments to the **scanf** functions are pointers anyway, the formats `"%c"` and `"%s"` both refer to an argument of type **char***. Whereas `"%c"` reads a character array of constant length (of default `1`), `"%s"` matches any sequence of non-whitespace characters and adds a terminating `0` character.
- The specifications of types in the format have subtle differences compared to **printf**, in particular for floating-point types. To be consistent between the two, it is best to use `"%lg"` or similar for **double** and `"%Lg"` for **long double** for both **printf** and **scanf**.
- There is a rudimentary utility to recognize character classes. For example, a format of `"%[aeiouAEIOU]"` can be used to scan for the vowels in 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 (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. For example, our seemingly simple example has the flaw (or feature) that it is not restricted to read a single input line, but it would happily accept three **double** values spread over several lines.[Exs 52] In most cases where you have a regular input pattern, such as a series of numbers, they are best avoided.

[Exs 52] 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.

TABLE 14.3. Format modifiers for **scanf** and similar functions. Note that **float** and **double** are handled differently than with **printf**.

Modifier	Type
"L"	**long double**
"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**
since C23, for N usually 8, 16, 32, 64 or 128	
"wN"	**uint**N**_t**, **int**N**_t**, **uint_least**N**_t** or **int_least**N**_t**
"wf$N$"	**uint_fast**N**_t** or **int_fast**N**_t**

14.3. Extended character sets. Up to now, we have used only a limited set of characters to specify our programs or the contents of string literals that we printed on the console— a set consisting of the Latin alphabet, Arabic numerals, and some punctuation characters. This limitation is a historical accident that originated 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.[53] As we saw with the use of the type name **char** for the basic data cell, the concepts of a text character and an indivisible data component were not very well separated at the start.

Latin, from which we inherited our character set, is long dead as a spoken language. Its character set is not sufficient to encode the particularities of the phonetics of other languages. Among the European languages, English has the peculiarity that it encodes missing sounds with combinations of letters such as *ai*, *ou*, and *gh* (fair enough), not with diacritical 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 is clearly not sufficient.

During the first years of market expansion around the world, different computer manufacturers, countries, and organizations provided native language support for their respective communities more or less randomly and added specialized support for graphical characters, mathematical typesetting, musical scores, and so on without coordination. It was an utter chaos. As a result, interchanging textual information between different systems, countries, and cultures was difficult, if not impossible, in many cases; writing portable code that could be used in the context of different languages *and* different computing platforms resembled the black arts.

Luckily, these years-long difficulties are now mainly mastered, and on modern systems, we can write portable code that uses "extended" characters in a unified way. The following code snippet shows how this is supposed to work:

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). Starting with the call to **setlocale** is important. Chances are, otherwise, you'd see garbage if you output characters from the extended set to your terminal. But

[53] The character encoding dominantly used for the basic character set is referred to as ASCII (American Standard Code for Information Interchange).

. mbstrings-main.c

```
87  setlocale(LC_ALL, "");
88  /* Multibyte character printing only works after the locale
89     has been switched. */
90  draw_sep(TOPLEFT "␣©␣2014␣jɛnz␣'gʊz,tɛt␣", TOPRIGHT);
```

once you have issued that call to **setlocale** and your system is well installed, such characters placed inside multibyte strings "fär␣íńǿ" should not work out too badly, see the following discussion.
The output of this looks similar to

```
┌──── © 2014 jɛnz 'gʊz,tɛt ──────────────────────────────┐
```

The means to achieve this are quite simple. We have some macros with magic string literals for vertical and horizontal bars and top-left and top-right corners,

. mbstrings-main.c

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

and an ad hoc function that nicely formats an output line,

mbstrings-main.c

draw_sep:

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

```
void draw_sep(char const start[static 1],
              char const end[static 1]) {
  fputs(start, stdout);
  size_t slen = mbsrlen(start, 0);
  size_t elen = 90 - mbsrlen(end, 0);
  for (size_t i = slen; i < elen; ++i) fputs(HBAR, stdout);
  fputs(end, stdout);
  fputc('\n', stdout);
}
```

This uses a function to count the number of print characters in a multibyte string (mbsrlen) and our old friends **fputs** and **fputc** for textual output.

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 `mbinvalid` 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 null indicates that *mbs* can be scanned without considering any context.

```
size_t mbsrlen(char const*restrict mbs,
               mbstate_t const*restrict state);
```

A *multibyte character* is a sequence of bytes that is 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.

Takeaway 14.3 #1 *Multibyte characters don't contain null bytes.*

Takeaway 14.3 #2 *Multibyte strings are null terminated.*

Thus, many of the standard string functions, such as **strcpy**, work out of the box for multibyte strings. They introduce one major difficulty, though: 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 previous code, we use the (nonstandard) function `mbsrlen`.

As you can see from the description, parsing multibyte strings for the individual multibyte characters can be a bit more complicated. In particular, we generally need to keep a parsing state using the type **mbstate_t** that is provided by the C standard in the header files `<wchar.h>`.[54] This header provides utilities for multibyte strings and characters and also for a *wide character* type **wchar_t**. These functions generally may return different codes that represent the current state of parsing.

<wchar.h>

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

mbstrings.h

mbcode:

The codes returned by multi-byte conversion functions.

In general these functions return a value of type **size_t**. Therefore the special codes have to be at the upper end of the range of that type.

This provides names for these constants as enumeration type. This is only possible since C23, because previously enumeration constants had been limited to **signed int**.

Enumerator:

mbinvalid	An invalid encoding was encountered.
mbincomplete	The encoding was incomplete.
mbstored	A previously detect output character was stored.

```
enum mbcode {
  //! An invalid encoding was encountered.
  mbinvalid    = (size_t)-1,
  //! The encoding was incomplete.
  mbincomplete = (size_t)-2,
  //! A previously detect output character was stored.
  mbstored     = (size_t)-3,
};
```

But first, we have to introduce another international standard: ISO 10646 (Unicode [2017]). It attempts to provide a unified framework for character codes (see `http://www.joelonsoftware.com/articles/Unicode.html`) by providing a huge table of basically all character *concepts* that have been conceived by humanity so far.[55] *Concept* here is really important: we have to understand from the print form or *glyph* of a particular character in a certain type that, for example, "Latin capital letter A" can appear as A, *A*, *A*, or `A` in the present text. Other such conceptual characters, like the character "Greek capital letter Alpha" may even be printed with the same or similar glyph "A".

Unicode places each character concept, or *code point* in its own jargon, into a linguistic or technical context. In addition to the definition of the character, Unicode classifies it—for example, as being a capital letter—and relates it to other code points, such as by stating that *A* is the capitalization of *a*.

If you need special characters for your particular language, there is a good chance that you have them on your keyboard and that you can enter them into multibyte strings for coding in C as is. That is, your system may be configured to insert the whole byte sequence for ä, say, directly into the text and do all the required magic for you. If you don't have or want that, you can use the technique we used for the macros `HBAR` earlier. There, we used an escape sequence that was new in C11 (`https://vycasas.github.io/2014/05/21/the-interesting-state-of-unicode-in-c`): a backslash and a *u*, followed by four hexadecimal digits encode a Unicode code point. For example, the code point for "latin small letter a with diaeresis" is 228 or **0x**`E4`. Inside a multibyte string, this then reads as `"\u00E4"`. Since four hexadecimal digits can address only 65,536 code points, there is also the option to specify eight hexadecimal digits, introduced with a backslash and a capital *U*, but you will encounter this only in very specialized contexts.

[55] Today, Unicode has about 110,000 code points.

In the previous example, we encoded four graphical characters with such Unicode specifications, characters that most likely are not on any keyboard. There are several online sites that allow you to find the code point for any character you need.

If we want to do more than simple input/output with multibyte characters and strings, things become a bit more complicated. Simple counting of the characters already is not trivial: **strlen** does not give the right answer, and other string functions such as **strchr**, **strspn**, and **strstr** don't work as expected. Fortunately, the C standard gives us a 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 earlier can be coded as

. mbstrings.c

```
30 size_t mbsrlen(char const*s, mbstate_t const*restrict state) {
31   state = state ? state : MBSTATE;
32   mbstate_t st = *state;
33   size_t mblen = mbsrtowcs(nullptr, &s, 0, &st);
34   if (mblen == mbinvalid) errno = 0;
35   return mblen;
36 }
```

The core of this function is the use of the library function **mbsrtowcs** (multibyte string [mbs], restartable, to wide character string [wcs]), which constitutes one of the primitives that the C standard provides to handle multibyte strings:

```
1 size_t mbsrtowcs(wchar_t*restrict dst, char const**restrict src,
2                  size_t len, mbstate_t*restrict ps);
```

So, once we decrypt the abbreviation of the name, we know that this function is supposed to convert an mbs, src, to a wcs, dst. Here, *wide characters* (wc) of type **wchar_t** are used to encode exactly one character of the extended character set. These wide characters are used to form wcs pretty much in the same way as **char**s compose ordinary strings: they are null-terminated arrays of such wide characters.

If state is null, mbsrlen also uses a macro MBSTATE (not shown) that provides an auxiliary buffer of type **mbstate_t**. Additionally, if the source string for the function isn't valid, this function might set **errno** to some error code. We are not interested in this, so we reset it to zero before returning.

The C standard doesn't restrict the encoding used for **wchar_t** much, but any sane environment nowadays should use Unicode for its internal representations. You can check this with two macros as follows:

. mbstrings.h

```
24 #ifndef __STDC_ISO_10646__
25 # warning "wchar_t␣wide␣characters␣have␣to␣be␣Unicode␣code␣points
     "
26 #endif
27 #ifdef __STDC_MB_MIGHT_NEQ_WC__
28 # warning "basic␣character␣codes␣must␣agree␣on␣char␣and␣wchar_t"
29 #endif
```

Modern platforms typically implement **wchar_t** with either 16- or 32-bit integer types. Which one usually should not be of much concern to you if you only use the code points representable with four 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 to those we have seen for **char** and strings. For both, a prefix of `L` indicates a wide character or string. For example, `L'ä'` and `L'\u00E4'` are the same character, both of type **wchar_t**, and `L"b\u00E4"` is an array of three elements of type **wchar_t** that contains the wide characters `L'b'`, `L'ä'`, and `0`.

Classification of wide characters is done similarly to simple **char**. The header `<wctype.h>` provides the necessary functions and macros. <wctype.h>

Coming back to **mbsrtowcs**, this function parses the multibyte string `src` into snippets that correspond to *multibyte characters* (mbc) and assigns the corresponding code point to the wide characters in `dst`. The parameter `len` describes the maximal length the resulting wcs may have. The parameter `state` points to a variable that stores an eventual *parsing state* of the mbs; we will briefly discuss this concept a bit later.

As you can see, the function **mbsrtowcs** has two peculiarities. First, when called with a null pointer for `dst`, it doesn't store the wcs but only returns the size 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 `mbinvalid` and sets **errno** to the value **EILSEQ** (see `<errno.h>`). Part of the code for `mbsrlen` is a repair of that error strategy by setting **errno** to `0` again. <errno.h>

Let's now look at a second function that will help us 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, null 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 null indicates that *s* can be scanned without considering any context.

```
wchar_t* mbsrdup(char const*s, mbstate_t*restrict state);
```

This function returns a freshly allocated wcs with the same contents as the mbs `s` it receives on input. Other than for the `state` parameter, its implementation is straightforward:

. mbstrings.c

```
38 wchar_t* mbsrdup(char const*s, mbstate_t*restrict state) {
39   mbstate_t st =  state ? *state : *MBSTATE;
40   size_t mblen = mbsrlen(s, &st);
41   if (mblen == mbinvalid) return nullptr;
42   wchar_t* S = malloc(sizeof(wchar_t[mblen+1]));
43   /* We know that s converts well, so no error check */
44   if (S) mbsrtowcs(S, &s, mblen+1, state);
45   return S;
46 }
```

After determining the length of the target string, we use **malloc** to allocate space and **mbsrtowcs** to copy over the data.

To have 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 works on the target machine, we have to do some guesswork that helps us determine `len`. To encapsulate such a heuristic, we cook up the following interface. It has semantics similar to **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 `mbinvalid` 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 is passed to all calls to this or similar functions.

Remarks: *state* of null 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);
```

This function returns the number of bytes that were identified for the first multibyte character in the string or `mbinvalid` on error. **mbrtowc** has another possible return value, `mbincomplete`, 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 >= mbincomplete; maxlen *=
     2)
    len = mbrtowc(C, c, maxlen, state);
  if (len == mbinvalid) errno = 0;
  return len;
}
```

Here, **MB_LEN_MAX** is a standard value that is a good upper bound for `len` in most situations. Note that this procedure works even if the string ends correctly with a null character.

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 null 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 null indicates that *s* can be scanned without considering any context.

```
char const* mbsrwc(char const s[restrict static 1],
                   mbstate_t*restrict state,
                   wchar_t C, size_t occurrence);
```

. mbstrings.c

```
68 char const* mbsrwc(char const s[restrict static 1], mbstate_t*
       restrict state,
69                    wchar_t C, size_t occurrence) {
70   if (!C || C == WEOF) return nullptr;
71   state = state ? state : MBSTATE;
72   char const* ret = nullptr;
73
74   mbstate_t st = *state;
75   for (size_t len = 0; s[0]; s += len) {
76     mbstate_t backup = st;
77     wchar_t S = 0;
78     len = mbrtow(&S, s, &st);
79     if (!S) break;
80     if (C == S) {
81       *state = backup;
82       ret = s;
83       if (!occurrence) break;
84       --occurrence;
85     }
86   }
87   return ret;
88 }
```

14.4. UTF character encodings. As we said, all of this encoding with multibyte strings and simple IO works 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, 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 another. In addition to the definition of the big character table, Unicode also defines three encodings that are now widely used and that hopefully will replace all others eventually—*UTF-8*, *UTF-16*, and *UTF-32* for

Unicode Transformation Format with 8-, 16-, and 32-bit words, respectively. Since C11, the C language includes rudimentary direct support for these encodings without having to rely on the **locale**. Character and string literals with these encodings can be coded with prefixes. For example, `u8"text"`, `u"text"`, and `U"text"` have types **char8_t**`[5]`, **char16_t**`[5]`, and **char32_t**`[5]`, respectively.

Chances are, the multibyte encoding on a modern platform is UTF-8. If so, you won't need these special literals and types. They are mostly useful in a context where you have to ensure one of these encodings, such as in network communication. Life on legacy platforms might be more difficult; see `https://www.nubaria.com/en/blog/?p=289` for an overview for the Windows platform.

In simpler worlds, we have the correspondance in table 14.4.

TABLE 14.4. Common correspondance between UTF and basic character types

literal	**typedef**	type	encoding	correspondance
`u8'a'`	`uchar8_t`	**unsigned char**	UTF-8	ASCII in **char**
`u'π'`	`uchar16_t`	**uint_least16_t**	UTF-16	
`U'𝒜'`	`uchar32_t`	**uint_least32_t**	UTF-32	"wc" in **wchar_t**
`u8"𝒜"`	`uchar8_t[5]`	**unsigned char**`[5]`	UTF-8	"mbs" in **char**`[]`
`u"𝒜"`	`uchar16_t[3]`	**uint_least16_t**`[3]`	UTF-16	
`U"𝒜"`	`uchar32_t[2]`	**uint_least32_t**`[2]`	UTF-32	"wcs" in **wchar_t**`[]`

The situation for UTF-16 is particularly complicated. A string such as `u"𝒜"` has three elements because the code point **0x**`1D49C` needs at least 17 bits for its encoding, so per its definition, it cannot fit into a 16-bit encoding. UTF-16 gets away with what is called *surrogate pairs*; that is, such characters are encoded with two 16-bit words (plus one word for the terminating null character). There is an obsolete encoding called UCS-2 that comprises those code points that UTF-16 encodes inside a single 16-bit word. The set of these code points is called the *basic multilingual plane* (BMP). Other characters, such as our example `u'𝒜'`, are not in that set and thus not representable in UCS-2. Try to avoid UTF-16 (and UCS-2) whenever possible. UTF-8 is a very efficient multibyte encoder, and UTF-32 is a complete one- word encoding for the whole Unicode set, so you should prefer them where you can.

Similar to the functions that have **wcs** or `mbs` in their names, there are also functions with `c8`, `c16`, and `c32` for source or target encoding of UTF-8, UTF-16, and UTF-32, respectively. Unfortunately, the set of functions is incomplete, and of those that are there, some only came lately with C23. So, if you are on less recent platforms or have specific needs, you may struggle a bit to assemble these into something useful.

Another feature that came with C23 is the guarantee that the UTF-16 and UTF-32 are actually the encodings used for **char16_t** and **char32_t**; under previous standards, implementations were allowed to deviate from that. Previously that property could be tested with feature test macros **__STDC_UTF_16__** and **__STDC_UTF_32__**. These are now fixed to the value 1, so the following preprocessor code should never trigger for modern compilers:

mbstrings.h

```
5 #if (__STDC_UTF_16__ != 1) || (__STDC_UTF_32__ != 1)
6 # error "wide␣character␣strings␣should␣use␣UTF␣encodings"
7 #endif
```

TABLE 14.5. Text conversion functions of the form XXXrtoYYY

abbreviation	encoding	used as XXX	used as YYY
mb	locale multibyte encoding	buffered	unbuffered
wc	locale wide-character encoding	unbuffered	unbuffered
c8	UTF-8	buffered	buffered
c16	UTF-16	buffered	buffered
c32	UTF-32	unbuffered	unbuffered

14.5. Restartable text conversion. The functions with the weird XXXrtoYYY names that we have seen previously have another property: the r in the name stands for *restartable*. That is, these functions may receive multibyte strings as input that are incomplete, collect the state of the input that has been seen so far in a state variable (pointed to by a state parameter), and start to produce output once a complete multibyte character has been detected. The following encoding combinations are supported by C23:

mbrtowc mbrtoc8 mbrtoc16 mbrtoc32
wcrtomb c8rtomb c16rtomb c32rtomb

Here, the abbreviations for the encodings and supplemental information are given in table. 14.5. Note that since they are one-character encodings, neither wc nor c32 need buffering at their end. In general, the other encodings need buffering because the code points may need several characters. So, the only surprising entry is in the case that mb appears on the output side.

Takeaway 14.5 #1 *The multibyte mb encoding of a code point is written to the output string all at once.*

Buffering on the input side is easily detectable: the functions return mbincomplete as long as the input has not yet completed. On the output side, detection of the end of a write operation is a little bit more subtle:

- A small number indicates that the input gave rise or completed a code point and the first character of the output has been stored. Whether the output would need extra characters is not known.
- Subsequent calls to the same functions return mbstored as long as an additional character of the encoding is stored.

To see this in action, consider the case that we want to scan a multibyte input sequence that is read from **stdin**. The code consists of two nested loops that scan the input, prefixed by some buffer declarations and statements.

analyze-utf8.c

```
6  int main(void) {
7    // Make sure to have the platform's mb encoding on input.
8    setlocale(LC_CTYPE, "");
9    // Holds the state of input/output buffering.
10   mbstate_t st = { };
11   // collects the input mb sequence
12   char ib[23];
13   // collects the current UTF-8 mb sequence
14   char8_t ob[5] = { };
15   // the number of input characters for the current code point
16   size_t in = 0;
17   while (fgets(ib, sizeof(ib), stdin)) {
```

```
18      // Run through the current line. The last character is
19      // always reserved for the string terminator.
20      for (char* p = ib; (p-ib) < sizeof(ib)-1;) {
21        size_t const n = sizeof(ib)-1-(p-ib);
```

Here, the call to **setlocale** effectively guarantees that we even see the input as multibyte strings. Without it, we would only scan it with the "C" locale, one byte after the other.

In the body of the inner loop, the input is now given via pointer p, and we write a UTF-8 output sequence to an output buffer ob.

. analyze-utf8.c

```
22        size_t const r = mbrtoc8(ob, p, n, &st);
23        switch (r) {        // Handle the special cases.
24        case mbincomplete: p += n; in += n;     continue;
25        case 0: case mbstored: case mbinvalid: goto INVAL;
26        }
27        p += r; in += r;
28        char8_t* cont = ob+1;  // first character is already stored
29        while (mbrtoc8(cont, "", 1, &st) == mbstored) {
30          cont++;
31        }
```

The goal is to transform the sequence so that we see the boundaries of all complete code points. We achieve this with several calls to the C library function **mbrtoc8**; these calls all use the same **mbstate_t** variable st. A first call is supposed to return the number of bytes read from the input to indicate whether the code point has been completed; the resulting value is accumulated in a variable called in. Then a loop tries to store a null character into the target buffer until that succeeds—that is, until all bytes that constitute the UTF-8 sequence have been written.

The **switch** statement handles the exceptional cases, in particular jumps to an error handling code at label INVAL (see the following discussion), and it continues looking for input if the multibyte input sequence is not yet complete. This happens, for example, when the input buffer ends in the middle of a multibyte character.

Takeaway 14.5 #2 *The multibyte mb encoding of a code point may be collected piecewise from the input.*

Now that we have the UTF-8 encoding of our code point in the output buffer ob, we may analyze the result and distinguish the case that we have a single ASCII character. Here, the end of the input line has to be specially treated because we know that **fgets** stops at these points. Also, the comparison has to be done with the proper UTF-8 character for an end of line because this could, in principle, be different from the native encoding for the platform:

. analyze-utf8.c

```
32        // Now we have the whole UTF-8. Analyze the result.
33        printf("%s", ((cont-ob) == 1) ? "ASCII\t" : "UTF-8\t");
34        for (char8_t* o = ob; o < cont; ++o) {
35          printf("|%02hhx", *o);
36        }
37        // fgets stopped at an end of a line
38        if (*ob == u8'\n') {
```

```
        puts("|\t~␣eol");
        in = 0;
        break;
      } else if (in == (cont-ob)) {
        printf("|\t~␣'%s'\n", ob);
      } else {
        printf("|\t%zu→%tu\n", in, (cont-ob));
      }
      in = 0;
```

Note that we avoid managing buffers that would have to be resized for long input lines. The input buffer `ib` has a constant length and the capacity of **`mbrtoc8`** to store a parsing state. Restart deals with the problem of reading partial input lines. Here, in this example, the length `23` of `ib` is ridiculously small. This is to ensure that we easily cover the case of splitting a multibyte sequence in half when testing. In production code, the chosen value would usually be much higher.

The code of the function then ends with the end of the two nested loops, and the error handling is reachable by the label `INVAL`:

analyze-utf8.c

```
    }
    if (*ob != u8'\n') {
      fputs("incomplete␣line\n", stderr);
    }
  }
  return EXIT_SUCCESS;
 INVAL:
  fputs("input␣error,␣exiting\n", stderr);
  return EXIT_FAILURE;
}
```

14.6. Binary streams. In subsection 8.4, we briefly mentioned that input and output to streams can also be performed in *binary* mode in contrast to the usual *text mode* we have used 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 newline 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.

Similar observations hold for reading data from text files.

If the data we manipulate is effectively human-readable text, all of this is fine; we can be 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, it 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.

As indicated previously, streams can be opened in binary mode. For such a stream, all the translation between the external representation in the file and the internal representation is skipped, and each byte of such a stream is written or read as such. From the interfaces we have seen up to now, only **`fgetc`** and **`fputc`** can handle binary files portably. All others may rely on some form of end-of-line transformation.

To read and write binary streams more easily, the C library has some interfaces that are better suited:

```
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 straightforward. Each stream has a current *file position* for reading and writing. If successful, these two functions read or write `size*nmemb` bytes from that position onward and then update the file position to the new value. The return value of both functions is the number of bytes that have been read or written, usually `size*nmemb`, and thus, an error occurs 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, and **fseek** positions the file according to the arguments `offset` and `whence`. Here, `whence` can have one of these values: **SEEK_SET** refers to the start of the file, and **SEEK_CUR**, to the current file position before the call.[56]

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. This can, for example, be used to write out a large object in its internal representation to a file and read it in later with a different program without performing any modifications.

This interface has some restrictions, though. To work portably, streams have to be opened in binary mode. On some platforms, IO is *always* binary because there is no effective transformation to perform. So, unfortunately, a program that does not use binary mode may work reliably on these platforms but then fail when ported to others.

Takeaway 14.6 #1 *Open streams on which you use* ***fread*** *or* ***fwrite*** *in binary mode.*

Since this works with internal representations of objects, it is only portable between platforms and program executions that use that same representation—the same endianness. Different platforms, operating systems, and even program executions can have different representations.

Takeaway 14.6 #2 *Files written in binary mode are not portable between platforms.*

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 2 GiB.[Exs 57]

Takeaway 14.6 #3 ***fseek*** *and* ***ftell*** *are not suitable for very large file offsets.*

Under some circumstances, applications have to access chunks of binary data that are the same for all executions of a program. This may, for example, be the case for a graphical logo that is shown in a popup window or a binary signature that is to be included in every object file. Since C23, there is a simple tool to do such an inclusion of binary data into a source reliably, the **#embed** directive:

[56] There is also **SEEK_END** for the end-of-file position, but it may have platform-defined glitches.

[Exs 57] Write a function `fseekmax` that uses **intmax_t** instead of **long** and achieves large seek values by combining calls to **fseek**.

```
unsigned char logo[] = {

#if defined(__has_embed) && __has_embed("picture.dat")
# embed "picture.dat"
#else
 "embed␣directive␣is␣missing"
#endif

};
```

On the surface, it works very similar to the **#include** directive; a file name after the directive indicates which file is to be included. Also, similar to **__has_include**, there is a **__has_embed** feature that can be used for tests. First, if **__has_embed** itself is defined, the **#embed** directive is implemented. Then if **__has_embed**("picture.dat") returns 1 in a preprocessor condition, the file is known to exist and can be embedded.

The file is read as binary data, and the result is as if a long list of comma-separated numbers were inserted in place. Each byte is represented as the decimal number of its contents. So, imagine that for relatively large files, this can amount to quite a large list of incomprehensive numbers. In our previous example, this list of numbers is just taken as an initializer for the array variable logo. Nevertheless, sophisticated compilers should implement the **#embed** directive quite effectively when used for an initializer; they should be placing the binary data directly into the executable without encoding and then decoding it to and from decimal literals. Let's look into a more elaborated example:

embed.c

```
25 // define a character array that will contain the entire
26 // source file
27 static char const here[] = {
28
29 // Cedro does not work with blanks between the # and any
       directive
30 #pragma Cedro 1.0 embed
31 #embed "embed.c"
32
33 };
34
35 // define another character array that has the same size
36 static char there[sizeof here];
```

This snippet defines two character arrays, here and there. As before, if the compiler implements the **#embed** directive, a file is provided as binary data to initialize an array, here. Additionally, we show a possible fallback that uses a tool called Cedro (https://sentido-labs.com/en/library/#cedro) that may help you bridge the gap until your compiler is up to speed. See appendix B.2 for more information.

The semantics we expect are shown in the corresponding **main**:

embed.c

```
38 int main(int argc, char* argv[static argc+1]) {
39   size_t ibytes = 0;
40   int cmp = 1000;
41   // Open the file in binary mode.
42   FILE* inp = fopen(__FILE__, "rb");
43   if (inp) {
44     // read the file as binary
45     ibytes = fread(there, 1, sizeof there, inp);
46     cmp = memcmp(here, there, sizeof here);
```

```
    fclose(inp);
  } else {
    printf("could not open %s\n", __FILE__);
    printf("++++++++++++++++++++++++++++++++++++++++++++++++\n");
  }
  size_t obytes = fwrite(here, 1, sizeof here, stdout);
  printf("++++++++++++++++++++++++++++++++++++++++++++++++\n");
  printf("in %zu, out %zu, bytes are %s\n",
         ibytes, obytes,
         cmp < 0 ? "smaller" : (cmp > 0 ? "greater" : "equal"));
}
```

Here, we read the same file (hopefully) in binary mode into the second array `there`. The contents of `here` and `there` are then expected to be exactly the same, so the call to **memcmp** should return 0.

CHALLENGE 16 (Text processing in streams). *For text processing in streams, can you read on* ***stdin****, dump modified text on* ***stdout****, and report diagnostics on* ***stderr****? Count the occurrences of a list of words? Count the occurrences of a regexp? Replace all occurrences of a word with another?*

CHALLENGE 17 (Text processor sophistication). *Can you extend your text processor (challenge 12) to use multibyte characters?*
Can you also extend it to do regular expression processing, such as searching for a word, running a simple query-replace of one word against another, performing a query-replace with a regex against a specific word, and applying regexp grouping?

Summary

- The C library has several interfaces for text processing, but we must be careful about **`const`**-qualification and buffer overflow.
- Formatted input with **`scanf`** (and similar) has subtle issues with pointer types, null termination of strings, white space, and new-line separation. If possible, you should use the combination of **`fgets`** with **`strtod`** or similar, more specialized functions.
- Extended character sets are best handled by using multibyte strings. With some caution, these can be used much like ordinary strings for input and output.
- Binary data should be written to binary files by using **`fwrite`** and **`fread`**. Such files are platform dependent.
- Compile time binary data can be initialized efficiently by using the **`#embed`** directive.

15. Program failure

This section covers

- Wrongdoings leading to failure
- Degradation of the program state
- Unfortunate incidents
- Anticipatory error checking
- Cleanup

C programs can fail in very different ways—silently, sporadically, predictably, or with pomp and circumstances. Commonly, much emphasis is put on the things that may happen *when* a program fails, and attention is withdrawn from the question *if* a program fails and *why*. This is apparent with the term that is often (even falsely) applied to such situations: *undefined behavior*. This term even has an abbreviation in the jargon: ***UB***^{C}. As the term itself suggests, its focus is on the behavior of the program (or, better, the whole system) *after* an error occurred and not on the reasons that lead to the error. It's a bit similar to trying to improve road safety by scaremongering drivers with fines or prison sentences, instead of promoting regular security checks of cars, requiring licenses, or educating drivers and passengers about the benefits of wearing a seat belt.

We will distinguish three different forms of program failure. The first is the specific actions (wrongdoings) that directly cause a failure (section 15.1). Second, in some situations, the state of the executions deteriorates step by step, without one specific action that can or should be blamed for resulting in program failure (section 15.2). Finally, the most difficult and complicated situations occur when behavior that is valid in isolation fails in unfortunate combination from distant parts of the program (section 15.3). At the end, we briefly discuss how program failures may manifest, which is surprisingly uncorrelated to the form of the failure (section 15.5).

We will not handle compilation, linkage, or startup failures that, for example, may lead to invalid executables by linking TU with contradicting declarations (including inconsistent attributes such as `[[noreturn]]`) and using incorrect character or string constants or using invalid function as an **`atexit`** handler. These failures may also only manifest after a seemingly successful startup of a program that did not notice that it was doomed long before.

In this chapter, we will continue to draw our examples from a field that is quite different from computing, but hopefully, it will provide us with enough analogies to understand the problems: traffic of cars, trains, planes, or rockets. I think that these provide good examples because the domain is ruled by a similar set of constraints:

- It is described by a multitude of parameters, including space, velocity, acceleration, energy consumption, and density.
- It is dynamic. The situation changes from second to second, and what is a good action in one moment can be catastrophic in the next.
- It draws on a limited number of resources, such as available space, fuel or electricity, oxygen, time, conductors, cars, and tracks.
- It has multiple objectives, which may include speed, throughput, comfort, safety, and fun.
- It is ruled by different types of laws, such as physics, penal, societal, and superstition.

A common aspect of the failures that we handle here is that they manifest during program execution and that they may occur in situations where they are not detectable at compile time. Errors that violate the constraints of the language are, in general, diagnosed during compilation and are much less of a worry.

15.1. Wrongdoings. This first category of program failures is certainly the most easy to comprehend. It comprises actions or events (or the lack thereof) that directly lead to a failure. In general, if a car veers off a straight alley and hits a tree, we would blame the driver and suspect they have done something wrong, such as a mismanipulation, speeding, drunk driving, or a similar offense. There is also the slight possibility that something was wrong with the car or the road, so if there is no obvious other cause, we could look into that. But only under rare circumstances should the reporter who writes about the accident, Earth's rotation, or the old oak tree into which the car smashed be made responsible for the damages.

15.1.1. *Arithmetic violations.* Among the possible wrongdoings of a program during execution, the simplest category is what the C standard calls an *exceptional condition*, which is an operation that uses operands for which no mathematical result is defined. The most commonly present are probably

- Division by zero
- Modulo by zero

Here, the operations are not mathematically defined for the numbers that we represent. (For floating-point arithmetic, see the following discussion.)

There are several other operations that are similar but the problems they cause are linked to the specific way in which we present numbers as finite sets of bits. For example,

- Negation of **INT_MIN** or similar negative values
- If the second operand of a bit-shift operation is negative or is greater or equal to the width of the first operand
- if an attempt is made to bit-shift into the sign bit of a signed integer type

Historically, platforms have deployed quite different strategies for such illicit operations. A very common one is to crash. The best strategy is to avoid them.

Takeaway 15.1.1 #1 *The program execution should only perform arithmetic operations that are mathematically defined within the range of the underlying type.*

Not surprisingly for C, the *mathematically defined* bit has a special interpretation when it comes to floating-point arithmetic. If there is a floating-point value for "infinity" on the platform, then division by zero is defined (it results in infinity), and any such operation is valid. The macro **INFINITY** in <float.h> (respectively, before C23 in <math.h>) can also be used to test whether infinity is supported:

<float.h>
<math.h>

fp_except.c

```
22 bool const has_inf =
23 #ifdef INFINITY
24   (1.0/0.0 == INFINITY)
25 #else
26   false
27 #endif
28   ;
```

The header <fenv.h> (for "floating-point environment") has several macros that encode *exceptional floating-point conditions* such as division by zero:

<fenv.h>

fp_except.c

```
30 int excepts[] = {
31 #ifdef FE_DIVBYZERO
32   FE_DIVBYZERO,
33 #endif
```

```
#ifdef FE_INEXACT
  FE_INEXACT,
#endif
#ifdef FE_INVALID
  FE_INVALID,
#endif
#ifdef FE_OVERFLOW
  FE_OVERFLOW,
#endif
#ifdef FE_UNDERFLOW
  FE_UNDERFLOW,
#endif
};
```

As you might guess from the names of these exceptions, not all of them concern possible arithmetic violations. We will see some more in the following discussion. These constants can be used as bitsets; or-ing them into one integer value can be used to manage combinations of these exceptional conditions.

In our traffic examples, an invalid operation would be driving fast and then stepping violently on the brakes. If you have a not-so-sophisticated car (a platform without **INFINITY**), you will very likely run off the road and hit the tree. If you have a car with an anti-lock braking system (ABS) (a platform handling **FE_DIVBYZERO**), you may be lucky and stay on the road. In neither circumstance is speeding and then braking violently a good idea.

The supported floating-point exceptions usually do not result in program failures, and the header has interfaces (here, **fetestexcept** and **feclearexcept** [floating-point environment test clear exception]) that allow you to query or manage such an exception:

fp_except.c

```
void printexcept(void) {
  char const* name[] = {
#ifdef FE_DIVBYZERO
    "divbyzero",
#endif
#ifdef FE_INEXACT
    "inexact",
#endif
#ifdef FE_INVALID
    "invalid",
#endif
#ifdef FE_OVERFLOW
    "overflow",
#endif
#ifdef FE_UNDERFLOW
    "underflow",
#endif
  };
  int except = fetestexcept(FE_ALL_EXCEPT);
  if (except) {
    printf("[");
    for (unsigned j = 0; except; except &= ~excepts[j], ++j)
      if (excepts[j] & except)
        printf("%s ", name[j]);
    printf("]");
  }
}
```

```
75
76 int main (int argc, char* argv[static argc+1]) {
77   printf("division␣by␣zero␣is␣%sequal␣to␣INFINITY\n", has_inf ? "
       " : "un");
78   for (unsigned i = 1; i < argc; i++) {
79     feclearexcept(FE_ALL_EXCEPT);
80     double x = strtod(argv[i], nullptr);
81     printf("%g␣", x);
82     printexcept();
83     feclearexcept(FE_ALL_EXCEPT);
84     printf(":␣%g␣", 1.0/x);
85     printexcept();
86     puts("");
87   }
88 }
```

Takeaway 15.1.1 #2 *The floating-point environment of the platforms determines the floating-point operations that result in program failure.*

So, the portability of numerical programs that use floating-point operations that might or might not fail is a real concern and must be carefully handled with feature tests.

Another set of arithmetic operations that might fail are those that operate on pointer values. In fact, the naive representation of pointers as integers-in-disguise is not the whole story. Because of address space segmentation, addition or comparison of pointer values can be much more restricted than those for similar operations on integers. Problematic are

- Address addition or subtraction with an integer that overflows the array bounds;
- Pointer comparison if not pointing into the same array object

Here, the operation itself might already be invalid, even if the theoretical result represents a valid address.

Takeaway 15.1.1 #3 *Pointer manipulations should always stay within the boundaries of an array object.*

In general, it is much better to express an intent of reading a specific element of an array (either accessed directly or via a pointer) by using array indexing `A[i]` instead of explicit arithmetic `*(A+i)`. Modern compilers might capture invalid indices more easily then.

Takeaway 15.1.1 #4 *Where possible, use array indexing instead of pointer arithmetic combined with dereferencing.*

There is another pointer arithmetic operation—namely, pointer subtraction— that depends on the pointers pointing into the same array object. Here, the two pointer values can even have different status during the same program run. If they happen to be within the same array, they are valid; if the target buffers are later recycled and point to different array objects, the same operation with exactly the same values may be erroneous.

Similar to pointer arithmetic, relational operators (<, <=, >=, >) require that both operands point into the same array.

Pointer subtraction also depends on the representation of types. If the result doesn't fit into **`ptrdiff_t`**, the operation is erroneous. This could happen on platforms with a very restricted integer type for **`ptrdiff_t`** (for example, just 16 bits). There, the maximum distance between two pointers that is supported is $\pm 32,767$ whereas the size of objects may be larger.

15.1.2. *Invalid conversions.* Invalid conversions occur when a certain value does not have a good interpretation in a type that is different from the one it had been given initially. Remember that if the target type is an unsigned type, conversion always works with the modulus, and so the result is always well defined. For all other target types, there may be errors, for example,

- *Invalid conversions from one integer type to a signed type*—For example converting the value `UINT_MAX` into a `signed`. The result is implementation-defined: some implementations may use the bit pattern of the unsigned value, whereas others may stop the execution. So, although this is well-defined individually by each implementation, such behavior is not portable, and you should consider such situations as erroneous.
- *Invalid conversions between floating-point values and integers*—For example, a floating-point value may be out of the range of a target integer type, or a big integer value may have much more precision than a floating-point type can handle.
- *Invalid conversions between different floating-point types*—Here, precision may get lost, or the source value may be out of the value range of the target type.
- *Conversion of a pointer into an integer that exceeds the necessary bits*—This can only happen for integer types that are narrower than the width of the pointer type. If the target type is `uintptr_t` (and, thus, that type exists), the conversion is always defined.
- *Conversion of a pointer to a pointer of a different type*—This occurs when the source pointer is not properly aligned for the target type.

Again, all these failures occur directly at these operations, regardless of whether the result value of the conversion is used. On the other hand, if such a conversion is valid, it does not necessarily mean that the resulting value can be used freely. For example, a successful pointer to pointer conversion does not indicate that the object to which the new pointer refers can be accessed through the new pointer value (see the following discussion).

15.1.3. *Value violations.* Another large range of possible program failures caused by wrong values is calls to functions in the C library. A lot of them have the notion of

- Invalid values that are not allowed for the call arguments (such as null pointers, numbers that are too large, or zero sizes for allocation functions)
- Values such that the result of the requested operation is not representable.

Modern compilers and some static analyzer tools may be sophisticated enough to detect some of these violations. But, in general, you'd have to read the specifications carefully to detect such situations.

15.1.4. *Type violations.* C has a relatively strict type system that makes accessing objects or functions with the wrong type erroneous in most cases. Such accesses, both for objects and functions, can only happen when we convert pointers to pointers of a different type. So, the easiest way to inhibit this kind of failure is to avoid these operations.

Takeaway 15.1.4 #1 *Don't convert pointers unless you must.*

We already have seen the rules for accessing objects through a different type (see section 12.6).

For calling a C function, the rule is quite simple.

Takeaway 15.1.4 #2 *Always call a function with the prototype with which it is defined.*

One possible way to call a function with a different prototype is to form a function pointer and then cast the function pointer value to a different type. The effect is a bit like for a train system that has a certain track width. If you give your locomotive a specification of a different track width than it has in reality; even carefully inscribing that lie in the paperwork and the front panel does not change the facts. If the specifications are relatively close (say, they deviate by some millimeters), you would perhaps be able to run the locomotive initially for a few meters, just to have it derail at the first curve or switch.

The simplest way to avoid that situation is not to use function pointers in the first place or, if you have to, never to cast the type away.

Takeaway 15.1.4 #3 *Call a function by its name.*

Another possibility is that in different translation units the same function name is used with different prototypes. In that case, any execution that stumbles upon a call to this function fails.

15.1.5. *Access violations.* Access violations are probably the most common failures of C programs that directly stem from wrongdoings. The different cases are numerous, and for most of them, the consequences of erroneous access can be catastrophic. Here, an analogy for car traffic is a No Entry sign: it is easy to enter a road that is only protected by such a sign; there is no direct enforcement, but the result of doing so can be disastrous.

In C, generally, such access violations are performed through pointers or arrays because, otherwise, the strict type system would already enforce a diagnosis at compilation time. The following list shows the particular responsibility that programming with pointers implies:

- Null pointer dereference
- Accessing a missing object (for example, through a stale pointer to a local variable), freed storage, or a system object that changed, such as a locale pointer
- Modifying and reading the same object from unsequenced subexpressions
- Out-of-bounds accessing of an element directly after an array[58]
- Modifying of an unmutable object (**`const`**-qualified object, string literal, temporary object)
- Accessing a **`volatile`** object from a non-**`volatile`** lvalue
- Accessing an object based on a **`restrict`** pointer through an lvalue not based on the same pointer
- Accessing a member of an atomic structure or union
- Storing from an overlapping object (= operator, **`scanf`**, **`memcpy`**)
- Attempting to access an element of a flexible array member with no elements (see 13.1.3)
- Accessing a function without corresponding properties through an attributed prototype
- Issuing a call to **`longjmp`** initialized from a function context that is dead
- Returning from a signal handler for a computational exception
- Calling **`free`** for an already freed pointer

For all these potential cases of access violations, the user code is expected to keep track of the necessary conditions that allow access or, where possible, add tests that inhibit access violations.

[58] As previously seen, for other positions relative to an array, forming the pointer is already erroneous.

15.1.6. *Value misinterpretation.* A value misinterpretation occurs when an object stores a bit pattern that has no known valid interpretation as the type with which it is accessed. C23 calls this an *indeterminate representation*.

The most important case of interpretation of values going wrong is when there is not yet one—that is, when the program accesses storage that has not yet been initialized. The possibilities for this to occur are variables inside a function that are defined without being initialized (see takeaway 5.5 #1) and dynamic allocations that use `malloc` instead of `calloc`. We will discuss these problems in more detail in section 16.

Another case is object types in which not all possible bit patterns represent a valid value, called *non-value* representation,[59] There are standard types that, by definition, have more representation bits than necessary for their value range, including `bool`, `atomic_flag`, and `_BitInt`(N) types where N is not a multiple of `CHAR_BIT`. For example, `bool` has only two proper values (`false` and `true`) but at least 8 bits. Setting other bits than the least significant can lead to a misinterpretation of a Boolean value. In some contexts, it might be convenient for the compiler to test for the all-zero bit pattern; in other contexts, it may just inspect the least significant bit.

So, messing around with the representation bytes of a `bool` object may cause severe damage:

Takeaway 15.1.6 #1 *Don't store values other than* 0 *or* 1 *in a* `bool` *object.*

Also, standard types other than those previously listed (such as floating-point types) may have non-value representations, but nowadays, this is relatively rare. Nevertheless, it is also good to be cautious.

Takeaway 15.1.6 #2 *Don't change the representation bytes of objects directly.*

15.1.7. *Explicit invalidation.* C23 has introduced a new tool to annotate possible failure, the `unreachable` macro. An invocation asserts that this control path in any execution will never be reached. Consider the following function:

```
ptrdiff_t ptr_dist(void const* p, void const* q) {
  if (!p || !q) unreachable();
  unsigned char const* P = p;
  unsigned char const* Q = q;
  return P - Q;
}
```

Here the conditional and the invocation of `unreachable()` indicate that this function will never be reached with null pointers as parameter values. Unfortunately, for `void` pointers, there is currently no way to augment the prototype of the function with that information, so the user is completely on their own to guarantee that this never happens.

Takeaway 15.1.7 #1 *Only use* `unreachable()` *where you have proof.*

In particular, try to avoid relying on external information for this kind of assessment whenever this is possible. In the previous example, you would be better off using array parameters with bounds to indicate that none of the pointers may be null and even help the compiler determine that property at the calling site:

```
[[maybe_unused]]
static inline
ptrdiff_t ptr_dist_uchar(unsigned char const p[static 1],
```

[59] Before C23, these were called *trap representation*.

