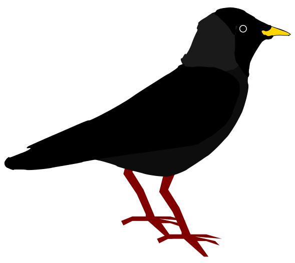


Modern C

Jens Gustedt

INRIA, France

ICube, Strasbourg, France

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

This is a preliminary version of this book compiled on June 12, 2014.
It contains feature complete versions of Levels 0 and 1.

You might find a more up to date version at
`http://icube-icps.unistra.fr/index.php/File:ModernC.pdf` (inline)
`http://icube-icps.unistra.fr/img_auth.php/d/db/ModernC.pdf` (download)

PRELIMINARIES.
It has been a long time since the programming language C is around; the main references for it was traditionally be the book written by its creators Kernighan and Ritchie [1978]. Since that time C has seen a fabulous distribution, programs and systems written in C are all around us, in personal computers, phones, cameras, set-top boxes, refrigerators, cars, mainframes, satellites, basically in any modern device that has a programmable interface.

In contrast to that ubiquitous presence, good knowledge of and about C seems to be much more scarcely distributed. Even confirmed C programmers often seem to be stuck in some self-inflicted ignorance about the modern evolution of the C language. One of the reasons for this is probably that C is an "easy to learn" language: even without much programming knowledge C lets us quickly write or just copy some code that will, often only seemingly, do the task that we want it to do. So somehow C is missing to offer an incentive to those working with it to climb to higher levels of knowledge.

This book is intended to change that general attitude. It is organized in chapters called "Levels" that summarize levels of familiarity with the C language and programming in general. Some features of the language are treated in parts on the different levels according to their difficulty. Most prominently pointers are only partially introduced at Level 1 and then fully treated at Level 2. As a consequence, we have a lot of forward references to stave off the impatient reader.

As the title of this book indicates, today's C is not exactly the same as the one that was originally designed by its creators, usually referred to as K&R C. In particular it has undergone an important normalization and extension process that is now driven by ISO, the International Standards Organization. This has lead to three major publications of C standards in the years 1989, 1999 and 2011, commonly referred to as C89, C99 and C11. The C standards committee puts a lot of efforts into guaranteeing backwards compatibility such that code written for one language, C89 say, should compile to a semantically equivalent executable with a compiler that implements a newer version. Unfortunately, this backwards compatibility also has the unwanted side effect that projects that could much benefit from new features miss the opportunity to update.

In this book we will mainly refer to C11, see JTC1/SC22/WG14 [2011], but at the time of this writing there are not even much compilers around that implement this standard completely. If you want to compile the examples of this book, you'd need at least a compiler that implements most of C99. For the Δ that C11 put on top of C99, using an emulation such as my macro package P99 might suffice, you may find it at `http://p99.gforge.inria.fr/`.

Programming has become a very important cultural and economical activity and C is just in the middle of it. As all human activities progress in C is driven by many factors, company or individual interest, politics, beauty, logic, luck, ignorance, selfishness, Ego, sectarianism, ... (add your primary motive, here). Thus the development of C has not been and cannot be ideal. It has flaws and artifacts that can only be understood from the historic and societal context.

An important context in which C developed was the early appearance of its sister language C++. One common misconception is that C++ had evolved from C by adding its particular features. Whereas this is historically correct (C++ evolved from an very early C) it is not much relevant, today. Fact is, that C and C++ separated from a common ancestor almost 30 years ago, and that they evolved separately since then. But this evolution of both languages has not taken place in isolation, both have integrated concepts of the other over the years. Some new features, such as the recent addition of atomics and threads have been designed in close relationship between the two committees.

Nevertheless, many differences prevail and generally all that is said in this book is about C and not C++. Most code examples that are given shouldn't even compile with a C++ compiler.

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

Contents

LEVEL 0

Encounter

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

1. Getting started

In this section I will try to introduce you to one simple program, that is supposed to contain already many of the constructs of the C language. If you already have experience in programming at a first glance you may find parts of it a needless repetition. If in contrast you don't have such an experience, you might feel deluged under a stream of new terms and concepts.

In either case be patient. For the ones that already know, there are many chances that you finally don't know exactly what we will be talking about; there are subtle difference between programming languages and even C is not always understood the same. For the ones that don't know (yet!), be assured that after these roughly ten first pages your understanding will have increased a lot, and you should be able to imagine much better what programming might represent.

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

Rule B *Don't Panik.*

It's not worth it. There are a lot of cross references, links, side information present in the text. There is an Index on page 120 follow them if you have a question. Or just take a break.

1.1. Imperative programming.

To get ourselves started and see what we are talking about consider our first program in Listing 1:

You probably see that this a sort of language, some weird words "**main**", "**`include`**", "**`for`**" etc laid out and colored quite bizarrely and mixed with a lot of weird characters, numbers, and other text "*Doing some work*" that looks like real English phrases. It is supposed to provide a link between us, the human programmers, and a machine, the computer, to tell him what to do. Give him "orders".

Rule 0.1.1.1 *C is an imperative programming language.*

In this book, we will not only encounter the programming language C, but also vocabulary from an English dialect, C *jargon*, the language that will help us *to talk about C*. Unfortunately, it will not be possible to immediately explain all the terms. But I will do, later, and all these terms are indexed such that you can easily cheat, and ***jump***^{C} to more explanatory text, at your own risk.

LISTING 1. A first example of a C programm

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

As you already guessed from this first example, such a C program has different components that form some mangled layers. Let's start to try to understand it from inside out.

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

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

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

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

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

- A funny looking text (the blue part), a so-called ***string literal***^{C}, that serves as ***format***^{C} for the output. Spread in the text are three markers (***format specifiers***^{C}), that mark the positions where in the output there will be numbers. These markers start with a `"%"` character. Also this format contains some mysterious ***escape characters***^{C} that start with a backslash, namely `"\t"` and `"\n"`.
- After a comma character comes the word "`i`". Whatever this "`i`" is, it will be printed at the occurrence of the first format, `"%zu"`.
- Another comma separates the next argument "`A[i]`". Whatever that is, it will be printed at the occurrence of the second format, the first `"%g"`.
- Last, again separated by comma, appears "`A[i]*A[i]`", corresponding to the last `"%g"`.

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

1.2. Compiling and running. As such the program text that we have listed above is not understood by your computer.

There is a special program, called a *compiler*, that translates the C text into something that can be understood by the machine, so-called ***binary code***^{C} or ***executable***^{C}. How that translated program looks like you shouldn't want to know at first. Just be happy that you found a tool that creates that format for you.

Rule 0.1.2.1 *C is a compiled programming language.*

The name of the compiler and its command line arguments depend a lot on the ***platform***^{C} on which you will be running your program. This for a simple reason, the target binary code is ***platform dependent***^{C}, that is its form and details depend on the computer on which you want to run it; a PC has different needs than a phone, your fridge doesn't talk the same as your settop box. In fact, that's one of the goals

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

It is the job of the compiler to make that happen and to ensure that our little program above, once translated for the appropriate platform, should run within your PC, your phone, your settop box and maybe even in your fridge.

That said there are good chances that a program named `c99` might be present on your PC and that this is in fact a C compiler. You could try to compile the example program as

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

The compiler should do that without complaining, and thereafter you should find an executable file `getting-started` in your directory.[Exs 1] In the above line

- `c99` is the compiler program.
- `-Wall` tells it to warn us about anything that it finds noticeable.
- `-lm` tells it to add mathematical functions if necessary, we will need that later.
- `-o getting-started` requests to store the ***compiler output***^{C} in a file named `getting-started`.
- `getting-started.c` names the ***source file***^{C}, namely the file that contains the C code that we have written. Observe the `.c` extension at the end of the file name, refering to the C programming language.

Now in a next step we can ***execute***^{C} our newly created ***executable***^{C}

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

and you should see exactly the same output as I have given you above. That's what portable should mean to you, wherever you run that program its ***behavior***^{C} is the same.

If you are not lucky and the compilation above didn't work, you'd have to look up in your documentation how to achieve that. The names of compilers may vary. Some alternatives that might do the trick look as

Terminal
```
> clang -Wall -lm -o getting-started getting-started.c
> gcc -std=c99 -Wall -lm -o getting-started getting-started.c
> icc -std=c99 -Wall -lm -o getting-started getting-started.c
```

and some of these even if they kind of work, might not compile the program without complaining.[Exs 2]

Now look at the program in Listing 2.

If you run your compiler on that one, it should give you some ***diagnostic***^{C}, something similar to this

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

So here we had a lot of long "warnings" lines but at the end the compiler produced an executable. Unfortunately, the output of running the program is different. So we have to be careful, we have been warned.

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

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

[Exs 2] Start a writing a textual report about your tests with this book. Note the command that worked for you.

LISTING 2. An example of a C programm with flaws

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

Terminal

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

but that is good. Its ***diagnostic output***^{C} is much more informative. In particular it gave us a hint that it expected us to have a line such as Line 3 of Listing 1 to specify where function **printf** comes from. And, at the end it doesn't produce an executable because it handles the problem in Line 22 fatal. Consider this to be a feature.

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

2. The principal structure of a program

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

Of the different aspects of a C program we have to distinguish two very different categories, syntactical aspects (how do we write up the program) and semantic aspects (what do we specify?). In the following three subsections we will introduce the syntactical aspects ("Grammar") and three different semantic aspects, namely declarative parts (what are the objects we are talking about?), definitions of objects (where are the objects?) and statements (what is this supposed to do?).

2.1. Grammar.

From a distance, we see that a C program is composed of different types of text elements that are assembled in some sort of grammar. These elements are:

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

punctuations^C**:** There are several punctuation concepts that C uses to structure the program text.

- There are five sorts of parenthesis: `{ ... }`, `( ... )`, `[ ... ]`, `/* ... */` and `< ... >`. Parenthesis *group* certain parts of the program together and should always come in pairs. Fortunately, the `< ... >` parenthesis are rare in C, and only used as in the example on the same line of text. The other four, are not limited to a text line, their contents might span several lines as we have seen for **printf**, above.
- There are two different separators or terminators, comma and semicolon. E.g in the **printf** discussion we have seen that the comma *separated* the four arguments to that function, in line 12 we see that a comma can just *terminate* the elements of a list.

comments^C**:** The construct `/* ... */` that we already have seen as parenthesis above changes everything that is put inside to a *comment*, see Line 5. Such a comment is otherwise ignored by the compiler. It is the perfect place to explain and document your code. Such an "in place" documentation can (and should) improve the readability and comprehensibility of your code a lot. Another form of comment are so-called C++-comments as in Line 15 that are introduced by `//`. Such a comment extends to the end of the line in which it is found.

literals^C**:** Our program contains several items that refer to fixed values that are part of the program: `0`, `1`, `3`, `4`, `5`, `9.0`, `2.9`, `3.E+25`, `.00007`, and `"element␣%zu␣is␣%g,␣\tits␣square␣is␣%g\n"`.

identifiers^C**:** These are "names" that we (or the C standard) gives to certain entities of the program. Here we have: `A`, `i`, **main**, **printf**, **size_t**, and **EXIT_SUCCESS**. Identifiers can play different roles in a program. Amongst others they may refer to:

- ***data objects***^{C} (such as `A` and `i`), these are also often referred to as ***variables***^{C}
- ***type***^{C} aliases, **size_t**, that specify the "sort" of a new object, here of `i`. (Observe the trailing `_t` in the name.)
- functions (**main** and **printf**),
- constants (**EXIT_SUCCESS**).

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

operators^C**:** Off the numerous C operators our program only uses a few:
- = for ***initialization***^{C} and ***assignement***^{C},
- < for comparison,
- ++ to increment a variable,
- $\star$ to perform the multiplication of two values.

2.2. Declarations. Declarations have to do with the ***identifiers***^{C} that we encountered above, as a general rule

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

In that identifiers are different from ***keywords***^{C}; these are predefined by the language, and mustn't be declared or redefined.

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

Above, we already mentioned the declaration of the **main** function. All three declarations, in isolation as "declarations only", look as the following:

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

These three follow a pattern. Each has an identifier (**main**, `A` or `i`) that is declared, that is where we specify certain properties that are associated with that identifier.

- `i` is of type **size_t**.
- **main** is additionally followed by parenthesis, `( ... )`, and thus declares a function of type `int`.
- `A` is followed by brackets, `[ ... ]`, and thus declares an array, here consisting of `5` items of type `double`.

So declarations are all introduced by a ***type***^{C}, here `int`, `double` and **size_t**. We will see later what that represents. For the moment it is sufficient that this specifies that all three identifiers when used in the context of a statement will be some sort of "numbers".

For the other three identifiers, **printf**, **size_t** and **EXIT_SUCCESS**, we don't see any declaration. In fact they are pre-declared identifiers, but as we have seen in the Listing 2 the knowledge of these identifiers doesn't come out of nowhere. We have to tell the compiler from 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 declaration of these identifiers could be something like

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

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

but actually you don't want to know all of this precisely. This information is normally hidden from you in these ***include files***^{C} or ***header files***^{C}. If you need to know the semantics of these, you better not look them up in the corresponding files, they tend to be barely readable. Search in the documentation that comes with your platform. For the brave, I always advise to have a look into the current C standard, this is where all of these come from. For the less courageous the following commands may help:

```
Terminal
> apropos printf
> man printf
> man 3 printf
```

Declarations may be repeated, if they specify exactly the same

Rule 0.2.2.2 *Identifiers may have several consistent declarations.*

Another property of declarations is that they might only be valid (***visible***^{C}) in some part of the program, not everywhere. A ***scope***^{C} is such a part of the program the validity of an identifier.

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

In our example we have declarations in different scopes.

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

In a slight abuse of terminology, the first two types of scope are called ***block scope***^{C}. The later for **main** is called ***file scope***^{C}, often such identifiers in file scope are referred to as *globals*.

2.3. Definitions. Generally, declarations only tell the kind of an identifier, not what the concrete value of an identifier is, nor where the object to which it might refer can be found. Everything only springs into life with a ***definition***^{C}.

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

We will later see that things are a little bit more complicated in real life, but here for the introductory part let's make a simplification

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

Technically the initializations are the additions to the declarations that give the object its initial value. E.g

```
size_t i = 0;
```

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

A is a bit more complex

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

this initializes the 5 items in A to the values 9.0, 2.9, 0.0, 0.00007 and 3.0E+25, in that order. The form of an initializer we see here is called ***designated***^{C}: a pair of brackets with an integer *designates* which item of the array is initialized with the corresponding value. E.g [4] = 3.E+25 sets the last item of the array A to 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.

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

For a function we have a definition (and not only a declaration) if it contains the braces { ... } for the code of the function.

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

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

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

Rule 0.2.3.4 *Objects must have exactly one definition.*

This rule concerns data objects as well as function objects.

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

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

  return EXIT_SUCCESS;
```

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

2.4.1. *Iteration.* The `for` statement is to execute the **printf** repeatedly, it is the simplest form of ***iteration***^{C} that C has to offer. It has four different parts: The `( ... )` part is divided in three by semicolon and the fourth is the `{ ... }` block that follows the `for ( ... )`.

(1) The declaration, definition and initialization of ***loop variable***^{C} `i` that we already have discussed above.
(2) A termination condition, `i < 5`, for the `for` iteration.
(3) Another statement, `++i`, to update `i` after each iteration. In this case to increase `i` by `1` each time.
(4) A ***depending***^{C} block, that in this case surrounds the **printf** call which we have already seen.

All of what this says is to perform the part in the block 5 times, for values of `i` set to `0`, `1`, `2`, `3`, and `4`. There is more than one way to do this in C, but this here is the easiest, cleanest and best tool for the task.

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

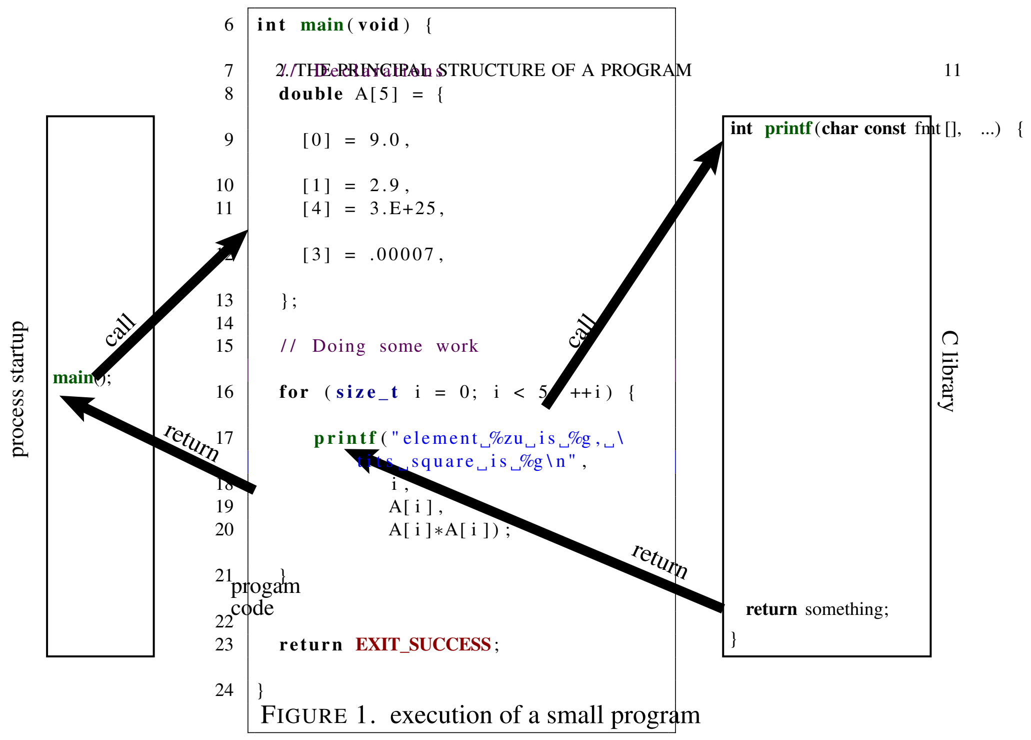
Even `for` could be written much differently from what we have seen here. Often people place the definition of the loop variable somewhere before the `for` or even reuse the same variable for several loops. Don't do that.

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

2.4.2. *Function return.* The last statement in **main** is a `return`. It tells the function, so **main**, to *return* to the statement from where it was called. Here, since **main** has `int` in its declaration, a `return` *must* send back a value of that type `int` to the calling statement, namely **EXIT_SUCCESS**.

Inside the function **printf**, that we don't see but only call, there must be a similar `return` statement. At the point where we call the function in Line 17, execution of the statements in **main** is temporarily suspended, execution continues in the **printf** function until a `return` is encountered, there. After the return from **printf** execution of the statements in **main** resumes.

In Figure 1 we have a schematic view about the execution of our little program. First a process startup routine (on the left) that is provided by our platform calls the user provided function **main** (middle). That in turn calls **printf**, a function of the ***C library***^{C}, on the right. Once a `return` is encountered there, control returns back to **main**, and on `return` there to the startup routine. The later, from a programmers point of view, is the end of the program as we know it.



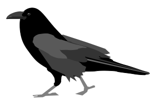
```
int main(void) {
  // Declarations
  double A[5] = {
    [0] = 9.0,
    [1] = 2.9,
    [4] = 3.E+25,
    [3] = .00007,
  };

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

  return EXIT_SUCCESS;
}
```

```
main();
```

```
int printf(char const fmt[], ...) {
  return something;
}
```

FIGURE 1. execution of a small program

LEVEL 1



Acquaintance

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

Having worked through this you should be able to write short code for everyday's needs, nothing too sophisticated, but useful and portable. All of this comes with some restriction. In parts, C is a permissive language, any programmer is allowed to shoot herself in the foot or other body parts, if she prefers. So in this chapter we have to see, that we don't hand out the guns, that the key to the gun safe is securely out of your reach, and that we mark the location of the gun safe with big and visible exclamation marks.

The main tool for the carnage in C are so-called ***casts***^{C}, so these are simply and plainly not introduced at this level. But there are many other pitfalls that are less obvious to avoid. We will approach some of them in a way that might look unfamiliar to some, in particular those that have learned some C basics from documents or teachers that have been stuck in the last millennium:

- We will emphasize mostly on the ***unsigned***^{C} versions of integer types.
- We will introduce pointers in steps: first, in disguise as parameters to functions (4.1.4) for their state (being valid or not, 4.2) and then, only very late (2, Lev. 2), for their whole potential.
- Instead, we will emphasize on the use of arrays.

Warning to experienced C programmers. For some of you that already have experience with programming in C[1] this may need some getting-used-to. Here are some of the things that may provoke allergic reactions. If you happen to break out in spots when you read some code here, try to take a deep breath and let it go.

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