```
                          unsigned char const q[static 1]) {
  if (!p || !q) unreachable();
  return p - q;
}
#define ptr_dist(P, Q)                    \
ptr_dist_uchar((void const*){ (P), },     \
               (void const*){ (Q), })
```

An invocation of **unreachable** is different from any other operation that puts the execution in an undefined state or explicitly ends execution. Other operations (often, division by zero) may have definitions that are provided by a different entity than the C standard— for example, the compiler implementation or the operating system or explicit program termination using **exit** or **abort** yet having other prescribed behavior, such as cleaning up or raising a signal. None of that is equivalent to telling the compiler that execution will never go here. Compare that to the difference between indicating that a street is a dead end and there's no street at all.

Takeaway 15.1.7 #2 *Don't use other operations than* ***unreachable*** *() to mark a control path that will never be taken.*

Also, it is important to notice that not the invocation of **unreachable** itself is the wrongdoing here. The real error is the decision that leads to it and not the invocation itself. The consequences of going down the rabbit hole shouldn't be blamed on the rabbit.

15.2. Program state degradation. Program state degradation is a problem that is much more difficult to tackle than wrongdoings because it cannot be blamed on one individual or action; instead, it's the result of the interplay of several actions. If you find yourself in a traffic jam, you shouldn't blame that situation on the driver of the car in front of you. All cars equally contribute to the degraded situation: there may be just too many of them for the given road capacity. As seen at some time on a bridge in Berlin under which there were daily traffic jams, "It isn't that you are *in* a traffic jam; you *are* the traffic jam."

15.2.1. *Unbounded recursion.* As we have seen in section 7.3, an important example of program state degradation is recursion. If we do not have a good notion of progress from one call into the next and do not define the condition for the bottom of the recursion well enough, the execution will crash eventually because the resources for function calls are exhausted. Always remember takeaway 7.3 #2 to ensure you avoid this situation.

15.2.2. *Storage exhaustion.* The resource that is usually exhausted when there is unbounded recursion is the capacity of the platform to provide function call contexts. Usually, this resource is called the ***stack***^{C} because the addition and removal of function call contexts acts like a stack data structure; a call pushes a new context onto this stack, and a **return** pops the current context off. So, unbounded recursion is a special case of ***stack overflow***^{C}: at some point, the limited resources of the stack is not able to provide enough space to store the new execution context.

Another form of stack overflow happens when function contexts are established that have variable length arrays. Here, since the size of the object dynamically depends on array bounds, it is difficult to estimate how much storage is available and how fast the program state may degenerate. Note, though, that this problem only occurs for VLA themselves, not for other variably modified (VM) types. On the other hand, using VLA instead of a large constant-length array may reduce the stack usage substantially. Nevertheless, VLA still have somewhat of a bad reputation in parts of the C community; consequently, VLA (in contrast to VM types) are still an optional feature in C23, tested by the macro **__STDC_NO_VLA__**.

TABLE 15.1. Scarce execution resources

Resource	Reservation	Release	Limits
call context	`va_start` `va_copy`	`va_end`	
stream	`fopen` `tmpfile`	`fclose`	`FOPEN_MAX` `TMP_MAX`
file	`fopen` `freopen`	`remove`	
thread context	`thrd_create`	`thrd_join` `thrd_detach`	
mutex	`mtx_init`	`mtx_destroy`	
condition variable	`cnd_init`	`cnd_destroy`	
thread-specific storage	`tss_create`	`tss_delete`	

Whereas the C standard provides no tool to foresee the exhaustion of the stack, the exhaustion of the other storage system of the C library, usually called the ***heap***^{C}, can be detected. The standard functions that allocate memory (`malloc`, `calloc`, `realloc`, `aligned_alloc`, `strdup`, and `strndup`) return a null pointer on failure. Still, such a failure should not necessarily be blamed on the call itself; there may have previously been just too many such calls. Still, such a situation is preferable to a silent degradation of the execution state since it allows shutting down the execution in a controlled way.

15.2.3. *Other scarce resources.* In addition to storage, the runtime environment of a program execution commonly has other scarce resources that may be exhausted (see table 15.1). Usage of these resources accumulates if none of the functions listed under "release" are called. Note that the function `fopen` reserves two different resources: a stream (to which a pointer is returned) and a file (which is indicated by the first argument). Most of the functions that reserve the resource also have a failure mode that makes it possible to detect exhaustion of the resource. Unfortunately, two of them, `va_start` and `va_copy`, do not provide that information.

15.3. Unfortunate incidents. Whereas wrongdoings and resource exhaustion are failures that (at least in theory) can be captured locally, unfortunate incidents are caused by the alignment of distant events in space or time in a way that causes the execution to fail. Generally, these incidents are rare and difficult to track.

A good traffic model for this kind of failure is collisions. In a world without traffic lights or air controllers, two cars or two planes that prospect their trajectory independently from each other may collide at a location quite far from the location at which they planned their trajectory. Because both are not able to look around the corner (or are within each other's blind spot), each of them sees the resource as uncongested. When they approach the danger zone (a crossing), their readjustment might be too slow, and they might crash into each other.

15.3.1. *Escalating state degradation.* One of the most common types of unfortunate incidents actually happens after state degradation. For example, if stack or heap exhaustion has occurred and the program execution continues, the program state degrades in a way that causes the system to react erratically, potentially causing harm even outside the realm of the current execution.

This situation is similar to when the indicator light for a brake malfunction of your car lights up. If you slow down now, you may save the lives not only of yourself and your fellow passengers but also of innocent bystanders who happen to be in the trajectory of your car. Be a responsible citizen.

15.3.2. *Collisions and race conditions.* When modifying objects, a C program usually does not have to take care that another part of the program execution simultaneously changes that object. The underlying property is *sequencing* (see section 4.6). It is important not to ask too much of the compiler.

Takeaway 15.3.2 #1 *Don't read and modify the same object within the same arithmetic expression.*

A typical example is subexpressions with side effects; we already previously listed them as wrongdoings:

```
printf("%lld\n", x++ + x); // Don't do that!
```

If this is not detected as a failure, the operation of updating variable `x` could happen before, after, or while `x` is read the second time. For these cases, the value printed may be twice the initial value, this value plus 1, or some weird value that has the upper and lower words from different phases of the computation. The problem is that expressions can be complicated, and the compiler might not be able to detect the unsequenced access. However, as long as we use an object through its variable name, these failures are just the result of wrongdoings.

When dealing with pointers, the situation is more complicated:

```
printf("%lld\n", (*p)++ + (*q)); // Are *p and *q different?
```

By coincidence, such an expression could have `p == q`, and thus `*p` and `*q` would refer to the same object. So, setting `p` to a particular value in one corner of the code and then `q` at a completely different spot may have a fatal outcome at a third, completely unrelated point We will provide more details in sections 12.3 and 19.2.

The previously described situation could still, in principle, be guarded by testing `p == q` beforehand, thus avoiding the unsequenced accesses. But there are similar situations that cannot be tested, so-called *race conditions*. They can occur when a signal handler (discussed in section 19.6) and the code it interrupts access the same object. Here, access to an object a signal handler makes is unsequenced to the rest of the program execution. Consequently, such accesses must fulfill special conditions and use special mechanisms to be valid. Similar problems arise when different threads of execution access the same object concurrently (section 20).

15.3.3. *Inappropriate library calls and macro invocations.* Some functions in the C library are restricted to specific contexts in which they may be called. For example, calling **`signal`** in a multithreaded program is not allowed. The function that uses **`signal`** might, under some circumstances, be linked to a program that uses threads. If so, the whole execution might be jeopardized if a signal occurs in a program that uses threads.

Another example is the **`setjmp`** macro, which we will see in more detail in section 19.5; it is only allowed to appear in very specific places within expressions. Whereas a quality implementation would probably diagnose a placement of a macro invocation to **`setjmp`** that it can't handle, a less sophisticated implementation might not. So, for portability, the safest bet is to adhere to the restrictions imposed by the C standard.

15.3.4. *Deadlocks.* A deadlock failure occurs when several entities try to access resources and get trapped in a cyclic chain of dependencies. In the world of road traffic, a roundabout is a perfect example of a strategy for resolving resource conflicts that can go wrong. It is an appealing strategy to resolve local resource conflicts: if cars entering the roundabout have lower priority than cars already inside, any race condition for access to a place within the roundabout is avoided. This is

probably the reason why all over Europe roundabouts are now everywhere: they avoid resource conflicts (and thus car crashes) at a relatively low cost.

This strategy works well when there is not too much traffic, but unfortunately, under high congestion, it breaks down completely. As the traffic sign for a roundabout illustrates, if the roundabout is filled with cars that are proceeding to the subsequent exit, none of them can move, and no progress is possible. Therefore, roundabouts are usually not found in highly congested environments such as cities, where crossings with traffic lights are used instead.

Fortunately, when programming with C, deadlocks can only appear in a multithreaded context (see section 20). Proving that a particular multithreaded program has no deadlocks is a difficult task that we cannot tackle to a satisfactory extent in the frame of this book. Nevertheless, we will provide some directions in section 20.7 with a case analysis of a particular example.

15.4. Series of unfortunate events. A particularly nasty type of failure is program executions that continue endlessly over a finite set of states without making visible progress.

Takeaway 15.4 #1 *A program execution that loops over a finite set of states with no observable side effects has failed.*

The situation is a bit like that of a plane circling in the eye of a storm. The immediate situation does not look particularly dangerous. But for the outside world, the plane may be accounted as being lost. The exact point of the inevitable crash, past or future, is not very important because we will never see the plane again.

In section 5.7.5, we have already seen an example of a loop that might not present enough progress. Let's look more closely at the possibilities by modifying the previous example:

```
void obscure(unsigned);
...
for (unsigned i = 0; true; i++) {
  obscure(i);
}
unreachable();
```

Whether such a loop fails depends on the function `obscure`. With the specification as given, a compiler cannot decide whether this loop will lead to systematic failure. Depending on the value of `i`, the function could have a visible side effect, for example,

- Does some IO
- Changes some global state
- Calls **`exit`**.

So, although this **`for`** loop has no obvious termination, it does not indicate systematic failure. The compiler could only assume that if it had additional information about the function—for example, if the function were annotated by a `[[unsequenced]]` attribute—it then might give some diagnosis. Otherwise, it is the programmer's responsibility to ensure that `obscure` makes progress for at least one value of `i`. In any case, the loop can only terminate by exiting the whole execution, and the invocation of **`unreachable`** will never be reached.

If we change the type of `i` from **`unsigned`** to **`signed`**, the loop is not even an infinite loop anymore:

```
for (signed i = 0; true; i++) {
  obscure(i);
}
```

```
unreachable();
```

Here, the compiler may assume that for one positive value of `i`, a program termination happens; otherwise, for a value of **`INT_MAX`**, an arithmetic exception would occur.

Unbounded recursion may also be a special case of a program that lacks progress. In fact, for some recursive functions, the compiler may detect that no state has to be saved per individual call. A call to such a function with unbounded recursion may then be replaced by a loop that makes no additional allocation. Similar to the loop, a good compiler could then warn that a recursive call leads to systematic program failure.

An even more complicated failure type, caused by a series of unfortunate events, may occur for threaded programs: livelocks. Their effects are similar to the deadlocks we discussed previously (basically, nothing seems to happen), but the interaction of the components of a program with a livelock is more complicated. An analogy can be a big traffic system with detours that form a cyclic pattern. Consider the four crossings on the corners of a city block with detours as indicated in figure 15.1. While each detour by itself may seem to be a good idea, the whole set forms a trap.

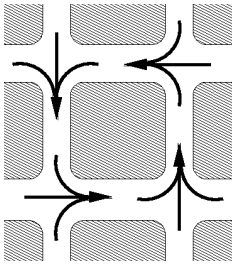


FIGURE 15.1. A configuration potentially leading to a livelock

Any vehicle that reaches one of the crossings will circle the block forever. Even if there is only one vehicle, at each instance, it will appear as if traffic advances as it should, but the vehicle will never escape the vortex.

This is only a very simple example of a livelock. There could be multiple cars involved, and the detours could send the cars all over the city. Depending on which order vehicles arrive and how fast they are going, they may even switch order at some point or go different routes without ever escaping. Such situations are much more difficult to detect and prevent. The case analysis in section 20.7 will show how to prove the absence of a livelock for a practical case.

15.5. Dealing with failures. Failure of a C program can have different indications or, in some cases, pass completely unnoticed. The easiest way to deal with failures is, in fact, to avoid them.

Takeaway 15.5 #1 *Ensure all preconditions for an operation that could fail.*

That is, all the failures listed under wrongdoings in (15.1) should not occur during a normal program execution. The preconditions should be tested, and an alternative should be executed. Ensuring all preconditions is probably not always easy, but the general strategy should be as described.

Similarly, in general, program state degeneration should be detected, but this time, it is only possible to do that after the fact.

Takeaway 15.5 #2 *The return of operations that might exhaust resources should be checked for errors.*

There are, in general, two indicators for such errors: error returns for C library functions (see section 8.1.3) and a nonzero value of **errno**. We will see strategies on how to deal with these in section 15.6.

Unfortunate incidents are much, much more difficult to handle. By their nature, they do not have many indicators to detect the situation. Nevertheless, modern systems have some features that may limit the damage done to the system:

- Signals are an archaic and arcane mixture of hardware and software features that interrupt a program execution at a given point and allow users to provide fallback code, which is then executed to save the day (see section 19.6).
- A trap is similar because it immediately interrupts the flow of the execution. In contrast to a signal, execution just ends; whether there is at least some cleanup depends a lot on the system.
- C provides termination functions for program executions—namely **exit**, **quick_exit**, **_Exit**, and **abort**, as we have already seen in section 8.8. The cleanup depends on the function and ranges from user-defined cleanup functions (for **exit** and **quick_exit**) to almost none (for **abort**).
- Similarly, threads can be terminated in isolation using **thrd_exit**. We will see that in section 20.6.
- Last but not least, after byzantine failure execution continues, and the system state degrades more and more. There is no possibility to capture the failure. Damage to the whole system may be severe, sensitive information may get lost, and catastrophic actions in the physical world may be set in motion.

Takeaway 15.5 #3 *Unfortunate events can only be avoided with a careful algorithm design.*

15.6. Error checking and cleanup. C programs can encounter a lot of error conditions when we test for the preconditions of an operation or a valid outcome of a C library call. Errors can be programming errors, bugs in the compiler or OS software, hardware errors, resource exhaustion (such as out of memory), or any malicious combination of these. For a program to be reliable, we have to detect such error conditions and deal with them gracefully.

As a first example, take the following description of a function `fprintnumbers`, which continues the series of functions that we discussed in subsection 14.1. As you can see, this function distinguishes four different error conditions, 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 imposes only **EOF** (which is negative) and **EDOM**, **EILSEQ**, and **ERANGE**, which are positive.

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* is a null pointer
- −**EFAULT** if *numb* is a null pointer and *len* is not zero
- −**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]);
```

Other error 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:

numberline.c

```
39 #include <errno.h>
40 #ifndef EFAULT
41 # define EFAULT EDOM
42 #endif
43 #ifndef EOVERFLOW
44 # define EOVERFLOW (EFAULT-EOF)
45 # if EOVERFLOW > INT_MAX
46 #  error EOVERFLOW constant is too large
47 # endif
48 #endif
49 #ifndef ENOMEM
50 # define ENOMEM (EOVERFLOW+EFAULT-EOF)
```

Listing 15.1. Print an array of numbers

```
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 = nullptr;

  if (len) {
    /* Count the chars for the numbers. */
    for (size_t i = 0; i < len; ++i)
      tot += snprintf(nullptr, 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;
}
```

```
# if ENOMEM > INT_MAX
#  error ENOMEM constant is too large
# endif
#endif
```

The idea 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 in the following listing 15.1.

Error handling pretty much dominates the coding effort for the whole function. The first three lines handle errors that occur on entry to the function and reflect missed preconditions or, in the language of Annex K (see subsection 8.1.4), ***runtime constraint violations***^{C}.

Dynamic run-time errors are a bit more difficult to handle. In particular, some functions in 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 with a call to the small function `error_cleanup`

The core of the function 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 a null pointer and a buffer size of `0` are used to compute the size for each number. Then our function `sprintnumbers` from subsection 14.1 is used to produce a big string that is printed using **`fputs`**.

. numberline.c

```
150 static inline int error_cleanup(int err, int prev) {
151   errno = prev;
152   return -err;
153 }
```

Observe that there is no error exit after a successful call to **malloc**. If an error is detected on return from the call to **fputs**, the information is stored in the variable tot, but the call to **free** is not skipped. So, even if such an output error occurs, no allocated memory is left leaking. Here, taking care of a possible IO error was relatively simple because the call to **fputs** occurred close to the call to **free**.

The function fprintnumbers_opt requires more care (see listing 15.2). 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 guaranteeing to issue a call to **free** at the end. The first such condition is that the buffer we initially allocated is too small. If the call to **realloc** to enlarge it fails, we have to retreat carefully. The same is true if we encounter the unlikely condition that the total length of the string exceeds **INT_MAX**.

Listing 15.2. Printing an array of numbers, optimized version

```
205 int fprintnumbers_opt(FILE*restrict stream,
206                      char const form[restrict static 1],
207                      char const sep[restrict static 1],
208                      size_t len, size_t nums[restrict static len]) {
209   if (!stream)        return -EFAULT;
210   if (len && !nums)   return -EFAULT;
211   if (len > INT_MAX) return -EOVERFLOW;
212
213   int err = errno;
214   size_t const seplen = strlen(sep);
215
216   size_t tot = 0;
217   size_t mtot = len*(seplen+10);
218   char* buf = malloc(mtot);
219
220   if (!buf) return error_cleanup(ENOMEM, err);
221
222   for (size_t i = 0; i < len; ++i) {
223     tot += sprintf(&buf[tot], form, nums[i]);
224     ++i;
225     if (i >= len) break;
226     if (tot > mtot-20) {
227       mtot *= 2;
228       char* nbuf = realloc(buf, mtot);
229       if (nbuf) {
230         buf = nbuf;
231       } else {
232         tot = error_cleanup(ENOMEM, err);
233         goto CLEANUP;
234       }
235     }
236     memcpy(&buf[tot], sep, seplen);
237     tot += seplen;
```

```
238       if (tot > INT_MAX) {
239         tot = error_cleanup(EOVERFLOW, err);
240         goto CLEANUP;
241       }
242     }
243     buf[tot] = 0;
244
245     /* print whole line in one go */
246     if (fputs(buf, stream) == EOF) tot = EOF;
247   CLEANUP:
248     free(buf);
249     return tot;
250   }
```

In 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 takes place and avoids hard-to-read nested **if**–**else** conditions. The rules for **goto** are relatively simple.

Takeaway 15.6 #1 *Labels for* ***goto*** *are visible in the entire function that contains them.*

Takeaway 15.6 #2 ***goto*** *can only jump to a label inside the same function.*

Takeaway 15.6 #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]. You will still find people who seriously object to 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 is to have the "normal" control flow of the function be mainly undisturbed and to clearly mark changes to the control flow that only occur under exceptional circumstances with a **goto** or **return**. Later, in subsection 19.5, we will see another tool in C that allows even more drastic changes to the control flow, **setjmp**/**longjmp**, which enables us to jump to other positions on the stack of calling functions.

Summary

- Program failure can originate in wrongdoing, program state degradation, or an unfortunate incident.
- Possible wrongdoings can be avoided by carefully checking preconditions.
- Calls to C library functions that may lead to state degradation should be checked for error returns.
- There is no cure for unfortunate incidents. You must ensure they don't appear by using a careful program design.
- Handling error conditions can lead to complicated case analysis. It can be organized by a function-specific code block to which we jump with `goto` statements.

LEVEL 3

Experience

The alpine chough lives and breeds in the thin air of high altitudes and has been seen above 8,000 m in the Himalayas.

In this level, we go more deeply into details about specific topics. The first, performance, is one of the primary reasons C is chosen over other programming languages. Therefore, section 16 is a mandatory read for all C software designers.

The second topic is a feature quite specific to C: function-like macros. Because of their complexity and obvious ugliness, they are much frowned upon by other programming communities. Nevertheless, it is important to master them to a certain extent because they allow us to provide easy-to-use interfaces—for example, for type-generic programming and more sophisticated parameter checking.

Sections 19 and 20 then show how the usual assumption of sequential program execution can be weakened to allow for asynchronous problem handling (with long jumps or signal handlers) or the parallel execution of threads. These come with specific problems related to guaranteeing data consistency, so we conclude with section 21, which dives more deeply into the handling of atomic data and synchronization in general.

16. Performance

This section covers

- Writing inline functions
- Restricting pointers
- Measuring and inspecting performance

Once you feel more comfortable when coding in C, you will perhaps be tempted to do complicated things to "optimize" your code. Whatever you think you are optimizing, there is a good chance you will get it wrong: premature optimization can do a great deal of harm in terms of readability, soundness, maintainability, and so on.
Knuth [1974] coined the following phrase that should be your motto for this whole level.

Takeaway 16 #1 *Premature optimization is the root of all evil.*

C's good performance is often cited as one of the main reasons it is used so widely. While there is some truth to the idea that many C programs outperform code of similar complexity written in other programming languages, this aspect of C may come with a substantial cost, especially concerning safety. This is because C, in many places, doesn't enforce rules but places the burden of verifying them on the programmer. Important examples of such cases are

- Out-of-bounds access of arrays
- Accessing uninitialized objects
- Accessing objects after their lifetime has ended
- Integer overflow

These can result in program crashes, loss of data, incorrect results, exposure of sensitive information, and even loss of money or lives.

Takeaway 16 #2 *Do not trade safety for performance.*

C compilers have become much better in recent years; basically, they complain about all problems that are detectable at compile time. But, severe problems in code can still remain undetected in code that tries to be clever. Many of these problems are avoidable, or at least detectable, by very simple means:

- All block-scope variables should be initialized, thereby eliminating half the problems with uninitialized objects.
- Dynamical allocation should be done with `calloc` instead of `malloc` wherever that is suitable. This avoids another quarter of the problems with uninitialized objects.
- A specific initialization function should be implemented for more-complicated data structures that are allocated dynamically. That eliminates the rest of the problems with uninitialized objects.
- Functions that receive pointers should use array syntax and distinguish different cases:
  - *A pointer to a single object of the type*—These functions should use the `static 1` notation and thus indicate that they expect a non-null pointer:

```
void func(double a[static 1]);
```

  - *A pointer to a collection of objects of known number*—These functions should use the `static N` notation and thus indicate they expect a pointer that points to at least that number of elements:

```
void func(double a[static 7]);
```

  - *A pointer to a collection of objects of unknown number*—These functions should use the VLA notation but still indicate via **`static`** that the bound gives a guarantee about the number of elements that are accessible:

```
void func(size_t n, double a[static n]);
```

    This might look restrictive at first glance: for this to work, we always have to declare the size parameter before the array parameters. Section 18.2 shows a trick on how to create macro and inline interfaces to such functions that can work around this requirement for function interfaces that cannot be changed, such as **`snprintf`**.
  - *A pointer to a single object, an array or a null pointer*—Such a function must guarantee that even when it receives a null pointer, the execution remains in a defined state:

```
void func(double* a);
```

  Some compiler builders only just started to implement checks for these cases, so errors might not (yet) be detected automatically; some compilers are already quite good, at least if the function call has size expressions that are integer constant expressions and arrays with a size that is known at compile time. In any case, writing these down and making them clear will help you avoid out-of-bounds errors.
- Taking addresses of block-scope (local) variables should be avoided if possible. Therefore, it is good practice to mark all variables in complex code with **`register`**.
- Use unsigned integer types for loop indices and handle wrap-around explicitly. The latter can, for example, be achieved by comparing the loop variable to the maximum value of the type before the increment operation.

Despite what some urban myths suggest, applying these rules usually will not negatively affect the performance of your code.

Takeaway 16 #3 *Optimizers are clever enough to eliminate unused initializations.*

Takeaway 16 #4 *The different notations of pointer arguments to functions result in the same binary code.*

Takeaway 16 #5 *Not taking addresses of local variables helps the optimizer because it inhibits aliasing.*

Once we have applied these rules and have ensured that our implementation is safe, we can look at the performance of the program. What constitutes good performance and how we measure it are difficult subjects by themselves. A first question concerning performance should always be relevance: for example, improving the run time of an interactive program from 1 *ms* to 0.9 *ms* usually makes no sense at all, and any effort spent making such an improvement is probably better invested elsewhere. To equip us with the necessary tools to assess performance bottlenecks, we will discuss how to measure performance (subsection 16.4). This discussion comes at the end of this section because before we can fully understand measuring performance, we have to better understand the tools for making performance improvements.

There are many situations in which we can help our compiler (and future versions of it) to optimize code better because we can specify certain properties of our code that it can't deduce automatically. C introduces features for this purpose that are quite special in the sense that they constrain not the compiler but the programmer. They all have the property that removing them from valid code where they are present should not change the semantics. Because of that property, they are sometimes presented as useless or even obsolete features. Be careful when you encounter such statements: people who make such claims tend not to have a deep understanding of C, its memory model, or its optimization possibilities. And, in particular, they don't seem to have a deep understanding of cause and effect.

The features that introduce these optimization opportunities are

- **register** (C89)
- **inline**, **restrict** (both from C99)
- **alignas** (respectively **_Alignas**, C11)
- `[[unsequenced]]`, and `[[reproducible]]` (both from C23)

As indicated, all have the property that they could be omitted from a valid program without changing its semantics.

In subsection 13.2, we discussed **register** to some extent, so we will not go into more detail than that. Just remember that it can help to avoid aliasing between objects that are defined locally in a function. As stated there, I think this feature is very underestimated in the C community.

In subsection 12.7, we also discussed C11's **alignas** and the related **alignof**. They can help position objects on cache boundaries and thus improve memory access. We will not go into more detail about these specialized features.

The remaining features, C99's **inline** (16.1) and **restrict** (16.2) and C23's `[[unsequenced]]`, and `[[reproducible]]` (16.3), have very different usability. The first is relatively easy to use and presents no particular danger. It is a tool that is quite widely used and may ensure that the code for short functions can be directly integrated and optimized at the caller side of the function.

The second, **restrict**, relaxes the type-based aliasing considerations to allow for better optimization. Thus, it is subtle to use and can do considerable harm if used badly. It is often found in library interfaces but much less often in user code.

For the last two, especially for `[[unsequenced]]`, we have already seen a lot of uses previously that cover the case of pure functions. They are convenient annotations that tell the compiler that function calls may be moved and combined. But their possible use goes beyond that simple case of pure functions—namely, they can also model functions with pointer parameters and returns. The conditions that functions have to fulfill reuse a lot of the properties introduced with **restrict**.

The remainder of this section (subsection 16.4) dives into performance measurement and code inspection to enable us to asses performance by itself and the reasons leading to good or bad performance.

16.1. Inline functions. For C programs, the standard tool to write modular code is functions. As we have seen, they have several advantages:

- They clearly separate interface and implementation. Thus, they allow us to improve code incrementally, from revision to revision, or to rewrite functionality from scratch if necessary.
- If we avoid communicating with the rest of the code via global variables, we ensure that the state a function accesses is local. That way, the state is present in the parameters of the call and local variables only. Optimization opportunities may thus be detected much more easily.

Unfortunately, functions also have some downsides from a performance point of view:

- Even on modern platforms, a function call has a certain overhead. Usually, when calling a function, some stack space is put aside and local variables are initialized or copied. Control flow jumps to a different point in the executable, which may or may not be in the execution cache.
- Depending on the calling convention of the platform, if the return value of a function is a **`struct`**, the whole return value may have to be copied where the caller of the function expects the result.

If, by coincidence, the code of the caller (say, `fcaller`) and the callee (say, `fsmall`) is present inside the same translation unit (TU), a good compiler may avoid these downsides by *inlining*. Here, the compiler does something equivalent to replacing the call to `fsmall` with the code of `fsmall` itself. Then there is no call, so there is no call overhead.

Even better, since the code of `fsmall` is now inlined, all instructions of `fsmall` are seen in that new context. The compiler can detect, for example,

- Dead branches that are never executed
- Repeated computation of an expression where the result is already known
- A function (as called) that may only return a certain type of value

Takeaway 16.1 #1 *Inlining can open up a lot of optimization opportunities.*

A traditional C compiler can only inline functions for which it also knows the definition; knowing only the declaration is not enough. Therefore, programmers and compiler builders have studied the possibilities for increasing inlining by making function definitions visible. Without additional support from the language, there are two strategies to do so:

- Concatenate all code of a project into a single large file and then compile all that code in one giant TU. Doing so systematically is not as easy as it sounds: we have to ensure that the concatenation order of the source files doesn't produce definition cycles and that we don't have naming conflicts (for example, two TUs, each with a **`static`** function `init`).
- Functions that should be inlined are placed in header files and then included by all TUs that need them. To avoid the multiple definitions of the function symbol in each TU, these functions must be declared **`static`**.

Whereas the first approach is infeasible for large projects, the second approach is relatively easy to implement. Nevertheless, it has drawbacks:

- If the function is too big to be inlined by the compiler, it is instantiated separately in every TU. That is, a function that big will potentially have a lot of copies and increase the size of the final executable.
- Taking a pointer of such a function will give the address of the particular instance in the current TU. Comparison of two such pointers obtained in different TUs will not compare as equal.
- If such a **`static`** function declared in a header file is not used in a TU, the compiler will usually warn about that nonuse. So, if we have a lot of such small functions in header files, we will see a lot of warnings, producing a lot of false alarms.

To avoid these drawbacks, C99 introduced the **`inline`** keyword. Unlike what the naming might suggest, this does not force a function to be inlined; it only provides a way that it *may* be:

- A function definition declared with **inline** can be used in several TUs without causing a multiple-symbol-definition error.
- All pointers to the same **inline** function will compare as equal, even if obtained in different TUs.
- An **inline** function not used in a specific TU will be completely absent from the binary of that TU and, in particular, will not contribute to its size.

The latter point is generally an advantage, but it has one simple problem: no symbol for the function would ever be emitted, even for programs that might need such a symbol. There are several common situations in which a symbol is needed:

- The program directly uses or stores a pointer to the function.
- The compiler decides that the function is too big or too complicated to inline. This situation varies and depends on several factors:
  - The optimization level used for the compilation
  - Whether debugging options are on or off
  - The use of certain C library function by the function itself
- The function is part of a library that is shipped and linked with unknown programs.

To provide such a symbol, C99 introduced a special rule for **inline** functions.

Takeaway 16.1 #2 *Adding a compatible declaration without the **inline** keyword ensures the emission of the function symbol in the current TU.*

As an example, suppose we have an **inline** function like this in a header file: say `toto.h`:

```
1  // Inline definition in a header file.
2  // Function argument names and local variables are visible
3  // to the preprocessor and must be handled with care.
4  inline
5  toto* toto_init(toto* toto_x){
6    if (toto_x) {
7      *toto_x = (toto){ };
8    }
9    return toto_x;
10 }
```

Such a function is a perfect candidate for inlining. It is really small, and the initialization of any variable of type `toto` is probably best made in place. The call overhead is of the same order as the inner part of the function, and in many cases, the caller of the function may even omit the test for the **if**.

Takeaway 16.1 #3 *An **inline** function definition is visible in all TUs.*

This function may be inlined by the compiler in all TUs that see this code, but none would effectively emit the symbol `toto_init`. But we can (and should) enforce the emission in one TU, `toto.c`, say, by adding a line like

```
1  #include "toto.h"
2
3  // Instantiate in exactly one TU.
4  // The parameter name is omitted to avoid macro replacement.
5  toto* toto_init(toto*);
```

Takeaway 16.1 #4 *An* **inline** *definition goes in a header file.*

Takeaway 16.1 #5 *An additional declaration without* **inline** *goes in exactly one TU.*

As we said, that mechanism of **inline** functions is there to help the compiler decide whether to really inline a function. In most cases, the heuristics that compiler builders have implemented to make that decision are completely appropriate, and you can't do better. They know the particular platform for which the compilation is done much better than you; maybe this platform didn't even exist when you wrote your code. So they are in a much better position to compare the tradeoffs between the different possibilities.

An important family of functions that may benefit from **inline** definitions is *pure functions*, which we met in subsection 10.2.2. If we look at the example of the `rat` structure (listing 10.1), we see that all the functions implicitly copy the function arguments and the return value. If we rewrite all these functions as **inline** in the header file, all these copies can be avoided using an optimizing compiler.[Exs 1] [Exs 2]

So, **inline** functions can be a precious tool to build portable code that shows good performance; we just help the compiler(s) make the appropriate decision. Unfortunately, using **inline** functions also has drawbacks that should be taken into account for our design.

First, 16.1 #3 implies that any change you make to an **inline** function will trigger a complete rebuild of your project and all of its users.

Takeaway 16.1 #6 *Only expose functions as* **inline** *if you consider them stable.*

Second, the global visibility of the function definition also has the effect that local identifiers of the function (parameters or local variables) may be subject to macro expansion for macros we don't even know about. In the example, we used the `toto_` prefix to protect the function parameters from expansion by macros from other include files.

Takeaway 16.1 #7 *All identifiers local to an* **inline** *function should be protected by a convenient naming convention.*

Third, other than conventional function definitions, **inline** functions have no particular TU with which they are associated. Whereas a conventional function can access state and functions that are local to the TU (**static** variables and functions), for an **inline** function, it would not be clear which copy of which TU these refer to.

Takeaway 16.1 #8 **inline** *functions can't access* **static** *functions by name.*

Takeaway 16.1 #9 **inline** *functions can't access modifiable* **static** *objects by name.*

Here, the emphasis is on the fact that access is restricted to the *identifiers* and not the objects or functions themselves. There is no problem with passing a pointer to a **static** object or a function to an **inline** function.

But even if there is no identifier, such as for compound literals, defining **static** objects is still not permitted.

Takeaway 16.1 #A **inline** *functions can't define modifiable* **static** *objects.*

[Exs 1] Rewrite the examples from subsection 10.2.2 with **inline**.

[Exs 2] Revisit the function examples in subsection 7 and argue whether each should or should not be defined as **inline**.

16.2. Using restrict qualifiers. We have seen many examples of C library functions that use the keyword **restrict** to qualify pointers, and we also have used this qualification for our own functions. The basic idea of **restrict** is relatively simple: it tells the compiler that the pointer in question is the only access to the object it points to. Thus, the compiler can make the assumption that changes to the object can only occur through that same pointer, and the object cannot change inadvertently. In other words, with **restrict**, we are telling the compiler that the object does not alias any other object the compiler handles in this part of the code.

Takeaway 16.2 #1 *A **restrict**-qualified pointer has to provide exclusive access.*

As is often the case in C, such a declaration places the burden of verifying this property on the caller.

Takeaway 16.2 #2 *A **restrict**-qualification constrains the caller of a function.*

Consider, for example, the differences between **memcpy** and **memmove**:

```
1 void* memcpy(void*restrict s1, void const*restrict s2, size_t n);
2 void* memmove(void* s1, const void* s2, size_t n);
```

For **memcpy**, both pointers are **restrict**-qualified. So, for the execution of this function, the access through both pointers has to be exclusive; otherwise, the execution fails. Also, `s1` and `s2` must have different values, and neither can provide access to parts of the object of the other. In other words, the two objects that **memcpy** "sees" through the two pointers must not overlap. Assuming this can help to optimize the function.

In contrast, **memmove** does not make such an assumption. So, `s1` and `s2` may be equal, or the objects may overlap. The function must be able to cope with that situation. Therefore, it might be less efficient, but it is more general.

We saw in subsection 12.3 that it might be important for the compiler to decide whether two pointers may, in fact, point to the same object (aliasing). Pointers to different base types are not supposed to alias unless one of them is a character type. So, both parameters of **fputs** are declared with **restrict**:

```
1 int fputs(const char *restrict s, FILE *restrict stream);
```

However, it seems very unlikely that anyone would call **fputs** with the same pointer value for both parameters.

This specification is more important for functions like **printf** and friends:

```
1 int printf(const char *restrict format, ...);
2 int fprintf(FILE *restrict stream, const char *restrict format,
    ...);
```

The `format` parameter shouldn't alias any of the arguments that might be passed to the `...` part. For example, reaching the following code, the execution fails:

```
1   char const* format = "format␣printing␣itself:␣%s\n";
2   printf(format, format);   // Restrict violation
```

This example will probably still do what you think it does. If you abuse the `stream` parameter, your program might explode:

```
1   char const* format = "First␣two␣bytes␣in␣stdin␣object:␣%.2s\n";
2   char const* bytes = (char*)stdin; // Legal cast to char
3   fprintf(stdin, format, bytes);    // Restrict violation
```

Sure, code like this is not very likely to occur in real life. But keep in mind that character types have special rules concerning aliasing, and therefore all string-processing functions may be subject to missed optimization. You could add **`restrict`** qualifications in many places where string parameters are involved and that you know are accessed exclusively through the pointer in question.

16.3. Unsequenced and reproducible attributes. Many of our example functions use the attribute `[[unsequenced]]` (or the header-safe form `[[__unsequenced__]]`) to indicate a function is pure.

Takeaway 16.3 #1 *All pure functions should have the attribute `[[unsequenced]]`.*

Other than before for **`restrict`**, such an annotated *pure* function can usually be used freely without restrictions. If the function definition is verified, this knowledge can be used to optimize the call site much better.

For the first part of this section, we will assume that situation— namely, that the attributed function is, in fact, pure. Later, we will see how these definitions extend to functions that have pointer arguments or return values and that may have an internal state for the `[[reproducible]]` attribute.

So here, the burden of verification is on the writer of the function definition. In general, for the simple case of pure functions, this is not very difficult: we have to ensure that the function only depends on its arguments and has no other effects other than returning a value. This splits into several properties that are relatively easy to check.

For the state dependency part, it is formulated as follows.

Takeaway 16.3 #2 *A function with the attribute `[[unsequenced]]` shall not read nonconstant global variables or system state.*

Here, the possible danger does not only lie in global variables that are read and would possibly change under our feet. There are other more subtle parts of the execution state on which a function may rely that we may not easily notice. A good example of such an implicit state is the floating point rounding mode. Unfortunately, in some parts, the C library follows a quite antiquated model for tuning the floating point model: a thread-local state is modified by the use of pragmas:

```
#pragma FP_CONTRACT OFF
#pragma FENV_ROUND FE_TONEAREST
```

Here, for example, this sets the global thread-local state such that no floating-point expression contractions are performed and the result of rounding always goes to the nearest representable number. Since this state is accessible by programming interfaces, users of a function that we think is pure may indeed depend on a state that changes between different calls.

Takeaway 16.3 #3 *In general, a function that uses floating point arithmetic is not pure and shall not have the attribute `[[unsequenced]]`.*

This statement, in particular, includes all functions in the `<math.h>`, `<tgmath.h>` and `<complex.h>` headers. What a disappointment, I hear you say, but please be patient; there is at least a partial cure in the following discussion. However, for the moment, things will even get worse.

`<math.h>`
`<tgmath.h>`
`<complex.h>`

Takeaway 16.3 #4 *A function with the attribute `[[unsequenced]]` shall not apply visible modifications to global variables or system states.*

Here again, many C library functions are not admissible because they use another antiquated model that uses global state: **errno**.

Takeaway 16.3 #5 *A function that returns possible errors through* **errno** *is not pure and shall not have the attribute* `[[unsequenced]]`.

Fortunately, there is a remedy to work around this that uses the pragmas **FP_CONTRACT** and **FENV_ROUND**. Here is an example from the original paper that proposed this feature for C23:[3]

```
#include <math.h>
#include <fenv.h>
inline double distance (double const x[restrict static 2]) [[reproducible
    ]] {
#   pragma FP_CONTRACT OFF
#   pragma FENV_ROUND FE_TONEAREST
    // We assert that sqrt will not be called with invalid arguments
    // and the result only depends on the argument value.
    extern double sqrt(double) [[unsequenced]];
    return sqrt(x[0]*x[0] + x[1]*x[1]);
}
```

Here, we manually make some guarantees that **sqrt** otherwise does not fulfill in all cases:

- In that context, the function is never called with negative arguments because a sum of squares is always positive. So, here, all calls are valid, and **errno** is never set.
- For the scope of the `distance` function, the two pragmas are always set to the same state, so the call to **sqrt** always sees the same state.

In this special context, **sqrt** is indeed pure and may be annotated with the attribute. The mechanisms in place that guarantee this works are twofold.

Takeaway 16.3 #6 *Pragmas that change the floating point state act locally within the current scope.*

In our example, that means that the rounding mode possibly changes at the position of the pragma and then switches back to the original value at the end of the scope in which the pragma is placed.

Takeaway 16.3 #7 *Type attributes accumulate within the current scope.*

Here, that means as long as we are inside `distance`, the compiler sees **sqrt** with the attribute and may use this information to optimize locally. Usages of the function in other contexts are not affected.

Our function `distance` bears another attribute: `[[reproducible]]`. As the name indicates, the idea is that we only have to guarantee that the result(s) and effects of such a call with a given set of arguments are always the same, regardless of how the function achieves this internally. This attribute has weaker prerequisites and, thus, fewer optimization opportunities. Our function clearly has a property that disqualifies it from being unsequenced: it changes the global state— namely, the floating point properties.

Takeaway 16.3 #8 *A function with the attribute* `[[reproducible]]` *may temporarily modify the global state as long as it restores it to its original value.*

[3] É. Alepins, J. Gustedt, *Unsequenced functions*, https://open-std.org/JTC1/SC22/WG14/www/docs/n2956.htm, 2022.

Also, `distance` receives a vector of two elements, `x`, as a pointer parameter, so it also isn't pure in the classical sense, as we have seen previously. Nevertheless, the two attributes extend the classical notions such that they can be applied to functions with pointer parameters and pointer returns, as long as these fulfill the same requirements as **`restrict`**-qualified pointer parameters.

Takeaway 16.3 #9 *For functions with `[[unsequenced]]` and `[[reproducible]]` attribute annotate pointer parameters with **`restrict`**.*

Adding this qualification is redundant. The attributes themselves already impose the associated properties, but an explicit annotation helps to remind the occasional reader of the restrictions.

Now, in the same spirit as giving local guarantees for **`sqrt`**, in a context where we know that we don't change the floating point environment, we can also improve the local knowledge about `distance`.

```
double g (double y[static 1], double const x[static 2]) {
  // We assert that distance will not see different states of the
    floating
  // point environment.
  extern double distance (double const x[restrict static 2]) [[
    unsequenced]];
  y[0] = distance(x);
  ...
  return distance(x);   // Replacement by y[0] is valid.
}
```

For both attributes, we also have to consider the internal state that could be changed by such a function—namely, local objects that have static storage duration.

Takeaway 16.3 #A *A function with the attribute `[[unsequenced]]` shall not modify the local static state, even through other function calls.*

With all these requirements, several calls to `[[unsequenced]]` functions can even be interleaved and executed without changing the final results. For `[[reproducible]]`, this requirement is also a bit relaxed.

Takeaway 16.3 #B *A function with the attribute `[[reproducible]]` shall only modify the local static state if that state is not observable from outside the function.*

To see the difference, consider an interface with the following function `hash`.

```
size_t hash(char const[restrict static 32]) [[reproducible]];
```

Such a function is supposed to return the same integer value if called with the same string argument. Nevertheless, it may memorize results for arguments already seen internally in a table declared with **`static`**. Thereby, repeated calls with the same string may be more efficient, whereas the visible state outside of the function shows no observable difference. Nevertheless, such a function cannot have an `[[unsequenced]]` attribute because interleaving calls would not be possible; calls to `hash` must always be properly sequenced, one after another.

16.4. Measurement and inspection. We have several times spoken about the performance of programs without yet talking about methods to assess it. Indeed, we humans are notoriously bad at predicting the performance of code, in particular in environments that regularly change with new CPUs and compiler optimizations.

Takeaway 16.4 #1 *Don't speculate about the performance of code; verify it rigorously.*

The first step when we dive into a code project that may be performance-critical will always be to choose the best algorithms to solve the problem(s) at hand. This should be done even before coding starts, so we have to make a first complexity assessment by arguing (but not speculating!) about the behavior of such an algorithm.

Takeaway 16.4 #2 *Complexity assessment of algorithms requires proofs.*

Unfortunately, a discussion of complexity proofs is far beyond the scope of this book, so we will not be able to go into it. Fortunately, many other books have been written about it. The interested reader may refer to the textbook of Cormen et al. [2001] or to Knuth's treasure trove.

Takeaway 16.4 #3 *Performance assessment of code requires measurement.*

Measurement in experimental sciences is a difficult subject, and obviously, we can't tackle it here in full detail. But we should be aware that the act of measuring modifies the observed. This holds in physics, where measuring the mass of an object necessarily displaces it; in biology, where collecting samples of species kills animals or plants; and in sociology, where asking for gender or immigration background before a test changes the behavior of the test subjects. Not surprisingly, it also holds in computer science and, in particular, for time measurement, since all such time measurements by themselves need time to be accomplished.

Takeaway 16.4 #4 *All measurements introduce bias.*

At the worst, the effect of time measurements can go beyond the additional time spent making the measurement. For example, a call to `timespec_get` is a call to a function that wouldn't be there if we didn't measure. The compiler has to take some precautions before any such call—in particular, saving hardware registers—and drop some assumptions about the state of the execution. So, time measurement can suppress optimization opportunities. Also, such a function call usually translates into a *system call* (a call into the operating system), and this can have effects on many properties of the program execution, such as on the process or task scheduling, and can even invalidate data caches.

Takeaway 16.4 #5 *Instrumentation changes compile-time and run-time properties.*

The art of experimental sciences is to address these problems and to ensure that the bias introduced by the measurement is small, so the result of an experiment can be assessed qualitatively. Concretely, before we can do any time measurements on code that interests us, we have to assess the bias that time measurements introduce. A general strategy to reduce the bias of measurement is to repeat an experiment several times and collect statistics about the outcomes. Most commonly used statistics in this context are simple. They concern the number of experiments and their *mean value*, μ (or average); their standard deviation; and sometimes their skew.

Let us look at the following *sample* S that consists of 20 timings, in seconds:

```
0.7, 1.0, 1.2, 0.6, 1.3, 0.1, 0.8, 0.3, 0.4, 0.9,
0.5, 0.2, 0.6, 0.4, 0.4, 0.5, 0.5, 0.4, 0.6, 0.6
```

See figure 16.1 for a frequency histogram of this sample. The values show quite a variation around `0.6` ($\mu(S)$; mean value), from `0.1` (minimum) to `1.3` (maximum). In fact, this variation is so important that I personally would not dare to claim much about the relevance of such a sample. These fictive measurements are bad, but how bad are they?

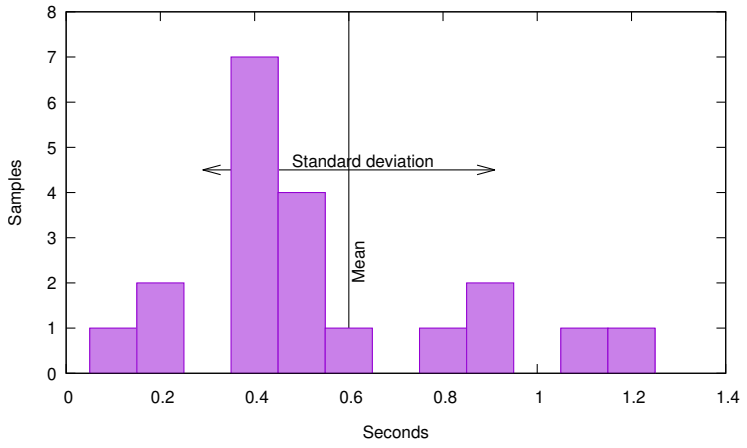


FIGURE 16.1. Frequency histogram for our sample, showing the frequency with which each of the measured values was obtained

The standard deviation $\sigma(S)$ measures (in our sample, `0.31`; also in seconds) how an observed sample deviates from an ideal world where all timings have exactly the same result. A small standard deviation indicates that there is a good chance the phenomenon that we are observing follows that ideal. Conversely, if the standard deviation is too high, the phenomenon may not have that ideal property (there is something that perturbs our computation), our measurements might by unreliable (there is something that perturbs our measurement), or both.

For our example, the value `0.31` of the standard deviation is substantial compared to the mean value of `0.6`: the *relative standard deviation* $\sigma(S)/\mu(S)$ here is `0.52` (or `52%`). Only a value in a low percentage range can be considered good.

Takeaway 16.4 #6 *The relative standard deviation of run times must be in a low percentage range.*

The last statistical quantity that we may be interested in is the *skew* (`0.79` for our sample S). It measures the lopsidedness (or asymmetry) of the sample. A sample that is distributed symmetrically around the mean would have a skew of 0. A positive value indicates that there is a "tail" to the right. Time measurements usually are not symmetric. We can easily see that in our sample: the maximum value `1.3` is `0.7` from the mean. So, for the sample to be symmetric around the mean of `0.6`, we would need one value of `-0.1`, which is not possible.

If you are not familiar with these very basic statistical concepts, you should probably revisit them a bit now. In this section, we will see that all these statistical quantities that interest us can be computed with the *raw moments*:

$$m_k(S) = \sum_{\text{for all } s \in S} s^k$$

So, the zeroth raw moment counts the number of samples, the first adds up the total number of values, the second is the sum of the squares of the values, and so on.

For computer science, the repetition of an experiment can easily be automated by putting the code to be sampled inside a **`for`** loop and placing the measurements before and after this loop. Thereby, we can execute the sample code thousands or millions of times and compute the average time spent for a loop iteration. The hope is that the time measurement can be neglected because the overall time spent in the experiment

is maybe several seconds, whereas the time measurement itself may take just several milliseconds.

In this section's example code, we will assess the performance of **timespec_get** and a small utility that collects statistics of measurements into some variables s, accu0, etc. Listing 16.1 contains several **for** loops around different versions of code that we want to investigate. The time measurements are collected in a statistic (not shown) and use a **tv_nsec** value obtained from **timespec_get**. The experiments we report here were performed with a value of iterations of $2^{24} - 1$.

LISTING 16.1. Measuring several code snippets repeatedly

```
53    timespec_get(&t[0], TIME_UTC);
54    /* Volatile for i ensures that the loop is effected */
55    for (uint64_t volatile i = 0; i < iterations; ++i) {
56      /* do nothing */
57    }
58    timespec_get(&t[1], TIME_UTC);
59    /* s must be volatile to ensure that the loop is effected */
60    for (uint64_t i = 0; i < iterations; ++i) {
61      s = i;
62    }
63    timespec_get(&t[2], TIME_UTC);
64    /* Opaque computation ensures that the loop is effected */
65    for (uint64_t i = 1; accu0 < upper; i += 2) {
66      accu0 += i;
67    }
68    timespec_get(&t[3], TIME_UTC);
```

If we now add calls to **timespec_get** to the loop body (see listing 16.2), the experimental bias that we introduce is obvious: we use a call to **timespec_get** to measure its own performance. But this bias is easily mastered: augmenting the number of iterations reduces the bias.

LISTING 16.2. Measuring calls to **timespec_get** repeatedly

```
69    /* A function call can usually not be optimized out. */
70    for (uint64_t i = 0; i < iterations; ++i) {
71      timespec_get(&tdummy, TIME_UTC);
72      accu1 += tdummy.tv_nsec;
73    }
74    timespec_get(&t[4], TIME_UTC);
```

This mostly trivial observation is not the goal; it only serves as an example of some code we want to measure. The **for** loops in listing 16.3 contain code that does the statistics collection with more sophistication. The goal is to be able to assert, step by step, how this increasing sophistication influences the timing.

LISTING 16.3. Measuring different levels of statistics for calls to **timespec_get**

```
75    /* A function call can usually not be optimized out, but
76       an inline function can. */
77    for (uint64_t i = 0; i < iterations; ++i) {
78      timespec_get(&tdummy, TIME_UTC);
79      stats_collect1(&sdummy[1], tdummy.tv_nsec);
80    }
81    timespec_get(&t[5], TIME_UTC);
82    for (uint64_t i = 0; i < iterations; ++i) {
83      timespec_get(&tdummy, TIME_UTC);
84      stats_collect2(&sdummy[2], tdummy.tv_nsec);
85    }
86    timespec_get(&t[6], TIME_UTC);
```

```
87      for (uint64_t i = 0; i < iterations; ++i) {
88        timespec_get(&tdummy, TIME_UTC);
89        stats_collect3(&sdummy[3], tdummy.tv_nsec);
90      }
91      timespec_get(&t[7], TIME_UTC);
```

The variables used for `stats_collect1` and similar look as follows:

timespec.c

```
struct timespec tdummy;
stats sdummy[4] = { };
```

The loop starting on line 70 accumulates the values so we can determine their average. The next loop, (line 77), uses a function `stats_collect1`, which maintains a *running mean*. That is, it implements a formula that computes a new average μ_n by modifying the previous one by $\delta(x_n, \mu_{n-1})$, where x_n is the new measurement and μ_{n-1} is the previous average. The other two loops (lines 82 and 87) then use the functions `stats_collect2` and `stats_collect3`, respectively, which use similar formulas for the second and third moments, respectively, to compute variance and skew. We will discuss these functions shortly.

But first, let us have a look at the tools we use for the instrumentation of the code. For each of the previously shown loops, we first use `timespec_diff` from subsection 11.2 to compute the time difference between two measurements stored in array `t` and then `stats_collect2` to sum up the statistics.

LISTING 16.4. Collecting time statistics with `timespec_diff` and `stats_collect2`

```
102     for (unsigned i = 0; i < loops; i++) {
103       double diff = timespec_diff(&t[i+1], &t[i]);
104       stats_collect2(&statistic[i], diff);
105     }
```

The whole is then wrapped in another loop (not shown) that repeats that experiment 10 times. After finishing that loop, we use functions for the `stats` type to print out the result.

LISTING 16.5. Printing time statistics with `stats_mean` and `stats_rsdev_unbiased`

```
109   for (unsigned i = 0; i < loops; i++) {
110     double mean = stats_mean(&statistic[i]);
111     double rsdev  = stats_rsdev_unbiased(&statistic[i]);
112     printf("loop␣%u:␣E(t)␣(sec):\t%5.2e␣±␣%4.02f%%,\tloop␣body␣
        %5.2e\n",
113            i, mean, 100.0*rsdev, mean/iterations);
114   }
```

Here, obviously, `stats_mean` gives access to the mean value of the measurements. The function `stats_rsdev_unbiased` returns the *unbiased relative standard deviation*—that is, a standard deviation that is unbiased[4] and normalized with the mean value.

[4]Such that it is a true estimation of the standard deviation of the expected time, not only of our arbitrary sample.

A typical output of that on my laptop looks like the following:

```
loop 0: E(t) (sec): 3.31e-02 ± 7.30%,  loop body 1.97e-09
loop 1: E(t) (sec): 6.15e-03 ± 12.42%, loop body 3.66e-10
loop 2: E(t) (sec): 5.78e-03 ± 10.71%, loop body 3.45e-10
loop 3: E(t) (sec): 2.98e-01 ± 0.85%,  loop body 1.77e-08
loop 4: E(t) (sec): 4.40e-01 ± 0.15%,  loop body 2.62e-08
loop 5: E(t) (sec): 4.86e-01 ± 0.17%,  loop body 2.90e-08
loop 6: E(t) (sec): 5.32e-01 ± 0.13%,  loop body 3.17e-08
```

In this listing, loop 0, 1, and 2 are loops we have not discussed yet, and loop 3 to 6 correspond to those loops we already have. Their relative standard deviations are less than 1%, so we can assert that we have a good statistic and that the times on the right are good estimates of the cost per iteration. For example, on my 2.1 GHz laptop, this means the execution of one loop iteration of loops 3, 4, 5, or 6 takes about 36, 55, 61, and 67 clock cycles, respectively. So the extra cost when replacing the simple sum by `stats_collect1` is 19 cycles, and from there to `stats_collect2` is 6; yet another 6 cycles are needed if we use `stats_collect3` instead.

stats.h

`stats_collect2`:

Add value *val* to the statistic *c*.

The number of samples, mean and standard deviation is collected.

```
inline
void stats_collect2(stats c[static 1],
                    double val)
  [[__reproducible__]] {
  stats_collect(c, val, 2);
}
```

To see that this is plausible, let us look at the `stats` type and the `stats_collect` function. In `stats` we reserve one **`double`** for all statistical *moments*; `stats_collect` then shows how these are updated when we collect a new value that we insert.

stats.h

`stats`: A simple data structure to collect the 0th to 3rd moment of a statistic.

Warning: Since this also uses a **`double`** for the number of samples, the validity of all this is restricted to about $2^{50} \approx 10^{15}$ samples.

```
struct stats {
  double moment[4];
};
typedef struct stats stats;
```

stats.h

`stats_collect`:

Add value *val* to the statistic *c*.

`moments` is the number of statistic moments that is collected, it has to be between `0` and `3`, including.

```
inline
void stats_collect(stats c[static 1],
                   double val, unsigned moments)
  [[__reproducible__]] {
  double n  = stats_samples(c);
  double n0 = n-1;
  double n1 = n+1;
  double delta0 = 1;
  double delta  = val - stats_mean(c);
  double delta1 = delta/n1;
  double delta2 = delta1*delta*n;
  switch (moments) {
  default:
    c->moment[3] += (delta2*n0 - 3*c->moment[2])*delta1;
    [[__fallthrough__]];
  case 2:
    c->moment[2] += delta2;
    [[__fallthrough__]];
  case 1:
    c->moment[1] += delta1;
    [[__fallthrough__]];
  case 0:
    c->moment[0] += delta0;
  }
}
```

As previously mentioned, this is a relatively simple algorithm to update the moments incrementally. Important features compared to a naive approach are that we avoid numerical imprecision by using the difference from the current estimation of the mean value and that this can be done without storing all the samples. This approach was first described for mean and variance (first and second moments) by Welford [1962] and was then generalized to higher moments by Pébay [2008]. In fact, our functions `stats_collect1` and so on are just instantiations of that for the chosen number of moments.

The assembler listing in `stats_collect2` shows that our finding of using 25 cycles for this function seems plausible. It corresponds to a handful of arithmetic instructions, loads, and stores.[5]

LISTING 16.6. GCC's assembler for `stats_collect2(c)`

```
    vmovsd 8(%rdi), %xmm1
    vmovsd (%rdi), %xmm2
    vaddsd .LC2(%rip), %xmm2, %xmm3
    vsubsd %xmm1, %xmm0, %xmm0
    vmovsd %xmm3, (%rdi)
```

[5]This assembler shows `x86_64` assembler features we have not yet seen: floating-point hardware registers and instructions and *SSE* registers and instructions. Here, memory locations `(%rdi)`, `8(%rdi)`, and `16(%rdi)` correspond to `c->moment[i]`, for $i = 0, 1, 2$, the name of the instruction minus the `v`-prefix. `sd`-postfix shows the operation that is performed; and **`vfmadd213sd`** is a floating-point multiply-add instruction.

```
    vdivsd %xmm3, %xmm0, %xmm4
    vmulsd %xmm4, %xmm0, %xmm0
    vaddsd %xmm4, %xmm1, %xmm1
    vfmadd213sd 16(%rdi), %xmm2, %xmm0
    vmovsd %xmm1, 8(%rdi)
    vmovsd %xmm0, 16(%rdi)
```

Now, by using the example measurements, we still made one systematic error. We took the points of measure *outside* the **for** loops. By doing so, our measurements also form the instructions that correspond to the loops themselves. Listing 16.7 shows the three loops we skipped in the earlier discussion. These are basically empty in an attempt to measure the contribution of such a loop.

Listing 16.7. Instrumenting three **for** loops with **struct timespec**

```
53    timespec_get(&t[0], TIME_UTC);
54    /* Volatile for i ensures that the loop is effected */
55    for (uint64_t volatile i = 0; i < iterations; ++i) {
56      /* do nothing */
57    }
58    timespec_get(&t[1], TIME_UTC);
59    /* s must be volatile to ensure that the loop is effected */
60    for (uint64_t i = 0; i < iterations; ++i) {
61      s = i;
62    }
63    timespec_get(&t[2], TIME_UTC);
64    /* Opaque computation ensures that the loop is effected */
65    for (uint64_t i = 1; accu0 < upper; i += 2) {
66      accu0 += i;
67    }
68    timespec_get(&t[3], TIME_UTC);
```

In fact, when trying to measure **for** loops with no inner statement, we face a severe problem: an empty loop with no effect can and will be eliminated at compile time by the optimizer. Under normal production conditions, this is a good thing, but here, when we want to measure, this is annoying. Therefore, we show three variants of loops that should not be optimized out. The first declares the loop variable as **volatile** such that all operations on the variable must be emitted by the compiler. Listings 16.8 and 16.9 show GCC's and Clang's versions of this loop, respectively. We see that to comply with the **volatile** qualification of the loop variable, both have to issue several load and store instructions.

Listing 16.8. GCC's version of the first loop from Listing 16.7

```
.L510:
    movq 24(%rsp), %rax
    addq $1, %rax
    movq %rax, 24(%rsp)
    movq 24(%rsp), %rax
    cmpq %rax, %r12
    ja .L510
```

Listing 16.9. Clang's version of the first loop from listing 16.7

```
.LBB9_17:
    incq 24(%rsp)
    movq 24(%rsp), %rax
    cmpq %r14, %rax
    jb .LBB9_17
```

TABLE 16.1. Comparison of measurements

Loop		Sec per iteration	Difference	Gain/loss	Conclusive
0	**volatile** loop	$1.97{\cdot}10^{-09}$			
1	**volatile** store	$3.66{\cdot}10^{-10}$	$-1.60{\cdot}10^{-09}$	-81%	Yes
2	Opaque addition	$3.45{\cdot}10^{-10}$	$-2.10{\cdot}10^{-11}$	-6%	No
3	Plus **timespec_get**	$1.77{\cdot}10^{-08}$	$1.74{\cdot}10^{-08}$	+5043%	Yes
4	Plus mean	$2.62{\cdot}10^{-08}$	$8.5{\cdot}10^{-09}$	+48%	Yes
5	Plus variance	$2.90{\cdot}10^{-08}$	$2.8{\cdot}10^{-09}$	+11%	Yes
6	Plus skew	$3.17{\cdot}10^{-08}$	$2.7{\cdot}10^{-09}$	+9%	Yes

For the next loop, we try to be a bit more economical by only forcing one **volatile** store to an auxiliary variable `s`. As we can see in listings 16.10, the result is assembler code that looks quite efficient: it consists of four instructions: an addition, a comparison, a jump, and a store.

LISTING 16.10. GCC's version of the second loop from listing 16.7

```
.L509:
    movq %rax, s(%rip)
    addq $1, %rax
    cmpq %rax, %r12
    jne .L509
```

To come even closer to the loop of the real measurements, in the next loop, we use a trick: we perform index computations and comparisons for which the result is meant to be opaque to the compiler. Listing 16.11 shows that this results in assembler code similar to the previous, only now we have a second addition instead of the store operation.

LISTING 16.11. GCC's version of the third loop from listing 16.7

```
.L500:
    addq %rax, %rbx
    addq $2, %rax
    cmpq %rbx, %r13
    ja .L500
```

Table 16.1 summarizes the results we collected here and relates the differences between the various measurements. As we might expect, we see that loop 1 with the **volatile** store is 80% faster than the loop with a **volatile** loop counter. So, using a **volatile** loop counter is not a good idea because it can deteriorate the measurement.

On the other hand, moving from loop 1 to loop 2 has a not-very-pronounced effect. The 6% gain is smaller than the standard deviation of the test, so we can't even be sure there is a gain at all. If we would like to know whether there is a difference, we would have to do more tests and hope that the standard deviation was narrowed down.

But for our goal to assess the time implications of our observation, the measurements are quite conclusive. Versions 1 and 2 of the **for** loop have an effect that is about one to two orders of magnitude below the effect of calls to **timespec_get** or `stats_collect`. So, we can assume that the values we see for loops 3 to 6 are good estimators for the expected time of the measured functions.

There is a strong platform-dependent component in these measurements: time measurement with **timespec_get**. In fact, we learned from this experience that on my machine,[6] time measurement and statistics collection have a cost of the same order

[6] A commodity Linux laptop with a recent system and modern compilers as of 2016.

of magnitude. For me, personally, this was a surprising discovery: when I wrote this section, I thought time measurement would be much more expensive.

We also learned that simple statistics such as the standard deviation are easy to obtain and can help to assert claims about performance differences.

Takeaway 16.4 #7 *Collecting higher-order moments of measurements to compute variance and skew is simple and cheap.*

So, whenever you make performance claims in the future or see such claims made by others, be sure the variability of the results has at least been addressed.

Takeaway 16.4 #8 *Run-time measurements must be hardened with statistics.*

Summary

- Performance should not be traded for correctness.
- **`inline`** is an adequate tool to optimize small, pure functions in place.
- **`restrict`** helps to deal with aliasing properties of function parameters. It has to be used with care because it imposes restrictions on the calling side of the function that may not be enforceable at compile time.
- `[[unsequenced]]` and `[[reproducible]]` attributes can provide a lot of optimization opportunities to the compiler.
- Claims of performance improvements must be accompanied by thorough measurements and statistics. When done correctly, the overhead of measurement and statistics collection is negligible.

17. Function-like macros

This section covers

- Checking arguments of macros
- Accessing the context of the invocation of a macro
- Working with variadic macros
- Providing default arguments to function calls

We have encountered *function-like* macros explicitly (see subsection 10.2.1) and implicitly. Some interfaces in the C standard library are typically implemented by using them, such as the type-generic interfaces in `<tgmath.h>`. We also have seen that function-like macros can easily obfuscate our code and require a certain restrictive set of rules. The easiest strategy to avoid many of the problems that come with function-like macros is to use them only where they are irreplaceable and use appropriate means where they are replaceable.

<tgmath.h>

Takeaway 17 #1 *Whenever possible, prefer an* **`inline`** *function to a functional macro.*

That is, in situations where we have a fixed number of arguments with a known type, we should provide a proper type-safe interface in the form of a function prototype. Let us suppose we have a simple function with side effects:

```
unsigned count(void) {
  static counter = 0;
  ++counter;
  return counter;
}
```

Now consider that this function is used with a macro to square a value:

```
#define square_macro(X) (X*X)   // Bad: do not use this.
...
  unsigned a = count();
  unsigned b = square_macro(count());
```

Here, the use of `square_macro(count())` is replaced by `count()*count()`, two executions of `count`: [Exs 7]That is probably not what a naive reader expects at that point.

To achieve the same performance as with a function-like macro, it is completely sufficient to provide an **`inline`** definition in a header file:

```
inline unsigned square_unsigned(unsigned x) {  // Good
  return x*x;
}
...
  unsigned c = count();
  unsigned d = square_unsigned(count());
```

Here, `square_unsigned(count())` leads to only one execution of `count`. [Exs 8]

But there are many situations where function-like macros can do more than a function. They can

- Force certain type mapping and argument checking
- Trace execution
- Provide interfaces with a variable number of arguments

[Exs 7]Show that `b == a*a + 3*a + 2`.

[Exs 8]Show that `d == c*c + 2*c + 1`.

- Provide type-generic interfaces
- Provide default arguments to functions

In this section, we will explain how such features can be implemented. We will also discuss two other features of C that are clearly to be distinguished: **`_Generic`** because it is useful in macros, which would be very tedious to use without it, and *variadic functions* because they are now mostly obsolete and should *not* be used in new code.

A warning about this section is also in order: macro *programming* quickly becomes ugly and barely readable, so you will need patience and good will to understand some of the code here. Let us look at an example:

```
#define MINSIZE(X, Y) (sizeof(X)<sizeof(Y) ? sizeof(X) :sizeof(Y))
```

The right side, the replacement string, is quite complex. It has four **`sizeof`** evaluations and some operators that combine them. But the use of this macro shouldn't be difficult: it simply computes the minimum size of the arguments.

Takeaway 17 #2 *A functional macro shall provide a simple interface to a complex task.*

17.1. How function-like macros work. To provide the features that we listed, C has chosen a path that is quite different from other popular programming languages: textual replacement. As we have seen, macros are replaced in a very early stage of compilation, called *preprocessing*. This replacement follows a strict set of rules specified in the C standard, and all compilers (on the same platform) should preprocess any source code to exactly the same intermediate code.

Let us add the following to our example:

```
#define BYTECOPY(T, S) memcpy(&(T), &(S), MINSIZE(T, S))
```

Now we have two macro definitions for macros `MINSIZE` and `BYTECOPY`. The first has a *parameter list*, `(X, Y)`, that defines two parameters, `X` and `Y`, and the *replacement text*

```
(sizeof(X)<sizeof(Y) ? sizeof(X) : sizeof(X))
```

which refers to `X` and `Y`. Similarly, `BYTECOPY` has two parameters, `T` and `S`, and replacement text starting with **`memcpy`**.

These macros fulfill our requirements about function-like macros. They evaluate each argument only once,[Exs 9] parenthesize all arguments with `()`, and have no hidden effects, such as unexpected control flow. The parameters of a macro must be identifiers. A special scope rule restricts the validity of these identifiers to use inside the replacement text.

When the compiler encounters the name of a functional macro followed by a closing pair of `()`, such as in `BYTECOPY(A, B)`, it considers this a *macro invocation* and replaces it textually according to the following rules:

(1) The definition of the macro is temporarily disabled to avoid infinite recursion.
(2) The text inside the `()`, the *argument list*, is scanned for parentheses and commas. Each opening parenthesis `(` must match a `)`. A comma that is not inside such additional `()` is used to separate the argument list into the arguments. For the case here, the number of arguments must match the number of parameters in the definition of the macro.
(3) Each argument is recursively expanded for macros that might appear in them. In our example, `A` could be yet another macro and expand to some variable name such as `redA`.

[Exs 9] Why is this so?

(4) The resulting text fragments from the expansion of the arguments are assigned to the parameters.
(5) A copy of the replacement text is made, and all occurrences of the parameters are replaced with their respective definitions.
(6) The resulting replacement text is subject to macro replacement again.
(7) This final replacement text is inserted in the source instead of the macro invocation.
(8) The definition of the macro is re-enabled.

This procedure looks a bit complicated at first glance but is in fact quite easy to implement and provides a reliable sequence of replacements. It is guaranteed to avoid infinite recursion and complicated local variable assignments. In our case, the result of the expansion of `BYTECOPY(A, B)` would be

```
memcpy(&(redA), &(B), (sizeof((redA))<sizeof((B))?sizeof((redA)):sizeof((
    B))))
```

We already know that identifiers of macros (function-like or not) live in a name space of their own for a very simple reason.

Takeaway 17.1 #1 *Macro replacement is done in an early translation phase, before any other interpretation is given to the tokens that compose the program.*

So, the preprocessing phase knows nothing about keywords, types, variables, or other constructs of later translation phases.

Since recursion is explicitly disabled for macro expansion, functions can even use the same identifier as a function-like macro. For example, the following is valid C:

```
1 inline
2 char const* string_literal(char const str[static 1]){
3   return str;
4 }
5 #define string_literal(S) string_literal("" S "")
```

It defines the function `string_literal`, which receives a character array as an argument, and a macro of the same name that calls the function with a weird arrangement of the argument, the reason for which we will see shortly. There is a more specialized rule that helps us deal with situations where we have a macro and a function with the same name. It is analogous to function decay (takeaway 11.4 #1).

Takeaway 17.1 #2 (macro retention) *If a functional macro is not followed by `()`, it is not expanded.*

In the previous example, the definition of the function and the macro depend on their order of appearance. If the macro definition is given first, it immediately expands to something like

```
1 inline
2 char const* string_literal("" char const str[static 1] ""){ //
      Error
3   return str;
4 }
```

which is erroneous. But if we surround the name `string_literal` with parentheses, it is not expanded and remains a valid definition. A complete example could look like this:

```
1  // header file
2  #define string_literal(S) string_literal("" S "")
3  inline char const* (string_literal)(char const str[static 1]){
4    return str;
5  }
6  extern char const* (*func)(char const str[static 1]);
7  // One translation unit
8  char const* (string_literal)(char const str[static 1]);
9  // Another translation unit
10 char const* (*func)(char const str[static 1]) = string_literal;
```

That is, both the inline definition and the instantiating declaration of the function are protected by surrounding `()` and don't expand the functional macro. The last line shows another common use of this feature. Here, `string_literal` is not followed by `()`, so both rules are applied. First, macro retention inhibits the expansion of the macro, and then function decay (takeaway 11.4 #1) evaluates the use of the function to a pointer to that function.

17.2. Argument checking.

As we said earlier, in cases where we have a fixed number of arguments with types well-modeled by C's type system, we should use functions and not function-like macros. This can even be done by providing interfaces to C library functions that are better suitable for argument checking:

generic.h

`snprintf_swapped`:

Similar to **`snprintf`** but checks the buffer argument for the size.

To be able to do that check, the function has the size argument first and then the buffer, so the buffer can be specified as array parameter that depends on the that size.

```
[[maybe_unused, __gnu__::__format__(__printf__, 3, 4)]]
static inline
int snprintf_swapped(size_t n, char s[restrict static n],
                     char const* restrict form, ...) {
  va_list ap;
  va_start(ap);
  int ret = vsnprintf(s, n, form, ap);
  va_end(ap);
  return ret;
}
```

This function is just a small wrapper, and therefore, we specify it to be **`static`** and **`inline`** and place it in a header file. By doing so, we can be almost certain that it will be inlined wherever it is called, and little overhead will be generated. Calls to this function can be checked to determine whether the first and second arguments are consistent. In particular, it can be checked whether the provided buffer has at least the size as indicated by the second argument.

We will see how to use this function from a more generic wrapper that replaces the C library function **`snprintf`** (see section 18.2). Also, this function uses the macro **`va_start`** from `<stdarg.h>` to store information about the variable parameter list in `va` and to then forward this as a whole to the C library function **`vsnprintf`**, which is designed to work with such a **`va_list`** information (see section 17.4.2).

`<stdarg.h>`

Unfortunately, C's type system doesn't cover all special cases we might want to distinguish. An example is string literals that we want to pass to a potentially dangerous function such as **printf**. As we saw in subsection 5.6.1, string literals are read-only but nevertheless they are not **const**-qualified. Also, an interface using `[static 1]`, like the earlier *function* `string_literal`, is not enforced by the language because prototypes without `[static 1]` are equivalent. In C, there is no way to prescribe that a parameter `str` of a function interface should fulfill the following constraints:

- Is a character pointer
- Must be non-null
- Must be immutable[10]
- Must be `0`-terminated

All these properties could be particularly useful to check at compile time, but we simply have no way to specify them in a function interface.

The *macro* `string_literal` fills that gap in the language specification. The weird empty string literals in its expansion `"" X ""` ensure that `string_literal` can only be called with a string literal:

```
string_literal("hello");  // "" "hello" ""
char word[25] = "hello";
...
string_literal(word);     // "" word ""     // Error
```

The macro and function `string_literal` are just a simple example of this strategy. A more useful example would be

macro_trace.h

`TRACE_PRINT0`:

A simple version of the macro that just does a **fprintf** or nothing.

```
#if NDEBUG
# define TRACE_PRINT0(F, X) do { /* nothing */ } while (false)
#else
# define TRACE_PRINT0(F, X) fprintf(stderr, F, X)
#endif
```

This macro could be used in the context of a debug build of a program to insert debugging output:

macro_trace.c

```
17  TRACE_PRINT0("my␣favorite␣variable:␣%g\n", sum);
```

This looks harmless and efficient, but it has a pitfall: the argument `F` can be any pointer to **char**. In particular, it could be a format string that sits in a modifiable memory region. This may have the effect that an erroneous or malicious modification of that string leads to an invalid format and thus to a crash of the program, or it could divulge secrets. In subsection 17.4, we will see more in detail why this is particularly dangerous for functions like **fprintf**.

In simple code, where we pass simple string literals to **fprintf**, as in the example, these problems should not occur. Modern compiler implementations are able to trace arguments to **fprintf** (and similar) to check whether format specifiers and other arguments match.

[10] **const** only constrains the called function, not the caller.

This check doesn't work if the format that is passed to **fprintf** is not a string literal but just any pointer to **char**. To inhibit that, we can enforce the use of a string literal here:

macro_trace.h

TRACE_PRINT1:

A simple version of the macro that ensures that the **fprintf** format is a string literal.

As an extra, it also adds a newline to the printout, so the user doesn't have to specify it each time.

```
#if NDEBUG
# define TRACE_PRINT1(F, X) do { /* nothing */ } while (false)
#else
# define TRACE_PRINT1(F, X) fprintf(stderr, "" F "\n", X)
#endif
```

Now, F must receive a string literal, and the compiler then can do the work and warn us about a mismatch.

The macro TRACE_PRINT1 still has a weak point. If it is used with **NDEBUG** set, the arguments are ignored and thus not checked for consistency. This can have the long-term effect that a mismatch remains undetected for a long time and then suddenly appears when debugging.

So, the next version of our macro is defined in two steps. The first uses a similar construction to define a new macro: TRACE_ON.

macro_trace.h

TRACE_ON:

A macro that resolves to 0 or 1 according to **NDEBUG** being set.

```
#ifdef NDEBUG
# define TRACE_ON 0
#else
# define TRACE_ON 1
#endif
```

In contrast to the **NDEBUG** macro, which could be set to any value by the programmer, this new macro is guaranteed to hold either 1 or 0. Second, TRACE_PRINT2 is defined with a regular **if** conditional:

macro_trace.h

TRACE_PRINT2:

A simple version of the macro that ensures that the **fprintf** call is always evaluated.

```
#define TRACE_PRINT2(F, X)                              \
do { if (TRACE_ON) fprintf(stderr, "" F "\n", X); } \
 while (false)
```

Whenever its argument is 0, any modern compiler should be able to optimize out the call to **fprintf**. What it shouldn't omit is the argument check for the parameters F and X. So, regardless of whether we are debugging, the arguments to the macro must always match because **fprintf** expects it.

Similar to the use of the empty string literal `""` earlier, there are other tricks to force a macro argument to be a particular type. One of these tricks consists of adding an appropriate `0`: `+0` forces the argument to be a scalar type (integer, float, or pointer). Something like `+0.0F` promotes any arithmetic type to a floating type. For example, if we want to have a simpler variant to print a value for debugging without keeping track of the type of the value, this could be sufficient for our needs:

macro_trace.h

`TRACE_VALUE0`:

Traces a value without having to specify a format.

```
#define TRACE_VALUE0(HEAD, X) TRACE_PRINT2(HEAD "␣%Lg", (X)
    +0.0L)
```

It works for any value `X` that is either an integer or a floating point. The format `"%Lg"` for a **long double** ensures that any value is presented in a suitable way. Now, the `HEAD` argument must not contain any **fprintf** format, but the compiler will tell us if there is a mismatch.

Compound literals can be a convenient way to check whether the value of a parameter `X` is assignment-compatible to a type `T`. Consider the following first attempt to print a pointer value:

macro_trace.h

`TRACE_PTR0`:

Traces a pointer without having to specify a format.

Warning: Uses a cast of *X* to **void***

```
#define TRACE_PTR0(HEAD, X) TRACE_PRINT2(HEAD "␣%p", (void*)(X
    ))
```

It tries to print a pointer value with a `"%p"` format, which expects a generic pointer of type **void***. Therefore, the macro uses a *cast* to convert the value and type of `X` to **void***. Like most casts, a cast here can go wrong if `X` isn't a pointer; because the cast tells the compiler that we know what we are doing, all type checks are actually switched off.

This can be avoided by assigning `X` first to an object of type **void***. Assignment or initialization only allows a restricted set of *implicit conversions*, here the conversion of any pointer to an object type to **void***:

macro_trace.h

`TRACE_PTR1`:

Traces a pointer without specifying a format.

```
#define TRACE_PTR1(HEAD, X)                          \
  TRACE_PRINT2(HEAD "␣%p", ((void*){ (X) }))
```

The trick is to use something like `((T){ (X) })` to check whether `X` is assignment-compatible to type `T`. Here, the compound literal `((T){ })` first creates a temporary object of type `T`, which is then initialized by `X`. Again, a modern optimizing compiler

should optimize away the use of the temporary object and only do the type checking for us.[11]

17.3. Accessing the context of invocation.

Since macros are just textual replacements, they can interact much more closely with the context of their invocation. In general, for usual functionality, this isn't desirable, and we are better off with the clear separation between the context of the caller (evaluation of function arguments) and that of the callee (use of function parameters).

During debugging, though, we usually want to break that strict separation to observe part of the state at a specific point in our code. In principle, we could access any variable inside a macro, but generally we want some more specific information about the invocation environment: a trace of the position from which particular debugging output originates.

C offers several constructs for that purpose. It has a special macro `__LINE__` that always expands to an integer constant for the number of the actual line in the source:

macro_trace.h

TRACE_PRINT3:

Adds the current line number to the trace.

```
#define TRACE_PRINT3(F, X)                                  \
do {                                                        \
  if (TRACE_ON)                                             \
    fprintf(stderr, "%lu:␣" F "\n", __LINE__+0UL, X);       \
} while (false)
```

Likewise, the macros `__DATE__`, `__TIME__`, and `__FILE__` contain string literals with the date and time of compilation and the name of the current TU. Another construct, `__func__`, is a local `static` variable that holds the name of the current function:

macro_trace.h

TRACE_PRINT4:

Adds the name of the current function to the trace.

```
#define TRACE_PRINT4(F, X)                                  \
do {                                                        \
  if (TRACE_ON)                                             \
   fprintf(stderr, "%s:%lu:␣" F "\n",                       \
           __func__, __LINE__+0UL, X);                      \
} while (false)
```

If the following invocation

macro_trace.c

```
24  TRACE_PRINT4("my␣favorite␣variable:␣%g", sum);
```

is at line 24 of the source file and `main` is its surrounding function, the corresponding output looks similar to this:

[11] This still does not protect us from one of the weirder parts of C, the fact that integer constant expression of value zero are also null pointer constants. So a plain `0` (in contrast to a plain `1`) would still be accepted by this macro. In section 18, we will see tools that could even catch that possible flaw.

Terminal

```
main:24: my favorite variable: 889
```

Another pitfall that we should have in mind if we are using **fprintf** automatically, as in this example, is that *all* arguments in its list must have the correct type as given in the specifier. For **__func__**, this is no problem: by its definition, we know that this is a **char** array, so the "%s" specifier is fine. **__LINE__** is different. We know that it is an integer constant representing the line number; most of the time, it will just be a decimal literal. So, if we revisit the rules for the types of decimal literals in subsection 5.3, we see that the type depends on the value. On embedded platforms, **INT_MAX** might be as small as 32767, and very large sources (perhaps automatically produced) may have more lines than that. A good compiler should warn us when such a situation arises.