```
char* name;
```

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

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

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

[1]If you are not an experience programmer you should just skip this part.

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

```
size_t strlen(char const string[static 1]);
int main(int argc, char* argv[argc+1]);
int atexit(void function(void));
```

The first stresses the fact that **strlen** must receive a valid pointer and will at least access the first element of `string`. The second summarizes the fact that **main** receives an array of pointers to **`char`**: the program name, `argc−1` program arguments and one null pointer that terminates the array.

We define variables as close to their first use as possible. Lack of variable initialization, especially for pointers, is one of the major pitfall for beginners with C. Therefore we simply avoid to separate a declaration of a variable from the first assignment to it: the real tool that is foreseen by C is a veritable definition, a declaration together with an initialization. This gives a name to a value, and introduces this name at the first place where can (and should) talk about it.

This is particularly convenient for **`for`**-loops: we only use the declarative version of it because an iterator variable of one loop is semantically a different object from the one in another loop.

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 it.
- The code inside is indented by one level.
- The terminating `}` starts a new line on the same level as the statement that introduced the block.
- Block statements that have a continuation after the `}` continue on the same line.

Examples:

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

1. Everything is about control

In our introductory example we have seen two different constructs that allowed us to control the flow of a program execution: functions and the **`for`**-iteration. Functions are a way to transfer control unconditionally. The call transfers control unconditionally *to* the function and a **`return`**-statement unconditionally transfers it *back* to the caller. We will come back to functions in Section 5.

The **`for`** statement is different in that it has a controlling condition (`i < 5` in the example) that regulates if and when the depending statement (`{ printf(...) }`) is executed. C has five such *control statements*: **`if`**, **`for`**, **`do`**, **`while`** and **`switch`**. These will be discussed throughout this section.

In additon, it also has two conditional expressions: the ***ternary operator***^{C} noted with something like "`cond ? A : B`", and type generic expressions, noted with the keyword **`_Generic`**. We will visit these in Sections 2.4 and 5, Lev. 3, respectively.

1.1. Conditional execution. The first construct that we will look at is introduced by the keyword **`if`**. It looks as the following:

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

Here we compare `i` against the value `25`, and only if it is larger than that `j` is set to the value `i - 25`. In that example `i > 25` is called the ***controlling expression***^{C}, and the part in `{ ... }` is called the ***depending block***^{C}.

So this form of an **`if`** statement is syntactically quite similar to the **`for`** statement that we already have encountered. Only that it is a bit simpler, the part inside the parenthesis isn't split up and as a whole it serves to determine what happens to the depending statement or block.

There is a more general form:

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

It has a second depending statement or block that is chosen if the controlling condition hasn't been fulfilled. Syntactically, this is done by introducing another keyword **`else`** that separates the two statements or blocks.

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

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

The possibilities for the controlling expression "`condition`" are numerous. They can range from simple comparisons as in this example to very complex nested expressions. We will present all the primitives that can be used in Section 2.3.2.

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

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

Here program text that chooses to proceed with **printf** (or not), is a simple `i`: a numerical value by itself is also interpreted as a condition. The result is that the text will only be printed in case `i` is not `0`.[Exs 2]

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

Rule 1.1.1.1 *The value `0` represents* **false**.

Rule 1.1.1.2 *Any value different from `0` represents* **true**.

Try to stick to these two rules as long as possible and don't write tautologies such as

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

Even worse would be something like

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

or even

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

#include <stdbool.h>

Here **bool**, included by `stdbool.h`, is the proper data type for truth values, and **false** and **true** are the proper values for that type, namely **false** is just another name for `0` and **true** for `1`. We will see more on that later in Section 3.5.4. For now we will just use **false** and **true** if we want to emphasize that a value is interpreted as a condition.

Also, notice the use of operators == and != that test for egality and inegality, respectively.

As the example shows such redundancy quickly becomes unreadable. Much easier to read, perhaps after some period of adjustment, is

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

That is taking a truth value directly as the condition. Generally:

Rule 1.1.1.3 *Don't compare to `0`,* **false** *or* **true**.

Doing otherwise obfuscates your code, and shows that you might have to revise one of the basic concepts of the C language:

[Exs 2] Add the **if** `(i)` condition to the program and compare the output to the previous.

Rule 1.1.1.4 *All scalar has a truth to it.*

This holds for all so-called ***scalar***^{C} types. These are first of all the numerical types such as **size_t**, **bool** or **int** that we already encountered, but also ***pointer***^{C} types, that we will introduce much later in Section 4.2.

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

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

So this statement is actually quite genereric. Usually "`expression-or-declaration`" is used to state an initial value for the iteration domain. "`condition`" tests if the iteration is finished (or not), "`expression`" updates the iteration variable and "`statement-or-block`" is performed in each iteration. Some advice

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

Let's see some more examples:

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

The first iteration counts `i` down from `10` to `1`, including. The condition is again just the evaluation of the variable `i`, no redundant test against value `0` is required. The second declares two variables `i` and `stop`. As before `i` is the loop variable, `stop` is use as final value in the condition. The third, also mysteriously counts down from `9` to `0`.[Exs 3]

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

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

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

The following example shows a typical use of the first:

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

...
double const a = 34.0;
double x = 0.5;
```

[Exs 3] Try to figure out why the iteration actually stops with the value `0`.

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

It simply iterates as long as given condition is fulfilled. The **do** loop is very similar, only that it investigates the condition *after* the depending block:

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

As for the **for** statement, for **do** and **while** it is advisable to use the `{ ... }` block variants. Still there is a subtle syntactical difference between the two, **do**–**while** always needs a semicolon `;` to terminate the statement. This is a feature.

All three iteration statements become even more flexible with **break** and **continue** statements. A **break** statement just stops the actual iteration without re-evaluating the termination condition:

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

Here this allows to separate the computation of the product `a*x`, the evaluation of the stop condition and the update of `x`. By that, now the condition of the **while** becomes trivial. There is a special dialect with the **for** statement that can be used for such unlimited iterations:

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

The **continue** statement is used less. As **break** it stops the current iteration, but then re-evaluates the condition.

#include <tgmath.h>

In the examples above we also have seen a use of the standard macro **fabs**, that comes with the `tgmath.h` header[4]. It calculates the absolute value of a **double**. For your pleasure, Listing 1.1 shows a more complete example of a program. Here the use of **fabs** is replaced by explicit comparisons.

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

Terminal

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

#include <stdlib.h>

To process the numbers on the command line the program uses another library function **strtod** from `stdlib.h`.[Exs 5][Exs 6][Exs 7]

[4]"tgmath" stands for *type generic mathematical functions*.

LISTING 1.1. A program to compute inverses of numbers

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

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

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

1.3. Multiple selection. The last control statement is called ***selection***^{C} or **`switch`** statement. It is mainly used when cascades of **`if`**-**`else`** constructs would be too tedious:

```c
  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");
  }
```

that is, if we have a choice that is more complex than a **false**−**true** decision and that can have several outcomes.

[Exs 5] Analyse Listing 1.1 by adding **printf** calls for intermediate values of `x`.

[Exs 6] Describe the use of the parameters `argc` and `argv` in Listing 1.1.

[Exs 7] Print out the mysterious values `eps1m01` and observe the output when change them slightly.

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

#include <stdio.h>

Here we select one of the **puts** calls according to the value of the `arg` variable. As **printf**, the function **puts** is provided by `stdio.h`. It just outputs a line with the string that is passed as an argument. We provide specific cases for characters `'e'`, `'t'`, `'d'`, `'c'` and a fall back case labeled **`default`** if `arg` doesn't match any of these.[Exs 8]

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

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

and also the semantics of it are quite straight forward: the **`case`** and **`default`** labels serve as jump targets. According to the value of the `expression`, control just continues at the statement that is labeled accordingly.

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

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

Because there is no **`break`** statement, once we have jumped into the block, the execution continues with all subsequent **puts** statements. For example the output for `count` with value `3` would be a triangle with three lines.

Terminal

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

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

Rule 1.1.3.1 **`case`** *values must be compile time integer expressions.*

And also it is more dangerous, since we might miss initializations of variables:

Rule 1.1.3.2 **`case`** *labels must not jump beyond a variable declaration.*

[Exs 8] Test the above **`switch`** statement in a program. Leave out some of the **`break`** statements.

2. Expressing computations

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

In this whole section, the values and objects on which we will do these computations will be mostly of type **size_t**, that we already met above. Such values correspond to "sizes", so they are numbers that cannot be negative. Their range of possible values starts at `0`. What we would very much like to represent are just the non-negative integers, in Mathematics often denoted as $\mathbb{N}$ or $\mathbb{N}_0$, "natural" numbers. Unfortunately computers are finite so we can't do that directly, but we can do something close. There is some big upper limit **SIZE_MAX** and we have

Rule 1.2.0.3 *The type* **size_t** *represents values in the range* `[0,` **SIZE_MAX**`]`.

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

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

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

#include `<stdint.h>`

This concept of "numbers that cannot be negative" for which **size_t** is just an example, are called ***unsigned integer types***^{C}. The funny symbols and combinations like + or != are called ***operators***^{C}, the parts to which they are applied are called ***operands***^{C}, so in something like "`a + b`", "+" is the operator and "`a`" and "`b`" are operands.

For an overview over C operators see the tables in the appendix; Table 2 for operators on values, Table 3 on objects and Table 4 on types.

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

2.1.1. *+, − and ∗.* Arithmetic operators +, − and ∗ mostly work as we would expect by computing the sum, the difference and the product of two values.

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

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

In addition, operators + and − also have unary variants. `-b` just gives the negative value of `b` and `+a` does not much more than to provide the value of `a`. The following would give `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 − is well defined. In fact, one of the miraculous properties of **size_t** is that +−∗ arithmetic always works where it can, that is provided the mathematical result falls into the range `[0,` **SIZE_MAX**`]` the value of the expression will be exactly that value.