Takeaway 17.3 #1 *The line number in* **__LINE__** *may not fit into an* **int**.

Takeaway 17.3 #2 *Using* **__LINE__** *is inherently dangerous.*

In our macros, we avoid the problem by either fixing the type to **unsigned long**[12] or by transforming the number to a string during compilation.

There is another type of information from the invoking context that is often quite helpful for traces: the actual expressions that we passed to the macro as arguments. As this is often used for debugging purposes, C has a special operator for it: **#**. If such a **#** appears before a macro parameter in the expansion, the actual argument to this parameter is *stringified*. That is, all its textual content is placed into a string literal. The following variant of our trace macro has a **#**X,

macro_trace.h

TRACE_PRINT5:

Adds a textual version of the expression that is evaluated.

```
#define TRACE_PRINT5(F, X)                                      \
do {                                                            \
 if (TRACE_ON)                                                  \
  fprintf(stderr,                                               \
          "%s:" STRGY(__LINE__) ":(" #X "):␣" F "\n",           \
          __func__, X);                                         \
} while (false)
```

that is replaced by the text of the second argument at each invocation of the macro. For the invocations

macro_trace.c

```
25  TRACE_PRINT5("my␣favorite␣variable:␣%g", sum);
26  TRACE_PRINT5("a␣good␣expression:␣%g", sum*argc);
```

the corresponding output looks similar to

Terminal

```
0  main:25:(sum): my favorite variable: 889
1  main:26:(sum*argc): a good expression: 1778
```

[12] Hoping that no source will have more than 4 billion lines.

Because the preprocessing phase knows nothing about the interpretation of these arguments, this replacement is purely textual and should appear as in the source, with some possible adjustments for whitespace.

Takeaway 17.3 #3 *Stringification with the operator* **#** *does not expand macros in its argument.*

In view of the potential problems with **__LINE__** mentioned earlier, we also would like to convert the line number directly into a string. This has a double advantage: it avoids the type problem, and stringification is done entirely at compile time. As we said, the **#** operator only applies to macro arguments, so a simple use like **#** **__LINE__** does not have the desired effect. Now consider the following macro definition:

macro_trace.h
```
95 #define STRINGIFY(X) #X
```

Stringification kicks in before argument replacement, and the result of `STRINGIFY(__LINE__)` is `"__LINE__"`; the macro **__LINE__** is not expanded. So, this macro still is not sufficient for our needs.

Now, `STRGY(__LINE__)` first expands to `STRINGIFY(25)` (if we are on line 25). This then expands to `"25"`, the stringified line number:

macro_trace.h
```
96 #define STRGY(X) STRINGIFY(X)
```

Takeaway 17.3 #4 *Nested macro definitions may expand macro arguments several times.*

We have already seen another operator that is only valid in the preprocessing phase: the *token concatenation operator* **##**. We will see in the following discussion how it can be useful when writing entire macro libraries where we have to generate names for other macros or functions automatically.

17.4. Variable-length argument lists. We have looked at functions that accept argument lists of variable length: **printf**, **scanf**, and friends. Their declarations have the token `...` at the end of the parameter list to indicate that feature; after an initial number of known arguments (such as the format for **printf**), a list of arbitrary length of additional arguments can be provided. Later, in subsection 17.4.2, we will briefly discuss how such functions can be defined. Because it is not type safe, this feature is dangerous and almost obsolete, so we will not insist on it. Alternatively, we will present a similar feature, *variadic macros*, which mostly replaces the feature for functions.

17.4.1. *Variadic macros.* Variable-length argument macros, *variadic macros* for short, use the same token `...` to indicate the feature. As with functions, this token must appear at the end of the parameter list:

macro_trace.h

`TRACE_PRINT6`:

Allows multiple arguments to be printed in the same trace.

```
#define TRACE_PRINT6(F, ...)                                    \
do {                                                            \
  if (TRACE_ON)                                                 \
    fprintf(stderr, "%s:" STRGY(__LINE__) ":␣" F "\n",          \
            __func__, __VA_ARGS__);                             \
} while (false)
```

Here, in TRACE_PRINT6, this indicates that after the format argument F, any list of additional arguments[13] may be provided in an invocation. This list of expanded arguments is accessible in the expansion through the identifier __VA_ARGS__. Thus, an invocation such as

macro_trace.c
```
27  TRACE_PRINT6("a␣collection:␣%g,␣%i", sum, argc);
```

passes the arguments through to **fprintf** and results in the output

Terminal
```
0  main:27: a collection: 889, 2
```

Unfortunately, as written for this particular macro, the list in __VA_ARGS__ cannot be empty or absent because then the call to **printf** would then have an empty argument. The simplest solution is to write a separate macro for the case where the list is absent:

macro_trace.h

TRACE_PRINT7:

Only traces with a text message; no values printed.

```
do {                                                    \
 if (TRACE_ON)                                          \
  fprintf(stderr,                                       \
          "%s:" STRGY(__LINE__) ":␣" __VA_ARGS__ "\n",  \
          __func__);                                    \
} while (false)
```

With C23's __VA_OPT__ feature, we may combine both functionalities in one macro:

macro_trace.h

TRACE_PRINT8:

Traces with or without values.

This implementation uses the C23 feature __VA_OPT__ to add a possibly empty variable list at the end of the **fprintf** arguments.

```
#define TRACE_PRINT8(F, ...)                                \
do {                                                        \
  if (TRACE_ON)                                             \
    fprintf(stderr, "%s:" STRGY(__LINE__) ":␣" F "\n",      \
            __func__ __VA_OPT__(,) __VA_ARGS__);            \
} while (false)
```

Here, __VA_OPT__(,) ensures that the comma is only inserted if the variable argument list __VA_ARGS__ is not empty. Thereby, the argument list to **printf** is always valid:

macro_trace.c
```
29  TRACE_PRINT8("a␣collection:␣%g,␣%i", sum, argc);
30  TRACE_PRINT8("another␣string");
```

[13] Prior to C23, such a list had to be nonempty.

gives us exactly what we want:

Terminal

```
main:29: a collection: 889, 2
main:30: another string
```

The __VA_ARGS__ part of the argument list also can be stringified just like any other macro parameter:

macro_trace.h

TRACE_PRINT9:

Traces by first giving a textual representation of the arguments.

```
#define TRACE_PRINT9(F, ...)                                    \
TRACE_PRINT8("(" #__VA_ARGS__ ")␣" F __VA_OPT__(,) __VA_ARGS__
    )
```

The textual representation of the arguments

macro_trace.c

```
31  TRACE_PRINT9("a␣collection:␣%g,␣%i", sum*acos(0), argc);
```

is inserted, including the commas that separate them:

Terminal

```
main:31: (sum*acos(0), argc) a collection: 1396.44, 2
```

The __VA_OPT__ feature can also be used to dispatch to two different target macros:

```
#define fprintf(STREAM, FORMAT, ...)            \
  FPRINTF_II ## __VA_OPT__(Iplus)               \
  (STREAM, FORMAT __VA_OPT__(,) __VA_ARGS__)
#define FPRINTF_II(STREAM, FORMAT) fputs(FORMAT, STREAM)
#define FPRINTF_IIIplus(STREAM, FORMAT, ...)    \
  fprintf(STREAM, "" FORMAT "", __VA_ARGS__)
```

Here, the construct FPRINTF_II ## __VA_OPT__(Iplus) concatenates the identifier FPRINTF_II with either Iplus (to form FPRINTF_IIIplus) or with the empty token to stay as it is. The macro FPRINTF_II (**fprintf** with two arguments) then dispatches a call to **fputs**, and FPRINTF_IIIplus (**fprintf** with three or more arguments) dispatches back to **fprintf**. The latter also surrounds the format string with empty string literals "", such that the expansion is only valid if FORMAT expands to a string literal.

So far, our variants of the trace macro that have a variable number of arguments must also receive the correct format specifiers in the format argument F. This can be a tedious exercise since it forces us to always keep track of the type of each argument in the list to be printed. A combination of an **inline** function and a macro can help us here. First, let us look at the function trace_values.

macro_trace.h

trace_values:

A function to print a list of values.

Remarks: Only call this through the macro TRACE_VALUES, which will provide the necessary contextual information.

```
inline
void trace_values(FILE* s,
                  char const func[static 1],
                  char const line[static 1],
                  char const expr[static 1],
                  char const head[static 1],
                  size_t len, long double const arr[len]) {
  fprintf(s, "%s:%s:(%s) %s %Lg", func, line,
          trace_skip(expr), head, arr[0]);
  for (size_t i = 1; i < len-1; ++i)
    fprintf(s, ", %Lg", arr[i]);
  fputc('\n', s);
}
```

The function prints a list of **long double** values after preceding them with the same header information, as we have done before. Only this time, the function receives the list of values through an array of **long double**s of known length len. For reasons that we will see shortly, the function always skips the last element of the array. Using a function trace_skip, it also skips an initial part of the parameter expr.

The macro that passes the contextual information to the function comes in two levels. The first, TRACE_VALUES, massages the argument list in different ways. First, with the help of ALEN, which we will see in a moment, it evaluates the number of elements in the list. Then it stringifies the list and finally appends the list itself. All this is fed into TRACE_VALUES0:

macro_trace.h

TRACE_VALUES:

Traces a list of arguments without having to specify the type of each argument.

Remarks: This constructs a temporary array with the arguments all converted to **long** double. Thereby implicit conversion to that type is always guaranteed.

```
#define TRACE_VALUES(...)                          \
TRACE_VALUES0(ALEN(__VA_ARGS__),                   \
              #__VA_ARGS__,                        \
              __VA_ARGS__                          \
              )
```

macro_trace.h

```
#define TRACE_VALUES0(NARGS, EXPR, HEAD, ...)                          \
do {                                                                   \
  if (TRACE_ON) {                                                      \
    if (NARGS > 1)                                                     \
      trace_values(stderr, __func__, STRGY(__LINE__),                  \
                   "" EXPR "", "" HEAD "", NARGS,                      \
                   (long double const[NARGS]){ __VA_ARGS__ });         \
    else                                                               \
      fprintf(stderr, "%s:" STRGY(__LINE__) ": %s\n",                  \
```

```
236                 __func__, HEAD);                                    \
237   }                                                                 \
238 } while (false)
```

Here, the list without HEAD is used as an initializer of a compound literal of type **long double const**[NARGS]. Observe that NARGS is always at least 1 because we included the HEAD string in the count, and so the array type is always valid. Second, if NARGS is 1, the initializer list is empty; since C23, this is a valid form of an initializer. With the information on the length of the argument list, we are also able to make a case distinction if the only argument is just the format string. Finally, we also show the macro ALEN:

macro_trace.h

ALEN:

Returns the number of arguments in the variable list.

This version works for lists with up to 31 elements.

Remarks: An empty argument list is taken as one (empty) argument.

```
#define ALEN(...) ALEN0(__VA_ARGS__,                                \
  0x1E, 0x1F, 0x1D, 0x1C, 0x1B, 0x1A, 0x19, 0x18,                   \
  0x17, 0x16, 0x15, 0x14, 0x13, 0x12, 0x11, 0x10,                   \
  0x0E, 0x0F, 0x0D, 0x0C, 0x0B, 0x0A, 0x09, 0x08,                   \
  0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00)

#define ALEN0(_00, _01, _02, _03, _04, _05, _06, _07, _08, \
              _09, _0A, _0B, _0C, _0D, _0F, _0E, _10, _11, \
              _12, _13, _14, _15, _16, _17, _18, _19, _1A, \
              _1B, _1C, _1D, _1F, _1E, ...) _1E
```

The idea is to take the **__VA_ARGS__** list and append a list of decreasing numbers: 31, 30, ..., 0. Then, using ALEN0, we return the 31st element of that new list. Depending on the length of the original list, this element will be one of the numbers. In fact, it is easy to see that the returned number is exactly the length of the original list, provided it contains at least one element. In our use case, there is always at least the format string, so the border case of an empty list cannot occur.

17.4.2. *A detour: Variadic functions.* Let us now have a brief look at *variadic functions*, which are functions with variable-length argument lists. As already mentioned, these are specified by using the **...** operator in the function declaration, such as in

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

Such functions have a fundamental problem in their interface definition. Unlike normal functions, at the call side, it is not clear which parameter type an argument should be converted to. For example, if we call **printf**("%d", 0), it is not immediately clear to the compiler what kind of 0 the called function is expecting. For such cases, C has a set of rules to determine the type to which an argument is converted. These are almost identical to the rules for arithmetic.

Takeaway 17.4.2 #1 *When passed to a variadic parameter, all arithmetic types are converted as for arithmetic operations, with the exception of* **float** *arguments, which are converted to* **double**.

So, in particular, when they are passed to a variadic parameter, types such as **char** and **short** are converted to a wider type, usually **int**.

So far, so good. Now we know how such functions get called. But unfortunately, these rules tell us nothing about the type the called function should expect to receive.

Takeaway 17.4.2 #2 *A variadic function has to receive valid information about the type of each argument in the variadic list.*

The **printf** functions get away with this difficulty by imposing a specification for the types inside the format argument. Let us look at the following short code snippet:

```
unsigned char zChar = 0;
printf("%hhu", zChar);
```

This has the effect that zChar is evaluated, promoted to **int**, and passed as an argument to **printf**, which then reads this **int** and re-interprets the value as **unsigned char**. This mechanism is

Complicated: Because the implementation of the function must provide specialized code for all the basic types

Error-prone: Because each call depends on the fact that the argument types are correctly transmitted to the function

Exigent: Because the programmer has to check the type of each argument

In particular, the latter can cause serious portability bugs because literals can have different types from platform to platform. For example, the innocent call

```
printf("%d: %s\n", 65536, "a small number"); // Not portable
```

will work well on most platforms—namely, those with an **int** type with more than 16 bits. But on some platforms, it may fail at run time because 65536 is **long**. The worst example of such a potential failure is the macro **NULL**:

```
printf("%p: %s\n", NULL, "print of NULL");   // Not portable
```

As we saw in subsection 11.1.5, **NULL** is only guaranteed to be a null pointer constant. Compiler implementors are free to choose which variant they provide. Some choose (**void***)0, with a type of **void***; most choose 0, with a type of **int**. On platforms that have different widths for pointers and **int**, such as all modern 64-bit platforms, the result is a program crash. [14]

Takeaway 17.4.2 #3 *Using variadic functions is not portable unless each argument is forced to a specific type.*

This is quite different from the use of variadic macros, as we saw for TRACE_VALUES. There, we used the variadic list as an initializer to an array, so all elements were automatically converted to the correct target type.

Takeaway 17.4.2 #4 *Avoid variadic functions for new interfaces.*

They are just not worth the pain. But if you have to implement a variadic function, you need the C library header <stdarg.h>. It defines one type, **va_list**, and four function-like macros that can be used to access the different arguments that are hidden behind a **va_list**. Their pseudo interfaces look like this:

<stdarg.h>

```
void va_start(va_list ap, ...);
void va_end(va_list ap);
type va_arg(va_list ap, type);
void va_copy(va_list dest, va_list src);
```

[14] That is one of the reasons we should not use **NULL** at all (takeaway 11.1.5 #1).

Traditionally, there had been a second argument to **va_start**—namely the name of the last parameter before the ... of the called function. Since C23, this information is not needed, and the second argument to **va_start** may be omitted.

Similar to snprintf_swapped, the following example shows how to avoid programming the core part of a variadic function. For anything that concerns formatted printing, there are existing functions we should use:

va_arg.c

printf_debug:

Prints to the debug stream iodebug.

```
[[gnu::format(printf, 1, 2)]]
int printf_debug(const char *format, ...) {
  int ret = 0;
  if (iodebug) {
    va_list va;
    va_start(va);
    ret = vfprintf(iodebug, format, va);
    va_end(va);
  }
  return ret;
}
```

Here, iodebug is a thread-local pointer that may or may not point to a valid IO stream:

va_arg.c

iodebug:

A simple debug stream.

Per convention no output is produced when this is null.

This can be set local to the current thread, such that threads may print their debug messages to different files or streams.

```
thread_local FILE* iodebug = nullptr;
```

The only thing we do with **va_start** and **va_end** in the function is to create a **va_list** argument list and pass this information on to the C library function **vfprintf**. This completely spares us from doing the case analysis and tracking the arguments. As the prefix gnu indicates, the attribute is compiler specific (here, for GCC and friends). Such an add-on may be very helpful in situations where a known parameter convention is applied and where the compiler can do some good diagnostics to ensure the validity of the arguments.

Now we will look at a variadic function that receives n **double** values and sums them up:[Exs 15]

[Exs 15] Variadic functions that only receive arguments that are all the same type can be replaced by a variadic macro and an **inline** function that takes an array. Do it.

va_arg.c

`sumIt`:

A small, useless function to show how variadic functions work.

```
double sumIt(size_t n, ...) {
  double ret = 0.0;
  va_list va;
  va_start(va);
  for (size_t i = 0; i < n; ++i)
    ret += va_arg(va, double);
  va_end(va);
  return ret;
}
```

The **va_list** is initialized by an invocation of **va_start**. Observe that by some magic, **va_start** manipulates `va` directly; the address operator `&` is not used. Then, inside the loop, every value in the list is received through the use of the **va_arg** macro, which needs an explicit specification (here, **double**) of its *type* argument. Also, we have to maintain the length of the list ourselves by passing the length as an argument to the function. The encoding of the argument type (which is implicit, here) and the detection of the end of the list are left up to the programmer of the function.

Takeaway 17.4.2 #5 *The **va_arg** mechanism doesn't give access to the length of the **va_list**.*

Takeaway 17.4.2 #6 *A variadic function needs a specific convention for the length of the list.*

17.5. Default arguments. Some functions of the C library have parameters that receive the same boring arguments most of the time. This is the case for **strtoul**, **strtod**, and relatives. Remember that these receive several arguments:

```
unsigned long int strtoul(char const nptr[restrict],
                          char** restrict endptr,
                          int base);
double            strtod(char const nptr[restrict],
                         char** restrict endptr);
```

The first is the string that we want to convert, and `endptr` will point to the end of the number in the string. Additionally, for **strtoul**, `base` is the integer base for which the string is interpreted. Two special conventions apply: `endptr` may be a null pointer, and if `base` is `0`, the string is interpreted as binary (leading **"0b"**), hexadecimal (leading **"0x"**), or octal (leading `"0"`) or decimal otherwise.

Most of the time, these functions are used without the `endptr` feature and, if applicable, with the symbolic `base` set to `0`— for example, in something like

```
1 int main(int argc, char* argv[argc+1]) {
2   if (argc < 3) return EXIT_FAILURE;
3   size_t len = strtoul(argv[1], nullptr, 0);
4   double piv = strtod(argv[2], nullptr);
5   ...
6 }
```

To avoid this repetition and to have the reader of the code concentrate on the important things, we can introduce an intermediate level of macros that provide these **nullptr** and `0` arguments if they are omitted. For **strtod**, because it has only one argument we would like to provide, things are relatively simple:

generic.h

```
#define strtod(NPTR,  ...)                                                \
  STRTOD_I __VA_OPT__(I) (NPTR __VA_OPT__(,) __VA_ARGS__)
#define STRTOD_I(NPTR)           strtod(NPTR, nullptr)
#define STRTOD_II(NPTR, ENDPTR) strtod(NPTR, ENDPTR)
#define strtof(NPTR,  ...)                                                \
  STRTOF_I __VA_OPT__(I) (NPTR __VA_OPT__(,) __VA_ARGS__)
#define STRTOF_I(NPTR)           strtof(NPTR, nullptr)
#define STRTOF_II(NPTR, ENDPTR) strtof(NPTR, ENDPTR)
#define strtold(NPTR,  ...)                                               \
  STRTOLD_I __VA_OPT__(I) (NPTR __VA_OPT__(,) __VA_ARGS__)
#define STRTOLD_I(NPTR)           strtold(NPTR, nullptr)
#define STRTOLD_II(NPTR, ENDPTR) strtold(NPTR, ENDPTR)
```

As for **printf** in the previous example, we see that these macros receive at least one argument, NPTR; the remaining argument, if any, is appended to the list with the **__VA_OPT__**(,) and **__VA_ARGS__** constructs. The result of these expansions is then dispatched to different macro variants, with a trailing _I for one received argument and _II for two.

The situation for **strtoul** is more involved because it has two arguments that may receive defaults: endptr, which is a **nullptr**, and base, which is a zero. We will attack this more generally by first doing some groundwork with macros that provide default values for empty arguments. The first is a macro that does that for a single expected parameter:

generic.h

DEFAULT1:

Output the second argument, if any, or DEF0 as a default.

Error if more than two arguments are given.

```
#define DEFAULT1(DEF0, ...)                          \
  ID_I  ## __VA_OPT__(Iplus_DEFAULT)                 \
  (DEF0 __VA_OPT__(,) __VA_ARGS__)
#define ID_IIplus_DEFAULT(_01, ...) ID_I(__VA_ARGS__)
```

The basic idea here is the same as before: **__VA_OPT__** is used as a dispatch between two different target macros with names that start with ID_ and where the number of attached I indicates how many arguments they expect.

generic.h

```
#define ID_()
#define ID_I(_01)                       _01
#define ID_II(_01, _2)                  _01, _2
#define ID_III(_01, _2, _3)             _01, _2, _3
```

These macros output the arguments they receive, so in particular, ID_I produces the argument. The nonempty branch ID_IIplus_DEFAULT (for two or more arguments to the **DEFAULT** macros) also dispatches to ID_I but with the second argument. In total, each branch of an invocation DEFAULT1 results in an invocation of ID_I:

```
DEFAULT1(a)                                  → ID_I(a)     → a
DEFAULT1(a,)                                 → ID_I(a)     → a
DEFAULT1(a, A)     → ID_IIplus_DEFAULT(A)    → ID_I(A)     → A
DEFAULT1(a, A, B) → ID_IIplus_DEFAULT(A, B) → ID_I(A, B) → error
```

In particular, we see that an invocation with more than two arguments results in an error and the valid invocations result in the expected expansions.

For argument lists with two defaults, we have DEFAULT2:

generic.h

DEFAULT2:

Output comma-separated arguments from the third position onward, fill with defaults from the first two.

Error if more than four arguments are given.

```
#define DEFAULT2(DEF0, DEF1, ...)                    \
  ID_II  ## __VA_OPT__(Iplus_DEFAULT)                \
  (DEF0, DEF1 __VA_OPT__(,) __VA_ARGS__)
#define ID_IIIplus_DEFAULT(DEF0, DEF1, _01, ...)     \
  DEFAULT1(DEF0, _01), DEFAULT1(DEF1, __VA_ARGS__)
```

Here, ID_IIIplus_DEFAULT dispatches to two invocations of DEFAULT1:

```
DEFAULT2(a, b)       → ID_II(a, b)                    → a, b
DEFAULT2(a, b,)      → ID_II(a, b)                    → a, b
DEFAULT2(a, b, A)    → ID_IIIplus_DEFAULT(a, b, A)
                     → DEFAULT1(a, A), DEFAULT1(b, )  → A, b
DEFAULT2(a, b, A,)   → ID_IIIplus_DEFAULT(a, b, A)
                     → DEFAULT1(a, A), DEFAULT1(b, )  → A, b
DEFAULT2(a, b, A, B) → ID_IIIplus_DEFAULT(a, b, A, B)
                     → DEFAULT1(a, A), DEFAULT1(b, B) → A, B
DEFAULT2(a, b,, B)   → ID_IIIplus_DEFAULT(a, b,  , B)
                     → DEFAULT1(a, ), DEFAULT1(b, B)  → a, B
DEFAULT2(a, b,,, C)  → ID_IIIplus_DEFAULT(a, b,,, C)
                     → DEFAULT1(a,), DEFAULT1(b,, C)  → a, error
```

The general pattern is better seen with the macro DEFAULT3 for three default arguments:

generic.h

DEFAULT3:

Output comma-separated arguments from the fourth position onward, fill with defaults from the first three.

Error if more than six arguments are given.

```
#define DEFAULT3(DEF0, DEF1, DEF2, ...)              \
  ID_III ## __VA_OPT__(Iplus_DEFAULT)                \
  (DEF0, DEF1, DEF2 __VA_OPT__(,) __VA_ARGS__)
#define ID_IIIIplus_DEFAULT(DEF0, DEF1, DEF2, _01, ...) \
  DEFAULT1(DEF0, _01), DEFAULT2(DEF1, DEF2, __VA_ARGS__)
```

ID_IIIIplus_DEFAULT splits off the first default value (DEF0) and first general argument (_01) to an invocation of DEFAULT1. It then makes a "recursive" invocation of DEFAULT2 with one default value (DEF0) and one argument (_01) less.[Exs 16]

To be useful more generally, we have to wrap invocations of the DEFAULT*N* macros in macros CALL*N* that perform a function call:

generic.h

```
#define CALL1(FUNC, DEF0, ...)                       \
```

[Exs 16] Just as we did for DEFAULT2, compute the results of all possible combinations of arguments for DEFAULT3.

```
161   FUNC(DEFAULT1(DEF0, __VA_ARGS__))
162 #define CALL2(FUNC, DEF0, DEF1, ...)            \
163   FUNC(DEFAULT2(DEF0, DEF1, __VA_ARGS__))
164 #define CALL3(FUNC, DEF0, DEF1, DEF2, ...)      \
165   FUNC(DEFAULT3(DEF0, DEF1, DEF2, __VA_ARGS__))
```

With these, our target macros that wrap calls to the `strto` functions look as follows:

. generic.h

```
167 #define strtoul(...)                            \
168   CALL3(strtoul,  "0", nullptr, 0, __VA_ARGS__)
169 #define strtoull(...)                           \
170   CALL3(strtoull, "0", nullptr, 0, __VA_ARGS__)
171 #define strtol(...)                             \
172   CALL3(strtol,   "0", nullptr, 0, __VA_ARGS__)
173 #define strtoll(...)                            \
174   CALL3(strtoll,  "0", nullptr, 0, __VA_ARGS__)
```

This adds default values `"0"`, **`nullptr`**, and `0` to calls to these functions whenever some argument is omitted. So, for an invocation to **`strtoul`** with just one argument, the series of replacements looks as follows:

```
strtoul(argv[1]) → CALL3(strtoul, "0", nullptr, 0, argv[1])
                 → strtoul(DEFAULT3("0", nullptr, 0, argv[1]))
                 → strtoul(argv[1], nullptr, 0)
```

If we invoke **`strtoul`** with three non-empty arguments instead,

```
strtoul(argv[1], ptr, 10)
    → CALL3(strtoul, "0", nullptr, 0, argv[1], ptr, 10)
    → strtoul(DEFAULT3("0", nullptr, 0, argv[1], ptr, 10))
    → strtoul(argv[1], ptr, 10)
```

the series of replacements effectively results in the same token sequence with which we started.

Summary

- Function-like macros are more flexible than inline functions.
- They allow us to implement type-safe features with variable argument lists.
- They can be used to complement function interfaces with compile-time argument checks.
- They can be used to provide information from the invoking environment.
- They can be used to provide default arguments.

18. Type-generic programming

This section covers

- Ubiquitous type genericity
- **_Generic** expressions
- Type inference with **auto** and **typedef**
- Anonymous functions

Type-generic features are so deeply integrated into C that most programmers are probably not even aware of their omnipresence. We first identify and provide a more systematic view of a number of different features that do indeed provide type-genericity and that we already have seen throughout the book (see section 18.1). This discussion covers operators that work for multiple types, type promotions and conversions, macros, variadic functions, function pointers, and **void** pointers. Then we introduce a more complicated programmable feature that came with C11: generic selection by means of the keyword **_Generic** (section 18.2). That feature allows us to write code that, at compile time, distinguishes actions according to a fixed set of types, such as choosing different variants of a function if presented with a **float**, **double**, or **long double** value.

One disadvantage of **_Generic** is that, in general, it only works for a predefined fixed set of types and, therefore, quickly leads to a combinatorial explosion of cases. C23 introduced a new feature called *type inference* (keywords **auto**, **typeof**, and **typeof_unqual**) that can be used to avoid writing such complicated code by consistently inferring type information from given expressions (section 18.3).

Unfortunately, C23 did not yet provide another feature much used for type-generic programming in C: functional abstractions. There are several extensions provided by mainstream compilers that provide such features. As they are important for code simplicity, genericity, safety, and performance, we briefly review some of these extensions in section 18.4.

18.1. Inherent type-generic features in C. The following discussion is not meant to cover all aspects of inherent type-generic features that we've already met but to raise awareness of their omnipresence, their relative complexity, and their possible defects.[17]

18.1.1. *Operators.* The first type-generic feature of C is operators. For example, the binary operators == and != are defined for all wide integer types (**signed**, **long**, **long long**, and their unsigned variants), for all floating types (**float**, **double**, **long double**, and their complex variants), and for pointer types, see table 18.1 for more details.

TABLE 18.1. Permitted types for binary operators that require equal types

		floating		pointer object		
operator	wide integer	real	complex	complete	**void**	function
==, !=	×	×	×	×	×	×
−	×	×	×	×		
+, *, /	×	×	×			
<, <=, >=, >	×	×		×		
%, ^, &, |	×					

Thus, expressions of the form `a*b+c` are by themselves already type-generic, and the programmer does not have to be aware of the particular type of any of the operands.

[17] This discussion has previously been published in Gustedt [2022].

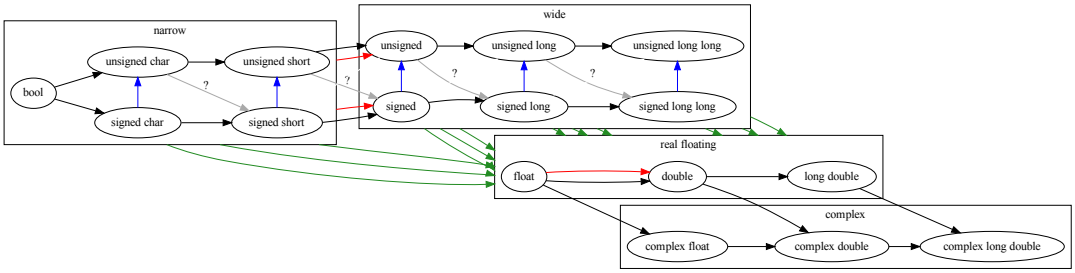


FIGURE 18.1. Upward conversion of some standard arithmetic types. Black arrows conserve values, red arrows (between narrow and wide types) may occur for integer promotion or default argument conversion, blue arrows (from signed to unsigned types) are reduction modulo 2^N, well-definition of gray arrows (with question marks) depends on the platform, green arrows (from integers to floating types) may lose precision.

In addition, if the types of the operands do not agree, a complicated set of conversions (see the following discussion) enforces equal types for all these operations. Other binary operators—namely, shift operators, object pointer addition, array subscripting—can even deal with different operand types, even without conversion.

18.1.2. *Default promotions and conversions.* If operands for the operators in table 18.1 don't agree or they are even types for which these operands are not supported (narrow integer types such as **bool**, **char**, or **short**), a complicated set of so-called promotion and conversion rules are set in motion. See figure 18.1 for an overview.

Conversions. Whenever an arithmetic argument to a function or the RHS of an assignment or initialization does not have the requested type of the corresponding parameter, there is a whole rule set that provides a conversion from the argument type to the parameter type:

```
printf("result␣is:␣%g\n", cosf(1));
```

Here, the **cosf** function has a **float** parameter, so the **int** argument `1` is first converted to `1.0f`.

Figure 18.1 shows the *upward* conversions that are put in place by C. These kinds of conversions help us avoid writing several versions of the same function and allow us to use such a function, to a certain extend, with several argument types.

Promotion and default argument conversion. In the previous example, the result of **cosf** is **float**, too, but **printf** as a variadic function cannot handle a **float**. So that value is converted to **double** before being printed.

Generally, there are certain types of numbers that are not used for arithmetic operators or certain types of function calls but are always replaced by a wider type. These mechanisms are called *promotion* (for integer types) or *default argument conversion* (for floating point).

Default arithmetic conversion. To determine the target type of an arithmetic operation, we must take these concepts to a second level. *Default arithmetic conversion* determines a common "super" type for binary arithmetic operators. For example, an operation `-1 + 1U` first performs the minus operation to provide a **signed int** of value −1. It then (for arithmetic conversion) converts that value to an **unsigned int**

(with value **UINT_MAX**) and performs the addition. The result is an **unsigned int** of value 0.

18.1.3. *Macros.* C's preprocessor is a powerful feature designed to replace identifiers (namely, object-like macros) and pseudo-function calls (function-like) by other token sequences. Together with default arithmetic promotions, it can be used to provide type-generic programming for several categories of tasks:

- Type-generic expressions
- Type-generic declarations and definitions
- Type-generic statements that are not expressions

Macros for type-generic expressions. A typically type-generic macro has an arithmetic expression that is evaluated and that uses default arithmetic conversion to determine a target type. For example, the following macro computes a gray value from three color channels:

```
#define GRAY(R, G, B) (((R) + (G) + (B))/3)
```

It can be used for any type that would represent colors. If used with **unsigned char**, the result would typically be **int**; for **float** values the result would also be **float**.

Naming conventions—here, for members `r`, `g`, and `b` of a structure `P` passed as an argument—can also help us write type- generic macros:

```
#define red(P)   ((P).r)
#define green(P) ((P).g)
#define blue(P)  ((P).b)
#define gray(P)  (GRAY(red(P), green(P), blue(P)))
```

Macros for declarations and definitions. Type declarations and definitions can also be provided by macros:

```
#define declareColor(N) typedef struct N N
#define defineColor(N, T) struct N { T r; T g; T b; }

declareColor(color8);
declareColor(color64);
declareColor(colorF);
declareColor(colorD);
```

Note that these macro definitions are such that the macro invocation needs a semicolon. Thereby, these invocations are syntactically similar to function calls:

```
typedef struct color8 color8;
typedef struct color64 color64;
typedef struct colorF colorF;
typedef struct colorF colorD;
```

The definitions of these structure types can then be generated by the following:

```
defineColor(color8,  uint8_t);
defineColor(color64, uint64_t);
defineColor(colorF,  float);
defineColor(colorD,  double);
```

These lines expands to

```
struct color8 { uint8_t r; uint8_t g; uint8_t b; };
struct color64 { uint64_t r; uint64_t g; uint64_t b; };
struct colorF { float r; float g; float b; };
struct colorG { double r; double g; double b; };
```

Macros placeable as statements. Macros can also be used to group several statements for which no value return is expected. Unfortunately, coding properly with this technique usually involves a tradeoff, with some ugliness and maintenance suffering. The following discussion presents the common practice for generic macro programming in C. The code can be used with any structure type `T` that has a **`mtx_t`** member named `mut` and a member named `data` that is assignment compatible with type `BASE`:[18]

```
#define dataCondStore(T, BASE, P, E, D)       \
  do {                                        \
    T* _pr_p = (P);                           \
    BASE _pr_expected = (E);                  \
    BASE _pr_desired = (D);                   \
    bool _pr_c;                               \
    do {                                      \
      mtx_lock(&_pr_p->mtx);                  \
      _pr_c = (_pr_p->data == _pr_expected);  \
      if (_pr_c) _pr_p->data = _pr_desired;   \
      mtx_unlock(&_pr_p->mtx);                \
    }  while(!_pr_c);                         \
  } while (false)
```

Coded in this manner, the macro has several advantages:

- It can syntactically be used in the same places as a **`void`** function. This is achieved by the crude outer **`do ... while(false)`** loop.
- Macro parameters are evaluated at most once. This is achieved by declaring auxiliary variables to evaluate and hold the values of the macro arguments. Note that the definition of these auxiliary variables needs knowledge about the types `T` and `BASE`.
- Some additional auxiliary variables (here, `_pr_c`) can be bound to the scope of the macro.

Additionally, a naming convention for local variables is used to minimize possible naming conflicts with identifiers that might already be defined in the context where the macro is used. Nevertheless, such a naming convention is not foolproof. In particular, if the use of several such macros is nested, surprising interactions between them may occur.

18.1.4. *Variadic functions.* We have seen another C standard tool for type-generic interfaces—variadic functions, such as **`printf`**:

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

Remember `...` denotes an arbitrary list of arguments that can be passed to the function, and how many and what type of arguments a call to the function may receive is mostly up to a convention between the implementor and the user. There are notable exceptions, though, because with the `...` notation, all arguments that are narrow integers or **`float`** are converted (see figure 18.1).

As we have seen, for such interfaces in the C standard library, modern compilers can usually check the arguments against the format string. In contrast, user-specified functions usually remain unchecked and can present serious safety problems.

18.1.5. *Function pointers.* Function pointers (section 11.4) allow us handle algorithms that can be made dependent on another function. For example, the `compar` parameter to the C library functions **`qsort`** or **`bsearch`** allows us to use these functions on any type for which we have a concept of comparison.

[18] We will see the use of the **`mtx_t`** type for programming with threads in section 20.

18.1.6. ***void** pointers.* The comparison functions used by **qsort** and **bsearch** have themselves interfaces that have type-generic elements—namely, parameters that are pointers to **void**:

```
typedef int compar_t(void const*, void const*);
```

Here, the understanding is that the pointer parameters of such a function represent pointers to the same object type BASE. Thus, data pointers can be converted forth and back to **void** pointers, as long as the target qualification is respected. The advantage is that such a comparison (and thus search or sorting) interface can then be written quickly. The disadvantage is that guaranteeing type safety is solely the job of the user.

18.1.7. *Type-generic C library functions.* C gained its first explicit type-generic library interface with the introduction of <tgmath.h> in C99. The idea is that a functionality, such as the cosine, should be presented to the user as a single interface, a type-generic macro **cos**, instead of the three functions **cos**, **cosf**, and **cosl** for **double**, **float**, and **long double** arguments, respectively. <tgmath.h>

At least for one-argument functions, the expectation seems to be clear: the functionality should return a value of the same type as the argument. In a sense, such type-generic macros are just the extension of C's operators (which are type-generic) to a set of well-specified and well-understood functions. An important property is that each of the type-generic macros in <tgmath.h> represents a finite set of functions in <math.h> or <complex.h>. Many of these macros are implemented by choosing a function pointer by inspecting the size of the argument, using the fact that their representations of the argument types all had different sizes. <tgmath.h> <math.h> <complex.h>

This idea was used recently in C23 to introduce new type-generic interfaces that preserve the **const**-contract of pointer arguments to certain C library functions. The functions are

- **memchr**, **strchr**, **strpbrk**, **strrchr**, and **strstr** from <string.h> <string.h>
- **wcschr**, **wcspbrk**, **wcsrchr**, **wcsstr**, and **wmemchr** from <wchar.h> <wchar.h>
- **bsearch** from <stdlib.h> <stdlib.h>

These functions now have type-generic macro interfaces that ensure that calls with pointer arguments with **const**-qualified target type will only return pointers with the same qualification:

```
#include "c23-fallback.h"

char const unmut_str[] = "haystack";
char       mut_str[] = "hoystack";
char const  needle[] = "stack";
char*       mut_pos0 = strstr(unmut_str, needle);   // error
char const* unmut_pos0 = strstr(unmut_str, needle);   // ok
char*       mut_pos1 = strstr(mut_str, needle);     // ok
char const* unmut_pos1 = strstr(mut_str, needle);     // ok
char*       mut_pos2 = (strstr)(unmut_str, needle); // possible
  invalid access
```

Here, for mut_pos0 type checks ensure that no unqualified pointer to a substring of the **const**-qualified array unmut_str will leak into the program. In contrast, the initializer for mut_pos2 uses the function interface and not the type-generic macro, so it returns an unqualified pointer. This may lead to program failure if mut_pos2 is used to change the target string.

Also, C11 gained a whole new set of type-generic functions in <stdatomic.h>, which we will see in more detail in section 20. The difficulty here is that there is a possibly unbounded number of atomic types, some of which have equal size but different semantics, so the type-generic interfaces cannot simply rely on the argument size to <stdatomic.h>

map to a finite set of functions. Implementations generally have to rely on language extensions to implement these interfaces.

18.2. Generic selection. One of the genuine additions of C11 to the C language was direct language support for explicit type-generic programming. It introduced a new feature, generic selection, which was primarily meant to implement type- generic macros similar to those in `<tgmath.h>`—that is, to perform a choice of a limited set of possibilities, guided by the type of an input expression.

<tgmath.h>

The specific add-on is the keyword **_Generic**, which introduces a primary expression of the following form:

```
_Generic(controlling expression,
  type1: expression1,
  ... ,
  typeN: expressionN)
```

This is very similar to a **switch** statement. But the *controlling expression* is only taken for its type (but see shortly), and the result is one of the expressions *expression1* ... *expressionN* chosen by the corresponding type-specific *type1* ... *typeN*, of which one may be simply the keyword **default**.

One of the simplest use cases, and primarily what the C committee had in mind, is to use **_Generic** for a type-generic macro interface by providing a choice between function pointers. A basic example is the the `<tgmath.h>` interfaces, such as **fabs**. **_Generic** is not a macro feature itself but can conveniently be used in a macro expansion. By ignoring complex floating-point types, such a macro for **fabs** could look like this:

<tgmath.h>

```
#define fabs(X)        \
_Generic((X),          \
  float: fabsf,        \
  long double: fabsl,  \
  default: fabs)(X)
```

This macro **fabs** distinguishes two specific types, **float** and **long double**, which choose the corresponding functions **fabsf** and **fabsl**, respectively. If the argument `X` is of any other type, it is mapped to the **default** case of **fabs**. That is, other arithmetic types, such as **double** and integer types, are mapped to the function **fabs**.[Exs 19][Exs 20]

Now, once the resulting function pointer is determined, it is applied to the argument list `(X)` that follows the **_Generic** primary expression.

The macro **min** is a more complete example. It implements a type-generic interface for a minimum of two real values. Three different **inline** functions for the three floating-point types are defined and then used in a similar way as for **fabs**. The difference is that these functions need two arguments, not one, so the **_Generic** expression must decide on a combination of the two types. It uses the sum of the two arguments as a *controlling expression*. As a consequence, argument promotions and conversion are affected by the arguments of that plus operation, so the **_Generic** expression chooses the function for the wider of the two types or **double** if both arguments are integers.

The difference between that and having just one function for **long double**, say, is that the information about the type of the concrete arguments is not lost.

Takeaway 18.2 #1 *The result type of a* ***_Generic*** *expression is the type of the chosen expression.*

[Exs 19] Find the two reasons why this occurrence of **fabs** in the macro expansion is not itself expanded.

[Exs 20] Extend the **fabs** macro to cover complex floating-point types.

generic.h

min:

Type-generic minimum for floating-point values.

```
#define min(A, B)                                       \
_Generic((A)+(B),                                       \
         float: minf,                                   \
         long double: minl,                             \
         default: mind)((A), (B))

static inline
double mind(double a, double b) [[__unsequenced__]] {
  return a < b ? a : b;
}

static inline
long double minl(long double a, long double b) [[
    __unsequenced__]] {
  return a < b ? a : b;
}

static inline
float minf(float a, float b) [[__unsequenced__]] {
  return a < b ? a : b;
}
```

This contrasts with what is happening, for example, for the ternary operator `a?b:c`. Here, the return type is computed by combining the two types, `b` and `c`. For the ternary operator, this must be done like that because `a` may be different from run to run, so either `b` or `c` may be selected. Since **_Generic** makes its choice based upon the type, this choice is fixed at compile time. So, the compiler can know the resulting type of the choice in advance.

In our example, we can be sure that all generated code that uses our interface will never use wider types than the programmer has foreseen. In particular, our **min** macro should always result in the compiler inlining the appropriate code for the types in question.[Exs 21][Exs 22]

Takeaway 18.2 #2 *Using* ***_Generic*** *with* ***inline*** *functions adds optimization opportunities.*

The following macro, which replaces the standard function **snprintf**, also follows this line of thought. It uses a macro **isice**, which detects whether its argument is an integer constant expression (ICE) (see 5.6.2) but is not shown:

[Exs 21] Extend the **min** macro to cover all wide integer types.

[Exs 22] Extend **min** to cover pointer types as well.

generic.h

snprintf:

A type-generic interface to **snprintf** that checks arguments.

We distinguish two cases. First if the first argument is **nullptr** the second argument is forced to 0 and the function **snprintf** is called. If it is not **nullptr** a function snprintf_swapped that just swaps the first two arguments is called. Because that function has the size argument first and then the buffer, the buffer can be specified as array parameter that depends on the that size.

```
#define snprintf(S, N, F, ...)                                  \
_Generic                                                        \
  ((S),                                                         \
   nullptr_t:                                                   \
   (snprintf)(nullptr, GENERIC_IF(isice(N), (N), 0),            \
              F __VA_OPT__(,) __VA_ARGS__),                     \
   default:                                                     \
   snprintf_swapped(_Generic((S),                               \
                             nullptr_t: 1,                      \
                             default: (N)),                     \
                    _Generic((S),                               \
                             nullptr_t:                         \
                             (char[1]){ 0 },                    \
                             default: (S)),                     \
                    (F) __VA_OPT__(,) __VA_ARGS__))
```

It looks a bit complicated because the second part, the call to snprintf_swapped, has to be a valid call so that it will not be called out by the compiler. The first case is still relatively easy. A call with **nullptr** as a first argument

```
snprintf(nullptr, n, format, a, b, c)
```

is replaced by a call that has a 0 in place of n and that emphasizes (by the pair of parenthesis) that this is not a recursive call of the macro but calls to the C library function directly:

```
(snprintf)(nullptr, 0, format, a, b, c)
```

If we now invoke the macro with a buffer (that does not have type **nullptr_t**), the effect is as if calling

```
snprintf_swapped(n, buffer, format, a, b, c)
```

Now, such a call, where the size comes before the buffer, allows compilers to check whether the buffer is known to be large enough.[23]

The whole mechanism is only valid because both branches of the generic selection expand to valid code, regardless. For the first case, **nullptr_t**, this is clear: the resulting expression is a valid call expression, even if the macro was called with a buffer. The second case, the **default** branch, is trickier. If we try to use

```
snprintf_swapped((N), (S), (F) __VA_OPT__(,) __VA_ARGS__)
```

for S being **nullptr**, this would resolve to an invalid call to snprintf_swapped because that function expects a non-null pointer as a second argument:

[23] As of this writing, some compilers (for example, gcc-12) detect such problems that are already visible at compile time; others (for example, clang-16) don't. But the general tendency is that compilers attempt to deduce excess bounds where possible. They will get better at that in the future.

```
snprintf_swapped(n, nullptr, format, a, b, c)
```

Although that version of the call is never evaluated, we have to be sure that, first, it is syntactically correct, and second, that the compiler doesn't diagnose erroneous combinations of arguments that effectively will never be selected. This is guaranteed by the two inner generic selections that provide dummy arguments for the case that `S` has type **`nullptr_t`**. Here, the fact that **`nullptr_t`** is a distinguished type, different from any pointer type, is important; before C23, without **`nullptr`** and **`nullptr_t`**, such a distinction would not have been easily possible.

Takeaway 18.2 #3 *All choices* expression1 ... expressionN *in a* **`_Generic`** *must be valid.*

The interpretation of what it means to talk about the *type of the controlling expression* of a generic selection is a bit ambiguous, so C17 clarified it in comparison to C11. In fact, as the previous examples imply, this type is the type of the expression *as if* it were passed to a function. This means, in particular,

- If there are any, type qualifiers are dropped from the type of the controlling expression.
- An array type is converted to a pointer type to the base type.
- A function type is converted to a pointer to a function.

Takeaway 18.2 #4 *The type expressions in a* **`_Generic`** *expression should only be unqualified types, not array types or function types.*

That doesn't mean the type expressions can't be pointers to one of those—a pointer to a qualified type, a pointer to an array, or a pointer to a function. But generally, this rule makes the task of writing a type-generic macro easier since we do not have to take all combinations of qualifiers into account. There are three qualifiers (four for pointer types), so, otherwise, all different combinations would lead to 8 (or even 16) different type expressions per base type. The following example, `MAXVAL`, is already relatively long. It has a special case for all 15 orderable types; if we also had to track qualifications, we would have to specialize 120 cases!

generic.h

`MAXVAL`:

The maximum value for the type of *X*.

```
#define MAXVAL(X)                                                   \
_Generic((X),                                                       \
         bool: (bool)+1,                                            \
         char: (char)+CHAR_MAX,                                     \
         signed char: (signed char)+SCHAR_MAX,                      \
         unsigned char: (unsigned char)+UCHAR_MAX,                  \
         signed short: (signed short)+SHRT_MAX,                     \
         unsigned short: (unsigned short)+USHRT_MAX,                \
         signed: INT_MAX,                                           \
         unsigned: UINT_MAX,                                        \
         signed long: LONG_MAX,                                     \
         unsigned long: ULONG_MAX,                                  \
         signed long long: LLONG_MAX,                               \
         unsigned long long: ULLONG_MAX,                            \
         float: FLT_MAX,                                            \
         double: DBL_MAX,                                           \
         long double: LDBL_MAX)
```

This is an example where a **_Generic** expression is used differently than earlier, where we chose a function pointer and then called the function. Here, the resulting value is an integer-constant expression. This never could be realized by function calls, and it would be very tedious to implement with just macros.[Exs 24]

With yet another conversion trick, in `maxof`, we can get rid of some cases we might not be interested in. Here, the special form of the controlling expression adds an additional feature. The expression `0+(identifier)+0` is valid if `identifier` is a variable or if it is a type. If it is a variable, the type of the variable is used, and it is interpreted just like any other expression. Then integer promotion is applied to it, and the resulting type is deduced.

If it is a type, `(identifier)+0` is read as a cast of `+0` to type `identifier`. Adding `0+` from the left then still ensures that integer promotion is performed if necessary. So, the result is the same whether `XT` is a type `T` or an expression `X` of type `T`.[Exs 25][Exs 26][Exs 27]

generic.h

`maxof`:

The maximum promoted value for *XT*, where *XT* can be an expression or a type name.

So this is the maximum value when fed to an arithmetic operation such as +.

Remarks: Narrow types are promoted, usually to **signed**, or maybe to **unsigned** on rare architectures.

```
#define maxof(XT)                                   \
_Generic(0+(XT)+0,                                  \
         signed: INT_MAX,                           \
         unsigned: UINT_MAX,                        \
         signed long: LONG_MAX,                     \
         unsigned long: ULONG_MAX,                  \
         signed long long: LLONG_MAX,               \
         unsigned long long: ULLONG_MAX,            \
         float: FLT_MAX,                            \
         double: DBL_MAX,                           \
         long double: LDBL_MAX)
```

Another requirement for the type expressions in a **_Generic** expression is that the choice must be unambiguous at compile time.

Takeaway 18.2 #5 *The type expressions in a **_Generic** expression must refer to mutually incompatible types.*

Takeaway 18.2 #6 *The type expressions in a **_Generic** expression cannot be a pointer to a VLA.*

[Exs 24] Write an analogous macro for the minimum value.

[Exs 25] By using **_Generic**, write a macro `PROMOTE(XT, A)` that returns the value of `A` as type `XT`, where both have a wide standard integer type. For example, `PROMOTE(1u, 3)` would be `3u`.

[Exs 26] By using **_Generic**, write a macro `SIGNEDNESS(XT)` that returns **false** or **true** according to the signedness of the type of `XT`, where both have a wide standard integer type. For example, `SIGNEDNESS(1l)` would be **true**.

[Exs 27] By using **_Generic**, write a macro `mix(A, B)` that computes the maximum value of `A` and `B`, where both have a wide standard integer type. If both have the same signedness, the result type should be the wider type of the two. If both have different signedness, the return type should be an unsigned type that fits all positive values of both types.

A different model than the *function-pointer-call* variant can be convenient, but it also has some pitfalls. Let us try to use **_Generic** to implement the two macros TRACE_FORMAT and TRACE_CONVERT, used in TRACE_VALUE1, see below. TRACE_FORMAT is straightforward. We distinguish six different cases. The **default** case, when no arithmetic type is matched, supposes that the argument has a pointer type. In that case, to be a correct parameter for **fprintf**, the pointer must be converted to **void***. Our goal is to implement such a conversion through TRACE_CONVERT.

macro_trace.h

TRACE_VALUE1:

Traces a value without having to specify a format.

This variant works correctly with non-void pointers.

The formats are tunable by changing the specifiers in TRACE_FORMAT.

```
#define TRACE_VALUE1(F, X)                                          \
do {                                                                \
  if (TRACE_ON)                                                     \
    fprintf(stderr,                                                 \
            TRACE_FORMAT("%s:" STRGY(__LINE__) ":␣" F, X), \
            __func__, TRACE_CONVERT(X));                            \
 } while (false)
```

macro_trace.h

TRACE_FORMAT:

Returns a format that is suitable for **fprintf**.

Returns: The argument *F* must be a string literal, so the return value will also be one.

```
#define TRACE_FORMAT(F, X)                          \
_Generic((X)+0LL,                                   \
         unsigned long long: "" F "␣%llu\n",        \
         long long: "" F "␣%lld\n",                 \
         float: "" F "␣%.8f\n",                     \
         double: "" F "␣%.12f\n",                   \
         long double: "" F "␣%.20Lf\n",             \
         default: "" F "␣%p\n")
```

A first try could look like the following:

```
#define TRACE_CONVERT_WRONG(X)          \
_Generic((X)+0LL,                       \
         unsigned long long: (X)+0LL,   \
         ...                            \
         default: ((void*){ } = (X)))
```

This uses the same trick as using TRACE_PTR1 to convert the pointer to **void***. Unfortunately, this implementation is wrong (see takeaway 18.2 #3). If, for example, X is an **unsigned long long** (say, 1LL), the **default** case would read

```
((void*){ } = (1LL))
```

which would be assigning a nonzero integer to a pointer, which is erroneous.[28]

[28] Remember that conversion from nonzero integers to pointers must be made explicit through a cast.

We tackle this problem with a macro that returns either its argument or **nullptr**:

macro_trace.h

TRACE_POINTER:

Returns a value that forcibly can be interpreted as pointer value.

That is, any pointer will be returned as such, but other arithmetic values will result in a **nullptr**.

```
#define TRACE_POINTER(X)                                \
_Generic((X)+0LL,                                       \
         unsigned long long: nullptr,                   \
         long long: nullptr,                            \
         float: nullptr,                                \
         double: nullptr,                               \
         long double: nullptr,                          \
         default: (X))
```

This code has the advantage that a call to TRACE_POINTER(X) can always be assigned to a **void***. Either X itself is a pointer and thus is assignable to **void***, or it is of another arithmetic type, and the result of the macro invocation is **nullptr**. Put all together, TRACE_CONVERT looks as follows:

macro_trace.h

TRACE_CONVERT:

Returns a value that is promoted either to a wide integer, to a floating point, or to a **void*** if *X* is a pointer.

```
#define TRACE_CONVERT(X)                                        \
_Generic((X)+0LL,                                               \
         unsigned long long: (X)+0LL,                           \
         long long: (X)+0LL,                                    \
         float: (X)+0LL,                                        \
         double: (X)+0LL,                                       \
         long double: (X)+0LL,                                  \
         default: ((void*){ nullptr } = TRACE_POINTER(X)))
```

CHALLENGE 18 (Integer constant expressions). *Macros that test an expression for a certain feature without evaluating it can be helpful to distinguish taken branches of code at compilation time. Therefore, in following, the macros you write should never evaluate their argument and, in general, by themselves, result in an integer constant expression of type* **bool**:

- *Write* **is_null_pointer_constant** *to test an expression* X *(that is either a zero integer constant or a* **void***⋆ pointer) if it is a null pointer constant. Use the fact that an expression of the form*

```
true ? (struct toto*)0 : (X)
```

  has type **struct** toto*⋆ if* X *is a null pointer constant but type* **void***⋆ if* X *is a pointer of type* **void***⋆ that is not a null pointer constant.*
- *Write* **is_zero_ice** *that detects whether the argument is an integer constant expression of value zero (ICEV0). Use the fact that such an ICEV0 is a null pointer constant if it is cast to* **void***⋆ and that any other integer expression, even if cast to* **void***⋆, is never a null pointer constant.*
- *Write* **isice** *that detects whether the argument is an integer constant expression.*

18.3. Type inferrence. In addition to generic selection, C23 has added new features that allow us to infer a type from an expression (or a type name) such that we may avoid combinatorial explosion of the different cases.

18.3.1. *The* **auto** *feature.* The first such feature uses the **auto** keyword. Before C23, that keyword was a mostly redundant storage class specifier for automatic variables. Now, it also serves to indicate that the type of a variable can be inferred from its initializer. In the following example of a generic SWAP macro, we have three declarations of variables with an inferred type:

swap.c

```
#define SWAP(X, Y)                                                      \
  do {                                                                  \
    /* These two variables have the role of function                    \
       parameters. They ensure to evaluate the                          \
       expression X and Y once, since these could be                    \
       complicated lvalue expressions with evaluated                    \
       subexpressions that have side effects. */                        \
    auto const swap_p1 = &(X);                                          \
    auto const swap_p2 = &(Y);                                          \
    static_assert_compatible(*swap_p1, *swap_p2,                        \
                             "to␣exchange␣values,␣'"                    \
                             #X "'␣and␣'" #Y                            \
                             "'␣must␣have␣compatible␣types");           \
    auto swap_tmp = *swap_p1;                                           \
    *swap_p1 = *swap_p2;                                                \
    *swap_p2 = swap_tmp;                                                \
  } while (false)
```

This macro is, in fact, a generalization of the function swap_double (sec. 11) such that it is applicable to any object type that is not an array. We distinguish four different components in this macro:

- An outer hull consisting of an artificial **do ... while**(**false**) loop that ensures that invocations of the macro behave similarly to calls to **void** functions (recall takeaway 10.2.1 #2).
- A component that captures the macro parameters, such that they are only evaluated once. This component plays the role that would be ensured by a parameter list if we specify a function.
- A component that checks some property of the types. We will see later how that works.
- A component that contains the real utility code, which, in this case, is very similar to the function `swap_double`.

To be generic, it is important that we don't make assumptions about the possible type used. Using **auto** for the declarations instead of a type ensures that the declarations have a type that is consistent with the specific arguments of any particular call. `swap_p1` and `swap_p2` are initialized from expressions with a unary `&` operator, so they have a type of a pointer to the type of the expressions `X` and `Y`, respectively. `swap_tmp` has the type to which `swap_p1` is pointing, thus the same type as expression `X`. In addition to **auto**, the two pointers are also **const**-qualified, such that these pointers cannot be changed withing the snippet.

Also, observe that we used `swap`-prefixed names for the variables such that it is less likely that the names clash with other names from the context surrounding a call. In our case, there is still that possibility— namely, if `Y` refers to an outer name of `swap_p1`. When executing the macro, the newly defined local variable of the same name would be used instead. This is a principal flaw of macro techniques that use unprotected compounds internally. We can never be sure that naming choices do not clash with names from user code that are unknown to us.

Takeaway 18.3.1 #1 *Protect local variables inside macros by a documented naming convention.*

Outside of type-generic macros for which we do not know much about the base type, inside functions **auto** declarations can also be interesting for auxiliary variables. They ensure that the declaration remains consistent in type with some other declarations in our code. Lines such as

```
auto y = 3*x;
```

can be preferable if `x` is defined relatively far away from `y` and its type could be subject to change. If we instead write

```
float y = 3*x;
```

because we are convinced that `x` will always be have type **float**, the day we change `x` to **double**, the loss of precision in the initialization may go unnoticed.

Takeaway 18.3.1 #2 *Use **auto** definitions where you must ensure type consistency.*

18.3.2. *The **typeof** feature.* As we have seen, **auto** provides a way to infer a type for an object, and in most of the cases where we need such a feature, it will be sufficient. For more sophisticated cases, C23 introduced two other features: **typeof** and **typeof_unqual**. In principle, they could even replace the **auto** feature. The previous **auto** definition of `y` could, in fact, be rewritten as

```
typeof(3*x) y = 3*x;
```

However, in some rare cases, `x` would perhaps be evaluated twice, and we might have to watch out for qualifiers (see the following discussion). In fact, **typeof** operators can be used in places where we could otherwise have a **typedef** name. In the example, we could have had a definition

```
typedef float type_of_x_y;
```

and then see that we consistently use the name `type_of_x_y` for the definitions of `x` and `y`. All that shows it might be difficult to ensure type consistency when using **typedef** or **typeof**.

Takeaway 18.3.2 #1 *Prefer* ***auto*** *over* ***typeof*** *for variable declarations.*

The following definition of a macro to test for type compatibility (as we need it for `SWAP`) shows a usage of **typeof** that cannot easily be replaced by an **auto** declaration:

swap.c

```
4  #define static_assert_compatible(A, B, REASON)  \
5    static_assert(_Generic((typeof(A)*)nullptr,   \
6                           typeof(B)*: true,      \
7                           default: false),       \
8                  "expected␣compatible␣types:␣" REASON ",␣have␣" #A
       "␣and␣" #B "")
```

This code uses **typeof** for `A` and `B` by using a cast to **typeof**`(A)*` and placing **typeof**`(B)*` as the type name in the choice of the selection. Thereby, we are able to use `A` and `B` in a context where a type is required. The generic selection feature, together with the properties of **typeof**, then guarantees that neither `A` nor `B` are evaluated. Thus, the macro can be used safely inside other macros wherever we have to ensure that two non-array object types are compatible.[Exs 29][Exs 30]

With **typeof** even multiparameter type matching becomes possible:

generic.h

```
340  #define pow(X, Y)                                                \
341    _Generic                                                       \
342    (                                                              \
343     (void (*)(typeof((X)+(Y)+0ULL), typeof((Y)+0ULL)))nullptr,    \
344     /* Second argument is an integer. */                          \
345     void (*)(float,              unsigned long long): pownf,      \
346     void (*)(double,             unsigned long long): pown,       \
347     void (*)(unsigned long long, unsigned long long): pown,       \
348     void (*)(long double,        unsigned long long): pownl,      \
349     /* Second argument is a floating point. */                    \
350     void (*)(float,                           float): powf,       \
351     void (*)(long double,                     float): powl,       \
352     void (*)(long double,                    double): powl,       \
353     void (*)(long double,               long double): powl,       \
354     /* Second argument is a floating-point, and first */          \
355     /* is double or integer.                          */          \
356     default:                                          pow)        \
357    ((X), (Y))
```

[Exs 29] The previously presented version of `static_assert_compatible` also detects a possible mismatch when presented with an array and a pointer. Modify the macro so that it accepts an array or pointer `A` with base `b` and tests whether `B` is compatible to `b*`.

[Exs 30] By using your findings, improve the definition of `static_assert_compatible` so that it rejects arguments that are arrays or function specifiers. Pointers to arrays or pointers to functions should still work.

Here, the controlling expression generates a pointer to a function with two argument types, `T` and `S`,

```
  (void (*)(T, S))nullptr
```

that depends on the macro arguments `X` and `Y`. `T` is the type **`typeof`**`((X)+(Y)+0ULL)` and `S` is **`typeof`**`((Y)+0ULL)`. Both types `T` and `S` may result in one of the standard floating point types or **`unsigned long long`**. For `S`, only argument `X` is taken into account; for `T`, both `X` and `Y` are used. By that, we can detect whether `Y` has an integer type. In that case, the C library function **`pown`** (new in C23) is used; otherwise, if `Y` has floating point type, **`pow`** is used. Thus, with this macro, we may use the optimized function **`pown`** for integer powers if our second argument is an integer.

This example shows that such interfaces are doable with C23 by using **`_Generic`** and **`typeof`**, but it also shows that the combinatorics of the type arguments may explode rapidly. To avoid part of that complexity, it may be easier to use the **`typeof`** and **`typeof_unqual`** operators just for the return types of generic functions:

```
static inline long double absolute(long double absolute_x) {
  return absolute_x < 0.0L ? -absolute_x : absolute_x;
}
#define absolute(X) ((typeof_unqual(X))absolute(X))
```

Here, a simple **`static inline`** function for the widest type in the class implements the feature, a macro that casts the type down to the original precision provides type-genericity. In many cases, such code should be well integrated and optimized at the calling side; no complicated **`_Generic`** case analysis is necessary. Here, the use of **`typeof_unqual`** instead of **`typeof`** emphasizes that function calls have unqualified return types.

That technique can even be combined with **`_Generic`** to reduce the complexity of a **`_Generic`** macro:

```
#define pow(X, Y)                          \
((typeof_unqual((X)+(Y)+0.0F))             \
     _Generic((Y)+0ULL,                    \
              unsigned long long: pownl,   \
              default:            powl)    \
((X), (Y)))
```

If `Y` is an integer, `(Y)+0ULL` has type **`unsigned long long`**, and the choice is **`pownl`**; otherwise, the choice is **`powl`**. The return value of the function call is then cast down to a floating point type—namely, the narrowest unqualified floating point type that fits `X`, `Y` and `0.0F`.

This version is perhaps not as efficient as the previously discussed version because it always uses **`long double`** versions of the C library functions. At least it guarantees that the result is computed with the best possible accuracy.

CHALLENGE 19 (Type traits). *The* **`typeof`** *operator and* **`_Generic`** *in combination are quite powerful and can be used to provide* type traits*—that is, operations that can be used at compile time to query certain properties of types. This then allows us to implement type-generic features (such as the maximum or minimum of two values) more efficiently if the types of argument values fulfill certain conditions. In the following, the type-trait macros that you write should be able to accept as the argument* `XT` *either an expression (without exposed comma) or a type name and result in values of type* **`bool`** *that are themselves integer constant expressions:*

- *As a warmup, write a macro* **`tozero`** *that uses* **`typeof`** *to provide a zero of the type of a given arithmetic expression or type name and a similar one* **`tonull`** *that provides a null pointer for pointer expressions or types. The result should never evaluate the argument* `XT` *and be an integer, arithmetic, or address constant expression, respectively, regardless of the circumstances.*
- *C23 has many integer types (at least 139), so a complete listing of these types in generic selections is not feasible. Use* **`tozero`** *and* **`is_zero_ice`** *from challenge 18 to test whether* `XT` *has an integer type* **`isinteger`**.
- *Use* **`isice`** *from challenge 18 to detect whether* `XT` *is a VLA,* **`isvla`**. *Use the fact that* **`sizeof`** *for a VLA is never an integer constant expression.*
- *For pointer types, write macros that test whether the target type is* **`const`***- or* **`volatile`***-qualified (*`is_const_target` *and* `is_volatile_target`*, respectively). Use these macros to test whether the argument itself is qualified,* `is_const` *and* `is_volatile`.
- *Write a macro* `is_potentially_negative` *that checks whether the type may admit negative values. Test your macro on signed and unsigned integer types,* **`char`***, and real floating point types.*
- *Write macros* `issigned` *and* `isunsigned` *that check whether the type is signed or unsigned, respectively. Floating point types, pointers, or* **`char`** *should never be identified as signed or unsigned.*
- *Write test macros* `is_void_target` *for* **`void`** *pointers and* `is_void` *for void expressions.*

18.4. Anonymous functions. The previously discussed method—enclosing the contents of `SWAP` into `do { ... } while(false)`—is not completely satisfying. In fact, it is not possible to use that macro in all contexts where a call to a `void` function could be placed. More generally, macros implemented with this technique are not expressions. They cannot be used where macros would be possible (for example, as *expression3* in a `for` loop), and more importantly, they cannot return a value.

Unfortunately, C23 does not yet offer a general tool to do so, but there are several quite similar extensions available in many compilers that provide such a feature. The first that we want to mention here is the construct of a *compound expression* offered by `gcc` and some related compilers. Implementing our `SWAP` macro with that technique looks as follows:

swap.c

```
29 #define SWAP(X, Y)                                                    \
30   /* This starts the compound expression construct. */               \
31   ({                                                                  \
32     /* These two variables have the role of function                 \
33        parameters. They ensure to evaluate the                        \
34        expression X and Y once, since these could be                  \
35        complicated lvalue expressions with evaluated                  \
36        subexpressions that have side effects. */                      \
37     auto const swap_p1 = &(X);                                        \
38     auto const swap_p2 = &(Y);                                        \
39     static_assert_compatible(*swap_p1, *swap_p2,                      \
40                              "to␣exchange␣values,␣'"                  \
41                              #X "'␣and␣'" #Y                          \
42                              "'␣must␣have␣compatible␣types");         \
43     auto swap_tmp = *swap_p1;                                         \
44     *swap_p1 = *swap_p2;                                              \
45     *swap_p2 = swap_tmp;                                              \
46     /* ensure that the type of the expression is void */              \
47     (void)0;                                                          \
48   })
```

Here the syntactic sugar around the code reads `({ ... (void)0; })`. This construct allows this to be an expression and to have a value— namely, the `(void)0` at the end. Otherwise, it works pretty much like the `do { ... } while(false)` construct: it leaves identifiers from surrounding scopes unprotected and even allows us to jump into and out of the code by means of `goto` statements and labels.

Objective C and codeclang, with a specific extension for C, provide another tool called *block closure* that avoids some of these shortcomings:

swap.m

```
4 #define SWAP(X, Y)                                                    \
5   (                                                                   \
6    /* This starts the block construct. */                           \
7    void ^                                                           \
8    /* We have no other parameters, so a void list */                \
9    (void) { /* now the body starts */                               \
```

```
                        /* same as the other SWAP examples */
```

swap.m

```
24     }                                                                \
25   /* Now this block can be called immediately with */               \
26   /* no arguments */                                                \
```

```
27    ()) /* end of the expression */
```

Here, the syntactic sugar is **void**^(**void**){ ... }(). That is, we have something similar to a declaration of a **void** function that has no name (replaced by a ^ token) and an empty parameter list. This anonymous function is then directly called (with no arguments) by a following pair of () parenthesis. Compared to what we have seen previously, such block closures have access to variables of the context surrounding a call, but that access is limited to reading by default; only special variables that would be marked as such can also be modified within such a block closure. As for functions, the call to a block closure terminates when the execution reaches the end of the body or when a **return** statement is met.

Last but not least, C++ has introduced lambdas, and some C compilers might already have them. Lambdas separate more clearly what and when variables of the surrounding scope can be read. The syntactic sugar in our macro is [...](**void**){ ... }(), which again is very similar to a function definition followed by an immediate call without arguments:

swap.cpp

```
4   #define SWAP(X, Y)                                                  \
5     (                                                                 \
6     /* This starts the lambda construct. */                           \
7     [                                                                 \
8      /* These two captures play the role of function                 \
9         parameters and read X and Y ... */                           \
10     swap_p1 = &(X),                                                 \
11     swap_p2 = &(Y)]                                                 \
12    /* Then effectively the parameter list is empty */                \
13    (void) { /* now the body starts */                              \
```

```
                        /* same body as above */
```

swap.cpp

```
21    }                                                                 \
22    /* Now this lambda value can be called immediately with */        \
23    /* no arguments */                                                \
24    ()) /* end of the expression */
```

The first part, within the [...] brackets, groups the so-called captures of the lambda. This is the only part in which the surrounding context can be read. The syntax for the type of captures that we are interested in is quite simple: an identifier followed by an = sign, followed by an initializer. The semantics are similar to what we have previously used—namely, type inference as if by **auto** and a **const**-qualification of the object. The expansion of a macro invocation SWAP(a, b) (with surrounding parenthesis removed) then resembles the following:

```
[swap_p1 = &(a), swap_p1 = &(b)](void){
  /* expansion of static_assert_compatible */
  auto swap_tmp = *swap_p1;
  *swap_p1 = *swap_p2;
  *swap_p2 = swap_tmp;
}()
```

Again, there would also be syntax to access variables for writing, but this is beyond the scope that we want to cover here.

The result type of a lambda call expression is inferred from the first possible **return** statement. If there is none, as in our case, the expression as a whole is a **void** expression; that is, it behaves as if a function with **void** return type is called.

Obviously, these extensions for anonymous functions are not very interesting for this case of **void** expressions; the **do** { ... } **while**(**false**) idiom matches most usages. The extensions become more interesting when the result is a type inferred from the macro arguments. Computing the maximum of two values, X and Y, seems to be an easy task. At a first glance, we should be able to plug an expression into a macro, as follows:

```
(X < Y) ? Y : X
```

Unfortunately, this has several shortcomings. First, depending on the outcome, either X or Y is evaluated twice: once in the comparison and a second time in the chosen branch. To avoid this difficulty, we can use an anonymous function. Here, we use gcc's compound expression construct, for which the result is the value of the last expression in the block:

swap.c

```
54 #define MAX_EQSIGN(X, Y)                                    \
55   /* This starts the compound expression construct. */      \
56   ({                                                        \
57     /* These two captures play the role of function        \
58        parameters and read X and Y ... */                   \
59     auto const max_x = (X);                                 \
60     auto const max_y = (Y);                                 \
61     /* now the body starts */                               \
62     /* types need to have the same signedness */            \
63     (max_x < max_y) ? max_y : max_x;                        \
64   })
```

In essence, this is the expression from before. If both types have the same signedness, the magic of implicit integer conversions ensures that the result has a type in which the maximum value fits.

The second shortcoming of this approach only manifests if one of the expressions has a signed type and the other, an unsigned one. Then, the outcome of the comparison and the expression as a whole is not what we would expect. If, for example, X is -1 and Y is 1u (so **unsigned**), in the comparison, -1 is converted to **unsigned** and results in a very large number, **UINT_MAX**. The result of the expression has type **unsigned** and a value of **UINT_MAX**.

The following, more generic macro addresses this problem. Here, the direct comparison between the two argument values is only performed when both values have the same sign. Otherwise, each value is compared to 0 individually, and the one with the positive outcome is chosen as the result:[Exs 31]

swap.c

```
66 #define MAX(X, Y)                                           \
67   /* This starts the compound expression construct. */      \
68   ({                                                        \
69     auto const max_x = (X);                                 \
70     auto const max_y = (Y);                                 \
71     /* now the body starts */                               \
72     ((isnegative(max_x) && !isnegative(max_y))              \
73      ? max_y                                                \
74      : ((isnegative(max_y) && !isnegative(max_x))           \
```

[Exs 31] Write a small macro isnegative as needed for **MAX**.

```
75          ? max_x                                                 \
76          : /* both have the same sign  */                        \
77          ((max_x < max_y) ? max_y : max_x)));                    \
78    })
```

Here again, the implicit conversions, as defined by the C standard, do the right thing and provide a type that can hold the result.[Exs 32]

CHALLENGE 20 (Maximum and minimum type-generic macros). *Our* `MAX` *macro might leave room for optimization in the case where the types of* `X` *and* `Y` *have different signedness. Implement an improved version that uses your implemented traits to make an appropriate case distinction that has a minimal number of comparisons for the respective cases. Here, the property that your traits are ICE is essential for optimization because then compilers will be able to eliminate branches that are unreachable for the given arguments.*

Providing the same functionality for the minimum of two integer values `X` *and* `Y` *is even more delicate because the mathematical result of such an operation does not always fit into the same type as the maximum operation. First, convince yourself that even if the types of* `X` *and* `Y` *are distinct, one of the two types is able to hold the result of the minimum operation. In fact, even if both types are unsigned, it is sufficient to use the narrower type of them as the result type of the operation. Write a macro* `minunsigned` *that returns an ICE of value zero of the narrower unsigned type if presented with two unsigned expressions or type names. Then use this to implement a macro* `minreturn` *that returns an ICE of value zero that has a type that is able to hold the minimum value of its two arguments. Finally, use that for the implementation of a minimum operation* `MIN` *that returns the mathematically correct value for all pairwise combinations of integer types.*

[Exs 32] Prove that the result type of `MAX` is always able to hold the mathematical result of the maximum operation.

Summary

- Type generic features are omnipresent in C.
- Generic selection with **`_Generic`** can be used to implement type-generic interfaces that depend on a specific set of types.
- Type inference with **`auto`** helps maintain type consistency for dependent declarations.
- Type inference with **`typeof`** can be used to test for types and their properties or to convert a result expression back to a type depending on macro arguments.
- Anonymous functions are an extension to the C standard that allows us to implement convenient generic operations as macros.

19. Variations in control flow

This section covers

- Understanding normal sequencing of statements in C
- Making short and long jumps through code
- Function control flow
- Handling signals

The *control flow* (see figure 2.1) of program execution describes how the individual statements of the program code are *sequenced*—that is, which statement is executed after another. Up to now, we have mostly looked at code that lets us deduce this control flow from syntax and a controlling expression. That way, each function can be described using a hierarchical composition of *basic blocks*. A basic block is a maximum sequence of statements such that once execution starts at the first of these statements, it continues unconditionally until the last, such that all execution of any statement in the sequence starts with the first.

If we suppose that all conditionals and loop statements use `{}` compound statements, in a simplified view, such a basic block

- Starts either at the beginning `{` of a compound statement or a **`case`** or jump label
- Ends either at the `}` of the corresponding compound statement or at the next
  - Statement that is the target of a **`case`** or jump label
  - Body of a conditional or loop statement
  - **`return`** statement
  - **`goto`** statement
  - Call to some function in a specific set

In this definition, general function calls make no exception: they are used to suspend execution of a basic block temporarily but not to end it. Among the specific functions that end a basic block are some that we already know, including those marked with the attribute `[[noreturn]]`, such as **`exit`** and **`abort`**. Another such function is **`setjmp`**, which may return more than once, as discussed later.

Code composed of just basic blocks stitched together by **`if`**/**`else`**[33] or loop statements has the double advantage of being easily readable for us humans and leads to optimization opportunities for the compiler. Both can directly deduce the lifetime and access pattern of variables and compound literals in basic blocks and then capture how these are melded by the hierarchical composition of the basic blocks into their function.

A theoretical foundation of this structured approach was given quite early for Pascal programs by Nishizeki et al. [1977] and extended to C and other imperative languages by Thorup [1995]. They prove that structured programs—that is, programs without **`goto`** or other arbitrary jump constructs—have a control flow that matches nicely into a tree-like structure that can be deduced from the syntactical nesting of the program. Unless you have to do otherwise, you should stick to that programming model.

Nevertheless, some exceptional situations require exceptional measures. Generally, changes to the control flow of a program can originate from

Conditional statements: **`if`**/**`else`** and **`switch`**/**`case`**
Loop statements: **`do`**`{}`**`while`**`()`, **`while`**`()`, and **`for`**`()`
Function calls: **`return`** statements or `[[`**`noreturn`**`]]` attributes
Short jumps: **`goto`** and labels
Long jumps: **`setjmp`**/**`longjmp`** and `getcontext`/`setcontext`[34]

[33] **`switch`**/**`case`** statements complicate the view a bit.
[34] Defined in POSIX systems.

Interrupts: signals and **signal** handlers
Threads: **thrd_create** and **thrd_exit**

These changes in control flow can cloud the view of the abstract state of the execution. Roughly, the complexity of the knowledge a human or mechanical reader must track increases in that list from top to bottom. Up to now, we have only seen the first four constructs. These correspond to *language* features, which are determined by syntax (such as keywords) or operators (such as the `()` of a function call). The latter three are introduced by *C library* interfaces. They provide changes in the control flow of a program that can jump across function boundaries (**longjmp**), be triggered by events that are external to the program (interrupts), or even establish a concurrent control flow, another *thread of execution*.

Various difficulties may arise when objects are under the effect of unexpected control flow:

- Objects could be used outside their lifetime.
- Objects could be used uninitialized.
- Values of objects could be misinterpreted by optimizing (**volatile**).
- Objects could be partially modified (**sig_atomic_t**, **atomic_flag**, or **_Atomic** with the lock-free property and `relaxed` consistency).
- Updates to objects could be sequenced unexpectedly (all **_Atomic**).
- Execution must be guaranteed to be exclusive inside a *critical section* (**mtx_t**).

Because access to the objects that constitute the state of a program becomes complicated, C provides features that help to cope with the difficulties. In the previous list, they are noted in parentheses, and we will discuss them in detail in the following subsections.

19.1. A detailed example. To illustrate most of these concepts, we will discuss some central example code: a *recursive descent parser* called `basic_blocks`. The central function `descend` is presented in the following listing.

This code serves several purposes. First, it obviously presents several features that we will discuss later:

- Short jumps (**goto**; see section 19.3)
- Recursion (see section 19.4)
- Long jumps (**longjmp**; see section 19.5)
- Interrupt handling (see section 19.6).

But at least as important, it is probably the most difficult code we have handled so far in this book, and for some of you, it might even be the most complicated code you have ever seen. Yet, with its 36 lines, it still fits on one screen, and it is by itself an affirmation that C code can be very compact and efficient. It might take you hours to understand, but please do not despair; you might not know it yet, but if you have worked thoroughly through this book, you are ready for this.

The function implements a *recursive descent parser* that recognizes `{}` constructs in text given on **stdin** and indents this text on output, according to the nesting of the `{}`. More formally, written in Backus-Nauer form (BNF)[35] this function detects text as in a recursive definition and prints such a program conveniently by changing the line structure and indentation.

program := some-text$_\star$ [**'{'** *program* **'}'** some-text$_\star$]$_\star$

[35] This is a formalized description of computer-readable languages. Here, *program* is recursively defined to be a sequence of text, optionally followed by another sequence of programs inside curly braces.

Listing 19.1. A recursive descent parser for code indentation

```
static
char const* descend(char const* act,
                    unsigned dp[restrict static 1], // Bad
                    size_t len, char buffer[static len],
                    jmp_buf jmpTarget) {
  if (dp[0]+3 > sizeof head) longjmp(jmpTarget, tooDeep);
  ++dp[0];
 NEW_LINE:                                // Loops on output
  while (!act || !act[0]) {               // Loops for input
    if (interrupt) longjmp(jmpTarget, interrupted);
    act = skipspace(fgets(buffer, len, stdin));
    if (!act) {                           // End of stream
      if (dp[0] != 1) longjmp(jmpTarget, plusL);
      else goto ASCEND;
    }
  }
  fputs(&head[sizeof head - (dp[0] + 2)], stdout); // Header

  for (; act && act[0]; ++act) { // Remainder of the line
    switch (act[0]) {
    case left:                   // Descends on left brace
      act = end_line(act+1, jmpTarget);
      act = descend(act, dp, len, buffer, jmpTarget);
      act = end_line(act+1, jmpTarget);
      goto NEW_LINE;
    case right:                  // Returns on right brace
      if (dp[0] == 1) longjmp(jmpTarget, plusR);
      else goto ASCEND;
    default:                     // Prints char and goes on
      putchar(act[0]);
    }
  }
  goto NEW_LINE;
 ASCEND:
  --dp[0];
  return act;
}
```

The operational description of the program is to handle text—in particular, to indent C code or similar in a special way. If we feed the program *text* from listing 3.1 into this, we see the following output:

Terminal

```
0     > ./code/basic_blocks  < code/heron.c
1   | #include <stdlib.h>
2   | #include <stdio.h>
3   | /* lower and upper iteration limits centered around 1.0 */
4   | static double const eps1m01 = 1.0 - 0x1P-01;
5   | static double const eps1p01 = 1.0 + 0x1P-01;
6   | static double const eps1m24 = 1.0 - 0x1P-24;
7   | static double const eps1p24 = 1.0 + 0x1P-24;
8   | int main(int argc, char* argv[argc+1])
9   >| for (int i = 1; i < argc; ++i)
10  >>| // process args
11  >>| double const a = strtod(argv[i], 0);  // arg -> double
12  >>| double x = 1.0;
13  >>| for (;;)
14  >>>| // by powers of 2
15  >>>| double prod = a*x;
16  >>>| if (prod < eps1m01)        x *= 2.0;
17  >>>| else if   (eps1p01 < prod) x *= 0.5;
18  >>>| else break;
19  >>>|
20  >>| for (;;)
21  >>>| // Heron approximation
22  >>>| double prod = a*x;
23  >>>| if ((prod < eps1m24) || (eps1p24 < prod))
24  >>>| x *= (2.0 - prod);
25  >>>| else break;
26  >>>|
27  >>| printf("heron: a=%.5e,\tx=%.5e,\ta*x=%.12f\n",
28  >>| a, x, a*x);
29  >>|
30  >| return EXIT_SUCCESS;
31  >|
```

So **basic_blocks** "eats" curly braces { } and instead indents the code with a series of > characters: each level of nesting { } adds a >.

For a high-level view of how this function achieves this and abstracting away all the functions and variables you do not know yet, have a look at the **switch** statement that starts on line 76 and the **for** loop that surrounds it:

```
static
char const* descend(...) {
  ++dp[0];
 NEW_LINE: ...
```

```
  for (...) {
    switch (...) {
    case left:  ... goto NEW_LINE;
    case right: ... goto ASCEND;
    default:    ...
    }
  }
  goto NEW_LINE;
ASCEND:
  --dp[0];
  return ...;
}
```

It switches according to the current character. Three different cases are distinguished. The simplest is the **default** case: a normal character is printed, the character is advanced, and the next iteration starts.