Rule 1.2.1.1 *Unsigned arithmetic is always well defined.*

Rule 1.2.1.2 *If it is representable, operations* +, − *and* ⋆ *on* **size_t** *provide the mathematically correct result.*

In case that we have a result that is not representable, we speak of ***overflow***^{C}.

2.1.2. *Division and remainder.* The operators `/` and `%` are a bit more complicated, because they correspond to integer division and remainder operation. You are perhaps not as used to them, as to the other three arithmetic operators. Operators `/` and `%` really come in pair: if we have `z = a / b` the remainder `a % b` could be computed as `a - r*b`:

Rule 1.2.1.3 *For unsigned values we have* `a == (a/b)*b + (a%b)`.

The easiest example for the `%` operator are the hours of the day, let's say in 12 hour counting: 6 hours after 8 o'clock is 2 o'clock. Most people should be able to compute clocks in 12 hour or 24 hour counting. Such clock computation corresponds to computation `a % 12`, in our example `(8 + 6)% 12`.[Exs 9] Another example in daily life for `%` is computation with minutes in the hour, namely computation of the form `a % 60`.

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

Rule 1.2.1.4 *The result of unsigned* `/` *and* `%` *is smaller than the operands.*

And thus

Rule 1.2.1.5 *Unsigned* `/` *and* `%` *can't overflow.*

There is only one exceptional value that is not allowed for these two operations: `0`.

Rule 1.2.1.6 *If the second operand is not* `0`*, unsigned* `/` *and* `%` *are well defined.*

The `%` operator can also be used to explain additive and multiplicative arithmetic on unsigned types a bit better. If, such an operation overflows, the result is reduced as if the `%` would be would be used.

Rule 1.2.1.7 *Arithmetic on* **size_t** *implicitly does computation* `%(`**SIZE_MAX**`+1)`.

Rule 1.2.1.8 *In case of overflow, unsigned arithmetic wraps around.*

This "wrapping around" is actually the magic that makes the − operators work for unsigned types. E.g if the value `-1` interpreted as a **size_t** is just **SIZE_MAX** and so adding `-1` to a value `a`, just evaluates to `a +` **SIZE_MAX** which wraps around to `a +` **SIZE_MAX** `- (`**SIZE_MAX**`+1)= a - 1`.

[Exs 9] Implement some computations in 24 hour clock, e.g 3 hours after ten, 8 hours after twenty.

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

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

```
an_object @= some_value;
```

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

```
an_object = (an_object @ some_value);
```

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

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

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

Rule 1.2.2.1 *Operators must have all their characters directly attached to each other.*

We already have seen two other operators that modify objects, namely the ***increment operator***^{C} `++` and the ***decrement operator***^{C} `--`:

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

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

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

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

Rule 1.2.2.2 *Side effects in value expressions are evil.*

Rule 1.2.2.3 *Never modify more than one object in a statement.*

For the increment and decrement operators there are even two other forms, namely ***postfix increment***^{C} and ***postfix decrement***^{C}. They differ from the one that we have seen in the result when they are used inside a larger expression. But since you will nicely obey to Rule 1.2.2.2, you will not be tempted to use them.

2.3. Boolean context. Several operators yield a value `0` or `1` according if some condition is verified or not, see Table 2. They can be grouped in two categories, comparisons and logical evaluation.

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

Rule 1.2.3.1 *Comparison operators return the values* **false** *or* **true**.

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

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

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

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

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

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

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

#include `<iso646.h>`

2.3.2. *Logic.* Logic operators operate on values that are already supposed to represent values **false** or **true**. If they are not, the rules that we described for conditional execution with Rules 1.1.1.1 and 1.1.1.2 apply first. The operator `!` (**not**) logically negates its operand, operator `&&` (**and**) is logical and, operator `||` (**or**) is logical or. The results of these operators are summarized in the following table:

TABLE 1. Logical operators

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

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

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

Similar as for the comparison operators we have

Rule 1.2.3.2 *Logic operators return the values* **false** *or* **true**.

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

```
double largeA[N] = { 0 };
...
```

```
/*  fill largeA somehow */

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

Here the expression `!!largeA[i]` applies the `!` operator twice and thus just ensures that `largeA[i]` is evaluated as a truth value according to the general Rule 1.1.1.4. As a result, the array elements `isset[0]` and `isset[1]` will hold the number of values that are equal to `0.0` and unequal, respectively.

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

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

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

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

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

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

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

#include `<tgmath.h>`

```
#include <tgmath.h>

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

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

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

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

Complex arithmetic and the tools used for it need the header `complex.h` which is indirectly included by `tgmath.h`. They will be introduced later in Section 3.5.6.

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

Rule 1.2.5.1 *`&&`, `||`, `?:` and `,` evaluate their first operand first.*

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

Rule 1.2.5.2 *Don't use the `,` operator.*

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

Rule 1.2.5.3 *Most operators don't sequence their operands.*

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

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

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

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

Rule 1.2.5.4 *Function calls don't sequence their argument expressions.*

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

Rule 1.2.5.5 *Functions that are called inside expressions should not have side effects.*

3. Basic values and data

We will now change the angle of view from the way "how things are to be done" (statements and expressions) to the things on which C programs operate, ***values***^{C} and ***data***^{C}.

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

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

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

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

Rule 1.3.0.6 *All values are numbers or translate to such.*

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

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

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

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

3.0.2. *Types.* An important step in that direction is the concept of ***types***^{C}. A type is an additional property that C associates with values. Up to now we already have seen several such types, most prominently **size_t**, but also **`double`** or **bool**.

Rule 1.3.0.7 *All value has a type that is statically determined.*

Rule 1.3.0.8 *A value's type determines its possible operations.*

Rule 1.3.0.9 *A value's type determines the results of all operations.*

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

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

E.g the operations on the type **size_t** can be entirely determined when inspecting the value of **SIZE_MAX** in addition to the operands. We call the model to represent values of a given type on a given platform the ***binary representation***^{C} of the type.

Rule 1.3.0.10 *A type's binary representation determines the results of all operations.*

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

Rule 1.3.0.11 *A type's binary representation is observable.*

This binary representation is still a model and so an *abstract representation* in the sense that it doesn't completely determine how values are stored in the memory of a computer or on a disk or other persistent storage device. That representation would be the *object representation*. In contrast to the binary representation, the object representation usually is of not much concern to us, as long as we don't want to hack together values of objects in main memory or have to communicate between computers that have a different platform model. Much later, in Section 2.9, Lev. 2, we will see that we may even observe the object representation *if* such an object is stored in memory *and* we know its address.

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

Rule 1.3.0.12 *Programs execute **as if** they were following the abstract state machine.*

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

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

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

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

or

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

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

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

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

Rule 1.3.0.13 *Type determines optimization opportunities.*

3.1. Basic types. C has a series of basic types and some means of constructing ***derived types***^{C} from them that we will describe later in Section 4.

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

We will start with the first level specification of such types. As we already discussed above in Rule 1.3.0.6, all basic values in C are numbers, but there are numbers of different kind. As a principal distinction we have two different classes of numbers, with two subclasses, each, namely ***unsigned integers***^{C}, ***signed integers***^{C}, ***real floating point numbers***^{C} and ***complex floating point numbers***^{C}

All these classes contain several types. They differ according to their ***precision***^{C}, which determines the valid range of values that are allowed for a particular type. Table 2 contains an overview of the 18 base types. As you can see from that table there are some types which we can't directly use for arithmetic, so-called ***narrow types***^{C}. As a rule of thumb we get

Rule 1.3.1.1 *Each of the 4 classes of base types has 3 distinct unpromoted types.*

Other than many people believe, the C standard doesn't even prescribe the precision of these 12 types, it only constrains them. They depend on a lot of factors that are ***implementation dependent***^{C}. Thus, to chose the "best" type for a given purpose in a portable way could be a tedious task, if we wouldn't get help from the compiler implementation.

TABLE 2. Base types according to the four main type classes. Types with a grey background don't allow for arithmetic, they are *promoted* before doing arithmetic. Type **`char`** is special since it can be unsigned or signed, depending on the platform. *All* types in the table are considered to be distinct types, even if they have the same class and precision.

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

Remember that unsigned types are the most convenient types, since they are the only types that have an arithmetic that is defined consistently with mathematical properties, namely modulo operation. They can't raise signals on overflow and can be optimized best. They are described in more detail in Section 3.5.1.

Rule 1.3.1.2 *Use* **size_t** *for sizes, cardinalities or ordinal numbers.*

Rule 1.3.1.3 *Use* **`unsigned`** *for small quantities that can't be negative.*

If your program really needs values that may both be positive and negative but don't have fractions, use a signed type, see Section 3.5.5.

Rule 1.3.1.4 *Use* **`signed`** *for small quantities that bear a sign.*

Rule 1.3.1.5 *Use* **ptrdiff_t** *for large differences that bear a sign.*

If you want to do fractional computation with values such as `0.5` or `3.77189E+89` use floating point types, see Section 3.5.6.

Rule 1.3.1.6 *Use* **`double`** *for floating point calculations.*

Rule 1.3.1.7 *Use* **`double`** **complex** *for complex calculations.*

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

TABLE 3. Semantic arithmetic types for specialized use cases

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

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

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

3.2. Specifying values. We have already seen several ways in which numerical constants, so-called ***literals***^{C} can be specified:

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

`077` ***octal integer constant***^{C}. This is specified by a sequence of digits, the first being `0` and the following between `0` and `7`, e.g `077` has the value 63. This type of specification has merely historical value and is rarely used nowadays.There is only one octal literal that is commonly used, namely `0` itself.

`0xFFFF` ***octal integer constant***^{C}. This is specified by a start of `0x` followed by a sequence of digits between `0`, ..., `9`, `a` ... `f`, e.g `0xbeaf` is value 48815. The `a .. f` and `x` can also be written in capitals, `0XBEAF`.

`1.7E-13` ***decimal floating point constants***^{C}. Quite familiar for the version that just has a decimal point. But there is also the "scientific" notation with an exponent. In the general form `mEe` is interpreted as $m \cdot 10^e$.