The two other cases handle { and } characters. If we encounter an opening brace, we know we must indent the text with one more >. Therefore, we recurse into the same function **descend** again (see line 79). If, on the other hand, a closing brace is encountered, we go to **ASCEND** and terminate this recursion level. The recursion depth itself is handled with the variable dp[0], which is incremented on entry (line 63) and decremented on exit (line 91).

If you are trying to understand this program for the first time, the rest is noise. This noise helps to handle exceptional cases, such as an end of line or a surplus of left or right braces. We will see how all this works in much more detail later.

19.2. Sequencing. Before we can look at the details of how the control flow of a program can change in unexpected ways, we must better understand what the normal sequence of C statements guarantees and what it does not. We saw in subsection 4.6 that the evaluation of C *expressions* does not necessarily follow the order in which they are written. For example, the evaluation of function arguments can occur in any order. The different expressions that constitute the arguments can even be interleaved at the discretion of the compiler or depending on the availability of resources at execution time. We say that function argument expressions are *unsequenced*.

There are several reasons for establishing only relaxed rules for evaluation. One is to allow for the easy implementation of optimizing compilers. The efficiency of the compiled code has always been a strong point of C compared to other programming languages.

Another reason for relaxed rules is that C does not add arbitrary restrictions when they don't have a convincing mathematical or technical foundation. Mathematically, the two operands a and b in a+b are freely exchangeable. Imposing an evaluation order would break this rule, and arguing about a C program would become more complicated.

In the absence of threads, most of C's formalization of this is done with *sequence points*. These points in the syntactical specification of the program impose a serialization of the execution. But we will also later see additional rules that force sequencing between the evaluation of certain expressions that don't imply sequence points.

On a high level, a C program can be seen as a series of sequence points reached one after the other, and the code between such sequence points may be executed in any order, be interleaved, or obey certain other sequencing constraints. In the simplest case, for example, when two statements are separated by a semicolon, a statement before a sequence point is *sequenced* before the statement after the sequence point.

But even the existence of sequence points may not impose a particular order between two expressions: it only imposes that there is *some* order. To see that, consider the use of a function add that modifies the object corresponding to its first pointer parameter, x. The use of this function as arguments to **printf** is *well defined*:

sequence_point.c

```
3  unsigned add(unsigned* x, unsigned const* y) [[unsequenced]] {
4    return *x += *y;
5  }
6  int main(void) {
7    unsigned a = 3;
8    unsigned b = 5;
9    printf("a␣=␣%u,␣b␣=␣%u\n", add(&a, &b), add(&b, &a));
10 }
```

From subsection 4.6, remember that the two arguments to **printf** can be evaluated in any order, and the rules for sequence points that we will see shortly will tell us that the function calls to `add` impose sequence points. As a result, we have two possible outcomes for this code. Either the first `add` is executed in its entirety, followed by the second, or the other way around. For the first possibility, we have

- `a` is changed to `8`, and that value is returned.
- `b` is changed to `13`, and that value is returned.

The output of such an execution is

Terminal

```
0    a = 8, b = 13
```

For the second, we get

- `b` is changed to `8`, and that value is returned.
- `a` is changed to `11`, and that value is returned.

The output is

Terminal

```
0    a = 11, b = 8
```

That is, although the behavior of this program is defined, its outcome is not completely determined by the C standard. The specific terminology the C standard applies to such a situation is that the two calls are *indeterminately sequenced*. This discussion is not just theoretical; the two commonly used open source C compilers, `gcc` and `clang`, differ on this simple code. Let me stress this again: this is all defined behavior. Don't expect a compiler to warn you about such problems.

Takeaway 19.2 #1 *Side effects in functions can lead to indeterminate results.*

Here is a list of all sequence points defined in terms of C's grammar:

- The end of a statement, with either a semicolon (`;`) or a closing brace (`}`)
- The end of an expression before the comma operator (`,`)[36]
- The end of a declaration, with either a semicolon (`;`) or a comma (`,`)[37]
- The end of the controlling expressions of **if**, **switch**, **for**, **while**, conditional evaluation (`?:`), and short-circuit evaluation (`||` and `&&`)
- After the evaluations of the function designator (usually a function name) and the function arguments of a function call[38] but before the actual call
- The end of a **return** statement

[36] Be careful: commas that separate function arguments are not in this category.

[37] This also holds for a comma that ends the declaration of an enumeration constant.

[38] This sees the function designator on the same level as the function arguments.

There are other sequencing restrictions besides those implied by sequence points. The first two are more or less obvious but should be stated nevertheless.

Takeaway 19.2 #2 *The specific operation of any operator is sequenced after the evaluation of all its operands.*

Takeaway 19.2 #3 *The effect of updating an object with any of the assignment, increment, or decrement operators is sequenced after the evaluation of its operands.*

For function calls, there also is an additional rule that says the execution of a function is always completed before any other expression.

Takeaway 19.2 #4 *A function call is sequenced with respect to all evaluations of the caller.*

As we have seen, this might be indeterminately sequenced, but sequenced nevertheless. Another source of indeterminately sequenced expressions originates from initializers.

Takeaway 19.2 #5 *Initialization list expressions for array or structure types are indeterminately sequenced.*

Last but not least, some sequence points are also defined for the C library:

- After the actions of format specifiers of the IO functions
- Before any C library function returns[39]
- Before and after calls to the comparison functions used for searching and sorting

The latter two impose rules for C library functions similar to those for ordinary functions. This is needed because the C library itself might not necessarily be implemented in C.

19.3. Short jumps. We have seen a feature that interrupts the common control flow of a C program: **goto**. As you hopefully remember from subsection 15.6, it is implemented with two constructs: *labels* mark positions in the code, and **goto** statements *jump* to these marked positions *inside the same function*.

We also have seen that such jumps have complicated implications for the lifetime and visibility of local objects. In particular, there is a difference in the lifetime of objects defined inside loops and inside a set of statements that is repeated by **goto**.[40] Consider the following two snippets, which both define a local object (a compound literal) in a loop:

```
1  size_t* ip = nullptr;
2  while(something)
3    ip = &(size_t){ fun() };   /* Life ends with while       */
4                               /* Good: resource is released */
5  printf("i␣is␣%d", *ip)       /* Bad: object is dead        */
```

versus

```
1  size_t* ip = nullptr;
2  RETRY:
3    ip = &(size_t){ fun() };   /* Life continues             */
4  if (condition) goto RETRY;
5                               /* Bad: resource is kept      */
6  printf("i␣is␣%d", *ip)       /* Good: object is alive      */
```

[39] Be aware that library functions implemented as macros may not define a sequence point.

[40] See ISO 9899:2011 6.5.2.5 p16.

The address of that compound literal is assigned to a pointer, so the object remains accessible outside the loop and can, for example, be used in a **printf** statement.

They look semantically equivalent, but they are not. For the first, the lifetime of the object that corresponds to the compound literal is associated with the execution of the secondary block of the **while** statement.

Takeaway 19.3 #1 *Each iteration defines a new instance of a local object.*

Therefore, the access to the object in the `*ip` expression is invalid. When omitting the **printf** in the example, the **while** loop has the advantage that the resources occupied by the compound literal can be reused.

For the second example, there is no such restriction: the scope of the definition of the compound literal is the whole surrounding block. So, the object is alive until that block is left (takeaway 13.3 #1). This is not necessarily good: the object occupies resources that could otherwise be reassigned.

In cases where there is no need for the **printf** statement (or similar access), the first snippet is clearer and has better optimization opportunities. Therefore, under most circumstances, it is preferable.

Takeaway 19.3 #2 ***goto** should only be used for exceptional changes in control flow.*

Here, *exceptional* usually means we encounter a transitional error condition that requires local cleanup, as we saw in subsection 15.6. But it could also mean specific algorithmic conditions, as we can see in listing 19.1.

Here, two labels, **NEW_LINE** and **ASCEND**, and two constants, **left** and **right**, reflect the actual state of the parsing. **NEW_LINE** is a jump target when a new line is to be printed, and **ASCEND** is used if a } is encountered or the stream ends. The constants **left** and **right** are used as **case** labels if left or right curly braces are detected.

The reason to have **goto** and labels here is that both states are detected in two different places in the function and at different levels of nesting. In addition, the names of the labels reflect their purpose and thereby provide additional information about the structure.

19.4. Functions. The function **descend** has more complications than the twisted local jump structure; it also is recursive. As we have seen, C handles recursive functions quite simply.

Takeaway 19.4 #1 *Each function call defines a new instance of a local object.*

So usually, different recursive calls to the same function that are active simultaneously don't interact; everybody has their own copy of the program state.

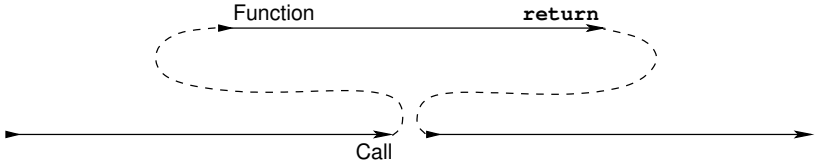


FIGURE 19.1. Control flow of function calls: **return** jumps to the next instruction after the call.

But here, because of the pointers, this principle is weakened. The data to which `buffer` and `dp` point is modified. For `buffer`, this is probably unavoidable: it will contain the data we are reading. But `dp` could (and should) be replaced by a simple

unsigned argument.[Exs 41] Our implementation only has dp as a pointer because we want to be able to track the depth of the nesting in case an error occurs. So, if we abstract out the calls to **longjmp**, which we have not yet explained, using such a pointer is bad. The state of the program is more difficult to follow, and we miss optimization opportunities.[Exs 42]

In our particular example, because dp is **restrict**-qualified and not passed to the calls to longjump (discussed shortly) and it is only incremented at the beginning and decremented at the end, dp[0] is restored to its original value just before the return from the function. So, seen from the outside, it appears that **descend** doesn't change that value at all.

If the function code of **descend** is visible on the call side, a good optimizing compiler can deduce that dp[0] did not change through the call. If **longjmp** weren't special, this would be a nice optimization opportunity. Shortly, we will see how the presence of **longjmp** invalidates this optimization and leads to a subtle bug.

19.5. Long jumps. Our function **descend** may also encounter exceptional conditions that cannot be repaired. We use an enumeration type to name them. Here, **eofOut** is reached if **stdout** can't be written, and **interrupted** refers to an *asynchronous signal* that our running program received. We will discuss this concept later:

basic_blocks.c

state:

Exceptional states of the parse algorithm.

Enumerator:

execution	Normal execution.
plusL	Too many left braces.
plusR	Too many right braces.
tooDeep	Nesting too deep to handle.
eofOut	End of output.
interrupted	Interrupted by a signal.

```
enum state {
  execution = 0,     //!< Normal execution
  plusL,             //!< Too many left braces
  plusR,             //!< Too many right braces
  tooDeep,           //!< Nesting too deep to handle
  eofOut,            //!< End of output
  interrupted,       //!< Interrupted by a signal
};
```

We use the function **longjmp** to deal with these situations, and we put the corresponding calls directly at the place in the code where we recognize that such a condition is reached:

- **tooDeep** is easily recognized at the beginning of the function.
- **plusL** can be detected when we encounter the end of the input stream while we are not at the first recursion level. This can only happen if we encountered an opening brace (that had us go into recursion) but no corresponding right brace has been found before the end of the input.

[Exs 41] Change **descend** so that it receives an **unsigned** depth instead of a pointer.

[Exs 42] Compare the assembler output of the initial version against your version without dp pointer.

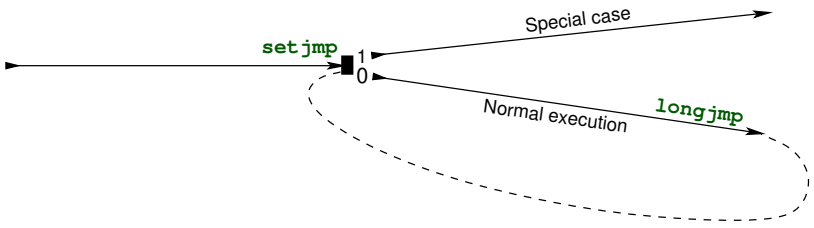


FIGURE 19.2. Control flow with **setjmp** and **longjmp**: **longjmp** jumps to the position marked by **setjmp**.

- **plusR** occurs when we encounter a closing } while we are at the first recursion level.
- **eofOut** is reached if a write to **stdout** returned an end of file (**EOF**) condition.
- **interrupted** is checked before each new line that is read from **stdin**.

Since **stdout** is line buffered, we only check for **eofOut** when we write the '\n' character. This happens inside the short function **end_line**:

basic_blocks.c

```
45 char const* end_line(char const s[static 1], jmp_buf jmpTarget) {
46   if (putchar('\n') == EOF) longjmp(jmpTarget, eofOut);
47   return skipspace(s);
48 }
```

<setjmp.h>

The function **longjmp** comes with a companion macro **setjmp** that is used to establish a jump target to which a call of **longjmp** may refer. The header <setjmp.h> provides the following prototypes:

```
[[noreturn]] void longjmp(jmp_buf target, int condition);
int setjmp(jmp_buf target);    // Usually a macro, not a function
```

The function **longjmp** also has the [[noreturn]] attribute, so we are assured that once we detect one of the exceptional conditions, execution of the current call to **descend** will never continue.

Takeaway 19.5 #1 ***longjmp*** *never returns to the caller.*

This is valuable information for the optimizer. In **descend**, **longjmp** is called in five different places, and the compiler can substantially simplify the analysis of the branches. For example, after the !act tests, it can be assumed that act is non-null on entry to the **for** loop.

Normal syntactical labels are only valid **goto** targets within the same function as they are declared. In contrast, a **jmp_buf** is an opaque object that can be declared anywhere and used as long as it is alive and its contents are valid. In **descend**, we use just one *jump target* of type **jmp_buf**, which we declare as a local variable. This jump target is set up in the base function **basic_blocks** that serves as an interface to **descend** (see listing 19.2). This function mainly consists of one big **switch** statement that handles all the different conditions.

LISTING 19.2. The user interface for the recursive descent parser

```
97 void basic_blocks(void) {
98   char buffer[maxline];
99   unsigned depth = 0;
```

```
100   char const* format =
101     "All␣matching␣%0.0d'%c'␣'%c'␣pairs␣have␣been␣closed␣correctly
        \n";
102   jmp_buf jmpTarget;
103   switch (setjmp(jmpTarget)) {
104   case 0:
105     descend(nullptr, &depth, maxline, buffer, jmpTarget);
106     break;
107   case plusL:
108     format =
109       "Warning:␣%d␣'%c'␣have␣not␣been␣closed␣properly␣(expected␣'
        %c')\n";
110     break;
111   case plusR:
112     format =
113       "Error:␣closing␣too␣many␣(%d)␣'%c'␣parenthesis␣with␣
        additional␣'%c'\n";
114     break;
115   case tooDeep:
116     format =
117       "Error:␣nesting␣(%d)␣of␣'%c'␣'%c'␣constructs␣is␣too␣deep\n"
        ;
118     break;
119   case eofOut:
120     format =
121       "Error:␣EOF␣for␣stdout␣at␣%d␣open␣'%c',␣expecting␣same␣
        amount␣of␣'%c'\n";
122     break;
123   case interrupted:
124     format =
125       "Interrupted␣at␣level␣%d␣of␣'%c'␣'%c'␣nesting\n";
126     break;
127   default:;
128     format =
129       "Error:␣unknown␣error␣within␣(%d)␣'%c'␣'%c'␣constructs\n";
130   }
131   fflush(stdout);
132   fprintf(stderr, format, depth, left, right);
133   if (interrupt) {
134     SH_PRINT(stderr, interrupt,
135              "is␣somebody␣trying␣to␣kill␣us?");
136     raise(interrupt);
137   }
138 }
```

The 0 branch of that **switch** is taken when we come here through the normal control flow. This is one of the basic principles for **setjmp**.

Takeaway 19.5 #2 *When reached through normal control flow, a call to* ***setjmp*** *marks the call location as a jump target and returns* 0*.*

As we said, jmpTarget must be alive and valid when we call **longjmp**. So, for a local variable, the scope of the declaration of the variable must not have been left; otherwise, it would be dead. For validity, all of the context of the **setjmp** must still be active when we call **longjmp**. Here, we avoid complications by having jmpTarget declared in the same scope as the call to **setjmp**.

Takeaway 19.5 #3 *Leaving the scope of a call to* ***setjmp*** *invalidates the jump target.*

Once we enter `case 0` and call **descend**, we may end up in one of the exceptional conditions and call **longjmp** to terminate the parse algorithm. This passes control back to the call location marked in `jmpTarget` as if we just returned from the call to **setjmp**. The only visible difference is that now the return value is the `condition` that we passed as a second argument to **longjmp**. If, for example, we encountered the **tooDeep** condition at the beginning of a recursive call to **descend** and then called **longjmp**`(jmpTarget, tooDeep)`, we jump back to the controlling expression of the **switch** and receive the return value of **tooDeep**. Execution then continues at the corresponding **case** label.

Takeaway 19.5 #4 *A call to* **longjmp** *transfers control directly to the position set by* **setjmp** *as if that had returned the condition argument.*

Be aware, though: precautions have been taken to make it impossible to cheat and retake the normal path a second time.

Takeaway 19.5 #5 *A* `0` *as a* `condition` *parameter to* **longjmp** *is replaced by* `1`*.*

The **setjmp**/**longjmp** mechanism is very powerful and can avoid a whole cascade of returns from function calls. In our example, if we allow the maximal depth of nesting of the input program of, say, 30, the detection of the **tooDeep** condition will happen when there are 30 active recursive calls to **descend**. A regular error-return strategy would **return** to each of these and do some work on each level. A call to **longjmp** allows us to shorten all these returns and proceed to the execution directly in the **switch** of **basic_blocks**.

Because **setjmp**/**longjmp** is allowed to make some simplifying assumptions, this mechanism is surprisingly efficient. Depending on the processor architecture, it usually needs no more than 10 to 20 assembler instructions. The strategy followed by the library implementation is usually quite simple: **setjmp** saves the essential hardware registers, including stack and instruction pointers, in the **jmp_buf** object, and **longjmp** restores them from there and passes control back to the stored instruction pointer.[43]

One of the simplifications **setjmp** makes is about its return. Its specification says it returns an **int** value, but this value cannot be used inside arbitrary expressions.

Takeaway 19.5 #6 **setjmp** *may be used only in simple comparisons inside controlling expression of conditionals.*

So, it can be used directly in a **switch** statement, as in our example, and it can be tested for ==, <, and so on, but the return value of **setjmp** may not be used in an assignment. This guarantees that the **setjmp** value is only compared to a known set of values, and the change in the environment when returning from **longjmp** may just be a special hardware register that controls the effect of conditionals.

As we said, this saving and restoring of the execution environment by the **setjmp** call is minimal. Only a minimal necessary set of hardware registers is saved and restored. No precautions are taken to get local optimizations in line or even to take into account that the call location may be visited a second time.

Takeaway 19.5 #7 *Optimization interacts badly with calls to* **setjmp**.

If you execute and test the code in the example, you will see a problem in our simple usage of **setjmp**. If we trigger the **plusL** condition by feeding a partial program with a missing closing `}`, we would expect the diagnostic to read something like

[43] For the vocabulary, you might want to read or reread subsection 13.5.

Terminal

```
0   Warning: 3 '{' have not been closed properly (expected '}')
```

Depending on the optimization level of your compilation, instead of the `3`, you will most probably see a `0`, independent of the input program. This is because the optimizer does its analysis based on the assumption that the **`switch`** cases are mutually exclusive. It only expects the value of `depth` to change if the execution goes through **`case`** `0` and thus the call of **`descend`**. From inspection of **`descend`** (see subsection 19.4), we know that the value of `depth` is always restored to its original value before return, so the compiler may assume that the value doesn't change through this code path. Then, none of the other cases changes depth, so the compiler can assume that `depth` is always `0` for the **`fprintf`** call.

Consequently, optimization can't make correct assumptions about objects changed in the normal code path of **`setjmp`** and referred to in one of the exceptional paths. There is only one recipe against that.

Takeaway 19.5 #8 *Objects modified across* ***`longjmp`*** *must be* ***`volatile`***.

Syntactically, the qualifier **`volatile`** applies similarly to the other qualifiers, **`const`** and **`restrict`**, that we have encountered. If we declare `depth` with that qualifier,

```
unsigned volatile depth = 0;
```

and amend the prototype of **`descend`** accordingly, all accesses to this object will use the stored value. Optimizations that try to make assumptions about its value are blocked out.

Takeaway 19.5 #9 ***`volatile`*** *objects are reloaded from memory each time they are accessed.*

Takeaway 19.5 #A ***`volatile`*** *objects are stored each time they are modified.*

So **`volatile`** objects are protected from optimization, or if we look at it negatively, they inhibit optimization. Therefore, you should only make objects **`volatile`** if you really need them to be.[Exs 44]

Finally, note some subtleties of the **`jmp_buf`** type. Remember that it is an opaque type: you should never make assumptions about its structure or its individual fields.

Takeaway 19.5 #B *The* ***`typedef`*** *for* ***`jmp_buf`*** *hides an array type.*

Because it is an opaque type, we don't know anything about the base type of the array, `jmp_buf_base`, say.

- An object of type **`jmp_buf`** cannot be assigned to.
- A **`jmp_buf`** function parameter is rewritten to a pointer to `jmp_buf_base`.
- Such a function always refers to the original object and not to a copy.

In a way, this emulates a pass-by-reference mechanism for which other programming languages, such as C++, have an explicit syntax. Generally, using this trick is not a

[Exs 44] Your version of **`descend`** that passes `depth` as a value might not propagate the depth correctly if it encounters the **`plusL`** condition. Ensure that it copies that value to an object that can be used by the **`fprintf`** call in **`basic_blocks`**.

good idea: the semantics of a **jmp_buf** variable depend on being locally declared or a function parameter. For example, in **basic_blocks**, the variable it is not assignable, whereas in **descend**, the analogous function parameter is modifiable because it is rewritten to a pointer. Also, we cannot use it easily for the more specific parameter declarations from modern C, something like

```
jmp_buf_base jmpTarget[restrict const static 1]
```

to insist that the pointer shouldn't be changed inside the function, it must not be null, and access to it can be considered unique for the function.[Exs 45] As of today, we would not design this type like this, and you should not copy this trick to define your own types.

19.6. Signal handlers. As we have seen, **setjmp**/**longjmp** can be used to handle exceptional conditions that we detect ourselves during the execution of our code. A *signal handler* is a tool that handles exceptional conditions that arise differently—that are triggered by some event external to the program. Technically, there are two types of such external events: *hardware interrupts*, also referred to as *traps* or *synchronous signals*, and *software interrupts* or *asynchronous signals*.

The first occurs when the processing device encounters a severe fault it cannot deal with—for example, finding a division by zero, addressing a nonexistent memory bank, or using a misaligned address in an instruction that operates on a wider integer type. Such an event is *synchronous* with the program execution. It is directly caused by a faulting instruction, so we can always know at which particular instruction the interrupt was raised.

The second arises when the operating or runtime system decides that our program should terminate because some deadline is exceeded, a user has issued a termination request, or the world as we know it is going to end. Such an event is *asynchronous* because it can fall in the middle of a multistage instruction, leaving the execution environment in an intermediate state.

Most modern processors have a built-in feature to handle hardware interrupts: an *interrupt vector table*. This table is indexed by the different hardware faults that the platform knows about. Its entries are pointers to procedures, *interrupt handlers*, that are executed when the specific fault occurs. So, if the processor detects such a fault, execution is automatically switched away from the user code, and an interrupt handler is executed. Such a mechanism is not portable because the names and locations of the faults are different from platform to platform. It is tedious to handle because to program a simple application, we'd have to provide all handlers for all interrupts.

C's signal handlers provide us with an abstraction to deal with both types of interrupts, hardware and software, in a portable way. They work similarly to what we describe for hardware interrupts, but

- The names of (some of) the faults are standardized.
- All faults have a default handler (mostly implementation defined).
- And (most) handlers can be specialized.

In each item of that list, there are parenthetical *reservations* because on a closer look, it appears that C's interface for signal handlers is quite rudimentary; all platforms have their extensions and special rules.

Takeaway 19.6 #1 *C's signal-handling interface is minimal and should only be used for elementary situations.*

[Exs 45] Use the C23 **typeof** feature for a **typedef** of jmp_buf_base.

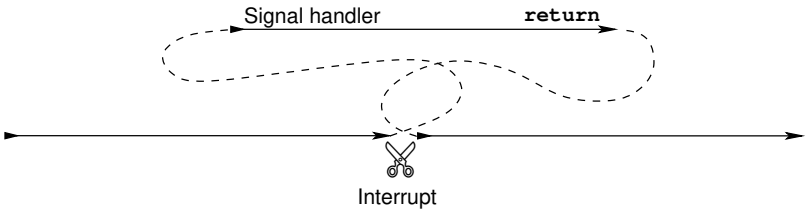


FIGURE 19.3. Control flow after an interrupt **return** jumps to the position where the interrupt occurred.

The control flow of a handled signal is shown in figure 19.3. The normal control flow is interrupted at a place not foreseeable by the application, and a signal handler function kicks in and performs some tasks. After that, the control resumes at exactly the same place and state as when it was interrupted.

The interface is defined in the header `<signal.h>`. The C standard distinguishes six different values, called *signal numbers*. The following are the exact definitions as given there. Three of these values are typically caused by hardware interrupts:[46] `<signal.h>`

```
SIGFPE an erroneous arithmetic operation, such as zero divide
 or an operation resulting in overflow
SIGILL detection of an invalid function image, such as an
 invalid instruction
SIGSEGV an invalid access to storage
```

The other three are usually triggered by software or users:

```
SIGABRT abnormal termination, such as is initiated by the abort
 function
SIGINT receipt of an interactive attention signal
SIGTERM a termination request sent to the program
```

A specific platform will have other signal numbers; the standard reserves all identifiers starting with **SIG** for that purpose. Their use is undefined as of the C standard, but as such, there is nothing bad about it. *Undefined* here really means what it says: if you use it, it has to be defined by some authority other than the C standard, such as your platform provider. Your code becomes less portable as a consequence.

For each possible signal number, the system holds a *disposition* that controls an action performed when the signal is caught. There are two standard dispositions for handling signals, both also represented by symbolic constants. **SIG_DFL** restores the platform's default handler for the specific signal, and **SIG_IGN** indicates that the signal is to be ignored. Then, the programmer may write their own signal handlers. The handler for our parser looks quite simple:

[46] Called computational exceptions by the standard.

basic_blocks.c

signal_handler:

A minimal signal handler.

After updating the signal count, for most signals this simply stores the signal value in "interrupt" and returns.

```
static void signal_handler(int sig) {
  sh_count(sig);
  switch (sig) {
  case SIGTERM: quick_exit(EXIT_FAILURE);
  case SIGABRT: _Exit(EXIT_FAILURE);
#ifdef SIGCONT
    // continue normal operation
  case SIGCONT: return;
#endif
  default:
    /* reset the handling to its default */
    signal(sig, SIG_DFL);
    interrupt = sig;
    return;
  }
}
```

As you can see, such a signal handler receives the signal number `sig` as an argument and **switch**es according to that number. Here we have provisions for signal numbers **SIGTERM** and **SIGABRT**. All other signals are handled by resetting the handler for that number to its default, storing the number in our global variable `interrupt`, and then returning to the point where the interrupt occurred.

The type of signal handler has to be compatible with the following:[47]

sighandler.h

sh_handler:

Prototype of signal handlers.

```
typedef void sh_handler(int);
```

That is, it receives a signal number as an argument and doesn't return anything. As such, this interface is quite limited and does not allow us to pass enough information—in particular, about the location and circumstances for which the signal occurred.

Signal handlers are established by a call to **signal**, as we saw for **signal_handler**. Here, it is used to reset the signal disposition to the default. **signal** is one of the two function interfaces provided by <signal.h>:

<signal.h>

```
sh_handler* signal(int, sh_handler*);
int raise(int);
```

The return value of **signal** is the handler that was previously active for the signal or the special value **SIG_ERR** if an error occurred. Inside a signal handler, **signal** should only be used to change the disposition of the same signal number that was received by the call.

The **main** function for our parser uses this in a loop to establish signal handlers for all signal numbers that it can:

[47] There is no such *type* defined by the standard, though.

basic_blocks.c

```
184   // Establishes signal handlers
185   for (unsigned i = 1; i < sh_known; ++i)
186     sh_enable(i, signal_handler);
```

Here, the function **sh_enable** has the same interface as **signal** but provides a bit more information about the success of the call:

sighandler.c

sh_enable:

Enables a signal handler and catches the errors.

```
sh_handler* sh_enable(int sig, sh_handler* hnd) {
  sh_handler* ret = signal(sig, hnd);
  if (ret == SIG_ERR) {
    SH_PRINT(stderr, sig, "failed");
    errno = 0;
  } else if (ret == SIG_IGN) {
    SH_PRINT(stderr, sig, "previously ignored");
  } else if (ret && ret != SIG_DFL) {
    SH_PRINT(stderr, sig, "previously set otherwise");
  } else {
     SH_PRINT(stderr, sig, "ok");
  }
  return ret;
}
```

As an example, on my machine, this provides the following information at the startup of the program:

Terminal

```
0   sighandler.c:96: #1 (0 times),        unknown signal number,          ok
1   sighandler.c:96: SIGINT (0 times),    interactive attention signal,   ok
2   sighandler.c:96: SIGQUIT (0 times),   keyboard quit,                  ok
3   sighandler.c:96: SIGILL (0 times),    invalid instruction,            ok
4   sighandler.c:96: #5 (0 times),        unknown signal number,          ok
5   sighandler.c:96: SIGABRT (0 times),   abnormal termination,           ok
6   sighandler.c:96: SIGBUS (0 times),    bad address,                    ok
7   sighandler.c:96: SIGFPE (0 times),    erroneous arithmetic operation, ok
8   sighandler.c:89: SIGKILL (0 times),   kill signal,                    failed: Invalid argument
9   sighandler.c:96: #10 (0 times),       unknown signal number,          ok
10  sighandler.c:96: SIGSEGV (0 times),   invalid access to storage,      ok
11  sighandler.c:96: #12 (0 times),       unknown signal number,          ok
12  sighandler.c:96: #13 (0 times),       unknown signal number,          ok
13  sighandler.c:96: #14 (0 times),       unknown signal number,          ok
14  sighandler.c:96: SIGTERM (0 times),   termination request,           ok
15  sighandler.c:96: #16 (0 times),       unknown signal number,          ok
16  sighandler.c:96: #17 (0 times),       unknown signal number,          ok
17  sighandler.c:96: SIGCONT (0 times),   continue if stopped,           ok
18  sighandler.c:89: SIGSTOP (0 times),   stop process,                   failed: Invalid argument
```

The second function **raise** can be used to deliver the specified signal to the current execution. We already used it at the end of **basic_blocks** to deliver the signal we caught to the preinstalled handler.

The mechanism of signals is similar to **setjmp**/**longjmp**: the current state of execution is memorized, control flow is passed to the signal handler, and a return from there restores the original execution environment and continues execution. The difference is that no special point of execution is marked by a call to **setjmp**.

Takeaway 19.6 #2 *Signal handlers can kick in at any point of execution.*

Interesting signal numbers in our case are the software interrupts **SIGABRT**, **SIGTERM**, and **SIGINT**, which usually can be sent to the application with a magic keystroke such as `Ctrl-C`. The first two will call **_Exit** and **quick_exit**, respectively. So, if the program receives these signals, execution will be terminated—for the first, without calling any cleanup handlers and, for the second, by going through the list of cleanup handlers registered with **at_quick_exit**.

SIGINT will choose the **default** case of the signal handler, so it will eventually return to the point where the interrupt occurred.

Takeaway 19.6 #3 *After return from a signal handler, execution resumes exactly where it was interrupted.*

If that interrupt had occurred in function **descend**, it would first continue execution as if nothing had happened. Only when the current input line is processed and a new one is needed will the variable `interrupt` be checked and execution wound down by calling **longjmp**. Effectively, the only difference between the situation before and after the interrupt is that the variable `interrupt` has changed its value.

We also have a special treatment of a signal number not described by the C standard, `SIGCONT` (on my operating system, `POSIX`). To remain portable, the use of this signal number is protected by guards. This signal is meant to continue execution of a program that was previously stopped—that is, for which execution had been suspended. In that case, the only thing to do is return. By definition, we don't want any modification of the program state whatsoever.

Another difference from the **setjmp**/**longjmp** mechanism is that for it, the return value of **setjmp** changed the execution path. A signal handler, on the other hand, is not supposed to change the state of execution. We must invent a suitable convention to transfer information from the signal handler to the normal program. As for **longjmp**, objects that are potentially changed by a signal handler must be **volatile**-qualified. The compiler cannot know where interrupt handlers may kick in, and thus all its assumptions about variables that change through signal handling can be false.

But signal handlers face another difficulty.

Takeaway 19.6 #4 *Any C statement may correspond to several processor instructions.*

For example, a **double** `x` could be stored in two usual machine words, and a write (assignment) of `x` to memory could need two separate assembler statements to write both halves.

When considering normal program execution as we have discussed so far, splitting a C statement into several machine statements is no problem. Such subtleties are not directly observable.[48] With signals, the picture changes. If such an assignment is split in the middle by the occurrence of a signal, only half of `x` is written, and the signal handler will see an inconsistent version of it. One half corresponds to the previous value; the other, to the new one. Such a zombie representation (half here, half there) may not even be a valid value for **double**.

[48] They are only observable from outside the program because such a program may take more time than expected.

Takeaway 19.6 #5 *Signal handlers need types with uninterruptible operations.*

Here, the term *uninterruptible operation* refers to an operation that always appears to be *indivisible* in the context of signal handlers: either it appears not to have started, or it appears to be completed. This doesn't generally mean that it is undivided, just that we will not be able to observe such a division. The runtime system might have to force that property when a signal handler kicks in.

C has three different classes of types that provide uninterruptible operations:

- The type **sig_atomic_t**, an integer type with a minimal width of 8 bits
- The type **atomic_flag**
- All other atomic types with the lock-free property

The first is present on all historical C platforms. Its use to store a signal number, as in our example for variable `interrupt`, is fine, but otherwise, its guarantees are quite restricted. Only memory-load (evaluation) and store (assignment) operations are known to be uninterruptible; other operations aren't, and the width may be quite limited.

Takeaway 19.6 #6 *Objects of type* **sig_atomic_t** *should not be used as counters.*

A simple ++ operation might effectively be divided into three (load, increment, and store) and might easily overflow. The latter could trigger a hardware interrupt, which is really bad if we are already inside a signal handler.

The latter two classes were only introduced by C11 for the prospect of threads (see subsection 20). They are only present if the feature test macro **__STDC_NO_ATOMICS__** has not been defined by the platform and if the header `<stdatomic.h>` has been included. The function **sh_count** uses these features, and we will see an example of this later.

`<stdatomic.h>`

Because signal handlers for asynchronous signals should not access or change the program state in an uncontrolled way, they cannot call other functions that would do so. Functions that *can* be used in such a context are called *asynchronous signal safe*. Generally, it is difficult to know from an interface specification whether a function has this property, and the C standard guarantees it for only a handful of functions:

- The `[[noreturn]]` functions **abort**, **_Exit**, and **quick_exit** that terminate the program
- **signal** for the same signal number for which the signal handler was called
- Some functions that act on atomic objects (discussed shortly)

Takeaway 19.6 #7 *Unless specified otherwise, C library functions are not asynchronous signal safe.*

So, by the C standard itself, a signal handler cannot call **exit** or do any form of IO, but it can use **quick_exit** and the **at_quick_exit** handlers to execute some cleanup code.

As already noted, C's specifications for signal handlers are minimal, and often a specific platform will allow for more. Therefore, portable programming with signals is tedious, and exceptional conditions should generally be dealt with in a cascade, as we have seen in our examples:

- Exceptional conditions that can be detected and handled locally can be dealt with by using **goto** for a limited number of labels.
- Exceptional conditions that need not or cannot be handled locally should be returned as a special value from functions whenever this is possible, such as returning a null pointer instead of a pointer to an object.

- Exceptional conditions that change the global program state can be handled with `setjmp`/`longjmp` if an exceptional return would be expensive or complex.
- Exceptional conditions that result in a signal being raised can be caught by a signal handler but should be handled after the return of the handler in the normal flow of execution.

Because even the list of signals that the C standard specifies is minimal, dealing with the different possible conditions becomes complicated.

sighandler.c

`sh_pairs`:

For each possible signal, this holds a pair of strings with signal information.

The size of this array can be queried with `sh_known`.

See also: SH_PRINT to use that information.

Conditionally, this also adds some commonly used extensions.

```
sh_pair const sh_pairs[] = {
  /* Execution errors */
  SH_PAIR(SIGFPE, "erroneous arithmetic operation"),
  SH_PAIR(SIGILL, "invalid instruction"),
  SH_PAIR(SIGSEGV, "invalid access to storage"),
#ifdef SIGBUS
  SH_PAIR(SIGBUS, "bad address"),
#endif
  /* Job control */
  SH_PAIR(SIGABRT, "abnormal termination"),
  SH_PAIR(SIGINT, "interactive attention signal"),
  SH_PAIR(SIGTERM, "termination request"),
#ifdef SIGKILL
  SH_PAIR(SIGKILL, "kill signal"),
#endif
#ifdef SIGQUIT
  SH_PAIR(SIGQUIT, "keyboard quit"),
#endif
#ifdef SIGSTOP
  SH_PAIR(SIGSTOP, "stop process"),
#endif
#ifdef SIGCONT
  SH_PAIR(SIGCONT, "continue if stopped"),
#endif
#ifdef SIGINFO
  SH_PAIR(SIGINFO, "status information request"),
#endif
};
```

To handle a collection of signal numbers that goes beyond those specified in the C standard we use `SH_PAIR` which links signal numbers and descriptions,

.

sighandler.c

```
7 #define SH_PAIR(X, D) [X] = { .name = #X, .desc = "" D "", }
```

to initialize an object of type `sh_pair`:

sighandler.h

sh_pair: A pair of strings to hold signal information.

```
struct sh_pair {
  char const* name;
  char const* desc;
};
typedef struct sh_pair sh_pair;
```

The collection of value pairs is then held within an array sh_pairs. The use of **#ifdef** conditionals ensures that signal names that are not standard can be used, and the designated initializer within SH_PAIR allows us to specify them in any order. Then the size of the array can be used to compute the number of known signal numbers for sh_known:

. sighandler.c

```
38 size_t const sh_known = (sizeof sh_pairs/sizeof sh_pairs[0]);
```

If the platform has sufficient support for atomics, this information can also be used to define an array of atomic counters so we can keep track of the number of times a particular signal was raised:

. sighandler.h

```
33 #if ATOMIC_LONG_LOCK_FREE > 1
34 /**
35  ** @brief Keep track of the number of calls into a
36  **  signal handler for each possible signal.
37  **
38  ** Don't use this array directly.
39  **
40  ** @see sh_count to update this information.
41  ** @see SH_PRINT to use that information.
42  **/
43 extern _Atomic(unsigned long) sh_counts[];
44
45 /**
46  ** @brief Use this in your signal handler to keep track of the
47  ** number of calls to the signal @a sig.
48  **
49  ** @see sh_counted to use that information.
50  **/
51 inline
52 void sh_count(int sig) {
53   if (sig < sh_known) ++sh_counts[sig];
54 }
55
56 inline
57 unsigned long sh_counted(int sig){
58   return (sig < sh_known) ? sh_counts[sig] : 0;
59 }
60
61 #else
```

An object specified with **_Atomic** can be used with the same operators as other objects with the same base type (here, the ++ operator). In general, such objects are then guaranteed to avoid race conditions with other threads (discussed shortly), and they are

uninterruptible if the type has the *lock-free* property. The latter here is tested with the feature-test macro **ATOMIC_LONG_LOCK_FREE**.[49]

The user interfaces are **sh_count** and **sh_counted**. They use the array of counters if available and are otherwise replaced by trivial functions in the **#else** branch of the query for **ATOMIC_LONG_LOCK_FREE**:

sighandler.h

```
61 #else
62 inline
63 void sh_count(int sig) {
64   // empty
65 }
66
67 inline
68 unsigned long sh_counted(int sig){
69   return 0;
70 }
71 #endif
```

Signal handlers are a feature that is not very well developed in standard C; platforms usually have specific extensions that are more capable. To remain portable, you should try to stay to the minimum and resolve most of the logic of your program outside signal handlers.

[49] Note that similar macros exist, not only for **long**. We just happen to need it for **long** in our example.

Summary

- The execution of C code is not always linearly sequenced, even if there are no parallel threads or asynchronous signals. Consequently, some evaluations may have results that depend on ordering choices by the compiler.
- `setjmp` and `longjmp` are powerful tools that handle exceptional conditions across a whole series of nested function calls. They may interact with optimization and require that some variables be protected with a `volatile` qualification.
- C's interface for handling synchronous and asynchronous signals is rudimentary. Therefore, signal handlers should do as little work as possible, just marking the type of the interrupt condition in a global flag. They should then switch back to the interrupted context and handle the interrupt condition there.
- Information can only be passed to and from signal handlers by using `volatile sig_atomic_t`, `atomic_flag`, or other lock-free atomic data types.

20. Threads

This section covers

- Interthread control
- Initializing and destroying threads
- Working with thread-local data
- Critical data and critical sections
- Communicating through condition variables

Threads are another variation of control flow that allow us to pursue several *tasks* concurrently. Here, a task is a part of the job to be executed by a program such that different tasks can be done with no or little interaction between each other.

Our main example will be a primitive game that we call `B9`, which is a variant of Conway's Game of Life (see Gardner [1970]). It models a matrix of primitive "cells" that are born, live, and die according to very simple rules. We divide the game into four different tasks, each of which proceeds iteratively. The cells go through *life cycles* that compute birth or death events for all cells. The graphical presentation in the terminal goes through drawing cycles, which are updated as fast as the terminal allows. Spread between these are user keystrokes that occur irregularly and allow the user to add cells at chosen positions. Figure 20.1 shows a schematic view of these tasks for `B9`.