`0x1.7aP-13` ***hexadecimal floating point constants***^{C}. Usually used to describe floating point values in a form that will ease to specify values that have exact representations. The general form `0XhPe` is interpreted as $h \cdot 2^e$. Here h is specified as an hexadecimal fraction. The exponent e is still specified as a decimal number.

`'a'` ***integer character constant***^{C}. These are characters put into `'` apostrophs, such as `'a'` or `'?'`. These have values that are only implicitly fixed by the C standard. E.g `'a'` corresponds to the integer code for the character "a" of the Latin alphabet.

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

`"hello"` ***string literals***^{C}. They specify text, e.g. as we needed it for the **printf** and **puts** functions. Again, the "\" character is special as in character constants.

All but the last are numerical constants, they specify numbers. An important rule applies:

Rule 1.3.2.1 *Numerical literals are never negative.*

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

In view of Rule 1.3.0.7 we know that all literals must not only have a value but also a type. Don't mix up the fact of a constant having a positive value with its type, which can be **`signed`**.

Rule 1.3.2.2 *Decimal integer constants are signed.*

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

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

Rule 1.3.2.3 *A decimal integer constant has the first of the 3 signed types that fits it.*

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

Rule 1.3.2.4 *The same value can have different types.*

Deducing the type of an octal or hexadecimal constant is a bit more complicated. These can also be of an unsigned type if the value doesn't fit for a signed one. In our example above the hexadecimal constant `0x7FFF` has the value 32767 and thus type **`signed`**. Other than for the decimal constant, the constant `0x8000` (value 32768 written in hexadecimal) then is an **`unsigned`** and expression `-0x8000` again is **`unsigned`**.[Exs 11]

Rule 1.3.2.5 *Don't use octal or hexadecimal constants to express negative values.*

Or if we formulate it postively

Rule 1.3.2.6 *Use decimal constants to express negative values.*

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

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

You remember that value `0` is important. It is so important that it has a lot of equivalent spellings: `0`, `0x0` and `'\0'` are all the same value, a `0` of type **`signed int`**. `0` *hasn't* a decimal integer spelling: `0.0` *is* a decimal spelling for the value `0` but seen as a floating point value, namely with type **`double`**.

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

[Exs 11] Show that if in that case the maximum **`unsigned`** is $2^{16}-1$ that then `-0x8000` has value 32768, too.

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

TABLE 4. Examples for constants and their types, under the supposition that **`signed`** and **`unsigned`** have the commonly used representation with 32 bit.

constant x	value	type	value of $-x$
`2147483647`	+2147483647	**`signed`**	−2147483647
`2147483648`	+2147483648	**`signed long`**	−2147483648
`4294967295`	+4294967295	**`signed long`**	−4294967295
`0x7FFFFFFF`	+2147483647	**`signed`**	−2147483647
`0x80000000`	+2147483648	**`unsigned`**	+2147483648
`0xFFFFFFFF`	+4294967295	**`unsigned`**	+1
`1`	+1	**`signed`**	−1
`1U`	+1	**`unsigned`**	+4294967295

Rule 1.3.2.7 *Different literals can have the same value.*

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

Rule 1.3.2.8 *The effective value of a decimal floating point constant may be different from its literal value.*

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

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

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

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

Rule 1.3.2.9 *Literals have value, type and binary representation.*

3.3. Initializers. We already have seen (Section 2.3, Lev. 0) that the initializer is an important part of an object definition. Accessing uninitialized objects has undefined behavior, the easiest way out is to avoid that situation systematically:

Rule 1.3.3.1 *All variables should be initialized.*

There are only few exception to that rule, VLA, see Section 4.1.3, that don't allow for an initializer, or code that must be highly optimized. The latter mainly occurs in situations that use pointers, so this is not yet relevant to us. For most code that we are able to write so far, a modern compiler will be able to trace the origin of a value to the last assignment or the initialization. Superfluous assignments will simply be optimized out.

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

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

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

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

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

Rule 1.3.3.2 *Use designated initializers for all aggregate data types.*

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

Rule 1.3.3.3 `{0}` *is a valid initializer for all types that are not VLA.*

There are several things, that ensure that this works. First, if we omit the designation (the `.fieldname` for **`struct`**, see Section 4.3 or `[n]` for arrays, see Section 4.1) initialization is just done in ***declaration order***^{C}, that is the `0` in the default initializer designates the very first field that is declared, and all other fields then are initialized per default to `0` as well. Then, the `{}` form of initializers for scalars ensures that `{ 0 }` is also valid for these.

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

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

```
char const*const animal[3] = {
  "raven",
  "magpie",
  "jay",
};
char const*const pronoun[3] = {
  "we",
  "you",
  "they",
};
char const*const ordinal[3] = {
```

[13]The exception are variable length arrays, see Section 4.1.3.

```
  "first",
  "second",
  "third",
};
...
for (unsigned i = 0; i < 3; ++i)
    printf("Corvid␣%u␣is␣the␣%s\n", i, animal[i]);
...
for (unsigned i = 0; i < 3; ++i)
    printf("%s␣pural␣pronoun␣is␣%s\n", ordinal[i], pronoun[
       i]);
```

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

Rule 1.3.4.1 *All constants with particular meaning must be named.*

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

Rule 1.3.4.2 *All constants with different meaning must be distinguished.*

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

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

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

Rule 1.3.4.3 *Enumeration constants have either an explicit or positional value.*

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

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

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

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

Rule 1.3.4.4 *Enumeration constants are of type* **signed int**.

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

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

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

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

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

Rule 1.3.4.5 *An integer constant expression doesn't evaluate any object.*

So principally an ICE may consist of any operations with integer literals, enumeration constants, **_Alignof** and **offsetof** sub-expressions and eventually some **sizeof** sub-expressions.[14]

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

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

```
# define M_PI 3.14159265358979323846
```

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

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

With this trick we can declare textual replacement for constants of **unsigned**, **size_t** or **double**. In fact the implementation imposed bound of **size_t**, **SIZE_MAX**, is such defined, but also many of the other system features that we already have seen: **EXIT_SUCCESS**,

[14]We will handle the later two concepts in Sections 2.8, Lev. 2 and 2.6, Lev. 2.

false, **true**, **not_eq**, **bool**, **complex** . . . Here, in this book such C standard macros are all printed in **dark red**.

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

Rule 1.3.4.6 *Macro names are in all caps.*

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

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

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

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

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

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

```
for (unsigned i = 0; i < corvid_num; ++i)
    printf("Corvid␣%u␣is␣the␣%s\n", i, CORVID_NAME[i]);
```

This form of macro has some pitfalls

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

Rule 1.3.4.7 *Don't hide a terminating semicolon inside a macro.*

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

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

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

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

Rule 1.3.4.8 **I** *is reserved for the imaginary unit.*

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

3.5. Binary represention.

Rule 1.3.5.1 *The same value may have different binary representations.*

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

Rule 1.3.5.2 *Unsigned arithmetic wraps nicely.*

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

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

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

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

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

Rule 1.3.5.3 *The maximum value of any integer type is of the form* $2^p - 1$.

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

Rule 1.3.5.4 *Arithmetic on an unsigned integer type is determined by its precision.*

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

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

bit op	value	hex	b_{15} ... b_0	set op	set
`V`	65535	`0xFFFF`	1111111111111111		$\{0,1,2,3,4,5,6,7,8,9,10, 11,12,13,14,15\}$
`A`	240	`0x00F0`	0000000011110000		$\{4,5,6,7\}$
`~A`	65295	`0xFF0F`	1111111100001111	$V \setminus A$	$\{0,1,2,3,8,9,10, 11,12,13,14,15\}$
`-A`	65296	`0xFF10`	1111111100010000		$\{4,8,9,10, 11,12,13,14,15\}$
`B`	287	`0x011F`	0000000100011111		$\{0,1,2,3,4,8\}$
`A|B`	511	`0x01FF`	0000000111111111	$A \cup B$	$\{0,1,2,3,4,5,6,7,8\}$
`A&B`	16	`0x0010`	0000000000010000	$A \cap B$	$\{4\}$
`A^B`	495	`0x01EF`	0000000111101111	$A \Delta B$	$\{0,1,2,3,5,6,7,8\}$

For the result of these operations the total size of the base set, and thus the precision p is not needed. As for the arithmetic operators, there are corresponding assignment operators `&=`, `|=`, and `^=`, respectively.[Exs 15][Exs 16][Exs 17][Exs 18]

There is yet another operator that operates on the bits of the value, the complement operator `~`. The complement `~A` would have value 65295 and would correspond to the set $\{0, 1, 2, 3, 8, 9, 10, 11, 12, 13, 14, 15\}$. This bit complement always depends on the precision p of the type.[Exs 19][Exs 20]

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

`#include <iso646.h>`

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

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

int main(void) {
  unsigned flock = FLOCK_EMPTY;

  ...

  if (something) flock |= FLOCK_JAY;
```

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

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

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

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

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

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

```

  ...

  if (flock&FLOCK_CHOUGH)
    do_something_chough_specific(flock);

}
```

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

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

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

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

Rule 1.3.5.5 *The second operand of a shift operation must be less than the precision.*

There is an analogous right shift operation >> that shifts the binary representation towards the less significant bits. Analogously this corresponds to an integer division by a power of two. Bits in positions less or equal to the shift value are omitted for the result. Observe that for this operation, the precision of the type isn't important.[Exs 21]

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

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

```
# define FLOCK_MAGPIE   (1U << magpie)
# define FLOCK_RAVEN  (1U << raven)
# define FLOCK_JAY      (1U << jay)
# define FLOCK_CHOUGH   (1U << chough)
# define FLOCK_EMPTY     0U
# define FLOCK_FULL    ((1U << corvid_num)-1)
```

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

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

`#include` `<stdbool.h>`

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

Treating **bool** as an unsigned type is a certain stretch of the concept. Assignment to a variable of that type doesn't follow the Modulus Rule 1.2.1.7, but a the special Rule 1.1.1.1.

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

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

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

Signed and unsigned types come in pairs, with the notable two exceptions from Table 2 **char** and **bool**. The binary representation of the signed type is constrained by corresponding unsigned type:

Rule 1.3.5.6 *Positive values are represented independently from signedness.*

Or stated otherwise, a positive value with a signed type has the same representation as in the corresponding unsigned type. That is the reason why we were able to express Rule 1.3.5.3 for *all* integer types. These also have a precision, p that determines the maximum value of the type.

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

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

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

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

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

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

```c
  for (unsigned i = 1; i; ++i) do_something();
  for (  signed i = 1; i; ++i) do_something();
```

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

For the second, all looks similar. But because here the behavior of overflow is undefined the compiler is allowed to *pretend* that it will never happen. Since it also knows that the value at start is positive it may assume that `i`, as long as the program has defined behavior, is never negative or 0. The *as-if* Rule 1.3.0.12 allows it to optimize the second loop to

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

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

Rule 1.3.5.7 *Once the abstract state machine reaches an undefined state no further assumption about the continuation of the execution can be made.*

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

Rule 1.3.5.8 *It is your responsibility to avoid undefined behavior of all operations.*

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

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

Rule 1.3.5.9 *Signed arithmetic may trap badly.*

One of the things that might already overflow for signed types is negation. We have seen above that **INT_MAX** has all bits but the sign bit set to 1. **INT_MIN** has then the "next" representation, namely the sign bit set to 1 and all other values set to 0. The corresponding value is not -**INT_MAX**.[Exs 22]

We then have

Rule 1.3.5.10 *In twos' complement representation* **INT_MIN** < -**INT_MAX**.

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

Rule 1.3.5.11 *Negation may overflow for signed arithmetic.*

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

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

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

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

Rule 1.3.5.12 *Use unsigned types for bit operations.*

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

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

#include <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 digit 3 in decimal representation or such as the value of π which is "transcendent" and so has an endless expansion in any representation, and which even doesn't repeat in any way.

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

In a view that is a bit simplified[23] a floating point value is computed from the following values:

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

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

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

The values p, $emin$ and $emax$ are type dependent, and therefore not represented explicitly in each number. They can be obtained through macros such as **DBL_MANT_DIG** (for p, typically 53) **DBL_MIN_EXP** (e_{min}, -1021) and **DBL_MAX_EXP** (e_{max}, 1024).

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

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

[23]see Section 5, Lev. 2 for the complete view

which corresponds to the decimal value `-0.15625`. From that calculation we see also that floating point values are always representable as a fraction that has some power of two in the denominator.[Exs 24]

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

Rule 1.3.5.13 *Floating point operations are neither* associative, commutative *or* distributive.

So basically they loose all nice algebraic properties that we are used to when doing pure math. The problems that arise from that are particularly pronounced if we operate with values that have very different orders of magnitude.[Exs 25] E.g adding a very small floating point value x with an exponent that is less than $-p$ to a value $y > 1$ just returns y, again. As a consequence it is really difficult to assert without further investigation if two computations have had the "same" result. We are only able to tell that they are "close":

Rule 1.3.5.14 *Never compare floating point values for equality.*

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

#include `<tgmath.h>`

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

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

4. Aggregate data types

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

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

Of these four, pointers are by far the most involved concept, and we will delay the full discussion on them to Section 2, Lev. 2. Below, in Section 4.2, we only will present them as opaque data type, without even mentioning the real purpose they fulfill.

Unions also need a deeper understanding of C's memory model and are not of much use in every day's programmers life, so they are only introduce in Section 2.10, Lev. 2. Here, for this level, we will introduce ***aggregate data types***^{C}, data types that group together several data to form one unit.

4.1. Arrays.

4.1. Arrays. Arrays allow us to group objects of the same type into an encapsulating object. We only will see pointer types later (Section 2, Lev. 2) but many people come to C with that confuse arrays and pointers. And this is completely normal, arrays and pointers are closely related in C and to explain them we face a *hen and egg* problem: arrays *look like* pointers in many contexts, and pointers refer to array objects. We chose an order of introduction that is perhaps unusual, namely we start with arrays and try to stay with them as long as possible before introducing pointers. This may seem "wrong" for some of you, but remember that everything which stated here has to be taken with the *as-if* rule: we will first describe arrays in a way that will be consistent with C's assumptions about the abstract state machine.

We start with a very important rule:

Rule 1.4.1.1 *Arrays are not pointers.*

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

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

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

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

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

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

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

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

Initializers can use *designated initializers* to pick the specific position to which an initialization applies. Example code on pages 35 and 43 contains such initializers.

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

Rule 1.4.1.2 *An array in a condition evaluates to* **true**.

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

Rule 1.4.1.3 *There are array objects but no array values.*

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

Rule 1.4.1.4 *Arrays can't be compared.*

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

Rule 1.4.1.5 *Arrays can't be assigned to.*

From that table we see that there are only four operators left that work on arrays as object operator. We already know the operator `[]`[26]. The "array decay" operation, the address operator `&` and the **`sizeof`** operator will be introduced later.

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

Rule 1.4.1.6 *VLA can't have initializers.*

Rule 1.4.1.7 *VLA can't be declared outside functions.*

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

Rule 1.4.1.8 *The length of an FLA is determined by an ICE or an initializer.*

In the first case, their length is know at compile time through a integer constant expression, ICE, as we have introduced them in Section 3.4.1. There is no type restriction for the ICE, any integer type would do. The only restriction is

Rule 1.4.1.9 *An array length specification must be strictly positive.*

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

[26]The real C jargon story about arrays and `[]` is a bit more complicated. Let us apply the **as-if** Rule 1.3.0.12 to our explanation. All C program behaves *as if* the `[]` are directly applied to an array object.

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

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

All other array variable declarations lead to VLA.

Rule 1.4.1.10 *An array with a length not an integer constant expression is an VLA.*

The length of an array can be computed with the **sizeof** operator. That operator provides the "size" of any object[27] and so the length of an array can be calculate by a simple division.

Rule 1.4.1.11 *The length of an array* A *is* (**sizeof** A)/(**sizeof** A[0]).

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

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

Rule 1.4.1.12 *The innermost dimension of an array parameter to a function is lost.*

Rule 1.4.1.13 *Don't use the* **sizeof** *operator on array parameters to functions.*

Array parameter are even more bizarre, because of Rule 1.4.1.3 array parameters cannot be passed by value; there is no such thing as an array value in C. As a first approximation of what is happening for array parameters to functions we have:

Rule 1.4.1.14 *Array parameters behave* as-if *the array is* **passed by reference**^{C}.

Take the following as an example:

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

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

[27] later we will see what the unit of measure for such sizes is

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

Rule 1.4.1.15 *A string is a* `0`*-terminated array of* **`char`**.

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

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

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

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

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

because it is not `0`-terminated.

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

The wast majority of platforms nowadays uses the so-called ASCII[28] to encode characters in the type **`char`**. We don't have to know how the particular encoding works as long as we stay in the basic character set, everything is done in C and its standard library that this encoding is used transparently.

`#include` `<string.h>`

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

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

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

String functions:

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

[28] American code for information interchange

- **strcmp**(s0, s1) compares two arrays in the lexicographic ordering similar to **memcmp**, but may not take some language specialties into account. The comparison stops at the first 0-character that is encountered in either s0 or s1. Again, both parameters have to be 0-terminated.
- **strcoll** (s0, s1) compares two arrays in the lexicographic ordering respecting language specific environment settings. We will learn how to properly ensure to set this in Section 6.5.
- **strchr**(s, c) is similar to **memchr**, only that the string s must be 0-terminated.

Rule 1.4.1.16 *Using a string function with a non-string has undefined behavior.*

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

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

In other words, be careful, and make sure that all your strings really are strings. If your platform already supports this, use the functions with bounds checking that were introduce with C11. There are **strnlen_s** and **strcpy_s** that additionally deal with the maximum length of their string parameters.[Exs 29]

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

LISTING 1.2. copying a string

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

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

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

[Exs 29] Use **memchr** and **memcmp** to implement a bounds checking version of **strcmp**.

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

Besides the bizarre return type of **strcpy**, this looks reasonable. The parameter arrays are arrays of "unknown" length, so the empty `[]`. **strlen** is to return a "size" and **strcmp** a negative, 0 or positive value according to the sort order of the arguments.

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

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

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

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

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

Rule 1.4.2.1 *Pointers are opaque objects.*

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

Rule 1.4.2.2 *Pointers are valid, null or indeterminate.*

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

Rule 1.4.2.3 *Initialization or assignment with `0` makes a pointer null.*

Usually we refer to a pointer in the null state as ***null pointer***^{C}. Surprisingly, disposing of null pointers is really a feature.

Rule 1.4.2.4 *In logical expressions, pointers evaluate to* **false** *iff they are null.*

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

Rule 1.4.2.5 *Indeterminate pointers lead to undefined behavior.*

In view of Rule 1.3.5.7, we need to make sure that pointers never reach an intermediate state. Thus, if we can't ensure that a pointer is valid, we *must* at least ensure that it is set to null:

Rule 1.4.2.6 *Always initialize pointers.*

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

As a first example, let us revisit the corvids on page 35. There, we used a trick with an enumeration type to keep track of our interpretation of the individual elements of an array name. C structures allow for a more systematic approach by giving names to so-called fields in an aggregate:

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

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

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

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

```
typedef signed calArray[6];
```

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

`#include` `<time.h>`

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

```c
  int tm_isdst;// Daylight Saving Time flag
};
```

This **struct** has *named fields*, such as **tm_sec** for the seconds or **tm_year** for the year. Encoding a date, such as the date of this writing

Terminal

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

is simple:

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

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

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

```c
  printf("this␣year␣is␣%d,␣next␣year␣will␣be␣%d\n",
         today.tm_year, today.tm_year+1);
```

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

There are three other fields in **struct tm** that we didn't even mention in our initializer list, namely **tm_wday**, **tm_yday** and **tm_isdst**. Since we don't mention them, they are automatically set to `0`.

Rule 1.4.3.1 *Omitted **struct** initializers force the corresponding field to* `0`.

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

Rule 1.4.3.2 *A **struct** initializer must initialize at least one field.*

In fact, in Rule 1.3.3.3 we have already seen that there is a default initializer that works for all data types: `{0}`.

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

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

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

Rule 1.4.3.3 **`struct`** *parameters are passed by value.*

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