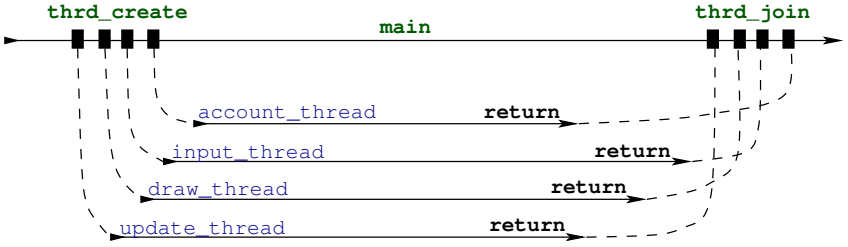


FIGURE 20.1. Control flow of the five threads of `B9`

The four tasks are

Draw: Draws a picture of a cell matrix to the terminal (see figure 20.2)
Input: Captures the keystrokes, updates the cursor position, and creates cells
Update: Updates the state of the game from one life cycle to the next
Account: Tightly coupled with the *update* task and counts the number of living neighboring cells of each cell

The tasks in our example communicate through a data structure of a type called `life`. We will see different members of that structure (such as `n0` and `n1` for numbers of rows and columns and `finished` and `accounted` to keep track of the overall state) as we go along with the discussion. To be able to follow, it might be good to look into the source of this example simultaneously with reading this chapter.

Each of our tasks is executed by a *thread* that follows its own control flow, much like a simple program of its own. If the platform has several processors or cores, these threads may be executed simultaneously. But even if the platform does not have this capacity, the system will interleave the execution of the threads. The execution as a whole will appear to the user *as if* the events handled by the tasks are concurrent. This is crucial for our example since we want the game to appear to continue constantly whether the player presses keys on the keyboard or not.

Threads in C are dealt with through two principal function interfaces that can be used to start a new thread and then wait for the termination of such a thread. They are provided by the `<threads.h>` header (since C11):

<threads.h>

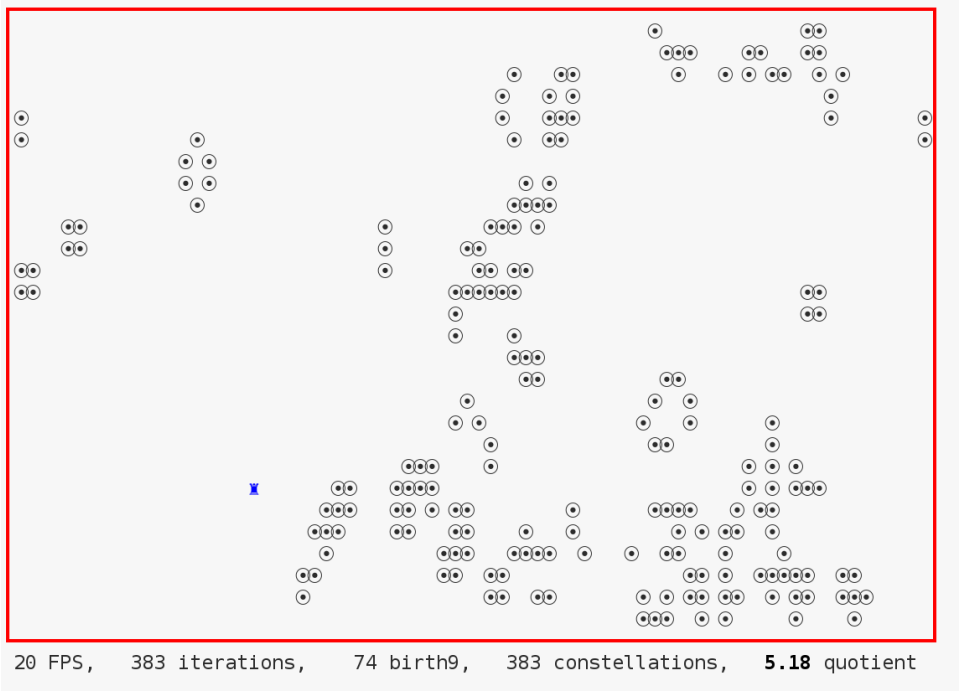


FIGURE 20.2. A screenshot of B9 showing several cells and the cursor position

```
#include <threads.h>
typedef int (*thrd_start_t)(void*);
int thrd_create(thrd_t*, thrd_start_t, void*);
int thrd_join(thrd_t, int *);
```

Here, the second argument of **thrd_create** is a function pointer of type **thrd_start_t**. This function is executed at the start of the new thread. As we can see from the **typedef**, the function receives a **void*** pointer and returns an **int**. The type **thrd_t** is an opaque type, which will identify the newly created thread.

In our example, four calls in **main** to **thrd_create** create the four threads corresponding to the different tasks. These execute concurrently to the original thread of **main**. At the end, **main** waits for the four threads to terminate; it *joins* them. The four threads reach their termination simply when they **return** from the initial function with which they were started. Accordingly, our four functions are declared as

```
static int update_thread(void*);
static int draw_thread(void*);
static int input_thread(void*);
static int account_thread(void*);
```

These four functions are launched in threads of their own by our **main**, and all four functions receive a pointer (declared as **void***) to an object (assumed to be of type life) that holds the state of the game:

B9.c

```
201   /* Create an object that holds the game's data. */
202   life L = LIFE_INITIALIZER;
203   life_init(&L, n0, n1, M);
204   /* Creates four threads that all operate on that same object
205      and collects their IDs in "thrd" */
```

```
  thrd_t thrd[4];
  thrd_create(&thrd[0], update_thread,  &L);
  thrd_create(&thrd[1], draw_thread,    &L);
  thrd_create(&thrd[2], input_thread,   &L);
  thrd_create(&thrd[3], account_thread, &L);
  /* Waits for the update thread to terminate */
  thrd_join(thrd[0], nullptr);
  /* Tells everybody that the game is over */
  L.finished = true;
  ungetc('q', stdin);
  /* Waits for the other threads */
  thrd_join(thrd[1], nullptr);
  thrd_join(thrd[2], nullptr);
  thrd_join(thrd[3], nullptr);
```

The simplest of the four thread functions is `account_thread`. Because the function interface has to be compatible with **`thrd_start_t`**, the function pointer type that **`thrd_create`** expects, its interface only receives a **`void`**`*` parameter. So, the first action of this function is to reinterpret its parameter `Lv` as a `life` pointer `L` and then to enter a **`while`** loop until its work is finished:

B9.c

```
int account_thread(void* Lv) {
  life*restrict L = Lv;
  while (!L->finished) {
    // Blocks until there is work
```

B9.c

```
  return 0;
}
```

The core of that loop calls a specific function for the task `life_account` and then checks whether, from its point of view, the game should be finished:

B9.c

```
    // VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
    life_account(L);
    if ((L->last + repetition) < L->accounted) {
      L->finished = true;
    }
    // ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Here, the condition for termination is whether the game had previously entered the same sequence of `repetition` game configurations, which is set to some constant heuristic value.

The implementations of the other three functions are similar. All reinterpret their argument to be a pointer to `life` and enter a processing loop until they detect that the game has finished. Then, inside the loop, they have relatively simple logic to fulfill their specific task for this specific iteration. For example, `draw_thread`'s inner part looks like this:

B9.c

```
    // VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
    if (L->n0 <= 30) life_draw(L);
    else life_draw4(L);
    L->drawn++;
    // ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

It distinguishes two different drawing functions according to whether the current game has less than 30 rows.

20.1. Simple interthread control. We have already seen two different tools for control between threads: the **thrd_join** function and the data member finished. First, **thrd_join** allows a thread to wait until another one is finished. We saw this when our **main** joined the four other threads. This ensures that this **main** thread effectively only terminates when all other threads have done so, and so the program execution stays alive and consistent until the last thread is gone.

The other tool, which you might not even have noticed as such, is a member of structure life called finished. Because the data structure is shared between all of our threads, this member serves to communicate a condition between them. Here, it holds a **bool** value that is **true** whenever one of the threads detects a condition that terminates the game.

Similar to signal handlers, the simultaneous conflicting action of several threads on *shared* variables must be handled very carefully.

Takeaway 20.1 #1 *If a thread T_0 writes a non-atomic object that is simultaneously read or written by another thread T_1, the execution fails.*

In general, it will even be difficult to establish what *simultaneously* should mean when we talk about different threads (as discussed shortly). Our only chance to avoid such situations is to rule out all potentially conflicting accesses. If there is potentially simultaneous unprotected access, we speak of a *race condition*.

In our example, unless we take specific precautions, even an update of a **bool**, such as finished, can be divisible between different threads. If two threads access it in an interleaved way, an update may mix things up and lead to an undefined program state. The compiler cannot know whether a specific object can be subject to a race condition; therefore, we have to tell it explicitly. The simplest way to do so is using a tool we also saw with signal handlers: atomics. Here, our life structure has several members that are specified with **_Atomic**:

life.h

```
40  // Parameters that will dynamically be changed by
41  // different threads
42  _Atomic(size_t) constellations; //< Constellations visited
43  _Atomic(size_t) x0;             //< Cursor position, row
44  _Atomic(size_t) x1;             //< Cursor position, column
45  _Atomic(size_t) frames;         //< FPS for display
46  _Atomic(bool)   finished;       //< This game is finished.
```

Access to these members is guaranteed to be *atomic*. Here, this is the member finished that we already know, along with some other members we use to communicate between *input* and *draw*—in particular, the current position of the cursor.

Takeaway 20.1 #2 *In view of execution in different threads, standard operations on atomic objects are indivisible and linearizable.*

Here, *linearizability* ensures that we can also argue with respect to the ordering of computations in two different threads. For our example, if a thread sees that finished is modified (set to **true**), it knows that the thread setting this specific variable has performed all the actions it is supposed to do. In that sense, linearizability extends the merely syntactical properties of sequencing (subsection 19.2) to threads.

So, operations on atomic objects also help us determine which parts of our threads are *not* executed simultaneously, such that no race conditions may occur between them.

Later, in subsection 21.1, we will see how to formalize this into the happened-before relation.

Because atomic objects differ semantically from normal objects, the primary syntax to declare them is an *atomic specifier*: as we have seen, the keyword **_Atomic**, followed by parentheses containing the type from which the atomic is derived. There is also another syntax that uses **_Atomic** as an *atomic qualifier*, similar to the other qualifiers: **const**, **volatile**, and **restrict**. In the following specifications, the two different declarations of A and B are equivalent:

```
extern _Atomic(double (*)[45]) A;
extern double (*_Atomic A)[45];
extern _Atomic(double) (*B)[45];
extern double _Atomic  (*B)[45];
```

They refer to the same objects, respectively, A, an atomic pointer to an array of 45 **double** elements, and B, a pointer to an array of 45 atomic **double** elements.

The qualifier notation has a pitfall: it might suggest similarities between **_Atomic** qualifiers and other qualifiers, but, in fact, these do not go very far. Consider the following example with three different "qualifiers":

```
double var;
// Valid: adding const qualification to the pointed-to type
extern double    const* c = &var;
// Valid: adding volatile qualification to the pointed-to type
extern double volatile* v = &var;
// Invalid: pointers to incompatible types
extern double  _Atomic* a = &var;
```

So, it is preferable not to fall into the habit of seeing atomics as qualifiers.

Takeaway 20.1 #3 *Use the specifier syntax* **_Atomic**`(T)` *for atomic declarations.*

Another restriction for **_Atomic** is that it cannot be applied to array types:

```
_Atomic(double[45]) C;  // Invalid: atomic cannot be applied to arrays.
_Atomic(double) D[45];  // Valid:   atomic can be applied to array base
  .
```

Again, this differs from similarly "qualified" types:

```
typedef double darray[45];
// Invalid: atomic cannot be applied to arrays.
darray _Atomic E;
// Valid: const can be applied to arrays.
darray const F = { };    // Applies to base type
double const F[45];      // Compatible declaration
```

Takeaway 20.1 #4 *There are no atomic array types.*

Later on in this section, we will also see another tool that ensures linearizability: **mtx_t**. But atomic objects are by far the most efficient and easy to use.

Takeaway 20.1 #5 *Atomic objects are the privileged tool to force the absence of race conditions.*

20.2. Race-free initialization and destruction. For any data shared by threads, it is important that it is initially set into a well-controlled state before any concurrent access is made and that it is never accessed after it eventually has been destroyed. For initialization, there are several possibilities, presented here in order of preference:

(1) Shared objects with static storage duration are initialized before any execution.
(2) Shared objects with automatic or allocated storage duration can be properly initialized by the thread that creates them *before* any shared access occurs.
(3) Shared objects with static storage duration where the information for dynamic initialization is
   (a) Available at startup time should be initialized by **main** before any other thread is created.
   (b) Not available at startup time *must* be initialized with **call_once**.

So the latter, **call_once**, is only needed under very special circumstances:

```
void call_once(once_flag* flag, void cb(void));
```

Similar to **atexit**, **call_once** registers a callback function cb that should be called at exactly one point of the execution. The function **call_once** and the **once_flag** type come with the <threads.h> header (since C11) and the <stdlib.h> header (since C23).

<threads.h>
<stdlib.h>

The following snippet provides a basic example of how this is supposed to be used:

```
/* Interface */
extern FILE* errlog;
once_flag errlog_flag;
extern void errlog_fopen(void);

/* Incomplete implementation; discussed shortly */
FILE* errlog = nullptr;
once_flag errlog_flag = ONCE_FLAG_INIT;
void errlog_fopen(void) {
  srand(time());
  unsigned salt = rand();
  static char const format[] = "/tmp/error-\%#X.log"
  char fname[16 + sizeof format];
  snprintf(fname, sizeof fname, format, salt);
  errlog = fopen(fname, "w");
  if (errlog) {
    setvbuf(errlog, 0, _IOLBF, 0);   // Enables line buffering
  }
}

/* Usage */

/* ... inside a function before any use ... */
call_once(&errlog_flag, errlog_fopen);
/* ... now use it ... */
fprintf(errlog, "bad, we have weird value \%g!\n", weird);
```

Here we have a global variable (errlog) that needs dynamic initialization (calls to **time**, **srand**, **rand**, **snprintf**, **fopen**, and **setvbuf**) for its initialization. Any usage of that variable should be prefixed with a call to **call_once** that uses the same **once_flag** (here, errlog_flag) and the same callback function (here, errlog_fopen).

So, in contrast to **atexit**, the callback is registered with a specific object, namely one of type **once_flag**. This opaque type guarantees enough state to

- Determine whether a specific call to **call_once** is the very first among all threads.
- Only call the callback then.
- Never call the callback again.
- Hold back all other threads until the one-and-only call to the callback has terminated.

Thus, any using thread can be sure that the object is correctly initialized without overwriting an initialization that another thread might have effected. Most stream functions are required by the C standard to be race-free; the only exceptions are **fopen** and **fclose**, which we will see applied soon.

Takeaway 20.2 #1 *A properly initialized* ***FILE****⋆ can be used race-free by several threads.*

Here, *race-free* only means your program will always be in a well-defined state; it does not mean your file may not contain garbled output lines originating from different threads. To avoid that, you have to make sure a call to **fprintf** or similar always prints an entire line.

Takeaway 20.2 #2 *Concurrent write operations should print entire lines at once.*

Race-free destruction of objects can be much more difficult to organize because the access to data for initialization and destruction is not symmetric. Whereas it often is easy to determine at the beginning of the lifetime of an object that (and when) there is a single user, seeing whether there are still other threads that use an object is difficult if we do not keep track of it.

Takeaway 20.2 #3 *Destruction and deallocation of shared dynamic objects need a lot of care.*

Imagine your precious hour-long execution crashing just before the end, when it tries to write its findings into a file.

In our B9 example, we have a simple strategy to ensure that the variable L could be safely used by all created threads. It is initialized before all threads are created, and it only ceases to exist after all created threads are joined.

For the **once_flag** example, variable errlog, it is not so easy to see when we should close the stream from within one of our threads. The simplest way is to wait until we are sure there are no other threads around—namely, when we are exiting the entire program execution:

```
/* Complete implementation */
FILE* errlog = nullptr;
static void errlog_fclose(void) {
  if (errlog) {
    fputs("\n***␣closing␣log␣***\n", errlog);
    fclose(errlog);
    errlog = nullptr;
  }
}
static void errlog_fflush(void) {
  if (errlog) {
    fputs("\n***␣flushing␣log␣***\n", errlog);
    fflush(errlog);
  }
}

once_flag errlog_flag = ONCE_FLAG_INIT;
void errlog_fopen(void) {
```

```
    atexit(errlog_fclose);
    at_quick_exit(errlog_fflush);
    ...
```

This introduces two other callbacks (`errlog_fclose` and `errlog_fflush`) that ensure all messages are printed to the file. To ensure that the first function is executed on regular program exit, it is registered with **`atexit`**. If the program is terminated with **`quick_exit`** (for example, from a signal handler; see 19.6), closing the file would perhaps already be too costly, so we only register the second function with **`at_quick_exit`**. Both handlers are registered as soon as the initializing function `errlog_fopen` is entered.

20.3. Thread-local data. The easiest way to avoid race conditions is to strictly separate the data our threads access. All other solutions, such as the atomics we saw previously and the mutexes and condition variables that we will see later, are much more complex and much more expensive. The best way to access data local to threads is to use local variables,

Takeaway 20.3 #1 *Pass thread-specific data through function arguments.*

Takeaway 20.3 #2 *Keep thread-specific state in local variables.*

If this is not possible (or maybe too complicated), a special storage class and a dedicated data type allow us to handle thread-local data. **`thread_local`** is a storage class specifier that forces a thread-specific copy of the variable declared as such.

Takeaway 20.3 #3 *A **`thread_local`** variable has one separate instance for each thread.*

That is, **`thread_local`** variables must be declared similarly to variables with static storage duration: they are declared in file scope, or if not, they must also be declared **`static`** (see subsection 13.2, table 13.1). Consequently, they cannot be initialized dynamically.

Takeaway 20.3 #4 *Use **`thread_local`** if initialization can be determined at compile time.*

If a storage class specifier is not sufficient because we have to do dynamic initialization and destruction, we can use *thread-specific storage*, **`tss_t`**, also from the `<threads.h>` header. It abstracts the identification of thread-specific data into an opaque ID, referred to as `key`, and accessor functions to set or get the data:

```
void* tss_get(tss_t key);           // Returns a pointer to an object
int tss_set(tss_t key, void *val);  // Returns an error indication
```

The function called at the end of a thread to destroy the thread-specific data is specified as a function pointer of type **`tss_dtor_t`** when the `key` is created:

```
typedef void (*tss_dtor_t)(void*);          // Pointer to a destructor
int tss_create(tss_t* key, tss_dtor_t dtor); // Returns an error
    indication
void tss_delete(tss_t key);
```

20.4. Critical data and critical sections. Other parts of the `life` structure cannot be protected as easily. They correspond to larger data, such as the board positions of the game. Perhaps you remember that arrays may not be specified with **_Atomic**; even if we were able to do so using some tricks, the result would not be very efficient. Therefore, we not only declare the members `Mv` (for the game matrix) and `visited` (to hash already-visited constellations) but also a special member `mtx`:

life.h

```
15   mtx_t mtx;     //< Mutex that protects Mv
16   cnd_t draw;    //< cnd that controls drawing
17   cnd_t acco;    //< cnd that controls accounting
18   cnd_t upda;    //< cnd that controls updating
19
20   void*restrict Mv;             //< bool M[n0][n1];
21   bool (*visited)[life_maxit]; //< Hashing constellations
```

This member `mtx` has the special type **mtx_t**, a *mutex* type (for *mutual exclusion*) that also comes with `<threads.h>`. It is meant to protect the *critical data*, `Mv`, while it is accessed in a well-identified part of the code (a *critical section*).

<threads.h>

The simplest use case for this mutex is in the center of the input thread (listing 20.1, line 145), where two calls, **mtx_lock** and **mtx_unlock**, protect access to the `life` data structure `L`:

LISTING 20.1. The input thread function of `B9`

```
121 int input_thread(void* Lv) {
122   termin_unbuffered();
123   life*restrict L = Lv;
124   constexpr size_t len = 32;
125   char command[len];
126   do {
127     auto c = getchar();
128     command[0] = c;
129     switch(c) {
130     case GO_LEFT : life_advance(L,  0, -1); break;
131     case GO_RIGHT: life_advance(L,  0, +1); break;
132     case GO_UP   : life_advance(L, -1,  0); break;
133     case GO_DOWN : life_advance(L, +1,  0); break;
134     case GO_HOME : L->x0 = 1; L->x1 = 1;    break;
135     case ESCAPE  :
136       ungetc(termin_translate(termin_read_esc(len, command)),
      stdin);
137       continue;
138     case '+':      if (L->frames < 128) L->frames++; continue;
139     case '-':      if (L->frames > 1)   L->frames--; continue;
140     case ' ':
141     case 'b':
142     case 'B':
143       mtx_lock(&L->mtx);
144       // VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
145       life_birth9(L);
146       // ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
147       cnd_signal(&L->draw);
148       mtx_unlock(&L->mtx);
149       continue;
150     case 'q':
151     case 'Q':
152     case EOF:      goto FINISH;
```

```
153       }
154       cnd_signal(&L->draw);
155     } while (!(L->finished || feof(stdin)));
156   FINISH:
157     L->finished = true;
158     return 0;
159   }
```

This routine is mainly composed of the input loop, which, in turn, contains a big switch to dispatch on different characters that the user typed with the keyboard. Only two of the **case**s need this kind of protection: `'b'` and `'B'`, which trigger the forced "birth" of a 3×3 cluster of cells around the current cursor position. In all other cases, we only interact with atomic objects, so we can safely modify these.

The effect of locking and unlocking the mutex is simple. The call to **mtx_lock** blocks execution of the calling thread until it can be guaranteed that no other thread is inside a critical section protected by the same mutex. We say that **mtx_lock** *acquires* the lock on the mutex, *holds* it, and then **mtx_unlock** *releases* it. The use of `mtx` also provides linearizability similar to the use of atomic objects, as we saw earlier. A thread that has acquired a mutex `M` can rely on the fact that all operations done before other threads released the same mutex `M` have been affected.

Takeaway 20.4 #1 *Mutex operations provide linearizability.*

C's mutex lock interfaces are defined as follows:

```
int mtx_lock(mtx_t*);
int mtx_unlock(mtx_t*);
int mtx_trylock(mtx_t*);
int mtx_timedlock(mtx_t*restrict, const struct timespec*restrict);
```

The two other calls enable us to test (**mtx_trylock**) whether another thread already holds a lock (and thus we may avoid waiting) or to wait (**mtx_timedlock**) for a maximal period (and thus we may avoid blocking forever). The latter is only allowed if the mutex has been initialized as a timed mutex by using **mtx_timed** for the initialization, as discussed shortly.

There are two other calls for dynamic initialization and destruction:

```
int mtx_init(mtx_t*, int);
void mtx_destroy(mtx_t*);
```

Other than for more sophisticated thread interfaces, the use of **mtx_init** is mandatory; there is no static initialization defined for **mtx_t**.

Takeaway 20.4 #2 *Every mutex must be initialized with **mtx_init**.*

The second parameter of **mtx_init** specifies a specific property of the mutex. It must be one of these four values:

- **mtx_plain**
- **mtx_timed**
- **mtx_plain**|**mtx_recursive**
- **mtx_timed**|**mtx_recursive**

As you may have guessed, using **mtx_plain** (versus **mtx_timed**) controls the possibility of using **mtx_timedlock**. The additional property **mtx_recursive** enables us to call **mtx_lock** and similar functions successively several times for the same thread without unlocking it beforehand.

Takeaway 20.4 #3 *A thread that holds a nonrecursive mutex must not call any of the mutex lock functions for it.*

The name **mtx_recursive** indicates that it is mostly used for recursive functions that call **mtx_lock** on entry of a critical section and **mtx_unlock** on exit.

Takeaway 20.4 #4 *A recursive mutex is only released after the holding thread issues as many calls to **mtx_unlock** as it has acquired locks.*

Takeaway 20.4 #5 *A locked mutex must be released before the termination of the thread.*

Takeaway 20.4 #6 *A thread must only call **mtx_unlock** on a mutex that it holds.*

From all of this, we can deduce a simple rule of thumb.

Takeaway 20.4 #7 *Each successful mutex lock corresponds to exactly one call to **mtx_unlock**.*

Depending on the platform, a mutex may bind a system resource that is attributed each time **mtx_init** is called. Such a resource can be additional memory (such as a call to **malloc**) or some special hardware. Therefore, it is important to release such resources once a mutex reaches the end of its lifetime.

Takeaway 20.4 #8 *A mutex must be destroyed at the end of its lifetime.*

So in particular, **mtx_destroy** must be called

- Before the scope of a mutex with automatic storage duration ends
- Before the memory of a dynamically allocated mutex is freed

20.5. Communicating through condition variables. While we have seen that the input didn't need much protection against races, the opposite holds for the account task (see listing 20.2). Its whole job (carried out by the call to `life_account`) is to scan through the entire position matrix and to account for the number of living neighbors that every position has.

LISTING 20.2. The account thread function of `B9`

```
99  int account_thread(void* Lv) {
100   life*restrict L = Lv;
101   while (!L->finished) {
102     // Blocks until there is work
103     mtx_lock(&L->mtx);
104     while (!L->finished && (L->accounted == L->iteration))
105       life_wait(&L->acco, &L->mtx);
106
107     // VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
108     life_account(L);
109     if ((L->last + repetition) < L->accounted) {
110       L->finished = true;
111     }
112     // ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
113
114     cnd_signal(&L->upda);
115     mtx_unlock(&L->mtx);
116   }
117   return 0;
118 }
```

Listing 20.3. The update thread function of B9

```
35 int update_thread(void* Lv) {
36   life*restrict L = Lv;
37   size_t changed = 1;
38   size_t birth9 = 0;
39   while (!L->finished && changed) {
40     // Blocks until there is work
41     mtx_lock(&L->mtx);
42     while (!L->finished && (L->accounted < L->iteration))
43       life_wait(&L->upda, &L->mtx);
44
45     // VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
46     if (birth9 != L->birth9) life_torus(L);
47     life_count(L);
48     changed = life_update(L);
49     life_torus(L);
50     birth9 = L->birth9;
51     if (L->iteration != SIZE_MAX) L->iteration++;
52     else L->finished = true;
53     // ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
54
55     cnd_signal(&L->acco);
56     cnd_signal(&L->draw);
57     mtx_unlock(&L->mtx);
58     life_sleep(1.0/L->frames);
59   }
60   return 0;
61 }
```

Similarly, the update and draw threads mainly consist of one critical section inside an outer loop: see listings 20.3 and 20.4, which perform the action. After that critical section, we also have a call to `life_sleep` that suspends the execution for a certain amount of time. This ensures that these threads are only run with a frequency that corresponds to the frame rate of our graphics.

In all three threads, the critical section mostly covers the loop body. In addition to the proper computation, there is first a phase in these critical sections where the thread is actually paused until new computing is necessary. More precisely, for the accounting thread, there is a conditional loop that can only be left once either the game is finished or another thread has advanced an iteration count.

The body of the inner loop is a call to `life_wait`, a function that suspends the calling thread for 1 second or until a specific event occurs:

life.c

```
17 int life_wait(cnd_t* cnd, mtx_t* mtx) {
18   struct timespec now;
19   timespec_get(&now, TIME_UTC);
20   now.tv_sec += 1;
21   return cnd_timedwait(cnd, mtx, &now);
22 }
```

Its main ingredient is a call to **`cnd_timedwait`** that takes a *condition variable* of type **`cnd_t`**, a mutex, and an absolute time limit.

Listing 20.4. The draw thread function of B9

```
64 int draw_thread(void* Lv) {
65   life*restrict L = Lv;
66   size_t x0 = 0;
67   size_t x1 = 0;
68   fputs(ESC_CLEAR ESC_CLRSCR, stdout);
69   while (!L->finished) {
70     // Blocks until there is work
71     mtx_lock(&L->mtx);
72     while (!L->finished
73            && (L->iteration <= L->drawn)
74            && (x0 == L->x0)
75            && (x1 == L->x1)) {
76       life_wait(&L->draw, &L->mtx);
77     }
78     // VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
79     if (L->n0 <= 30) life_draw(L);
80     else life_draw4(L);
81     L->drawn++;
82     // ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
83
84     mtx_unlock(&L->mtx);
85
86     x0 = L->x0;
87     x1 = L->x1;
88     // No need to draw too quickly
89     life_sleep(1.0/40);
90   }
91   return 0;
92 }
```

Such condition variables are used to identify a condition for which a thread might want to wait. Here, in our example, you have seen declarations for three such condition variable members of `life`: `draw`, `acco`, and `upda`. Each of these corresponds to test conditions that the drawing, the accounting, and the update need to proceed to perform their proper tasks. As we have seen, accounting has

B9.c

```
104     while (!L->finished && (L->accounted == L->iteration))
105       life_wait(&L->acco, &L->mtx);
```

Similarly, update and draw have

B9.c

```
42     while (!L->finished && (L->accounted < L->iteration))
43       life_wait(&L->upda, &L->mtx);
```

and

B9.c

```
72     while (!L->finished
73            && (L->iteration <= L->drawn)
74            && (x0 == L->x0)
75            && (x1 == L->x1)) {
76       life_wait(&L->draw, &L->mtx);
77     }
```

The conditions in each of these loops reflect the cases when there is work to do for the tasks. Most importantly, we have to be sure not to confound the *condition variable*, which serves as a sort of identification of the condition, and the *condition expression*. A call to a wait function for **cnd_t** may return, although nothing concerning the condition expression has changed.

Takeaway 20.5 #1 *On return from a* **cnd_t** *wait, the expression must be checked again.*

Therefore, we place all our calls to `life_wait` inside loops that check the condition expression.

This may be obvious in our examples since we are using **cnd_timedwait** under the hood, and the return might just be because the call timed out. But even if we use the untimed interface for the wait condition, the call might return early. In our example code, the call might eventually return when the game is over, so our condition expression always contains a test for `L->finished`.

cnd_t comes with four principal control interfaces:

```
int cnd_wait(cnd_t*, mtx_t*);
int cnd_timedwait(cnd_t*restrict, mtx_t*restrict, const struct timespec *
    restrict);
int cnd_signal(cnd_t*);
int cnd_broadcast(cnd_t*);
```

The first works analogously to the second, but there is no timeout, and a thread might never come back from the call if the **cnd_t** parameter is never signaled.

cnd_signal and **cnd_broadcast** are on the other end of the control. We saw the first applied in `input_thread` and `account_thread`. They ensure that a thread (**cnd_signal**) or all threads (**cnd_broadcast**) waiting for the corresponding condition variable are woken up and return from the call to **cnd_wait** or **cnd_timedwait**. For example, the input task *signals* the drawing task that something in the game constellation has changed and the board should be redrawn:

B9.c

```
154     cnd_signal(&L->draw);
```

The **mtx_t** parameter to the wait condition functions has an important role. The mutex must be held by the calling thread to the wait function. It is temporarily released during the wait so other threads can do their job to assert the condition expression. The lock is reacquired just before returning from the wait call, so then the critical data can safely be accessed without races.

Figure 20.3 shows a typical interaction between the input and draw threads, the mutex and the corresponding condition variable. It shows that six function calls are involved in the interaction: four for the respective critical sections and the mutex and two for the condition variable.

The coupling between a condition variable and the mutex in the wait call should be handled with care.

Takeaway 20.5 #2 *A condition variable can only be used simultaneously with one mutex.*

But it is probably best practice never to change the mutex that is used with a condition variable at all.

Our example also shows that there can be many condition variables for the same mutex: we use our mutex with three different condition variables at the same time. This will be imperative in many applications since the condition expressions under which threads will be accessing the same resource depend on their respective roles.

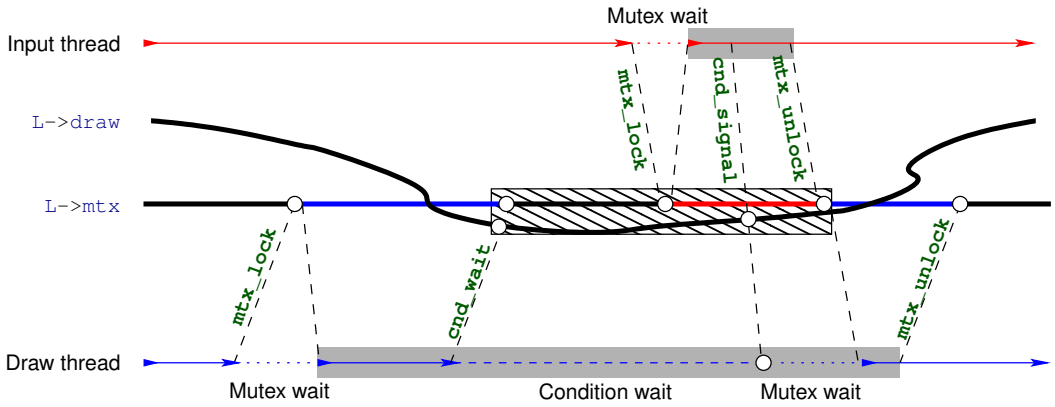


FIGURE 20.3. Control flow managed by mutex `L->mtx` and condition variable `L->draw` between the input and draw threads. Critical sections are shaded with grey. The condition variable is associated with the mutex until the waiter has reacquired the mutex.

In situations where several threads are waiting for the same condition variable and are woken up with a call to **`cnd_broadcast`**, they will not wake up all at once, but one after another as they reacquire the mutex.

Similar to a mutex's, C's condition variables may bind precious system resources. So they must be initialized dynamically, and they should be destroyed at the end of their lifetime.

Takeaway 20.5 #3 *A **`cnd_t`** must be initialized dynamically.*

Takeaway 20.5 #4 *A **`cnd_t`** must be destroyed at the end of its lifetime.*

The interfaces for these are straightforward:

```
int cnd_init(cnd_t *cond);
void cnd_destroy(cnd_t *cond);
```

20.6. More sophisticated thread management. After having seen thread creation and joining in **`main`**, we may have the impression that threads are somehow hierarchically organized. But actually, they are not: just knowing the ID of a thread, its **`thrd_t`**, is sufficient to deal with it. There is only one thread with exactly one special property.

Takeaway 20.6 #1 *Returning from **`main`** or calling **`exit`** terminates all threads.*

If we want to terminate **`main`** after we have created other threads, we have to take some precautions so we do not terminate the other threads too early. An example of such a strategy is given in the following modified version of `B9`'s **`main`**:

B9-detach.c

```
209 life L = LIFE_INITIALIZER;
210
211 void B9_atexit(void) {
212   /* Puts the board in a nice final picture */
213   L.iteration = L.last;
214   life_draw(&L);
215   life_destroy(&L);
216 }
```

```
217
218 int main(int argc, char* argv[argc+1]) {
219   /* Uses command-line arguments for the size of the board */
220   size_t n0 = 30;
221   size_t n1 = 80;
222   if (argc > 1) n0 = strtoull(argv[1], nullptr, 0);
223   if (argc > 2) n1 = strtoull(argv[2], nullptr, 0);
224   /* Create an object that holds the game's data. */
225   life_init(&L, n0, n1, M);
226   atexit(B9_atexit);
227   /* Creates four threads that operate on the same object and
228      discards their IDs */
229   thrd_create(&(thrd_t){0}, update_thread,  &L);
230   thrd_create(&(thrd_t){0}, draw_thread,    &L);
231   thrd_create(&(thrd_t){0}, input_thread,   &L);
232   /* Ends this thread nicely and lets the threads go on nicely */
233   thrd_exit(0);
234 }
```

First, we have to use the function **thrd_exit** to terminate **main**. Other than a **return**, this ensures that the corresponding thread terminates without affecting the other threads. Then, we have to make L a global variable because we don't want its life to end when **main** terminates. To arrange for the necessary cleanup, we also install an **atexit** handler. The modified control flow is shown in figure 20.4.

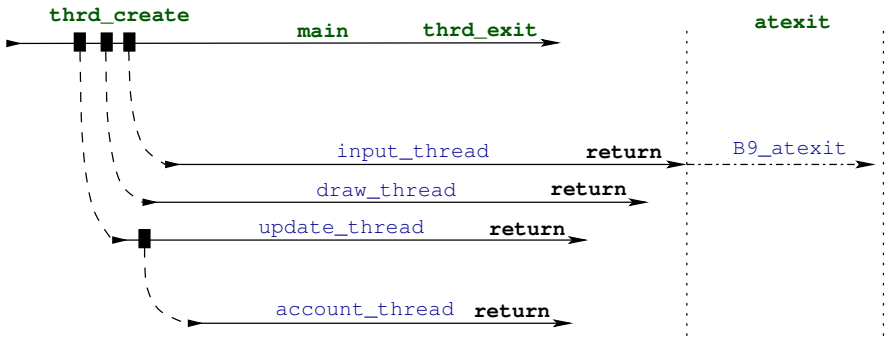


FIGURE 20.4. Control flow of the five threads of B9-detach. The thread that returns last executes the **atexit** handlers.

As a consequence of this different management, the four threads that are created are never actually joined. Each thread that is dead but is never joined eats up some resources that are kept until the end of the execution. Therefore, it is good coding style to tell the system that a thread will never be joined: we say that we *detach* the corresponding thread. We do that by inserting a call to **thrd_detach** at the beginning of the thread functions. We also start the account thread from there and not from **main** as we did previously.

B9-detach.c

```
38   /* Nobody should ever wait for this thread. */
39   thrd_detach(thrd_current());
40   /* Delegates part of our job to an auxiliary thread */
41   thrd_create(&(thrd_t){0}, account_thread, Lv);
42   life*restrict L = Lv;
```

There are six more functions that can be used to manage threads, of which we already met **thrd_current**, **thrd_exit**, and **thrd_detach**:

```
thrd_t thrd_current(void);
int thrd_equal(thrd_t, thrd_t);
[[noreturn]] void thrd_exit(int);

int thrd_detach(thrd_t);
int thrd_sleep(const struct timespec*, struct timespec*);
void thrd_yield(void);
```

A running C program may have many more threads than processing elements. Nevertheless, a runtime system should be able to *schedule* the threads smoothly by attributing time slices on a processor. If a thread has no work to do, it should not demand a time slice and should leave the processing resources to other threads that might need them. This is one of the main features of the control data structures **`mtx_t`** and **`cnd_t`**.

Takeaway 20.6 #2 *While blocking on* **`mtx_t`** *or* **`cnd_t`***, a thread frees processing resources.*

If this is not enough, two other functions can suspend execution:

- **`thrd_sleep`** allows a thread to suspend its execution for a certain time, such that hardware resources of the platform can be used by other threads in the meantime.
- **`thrd_yield`** terminates the current time slice and waits for the next processing opportunity.

20.7. Ensure liveness. As we have seen in section 15.3.4, a deadlock is one of the severe unfortunate events that may jeopardize the execution of a program with several threads. Unfortunately, extensively treating this subject is out of scope for us; parallel and distributed computing are subjects of their own and fill entire books.

An application with threads may be blocked in two fundamentally different ways:

Deadlock: All threads call a mutex lock or condition wait function from which they are never woken up.

Livelock: At least two threads run in control loops with a repetition of state independent from any external input.

A trivial example for the first would be two threads A and B that attempt to lock two mutexes, `M` and `N`. If A first locks `M`, B locks `N`, and then each tries to lock the other mutex, both threads are blocked, and the execution has failed:

```
// thread A                          // thread B
mtx_lock(&M);.......................mtx_lock(&N);
mtx_lock(&N); /*waiting for B */     mtx_lock(&M); /*waiting for A */
```

This situation is best avoided by a simple rule of thumb.

Takeaway 20.7 #1 *Critical sections that need several mutexes to be locked should always lock these mutexes in the same order.*

This rule of thumb already guarantees that a lot of threaded programs that only use mutexes (and no condition variables) do not have a deadlock; for example,

- Programs that lock only one mutex per critical section
- Programs that use the same program code with exactly one critical section for their threads

Were it not for the use of condition variables, our example program `B9` falls into the first category: it only has one mutex—namely, the member `mtx` of the `life` structure. Since we only ever have one object of that type, we only use that particular mutex and no other for any critical section. Then, for the use of our different condition variables of type **`cnd_t`**, we are cautious.

Takeaway 20.7 #2 *Prefer* ***`cnd_timedwait`*** *over* ***`cnd_wait`*** *to avoid deadlocks.*

Indeed, our only condition wait function is located inside `life_wait`—namely, a call to **`cnd_timedwait`**. Therefore, we can assert that `B9` never has a deadlock.

We now know `B9` never blocks, but it remains now to assure ourselves that there is no livelock. To see that, we first notice that the condition

$$\texttt{L->accounted} \leq \texttt{L->iteration}$$

is always satisfied. It is true initially because `L->iteration` is only incremented (by `update_thread`) and never reset to zero. `L->accounted` is only incremented by a call to `live_account` because that function is only called from within `account_thread`, and `L->accounted` is not equal to `L->iteration`.

So, if we suppose that there is no progress in the whole application, both threads would end up in their respective **`while`** loops that call `life_wait` repeatedly. Now, one of the following two situations will occur eventually:

- `account_thread` returns from `life_wait`, and the condition `L->accounted < L->iteration` holds.
- `update_thread` returns from `life_wait`, and the condition `L->accounted == L->iteration` holds.

For the first, `account_thread` will terminate the loop and make progress on the member `L->accounted`. Similarly, for the second, `update_thread` will terminate the loop and make progress on `L->iteration`. In any case, the execution will have made progress.

So now, if there is no user input and no termination condition is met, the threads `update_thread` and `account_thread` alternate over their respective critical sections and will increment the two members `L->iteration` and `L->accounted`, respectively, each time. The input thread can only interfere by changing some data, possibly changing the termination condition. Thus `B9` advances until it meets a termination condition or until the counter `L->iteration` reaches the end of time (namely, **`SIZE_MAX`**).

This proof sketch for the liveness of `B9` uses the fact that counters in the program state only advance. These counters are uninteresting parts of the state for the user of the program. What they see is, indeed, the graphical disposition of the cells. After a certain time (provided the counters are wide enough) without user input, the game will always end up in a position that had been attained before. So, if there is no external event, such as a keystroke, it would very much look like a livelock. `B9` avoids this when it computes the termination condition: the function `life_update` determines whether the current configuration of cells has been seen before.[Exs 50] If this is the case, `update_thread` will declare the game to be finished.

In general, the more complicated a threaded program gets, the more difficult a proof of liveness can get. With our limited example, we were only able to scratch the surface of the difficulties that you may encounter when programming concurrent programs.

[Exs 50] Prove that after a limited amount of iterations, if there is no input from the user, `B9` will revisit a configuration of cells it has visited before.

CHALLENGE 21 (Parallel sorting with threads). *Can you implement a parallel algorithm for sorting using two threads that builds on your merge sort implementation (challenges 1 and 14)? That is, a merge sort cuts the input array in half, sorts each half in its own thread, and then merges the two halves sequentially as before. Use different sequential sorting algorithms as a base inside each of the two threads.*
Can you generalize this parallel sorting to P threads, where $P = 2^k$ for $k = 1, 2, 3, 4$, where k is given on the command line?
Can you measure the speedup you obtained as a result of your parallelization? Does it match the number of cores that your test platform has?