```c
  today = time_set_yday(today);
```

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

Rule 1.4.3.4 *Structures can be assigned with = but not compared with == or !=.*

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

`#include` `<time.h>`

Rule 1.4.3.5 *A structure layout is an important design decision.*

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

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

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

So here we see the opaque type **time_t** that we already know from Table 3 for the seconds, and a **`long`** for the nanoseconds. Again, the reasons for this choice are just

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

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

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

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

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

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

historic, nowadays the chosen types would perhaps be a bit different. To compute the difference between two **struct timespec** times, we can easily define a function:

```
double timespec_diff(struct timespec a, struct timespec b){
  double ret = difftime(a.tv_sec, b.tv_sec);
  ret += (a.tv_nsec - b.tv_nsec) * 1E-9;
  return ret;
}
```

Whereas the function **difftime** is part of the C standard, this functionality here is very simple and isn't based on platform specific implementation. So it can easily be implemented by anyone who needs it. A more complete example for **struct timespec** will be given later on Level 2 on page 99.

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

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

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

Rule 1.4.3.6 *All* **struct** *declarations in a nested declaration have the same scope of visibility.*

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

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

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

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

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

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

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

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

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

Rule 1.4.4.1 *Forward-declare a* **`struct`** *within a* **`typedef`** *using the same identifier as the tag name.*

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

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

```
typedef double vector[64];
typedef vector vecvec[16];
vecvec A;
typedef double matrix[16][64];
matrix B;
double C[16][64];
```

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

The C standard also uses **`typedef`** a lot internally. The semantic integer types such as **size_t** that we have seen in Section 3.1 are declared with this mechanism. Here the standard often uses names that terminate with "`_t`" for "**`typedef`**". This naming convention ensures that the introduction of such a name by an upgraded version of the standard will not conflict with existing code. So you shouldn't introduce such names yourself in your code.

Rule 1.4.4.2 *Identifier names terminating with* `_t` *are reserved.*

5. Functions

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

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

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

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

- **exit**, **_Exit**, **quick_exit** and **abort** terminate the program execution, see Section 6.6.
- **goto** transfers control within a function body, see Section 8.1, Lev. 2.
- **setjmp** and **longjmp** can be used to return unconditionally to a calling context, see Section 3.2, Lev. 3.

5.1. Simple functions.

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

All the functions that we have seen so far have a ***prototype***^{C}, i.e their declaration and definition included a parameter type-list. If this list was to be empty it was replaced by the keyword **void**. Then they also specified the type of the value that is to be returned from the particular function, or, again, **void** if there is none.

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

```c
extern double fbar(double x);

...
double fbar2 = fbar(2)/2;
```

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

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

Rule 1.5.1.1 *All functions must have prototypes.*

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

`#include <stdargs.h>`

We will see later (**??**, Lev. 0) how this works, but this feature is to be avoided in any case. Already from your experience with **printf** you can imagine why such an interface poses difficulties. You as a programmer of the calling code have to ensure consistency by providing the correct `"%XX"` format specifiers.

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

Rule 1.5.1.2 *Functions only have one entry but several* **`return`**.

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

Rule 1.5.1.3 *A function* **`return`** *must be consistent with its type.*

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

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

Rule 1.5.1.4 *Reaching the end of the* `{ }` *block of a function is equivalent to a* **`return`** *statement without expression.*

This implies

Rule 1.5.1.5 *Reaching the end of the* `{ }` *block of a function is only allowed for* **`void`** *functions.*

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

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

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

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

- On some embedded platforms where **main** is not expected to return to the runtime system the return type may be **`void`**.

- On many platforms a third parameter can give access to the "environment".

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

Rule 1.5.2.1 *Use* **EXIT_SUCCESS** *or* **EXIT_FAILURE** *as return values of* **main***.*

For **main** as an exception of Rule 1.5.2.2

Rule 1.5.2.2 *Reaching the end of the* `{}` *block of* **main** *is equivalent to a* **return EXIT_SUCCESS;***.*

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

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

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

Rule 1.5.2.3 *Calling* **exit** `(s)` *is equivalent evaluation of* **return** `s` *in* **main***.*

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

Rule 1.5.2.4 **exit** *never fails and never returns to its caller.*

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

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

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

Rule 1.5.2.5 *All commandline arguments are transferred as strings.*

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

Of the strings of `argv` two elements hold special values:

Rule 1.5.2.6 *Of the arguments to* **main***,* `argv[0]` *holds the name of the program invocation.*

There is no strict rule of what that "program name" should be, but usually this is just taken as the name of the program executable.

Rule 1.5.2.7 *Of the arguments to* **main***,* `argv[argc]` *is* `0`*.*

In the `argv` array, the last argument could always be identified by that property, but this feature isn't too useful: we have `argc` to process that array.

5.3. Recursion. An important feature of functions is encapsulation: local variables are only visible and alive until we leave the function, either via an explicit **`return`** or because execution falls out of the last enclosing brace of the function's block. Their identifiers ("names") don't conflict with other similar identifiers in other functions and once we leave the function all the mess that we leave behind is cleaned up.

Even better: whenever we call a function, even one that we have called before, a new set of local variables (including function parameters) is created and these are newly initialized. This holds also, if we newly call a function for which another call is still active in the hierarchy of calling functions. A function that directly or indirectly calls itself is called *recursive*, the concept itself is called *recursion*.

Recursive functions are crucial for understanding C functions: they demonstrate and use main features of the function call model and they are only fully functional with these features. As a first example we show an implementation of Euclid's algorithm to compute the *greatest common divisor*, gcd, of two numbers.