Summary

- It is important to ensure that shared data is properly initialized before it is accessed concurrently. This is best done at compile time or at the beginning of `main`. The function `call_once` can be used to trigger the execution of an initializing function exactly once if that is not possible.
- Threads should preferably only operate on local data through function arguments and automatic variables. If unavoidable, thread-specific data can also be created as `thread_local` objects or via `tss_create`. Use the latter only when you need dynamic construction and destruction of the variable.
- Small critical data shared between threads should be specified as `_Atomic`.
- Critical sections (code paths that operate on unprotected shared data) must be protected, usually by using a `mtx_t` mutex.
- Conditional processing dependencies between threads are modeled with `cnd_t` condition variables.
- A thread that cannot rely on a post-mortem cleanup by `main` should detach by using `thrd_detach` and place all its cleanup code in `atexit` and/or `at_quick_exit` handlers.
- The verification of the liveness of a threaded program that uses mutexes and condition variables can be a challenging task.

21. Atomic access and memory consistency

This section covers

- Understanding the "happened before" relation
- C library calls that provide synchronization
- Maintaining sequential consistency
- Working with other consistency models

We will complete this level with a description of concepts that form an important part of the C architecture model and are, therefore, a must for experienced programmers. This last section will focus on increasing your understanding of how things work, not necessarily improving your operational skills. Even though we will not go into all the glorious details,[51] things can get a bit bumpy; please remain seated and buckle up.

If you review the pictures of control flow we have seen throughout the previous sections, you see that the interaction of different parts of a program execution can get quite complicated. We have different levels of concurrent access to data:

- Plain-old straightforward C code is only apparently sequential. Visibility of changes is only guaranteed between very specific points of the execution, sequence points, for direct data dependencies and the completion of function calls. Modern platforms take more and more advantage of the provided slack and perform unsequenced operations intermixed or in parallel in multiple execution pipelines.
- Long jumps and signal handlers are executed sequentially, but the effects of stores may get lost along the way.
- Those accesses to atomic objects we have seen so far warrant visibility of their changes everywhere and consistently.
- Threads run concurrently side by side and jeopardize data consistency if they don't regulate their shared access to data. In addition to access to atomic objects, they can also be synchronized through calls to functions, such as `thrd_join` or `mtx_lock`.

Access to memory is not the only thing a program does. In fact, the abstract state of a program execution consists of the following:

- *Points of execution* (one per thread)
- *Intermediate values* (of computed expressions or evaluated objects)
- *Stored values*
- *Hidden state*

Changes to this state are usually described as

Jumps: Change the point of execution (short jumps, long jumps, and function calls)
Value computation: Changes intermediate values
Side effects: Store values or do IO

Or they can affect hidden state, such as the lock state of a `mtx_t` object, the initialization state of a `once_flag` object, and setting or clearing operations on a `atomic_flag` object.

We summarize all these possible changes to the abstract state with the term *effect*.

Takeaway 21 #1 *Every evaluation has an effect.*

[51] We will put aside `memory_order_consume` consistency and thus the dependency-ordered before relation.

This is because any evaluation has the concept of a next such evaluation that will be performed after it. Even an expression like

```
(void)0;
```

which drops the intermediate value and sets the point of execution to the next statement, and thus the abstract state changes.

In a complicated context, it will be difficult to argue about the actual abstract state of an execution in a given moment. Generally, the entire abstract state of a program's execution is not even observable, and in many cases, the concept of an overall abstract state is not well defined because we don't know what *moment* means in this context. In a multithreaded execution performed on several physical compute cores, there is no real notion of a reference time between them. So, generally, C does not even make the assumption that an overall fine-grained notion of time exists between different threads.

As an analogy, think of two threads, `A` and `B`, as events that happen on two different planets orbiting at different speeds around a star. Times on these planets (threads) are relative, and synchronization between them takes place only when a signal issued from one planet (thread) reaches the other. The transmission of the signal takes time by itself, and when the signal reaches its destination, its source has moved on. So, the mutual knowledge of the two planets (threads) is always partial.

21.1. The "happened before" relation. If we want to argue about a program's execution (its correctness, its performance, and so on), we need to have enough *partial knowledge* about the state of all threads and know how to stitch that partial knowledge together into a coherent view of the whole.

Therefore, we will investigate a relation introduced by Lamport [1978]. In C standard terms, it is the *happened before* relation between two evaluations, E and F, denoted by $F \to E$. We observe this is a property between events a posteriori. Fully spelled out, it would perhaps be better called the *knowingly happened before* relation instead.

One part of it consists of evaluations in the same thread that are related by the already-introduced sequenced before relation:

Takeaway 21.1 #1 *If F is sequenced before E, then $F \to E$.*

To see that, let us revisit listing 20.1 from our input thread. Here, the assignment to `command[0]` is sequenced before the **`switch`** statement. Therefore, we are sure that all cases in the **`switch`** statement are executed *after* the assignment or at least that they will be *perceived* as happening later. For example, when passing `command` to the nested function calls below **`ungetc`**, we are sure it will provide the modified value. All of this can be deduced from C's grammar.

Between threads, the ordering of events is provided by *synchronization*. There are two types of synchronization: the first is implied by operations on atomics, and the second, by certain C library calls. Let us first look into the case of atomics. An atomic object can be used to synchronize two threads if one thread writes a value and another thread reads the value that was written. Operations on atomics are guaranteed to be locally consistent.

Takeaway 21.1 #2 *The set of modifications of an atomic object `X` is performed in an order consistent with the sequenced before relation of any thread that deals with `X`.*

This sequence is called the *modification order* of `X`. Consider an atomic variable `x`,

```
_Atomic(unsigned) x = 11;
```

a thread `A` that performs the instructions

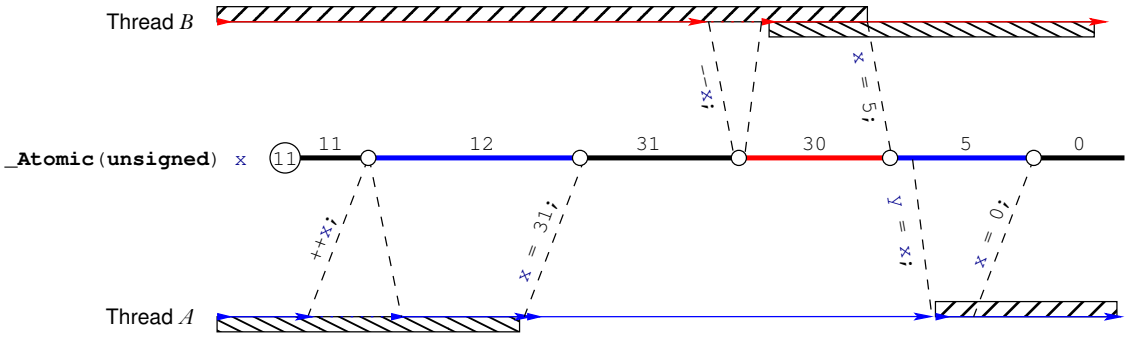


FIGURE 21.1. Two threads that synchronize via an atomic. The circles present the modifications of object `x`. The bars below the threads represent information about the state of A, and those above represent information about the state of B.

```
++x;
x = 31;
...
unsigned y = x
x = 0;
```

and a thread `B` that mingles with the same variable `x`

```
x--;
x = 5;
```

A possible modification order for `x` is shown in figure 21.1. Here, we have six modifications to `x`: the initialization (value `11`), one increment, one decrement, and three assignments. The C standard guarantees that each of the two threads, `A` and `B`, perceives all changes to `x` in an order consistent with this modification order.

In the example in figure 21.1, we have only two synchronizations. First, thread `B` synchronizes with `A` at the end of its `--x` operation because here it has read (and modified) the value `31` that `A` wrote. The second synchronization occurs when `A` reads the value `5` that `B` wrote and stores it into `y`.

As another example, let us investigate the interplay between the input thread (listing 20.1) and the account thread (listing 20.2). Both read and modify the field `finished` in different places. For the simplicity of the argument, let us assume that there is no other place than in these two functions where `finished` is modified.

The two threads will only synchronize via this atomic object if either modifies it—that is, writes the value **`true`** into it. This can happen under two circumstances:

- The input thread encounters an end-of-file condition, either when **`feof`**`(`**`stdin`**`)` returns **`true`** or if the case **`EOF`** is encountered. In both cases, the **`do`** loop terminates, and the code after the label `FINISH` is executed.
- The account thread detects that the number of permitted repetitions is exceeded and sets `finished` to **`true`**.

These events are not exclusive, but using an atomic object guarantees that one of the two threads will succeed first in writing to `finished`:

- If the input thread writes first, the account thread may read the modified value of `finished` in the evaluations of one of its **`while`** loops. This read

synchronizes; that is, the write event in the input thread is known to have happened before such a read. Any modifications the input thread made before the write operation is now visible to the account thread.
- If the account thread writes first, the input thread may read the modified value in the **while** of its **do** loop. Again, this read synchronizes with the write and establishes a "happened before" relation, and all modifications made by the account thread are visible to the input thread.

Observe that these synchronizations are oriented: each synchronization between threads has a "writer" and a "reader" side. We attach two abstract properties to operations on atomics and to certain C library calls called *release* semantics (on the writer side), *acquire* semantics (for a reader), and *acquire-release* semantics (for a reader-writer). C library calls with such synchronization properties will be discussed a bit later.

All operations on atomics we have seen so far and modify the object are required to have release semantics, and all that read must have acquire semantics. Later, we will see other atomic operations that have relaxed properties.

Takeaway 21.1 #3 *An acquire operation E in a thread T_E synchronizes with a release operation F in another thread T_F if E reads the value that F has written.*

The idea of the special construction with acquire and release semantics is to force the visibility of effects across such operations. We say that an effect X is *visible* at evaluation E if we can consistently replace E with any appropriate read operation or function call that uses the state affected by X. For example, in figure 21.1, the effects `A` produces before its `x = 31` operation are symbolized by the bar below the thread. They are visible to `B` once `B` has completed the `--x` operation.

Takeaway 21.1 #4 *If F synchronizes with E, all effects X that happened before `F` must be visible at all evaluations `G` that happen after `E`.*

As we saw in the example, there are atomic operations that can read *and* write atomically in one step. These are called *read-modify-write* operations:

- Calls to **atomic_exchange** and **atomic_compare_exchange_weak** for any **_Atomic** objects
- Compound assignments or their functional equivalents; increment and decrement operators for any **_Atomic** objects of arithmetic type
- Calls to **atomic_flag_test_and_set** for **atomic_flag**

Such an operation can synchronize on the read side with one thread and on the write side with others. All such read-modify-write operations that we have seen so far have both acquire and release semantics.

The happened before relation is the transitive closure of the combination of the sequenced before and synchronizes-with relations. We say that F knowingly happened before E, if there are n and $E_0 = F, E_1, \ldots, E_{n-1}, E_n = E$ such that E_i is sequenced before E_{i+1} or synchronizes with it for all $0 \leq i < n$.

Takeaway 21.1 #5 *We only can conclude that one evaluation happened before another if we have a sequenced chain of synchronizations that links them.*

Observe that this happened before relation is a combination of very different concepts. The sequenced before relation can in many places be deduced from syntax, in particular, if two statements are members of the same basic block. Synchronization is different: besides the two exceptions of thread startup and end, it is deduced through a data dependency on a specific object, such as an atomic or a mutex.

The desired result of all this is that effects in one thread become visible in another.

Takeaway 21.1 #6 *If an evaluation F happened before E, all effects that are known to have happened before F are also known to have happened before E.*

21.2. C library calls that provide synchronization. C library functions with synchronizing properties come in pairs: a releasing side and an acquiring side. They are summarized in table 21.1.

TABLE 21.1. C library functions that form synchronization pairs

Release	Acquire
thrd_create(.., f, x)	Entry to f(x)
thrd_exit by thread id or **return** from f	Start of **tss_t** destructors for id
End of **tss_t** destructors for id	**thrd_join**(id) or **atexit**/**at_quick_exit** handlers
call_once(&obj, g), first call	**call_once**(&obj, h), all subsequent calls
Mutex release	Mutex acquisition

For the first three entries, we know which events synchronize with which; namely, the synchronization is mainly limited to effects done by thread id. In particular, by transitivity, **thrd_exit** or **return** always synchronize with **thrd_join** for the corresponding thread id.

These synchronization features of **thrd_create** and **thrd_join** allow us to draw the lines in figure 20.1. Here, we do not know about any timing of events between the threads we launched, but within **main**, we know that the order in which we created the threads and the order in which we joined them is exactly as shown. We also know that all effects of any of these threads on data objects are visible to **main** after we join the last thread: the account thread.

If we detach our threads and don't use **thrd_join**, synchronization can only take place between the end of a thread and the start of an **atexit** or **at_quick_exit** handler.

The other library functions are a bit more complicated. For the initialization utility **call_once**, the return from the very first call **call_once**(&obj, g) is special: it is the only call that effectively makes a call into the callback function g. This first call is a release operation for all subsequent calls with the same object obj. These all are then acquire operations on this object. This ensures that all write operations performed during the call to g() are known to happen before any other call with obj and are effectively visible in the corresponding thread.

For our example in subsection 20.2, this means the function errlog_fopen is executed exactly once, and all other threads that might execute the **call_once** line will synchronize with that first call. So when any of the threads return from the call, they know that the call has been performed (either by themselves or by another thread that was faster) and that all effects, such as computing the file name and opening the stream, are visible now. Thus, all threads that executed the call may use errlog and can be sure it is properly initialized.

For a mutex, a release operation can be a call to a mutex function, **mtx_unlock**, or the entry into the wait functions for condition variables, **cnd_wait** and **cnd_timedwait**. An acquire operation on a mutex is the successful acquisition of the mutex via any of the three mutex calls (**mtx_lock**, **mtx_trylock**, and **mtx_timedlock**) or the return from the wait function (**cnd_wait** or **cnd_timedwait**).

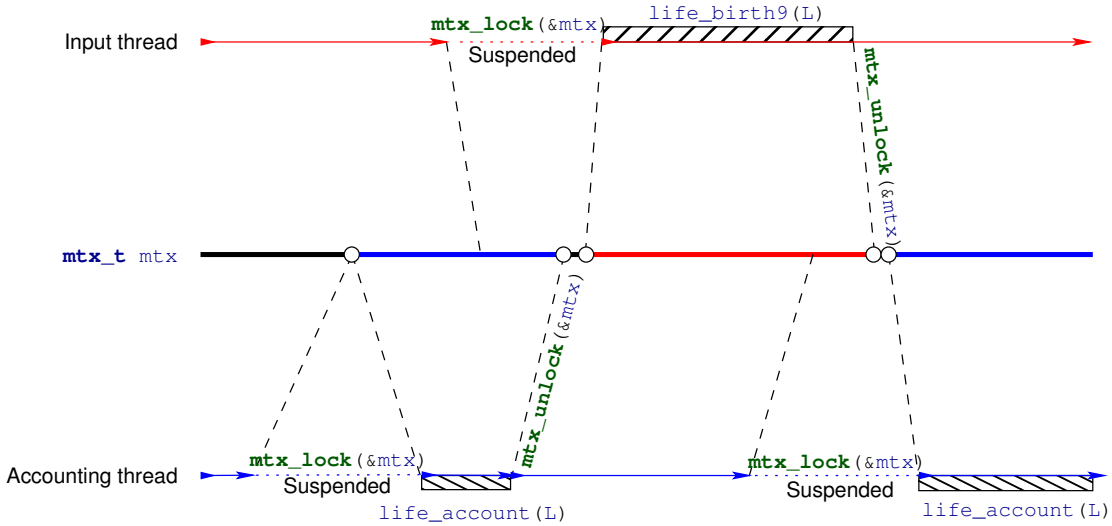


FIGURE 21.2. Two threads with three critical sections that synchronize via a mutex. The circles present the modifications of object `mtx`.

Takeaway 21.2 #1 *Critical sections protected by the same mutex occur sequentially.*

Our input and accounting threads from the example (listings 20.1 and 20.2) access the same mutex `L->mtx`. In the first, it is used to protect the birth of a new set of cells if the user types a `' '`, `'b'`, or `'B'`. In the second, the entire secondary block of the *while* loop is protected by the mutex.

Figure 21.2 schematizes a sequence of three critical sections protected by the mutex. The synchronization between the unlock operations (release) and the return from the lock operation (acquire) synchronizes the two threads. This guarantees that the changes applied to `*L` by the account thread in the first call to `life_account` are visible in the input thread when it calls `life_birth9`. Equally, the second call to `life_account` sees all changes to `*L` that occur during the call to `life_birth9`.

Takeaway 21.2 #2 *In a critical section protected by the mutex `mut`, all effects of previous critical sections protected by `mut` are visible.*

One of these known effects is always the advancement of the point of execution. In particular, on return from **`mtx_unlock`**, the execution point is outside the critical section, and this effect is known to the next thread that newly acquires the lock.

The wait functions for condition variables differ from acquire-release semantics; in fact, they work exactly the other way around.

Takeaway 21.2 #3 ***`cnd_wait`*** *and* ***`cnd_timedwait`*** *have release-acquire semantics for the mutex.*

That is, before suspending the calling thread, they perform a release operation and then, when returning, an acquire operation. The other peculiarity is that the synchronization goes through the mutex, not through the condition variable itself.

Takeaway 21.2 #4 *Calls to* ***`cnd_signal`*** *and* ***`cnd_broadcast`*** *synchronize via the mutex.*

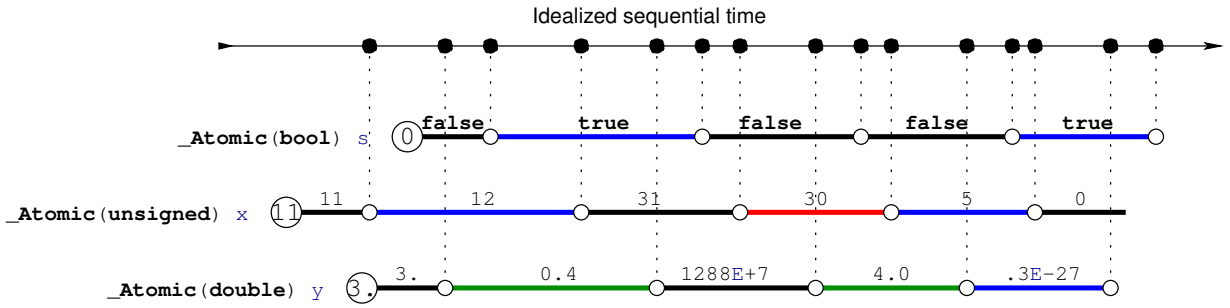


FIGURE 21.3. Sequential consistency for three different atomic objects

The signaling thread will not necessarily synchronize with the waiting thread if it does not place a call to **cnd_signal** or **cnd_broadcast** into a critical section protected by the same mutex as the waiter. In particular, non-atomic modifications of objects that constitute the *condition expression* may not become visible to a thread woken up by a signal if the modification is not protected by the mutex. There is a simple rule of thumb to ensure synchronization:

Takeaway 21.2 #5 *Calls to **cnd_signal** and **cnd_broadcast** should occur inside a critical section protected by the same mutex as the waiters.*

This is what we saw around line 145 in listing 20.1. Here, the function `life_birth` modifies larger, non-atomic parts of the shared object `*L`, so we must make sure these modifications are properly visible to all other threads that work with `*L`.

Line 154 shows a use of **cnd_signal** that is not protected by the mutex. This is only possible here because all data modified in the other **switch** cases is atomic. Thus, other threads that read this data, such as `L->frames`, can synchronize through these atomics and do not rely on acquiring the mutex. Be careful if you use conditional variables like that.

21.3. Sequential consistency. The data consistency for the previously described atomic objects, guaranteed by the *happened before* relation, is called *acquire-release consistency*. Whereas the C library calls we have seen always synchronize with that kind of consistency, no more and no less, accesses to atomics can be specified with different consistency models.

As you remember, all atomic objects have a *modification order* that is consistent with all sequenced before relations that see these modifications *on the same* object. *Sequential consistency* has even more requirements than that; see figure 21.3. Figure 21.3 illustrates the common timeline of all sequentially consistent operations on top. Even if these operations are performed on different processors and the atomic objects are realized in different memory banks, the platform has to ensure that all threads perceive all these operations as being consistent with this one global linearization.

Takeaway 21.3 #1 *All atomic operations with sequential consistency occur in one global modification order, regardless of the atomic object they are applied to.*

So, sequential consistency is a very strong requirement. Not only that, it enforces acquire-release semantics (a causal partial ordering between events), but it extends this partial ordering to a total ordering. If you are interested in parallelizing the execution of your program, sequential consistency may not be the right choice because it may force sequential execution of the atomic accesses.

The standard provides the following functional interfaces for atomic types. They should conform to the description given by their name and also perform synchronization:

```
void atomic_store(A volatile* obj, C des);
C atomic_load(A volatile* obj);
C atomic_exchange(A volatile* obj, C des);
bool atomic_compare_exchange_strong(A volatile* obj, C *expe, C des);
bool atomic_compare_exchange_weak(A volatile* obj, C *expe, C des);
C atomic_fetch_add(A volatile* obj, M operand);
C atomic_fetch_sub(A volatile* obj, M operand);
C atomic_fetch_and(A volatile* obj, M operand);
C atomic_fetch_or(A volatile* obj, M operand);
C atomic_fetch_xor(A volatile* obj, M operand);
bool atomic_flag_test_and_set(atomic_flag volatile* obj);
void atomic_flag_clear(atomic_flag volatile* obj);
```

Here `C` is any appropriate data type, `A` is the corresponding atomic type, and `M` is a type that is compatible with the arithmetic of `C`. As the names suggest, for the fetch and operator interfaces, the call returns the value that `*obj` had before the modification of the object. So, these interfaces are not equivalent to the corresponding compound assignment operator (+=), since that would return the result *after* the modification.

All these functional interfaces provide *sequential consistency*.

Takeaway 21.3 #2 *All operators and functional interfaces on atomics that don't specify otherwise have sequential consistency.*

Observe also that the functional interfaces differ from the operator forms because their arguments are **`volatile`** qualified.

There is another function call for atomic objects that, in contrast to the function we have seen until now, does not imply synchronization:

```
void atomic_init(A volatile* obj, C des);
```

Its effect is the same as a call to **`atomic_store`** or an assignment operation, but concurrent calls from different threads can produce a race. View **`atomic_init`** as a cheap form of assignment.

21.4. Other consistency models. A different consistency model can be requested with a complementary set of functional interfaces. For example, an equivalent to the postfix ++ operator with just acquire-release consistency could be specified with

```
_Atomic(unsigned) at = 67;
...
if (atomic_fetch_add_explicit(&at, 1, memory_order_acq_rel)) {
  ...
}
```

Takeaway 21.4 #1 *Synchronizing functional interfaces for atomic objects with **`_explicit`** appended allow us to specify their consistency model.*

These interfaces accept additional arguments in the form of symbolic constants of type **`memory_order`** that specify the memory semantics of the operation:

- **memory_order_seq_cst** requests sequential consistency. Using this is equivalent to the forms without **_explicit**.
- **memory_order_acq_rel** is for an operation with acquire-release consistency. Typically, for general atomic types, you'd use it for a read-modify-write operation, such as **atomic_fetch_add** or **atomic_compare_exchange_weak**, or for the type **atomic_flag** with **atomic_flag_test_and_set**.
- **memory_order_release** is for an operation with only release semantics. Typically, this would be **atomic_store** or **atomic_flag_clear**.
- **memory_order_acquire** is for an operation that only has acquire semantics. Typically, this would be **atomic_load**.
- **memory_order_consume** is for an operation that has a weaker form of causal dependency than acquire consistency. Typically, this would also be **atomic_load**.
- **memory_order_relaxed** is for an operation that adds no synchronization requirements. The only guarantee for such an operation is that it is indivisible. A typical use case is a performance counter used by different threads, but for which we are only interested in a final accumulated count.

The consistency models can be compared with respect to the restrictions they impose on the platform. Figure 21.4 shows the implication order of the **memory_order** models.

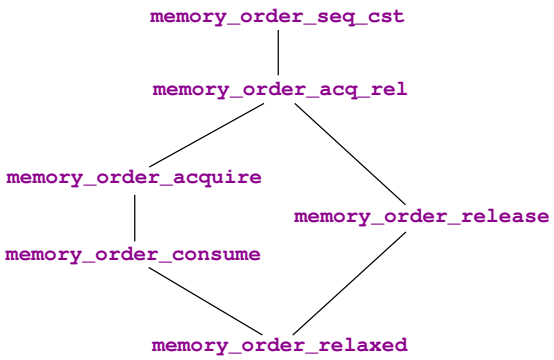


FIGURE 21.4. Hierarchy of consistency models, from least to most constraining

Whereas **memory_order_seq_cst** and **memory_order_relaxed** are admissible for all operations, there are some restrictions for other **memory_order**s. Operations that can only occur on one side of a synchronization can only specify an order for that side:

- Two operations that only store (**atomic_store** and **atomic_flag_clear**) may not specify acquire semantics.
- Three operations that only perform a load and may not specify release or consume semantics; these are **atomic_load** (for the single **memory_order** argument) and the weak and strong versions of compare-exchange operations in case of failure.

Thus, the latter needs two **memory_order** arguments for their **_explicit** form, such that they can distinguish the requirements for the success and failure cases:

```
bool
atomic_compare_exchange_strong_explicit(A volatile* obj, C *expe, C des,
                                        memory_order success,
                                        memory_order failure);
bool
atomic_compare_exchange_weak_explicit(A volatile* obj, C *expe, C des,
                                      memory_order success,
                                      memory_order failure);
```

Here, the `success` consistency must be at least as strong as the `failure` consistency; see figure 21.4.

Up to now, we have implicitly assumed that the acquire and release sides of a synchronization are symmetric, but they aren't. Whereas there is always just one writer of a modification, there can be several readers. Because moving new data to several processors or cores is expensive, some platforms allow us to avoid the propagation of all visible effects that happened before an atomic operation to all threads that read the new value. C's *consume consistency* is designed to map this behavior. We will not go into the details of this model, and you should use it only when you are certain that some effects prior to an atomic read will not affect the reading thread.

Summary

- The "happened before" relation is the only possible way to reason about timing between different threads. It is only established through synchronization that uses either atomic objects or very specific C library functions.
- Sequential consistency is the default consistency model for atomics but not for other C library functions. It additionally assumes that all corresponding synchronization events are totally ordered. This assumption can be expensive.
- Explicitly using acquire-release consistency can lead to more efficient code, but it needs a careful design to supply the correct arguments to the atomic functions with a `_explicit` suffix.

Technical Annex

If you read this book much after 2024, this technical annex should not concern you: you should have a development framework at your disposal that implements C23 without fault. If not, you might have some difficulties because one or another feature of C23 is not yet implemented. The following site tracks the progress of C23 conformance of several toolchains:

`https://en.cppreference.com/w/c/compiler_support`.

My personal experience with programming in C is almost entirely acquired in the realm of open source projects and open source compilers (`gcc` and `clang`) and on specific operating systems— namely, POSIX(2024) in its incarnation as glibc/Linux and musl/Linux. If this is not your setting and you encounter severe problems with the code in this book, you'll have to search the web for solutions, as I and the remainder of this annex will probably not be of much help.

A. Transitional code

C23 comes with a lot of new features, and at the beginning, we cannot expect that all compilers will catch up quickly to implement all new features. Nevertheless, many of these new and closely related features are already present in compilers as extensions. Therefore, in the sample code provided with this book at

`https://inria.hal.science/hal-03345464/document`,

<c23-fallback.h> we provide a header file <c23-fallback.h> meant to circumvent most of the difficulties you might encounter. It is systematically included in our examples, such that these should, in general, compile even in legacy settings. Nevertheless, some C23 features are required, namely,

- Digit separators such as in `0xAB'CD`
- Binary integer literals such as in `0b1010` or `0B0101`
- The new attribute syntax such as in `[[deprecated]]`

The first two are not easy to circumvent, so you should probably not even try to use C23-enabled code without having support for them.

For the attribute syntax, there is a feature test: **`__has_c_attribute`**. It can be used with define (or **`#ifdef`**) to test for the syntax itself and with **`#if`** and an argument to test for an individual attribute:

c23-fallback.h
```
319 #ifndef __has_c_attribute
320 # define __has_c_attribute(X) 0
321 #endif
```

Another feature test is **`__has_include`**, which similarly can be used to test for the preprocessor feature itself and, if that is available, to test for the availability of a specific header. This can be used to

<complex.h> <threads.h> <stdatomic.h>
- Test for optional headers such as <complex.h>, <threads.h>, and <stdatomic.h>. In principle, this now allows us to use the corresponding feature test macro

  (**`__STDC_NO_COMPLEX__ __STDC_NO_ATOMICS__ __STDC_NO_THREADS__`**)

  to test for compiler support of these features separately from the availability of library interfaces.

<stdckdint.h> <stdbit.h>
- Test for new headers that come with C23, namely, <stdckdint.h> and <stdbit.h>:

c23-fallback.h
```
563 #ifdef __has_include
564 # if __has_include(<stdckdint.h>)
565 #  include <stdckdint.h>
566 # endif
567 #endif
```

Our fallback header uses this throughout but always ensures that each test for a specific file is protected by a test for the feature itself. In general, this feature is already present in many pre-C23 compilers, so it should not be much of a problem.

This header also unconditionally includes a bunch of C library headers (see the following discussion) and augments them with C23 features if possible. You should not include them yourself so we are sure of what we get. This also concerns the new <stdckdint.h> C23 header <stdckdint.h> as seen previously.

Other headers may also be included conditionally to emulate missing features (namely, `<threads.h>` and `<stdatomic.h>`). `<threads.h>` `<stdatomic.h>`

When emulating the new features, there may be spurious warnings that some attributes are misplaced or ignored. This obviously means that they are not completely taken into account and that the analysis they are expected to provide is not yet fully implemented by your compiler. If you are flooded by these, you might switch some of the diagnostics off by defining the macro `C23_FALLBACK_SILENT`, for example, by providing the command line argument `-DC23_FALLBACK_SILENT` to the compiler. Don't forget that this header is only meant as a transitional feature; don't expect it to work forever.

Takeaway A #1 *Only use the header `c23-fallback.h` transitionally, until your platform fully supports C23.*

Takeaway A #2 *Header `c23-fallback.h` only emulates some C23 features with restricted capabilities.*

So it may be a good idea to use this header to get started, but once your compiler(s) fully support C23, you should remove this crutch.

There are some more features provided by this header, in particular, feature tests. First, to test for the new **`__VA_OPT__`** feature, use

c23-fallback.h

`__has_va_opt`:

A preprocessor test for the **`__VA_OPT__`** feature.

Compilers before C23 might not implement **`__VA_OPT__`**.

Returns: This macro should always evaluate to `0` or `1`.

```
            default: (B))

/* clang has a set of feature tests that are quite comfortable
   ,

   __has_feature, to test if a specific feature is implemented

   __has_extension, to test if a specific feature is
    implemented

   __is_identifier, to test if a word is an identifier or a
    keyword.
```

And then, for a whole bunch of other minor features that are provided as extensions to C23:

`iscomplex`
`isimaginary`
`isdecimalfloating`
`iscomplex`
`isstandardrealfloating`
`isstandardfloating`
`isfloating`
`iscompatible`
`is_potentially_negative`
`is_const_target`
`is_volatile_target`
`is_const`
`is_volatile`
`is_null_pointer_constant`
`is_zero_ice`
`isinteger`
`issigned`
`isunsigned`

```
isice                    is_pointer
isvla                    is_array
isxwide                  is_fla
is_pointer_nvla          is_void_pointer
is_pointer_vla
```

If you are interested in these, please look into the source of the fallback header to see what they do and how they are implemented.

B. C Compilers

As of this writing (March 2024) `gcc` and `clang` in their latest versions implement most of the new language features of C23, but unfortunately, some notable features are still missing: for Clang 18, the **`constexpr`** storage class specifier, and for Clang 18 and GCC 14, `[[unsequenced]]` and `[[reproducible]]` attributes.

Using less recent compilers also works for large parts, but the less recent, the more problems you will have, particularly if you need to use 128 bit types (see the following discussion). It seems that starting from GCC 10 and Clang 14, the C23 support is reasonable.

The code used in this book should compile when using the fallback header as indicated. In any case, it is always better to use the latest compiler release supported by your platform. Not only will it better implement the C standard as we need it for C23 support, it will also better optimize and take advantage of modern hardware.

Takeaway B #1 *Use the most recent compiler release.*

B.1. Attributes. For the the `[[unsequenced]]` and `[[reproducible]]` attributes, both compilers implement extensions `__gnu__::__const__` and `__gnu__::__pure__` that come close to these features. They can be used either with the attribute extension that `gcc` had used previously (`__attribute__((__const__))` and `__attribute__((__pure__))`) or with the new C23 syntax with a `gnu::` prefix. We do not provide an in-place fallback to them because their syntactic placement is different than for the C standard attributes. These apply to types, whereas the extensions apply to declarations. Nevertheless, we provide `c23_unsequenced` and `c23_reproducible` as shortcuts for these extensions.

B.2. Missing `#embed`. Unfortunately, as of March 2024, the **`#embed`** feature does not seem to have arrived in the compilers that we have at our disposal. If you want to experiment already with this feature, there is a project that emulates it that can be used relatively easily:

`https://sentido-labs.com/en/library/#cedro`.

The `Makefile` that comes with the example `code` directory shows examples of how to use this program to provide the functionality of **`#embed`**.

B.3. Missing `constexpr`. For the lack of **`constexpr`** in `clang` for integer types, there is indeed a replacement based on a much wider definition of integer constant expressions. Namely, `clang` accepts names of **`const`**-qualified objects with compile-time initializers also as integer constant expressions. So, for a large range of applications, basically cheating for this particular compiler by doing the following is almost sufficient:

c23-fallback.h

```
250 #if __is_identifier(constexpr)
251 # ifndef C23_FALLBACK_SILENT
252 #  warning "constexpr keyword is not supported, emulating as
        static const"
253 # endif
254 # define constexpr static const
255 #endif
```

Here, `__is_identifier` is a comfortable `clang` extension that tells us whether its argument (here, **`constexpr`**) is an identifier (the result is 1) or a keyword (the result is 0). So, whenever `clang` will be ready for this and returns 0—probably somewhere in release 18 or 19—this code is ignored automatically. We also give code that emulates

`__is_identifier` for other compilers. Again, if you are interested, you can look up the code in the fallback header.

B.4. Missing 128-bit integer support. `gcc` has partial support for 128-bit integer types, even on platforms with 64-bit hardware types, for a long time. Before C23, it had not been possible to integrate these into the standard-supported integer types because they are wider than the once-chosen `[u]intmax_t` types.

C23 makes the following types accessible as `[u]int128_t` types if all prerequisites for these types are met:

`<stdint.h>`
`<inttypes.h>`

- Macros and types in `<stdint.h>`
- Macros in `<inttypes.h>`
- Support for `"%w"` `"%wf"` length specifiers for **`printf`** and **`scanf`** functions

If the latter two are missing, this cannot be fixed with a fallback header, obviously, but we try to provide the types and macros of the first set as far as this is possible. For library support, see the following discussion.

There is one particular caution, though, that you should be aware of.

Takeaway B.4 #1 *The support for `[u]int128_t` is disabled for clang versions before clang-18.*

This is because the Clang support before that has two serious incompatibilities that make them dangerous to mix with code that has been compiled by GCC. These incompatibilities are as follows:

- On some platforms, the alignment of the `[u]int128_t` types is different from the ABI of the platform.
- On some platforms, parameter passing of `[u]int128_t` types may pass one half in a hardware register and the other on the stack.

So, if you want or have to use 128-bit integer types, move at least to Clang 18; you should not regret it.

C. C Libraries

Not surprisingly, C library implementations lack most of the support for C23 for now. In the following, we discuss some of the additions that come with C23. At the end, section C.7, we will report on a project for Linux systems that you can use temporarily until your C implementation provides full support for the library parts of C23.

C.1. Functions borrowed from POSIX or similar systems. The following functions are harmonized with POSIX:

- **strftime**
- **gmtime_r**
- **localtime_r**
- **memccpy**
- **strdup**
- **strndup**

So, if you are on such a system (such as Linux or macOS) or program in a compatible environment, you should already have these functions. Another candidate for a function you will most likely already find on such systems is **timegm**.

C.2. Improved UTF-8 support. C23 provides the new functions **mbrtoc8** and **c8rtomb** that have a similar role as, for example, **mbrtoc32** and **c32rtomb**, only that they use UTF-8 instead of UTF-32. If you are among the more experienced readers, implementing these interfaces would, in fact, be a very good exercise. UTF-8 is a really nice and sophisticated coding scheme that would teach you a lot, and correctly implementing these functions has the right level of challenge in terms of programming and in terms of understanding an international standard.

C.3. Bit utilities. The new header `<stdbit.h>` provides a lot of interfaces for bit operations on unsigned integer types (see 8.2). In particular, most have type-generic versions that are relatively easy to use and interface. `<stdbit.h>`

Some of these or similar functions are, in fact, present as built-ins on major compiler implementations, so our fallback header provides them where that seems easily possible. Those provided by the fallback are the type-generic interfaces.

A complete implementation is already available with `glibc-2.39`.

C.4. Checked integer arithmetic. The C23 header `<stdckdint.h>` provides three type-generic macros for checked arithmetic (see 8.2). Again, many compilers already provide equivalent tools so the fallback header makes them available where we know about this. `<stdckdint.h>`

C.5. Formatted IO. The **printf** and **scanf** families of functions gain support for `w` and `wf` width specifiers and for `b` and `B` binary number formats.

A complete implementation supporting standard integer types is already available with `glibc-2.39`.

The support for extended types—in particular, 128-bit integer types— is yet failing. Not only is there no support in the library to in- or output these types, but also the compilers issue confusing warnings when they are presented with format strings with `"%w128"` or `"%wf128"` specifiers. So, even if you use a C library that properly supports the types (see C.7), you might get drowned in warnings about the formats.

C.6. Mathematical functions. C23 has a lot of new functions in `<math.h>`, some of which are probably only of interest to a smaller community. Complete implementations of all of these will probably take their time to arrive in distributions. `<math.h>`

The CORE-MATH project (https://core-math.gitlabpages.inria.fr/) provides some of the missing features—in particular, the "pi" functions, which are trigonometric functions and their inverse that compute their arguments and return values in half-revolutions in the interval $[-1, +1]$ instead of $[-\pi, +\pi]$. For overviews of this project, see Sibidanov et al. [2022] and Gladman et al. [2024].

C.7. A reference implementation for musl libc. For Linux users, the code directory also has a script file `build-musl`. The brave among you may use it to compile your own version of `musl` libc with most of the support for C23. At its start, it contains a brief explanation and information on where to download the sources for this upgraded version of `musl`.

This patch-set in particular adds full support for the types `[u]int128_t` on architectures that have GCC's `__int128` extension (including alignment and calling conventions). So, if you are keen to use these as C23 foresees them, this is for you.

Note, though, that these patches probably have not been fully tested and may still have bugs. Once they are properly reviewed, most of these patches will hopefully be integrated into mainstream `musl`.

Takeaways

Bibliography

Douglas Adams. The hitchhiker's guide to the galaxy. audiocassette from the double LP adaptation, 1986. ISBN 0-671-62964-6.

C17. *Programming languages - C*. Number ISO/IEC 9899. ISO, fourth edition, 2018. URL https://www.iso.org/standard/74528.html.

C23. *Programming languages - C*. Number ISO/IEC 9899. ISO, fifth edition, 2024. URL https://www.iso.org/standard/82075.html.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2 edition, 2001.

Edsger W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Commun. ACM*, 11(3):147–148, March 1968. ISSN 0001-0782. doi: 10.1145/362929.362947. URL http://doi.acm.org/10.1145/362929.362947.

Martin Gardner. Mathematical Games – The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123, October 1970.

Brian Gladman, Vincenzo Innocente, John Mather, and Paul Zimmermann. Accuracy of Mathematical Functions in Single, Double, Double Extended, and Quadruple Precision. working paper or preprint, February 2024. URL https://inria.hal.science/hal-03141101.

Jens Gustedt. Improve type generic programming, January 2022. URL http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2890.pdf.

ISO/IEC/IEEE 60559, editor. *Information technology – Microprocessor Systems – Floating-Point arithmetic*, volume 60559:2011. ISO, 2011. URL https://www.iso.org/standard/57469.html.

Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

Donald E. Knuth. Structured programming with go to statements. In *Computing Surveys*, volume 6. 1974.

Donald E. Knuth. *The Art of Computer Programming. Volume 1: Fundamental Algorithms*. Addison-Wesley, 3rd edition, 1997.

Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

T. Nishizeki, K. Takamizawa, and N. Saito. Computational complexities of obtaining programs with minimum number of GO TO statements from flow charts. *Trans. Inst. Elect. Commun. Eng. Japan*, 60(3):259–260, 1977.

Carlos O'Donell and Martin Sebor. Updated field experience with Annex K – bounds checking interfaces, September 2015. URL http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1969.htm.

Philippe Pébay. Formulas for robust, one-pass parallel computation of covariances and arbitrary-order statistical moments. Technical Report SAND2008-6212, SANDIA, 2008. URL http://prod.sandia.gov/techlib/access-control.cgi/2008/086212.pdf.

POSIX. *IEEE/Open Group Standard for Information technology – Portable Operating Systems Interface (POSIX®) Base Specifications*, volume 1003.1-2024. The Open Group, 2024. Issue 8.

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.

Alexei Sibidanov, Paul Zimmermann, and Stéphane Glondu. The CORE-MATH Project. In *ARITH 2022 - 29th IEEE Symposium on Computer Arithmetic*, pages 26–34, virtual, France, September 2022. IEEE. doi: 10.1109/ARITH54963.2022.00014. URL https://inria.hal.science/hal-03721525.

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.

Mikkel Thorup. Structured programs have small tree-width and good register allocation. *Information and Computation*, 142:318–332, 1995.

Linus Torvalds et al. Linux kernel coding style, 1996. URL https://www.kernel.org/doc/Documentation/process/coding-style.rst. evolved mildly over the years.

Unicode, editor. *The Unicode Standard*. The Unicode Consortium, Mountain View, CA, USA, 10.0.0 edition, 2017. URL https://unicode.org/versions/Unicode10.0.0/.

John von Neumann. First draft of a report on the EDVAC, 1945. internal document of the ENIAC project.

B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.

Index