```
 5  size_t gcd2(size_t a, size_t b) {
 6    assert(a <= b);
 7    if (!a) return b;
 8    size_t rem = b % a;
 9    return gcd2(rem, a);
10  }
```

As you can see, this function is short and seemingly nice. But to understand how it works we need to understand well how functions work, and how we transform mathematical statements into algorithms.

Given two integers $a, b > 0$ the gcd is defined to be the greatest integer $c > 0$ that divides both a and b or as formulas:

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

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

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

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

That is the gcd doesn't change if we subtract the smaller one or if we replace the larger of the two by the modulus of the other. These formulas are used since antique Greek mathematics to compute the gcd. They are commonly attributed to Euclid (Εὐκλείδης, around 300 B.C) but may have been known even before him.

Our C function `gcd2` above uses Equation (3). First (Line 6) it checks if a precondition for the execution of this function is satisfied, namely if the first argument is less or equal to the second. It does this by using the macro **assert** from `assert.h`. This would abort the program with some informative message in case the function would be called with arguments that don't satisfy that condition, we will see more explanations of **assert** in Section 6.6.

`#include` `<assert.h>`

Rule 1.5.3.1 *Make all preconditions for a function explicit.*

Then, Line 7 checks if `a` is `0`, in which case it returns `b`. This is an important step in a recursive algorithm:

Rule 1.5.3.2 *In a recursive function, first check the termination condition.*

TABLE 5. Recursive call `gcd2(18, 30)`

call level 0						
`a = 18`						
`b = 30`						
`!a` $\Longrightarrow$ **false**						
`rem = 12`						
`gcd2(12, 18)`	$\Longrightarrow$					
		call level 1				
		`a = 12`				
		`b = 18`				
		`!a` $\Longrightarrow$ **false**				
		`rem = 6`				
		`gcd2(6, 12)`	$\Longrightarrow$			
				call level 2		
				`a = 6`		
				`b = 12`		
				`!a` $\Longrightarrow$ **false**		
				`rem = 0`		
				`gcd2(0, 6)`	$\Longrightarrow$	
						call level 3
						`a = 0`
						`b = 6`
						`!a` $\Longrightarrow$ **true**
					$\Longleftarrow$ 6	**`return`** `6`
			$\Longleftarrow$ 6	**`return`** `6`		
	$\Longleftarrow$ 6	**`return`** `6`				
`return` `6`						

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 better shouldn't try this.

Otherwise, we compute the remainder `rem` of `b` modulo `a` (Line 8) and then the function is called recursively with `rem` and `a` and the return value of that, is directly returned.

Table 5 shows an example for the different recursive calls that are issued from an initial call `gcd2(18, 30)`. Here, the recursion goes 4 levels deep. Each level implements its own copies of the variables `a`, `b` and `rem`.

For each recursive call, Rule 1.2.1.4, 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}:

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

Rule 1.5.3.3 *Ensure the preconditions of a recursive function in a wrapper function.*

This avoids that the precondition has to be checked at each recursive call: the **assert** macro is such that it can be disabled in the final “production” object file.

Another famous example for a recursive definition of an integer sequence are *Fibonnacci numbers*, a sequence of numbers that appears as early as 200 B.C in Indian texts. In modern terms the sequence can be defined as

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

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

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

The sequence of Fibonacci numbers is fast growing, its first elements are 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 377, 610, 987.

With the golden ratio

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

asymptotically we have

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

So the growth of F_n is exponential.

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

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

Here, again, we first check for the termination condition, namely if the argument to the call, `n`, is less then `3`. If it is the return value is `1`, otherwise we return the sum of calls with argument values `n-1` and `n-2`.

Table 6 shows an example of a call to `fig` with a small argument value. We see that this leads to 3 levels of “stacked” calls to the same function with different arguments. Because Equation (6) uses two different values of the sequence, the scheme of the recursive calls is much more involved than the one for `gcd2`. In particular, there are 3 so-called *leaf*

TABLE 6. Recursive call `fib(4)`

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

calls, calls to the function that fulfill the termination condition, and thus do by themselves not go into recursion. [Exs 30]

Implemented like that, the computation of the Fibonacci numbers is quite slow. [Exs 31] In fact it is easy to see that the recursive formula for the function itself also leads to an analogous formula for the execution time of the function:

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

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

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

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

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

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

with some other platform dependent constant C_2. So the execution time of `fib(n)` is exponential in `n`, which usually rules out such a function from being used in practice.

Rule 1.5.3.4 *Multiple recursion may lead to exponential computation times.*

[Exs 30] Show that a call `fib(n)` induces F_n leaf calls.

[Exs 31] Measure times for calls `fib(n)` with n set to different values. On POSIX systems you can use `/bin/time` to measure the run time of a program execution.

If we look at the nested calls in Table 6, we see that we have the same call `fib(2)`, twice, and thus all the effort to compute the value for `fib(2)` is repeated. The following function `fibCacheRec` avoids such repetitions. It receives an additional argument, `cache`, which is an array that holds all values that already have been computed.

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

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

By trading storage against computation time, the recursive calls only are effected if the value has not yet been computed. By that, the call `fibCache(i)`, has an execution time that is linear in n, namely

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

for some platform dependent parameter C_3.[Exs 32] Just by changing the algorithm that implements our sequence, we are able to reduce the execution time from exponential to linear! We didn't (and wouldn't) even discuss implementation details, nor did we perform concrete measurements of execution time: [Exs 33]

Rule 1.5.3.5 *A bad algorithm will never lead to a performing implementation.*

Rule 1.5.3.6 *Improving an algorithm can dramatically improve performance.*

[Exs 32] Show Equation (13).

[Exs 33] Measure times for calls `fibCache(n)` 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 an FLA instead a VLA.

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

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

Whether or not this is really faster, and how to prove that this version is still correct, is left as an exercise.[Exs 34] [Exs 35]

6. C Library functions

The functionality that the C standard provides is separated into two big parts. One is the proper C language, the other is the *C library*. We already have seen several functions that come with the C library, e.g **printf**, **puts** or **strtod**, so you should have a good idea what you may expect: basic tools that implement features that we need in every day's programming and for which we need clear interfaces and semantics to ensure portability.

On many platforms, the clear specification through an *application programmable interface*, *API*, also allows to separate the compiler implementation from the one of the library. E.g on Linux systems we have the choice between different compilers, most commonly `gcc` and `clang`, and different C library implementations, e.g the GNU C library (`glibc`), `dietlibc` or `musl`, and potentially any of the two choices can be used to produce an executable.

Roughly, library functions target one or two different purposes:

Platform abstraction layer. Functions that abstract from the specific properties and needs of the platform. These are functions that that need platform specific bits to implement basic operations such as IO, that could not be implemented without deep knowledge of the platform. E.g. **puts** has to have some concept of a "terminal output" and how to address that. Implementing these functionalities by herself would exceed the knowledge of most C programmers, because it needs operating system or even processor specific magic. Be glad that some people did that job for you.

Basic tools. Functions that implement a task (such as e.g **strtod**) that often occurs in programming in C for which it is important that the interface is fixed. These should be implemented relatively efficient, because they are used a lot, and they should be well tested and bug free such that we can rely safely on them. Implementing such functions should in principle be possible by any confirmed C programmer.[Exs 36]

[Exs 34] Measure times for calls `fib2(n)` with the same values as for `fib`.

[Exs 35] Use an iteration statement to transform `fib2rec` into a non-recursive function `fib2iter`.

[Exs 36] Write a function `my_strtod` that implements the functionality of **strtod** for decimal floating point constants.

TABLE 7. C library headers

`<assert.h>`	assert run time conditions	6.6
`<complex.h>`	complex numbers	5, Lev. 2
`<ctype.h>`	character classification and conversion	
`<errno.h>`	error codes	
`<fenv.h>`	floating-point environment.	
`<float.h>`	properties of floating point types	3.5
`<inttypes.h>`	exact width integer types	4, Lev. 2
`<iso646.h>`	alternative spellings for operators	2.1
`<limits.h>`	properties of integer types	3.0.3
`<locale.h>`	internationalization	
`<math.h>`	type specific mathematical functions	6.1
`<setjmp.h>`	non-local jumps	3.2, Lev. 3
`<signal.h>`	signal handling functions	
`<stdalign.h>`	alignment of objects	
`<stdarg.h>`	functions with varying number of arguments	
`<stdatomic.h>`	atomic operations	
`<stdbool.h>`	Booleans	1.1
`<stddef.h>`	basic types and macros	
`<stdint.h>`	exact width integer types	4, Lev. 2
`<stdio.h>`	input and output	6.2
`<stdlib.h>`	basic functions	
`<stdnoreturn.h>`	non-returning functions	5
`<string.h>`	string handling	6.3
`<tgmath.h>`	type generic mathematical functions	6.1
`<threads.h>`	threads and control structures	4, Lev. 3
`<time.h>`	time handling	6.4
`<uchar.h>`	Unicode characters	
`<wchar.h>`	wide string	
`<wctype.h>`	wide character classification and conversion	

A function as **printf** can be seen to target both purposes, it can effectively be separated in two, a formatting phase and an output phase. There is a function **snprintf** that provides the same formatting functionalities as **printf** but stores the result in a string. This string could then be printed with **puts** to have the same output as **printf** as a whole.

The C library has a lot of functions, far more than we can handle in this book. Here on this level, we will discuss those functions that are necessary for a basic programming with the elements of the language that we have seen so far. We will complete this on higher levels, as soon as we discuss a particular concept. Table 7 has an overview of the different standard header files.

Interfaces. Most interfaces of the C library are specified as functions, but implementations are free to chose to implement them as macros, were this is appropriate. Compared to those that we have seen in Section 3.4.2 this uses a second form of macros that are syntactically similar to functions, ***functionlike macros***^{C}.

```
#define putchar(A) putc(A, stdout)
```

As before, these are just textual replacements, and since the replacement text may contain a macro argument several times, it would be quite bad to pass any expression with

TABLE 8. Error return strategies for C library functions. Some functions may also indicate a specific error condition through the value of the macro **errno**.

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

side effects to such a macro-or-function. Fortunately, because of Rule 1.2.2.2 you don't do that, anyhow.

Some of the interfaces that we will see below will have arguments or return values that are pointers. We can't handle these completely, yet, but in most cases we can get away by passing in some "known" pointers or `0` for pointer arguments. Pointers as return values will only occur in situations where they can be interpreted as an error condition.

Error checking. C library functions usually indicate failure through a special return value. What value indicates the failure can be different and depends on the function itself. Generally you'd have to look up the specific convention in the manual page for the functions. Table 8 gives an rough overview of the different possibilities. There are three categories that apply: a special value that indicates an error, a special value that indicates success, and functions that return some sort of positive counter on success and a negative value on failure.

A typical error checking code in would look like the following

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

Here we see that **puts** falls into the category of functions that return a special value on error, **EOF**, "end-of-file". The function **perror** from `stdio.h` is then used provide an additional diagnostic that depends on the specific error; **exit** ends the program execution. Don't wipe failures under the carpet, in programming

`#include <stdio.h>`

Rule 1.6.0.7 *Failure is always an option.*

Rule 1.6.0.8 *Check the return value of library functions for errors.*

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

Rule 1.6.0.9 *Fail fast, fail early and fail often.*

C has one major state "variable" that tracks errors of C library functions, a dinosaur called **errno**. The function **perror** uses this state under the hood, to provide its diagnostic. If a function fails in a way that allows us to recover, we have to ensure that the error state also is reset, otherwise library functions or error checking might get confused.

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

Bounds-checking interfaces. Many of the functions in the C library are vulnerable to ***buffer overflow***^{C} if they are called with an inconsistent set of parameters. This lead (and still leads) to a lot of security bugs and exploits and is generally something that should be handled very carefully.

C11 addressed this sort of problems by deprecating or removing some functions from the standard and by adding an optional series of new interfaces that check consistency of the parameters at runtime. These are the so-called bounds-checking interfaces of *Annex K* of the standard. Other than for most other features, this doesn't come with its own header file but adds interfaces to others. Two macros regulate access to theses interfaces, __STDC_LIB_EXT1__ tells if this optional interfaces is supported, and __STDC_WANT_LIB_EXT1__ switches it on. The later must be set **before** any header files are included:

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

#include <stdio.h>

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

Annex K

Optional features such as these are marked as this paragraph, here. The bounds-checking functions usually use the suffix _s to the name of the library function they replace, such as printf_s for printf. So you should not use that suffix for code of your own.

Rule 1.6.0.10 *Identifier names terminating with _s are reserved.*

If such a function encounters an inconsistency, a ***runtime constraint violation***^{C}, it usually should end program execution after printing a diagnostic.

Platform preconditions. An important goal by programming with a standardized language such as C is portability. We should make as few assumptions about the execution platform as possible and leave it to the C compiler and library to fill out the gaps. Unfortunately this is not always possible, but if not, we should mark preconditions to our code as pronounced as possible.

Rule 1.6.0.11 *Missed preconditions for the execution platform must abort compilation.*

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

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

As you can see, such a conditional starts with the token sequence **# if** on a line and terminates with another line containing the sequence **# endif**. The **# error** directive in the middle is only executed if the condition (here !**__STDC_LIB_EXT1__**) is true. It aborts the compilation process with an error message. The conditions that we can place in such a construct are limited.[Exs 37]

Rule 1.6.0.12 *Only evaluate macros and integer literals in a preprocessor condition.*

As an extra feature in these conditions we have that identifiers that are unknown just evaluate to 0. So in the above example the expression is always valid, even if **__STDC_LIB_EXT1__** is completely unknown at that point.

Rule 1.6.0.13 *In preprocessor conditions unknown identifiers evaluate to* 0.

If we want to test a more sophisticated condition we have **_Static_assert** and **static_assert** from the header assert.h with a similar effect at our disposal. **#include** <assert.h>

```
#include <assert.h>
static_assert(sizeof(double) == sizeof(long double),
  "Extra␣precision␣needed␣for␣convergence.");
```

6.1. Mathematics. Mathematical *functions* come with the math.h header, but it is much simpler to use the type generic macros that come with tgmath.h. Basically for all functions this has a macro that dispatches an invocation such as **sin**(x) or **pow**(x, n) to the function that inspects the type of x for its argument and for which the return value then is of that same type. **#include** <math.h> **#include** <tgmath.h>

The type generic macros that are defined are **acos**, **acosh**, **asin**, **asinh**, **atan**, **atan2**, **atanh**, **carg**, **cbrt**, **ceil**, **cimag**, **conj**, **copysign**, **cos**, **cosh**, **cproj**, **creal**, **erf**, **erfc**, **exp**, **exp2**, **expm1**, **fabs**, **fdim**, **floor**, **fma**, **fmax**, **fmin**, **fmod**, **frexp**, **hypot**, **ilogb**, **ldexp**, **lgamma**, **llrint**, **llround**, **log**, **log10**, **log1p**, **log2**, **logb**, **lrint**, **lround**, **nearbyint**, **nextafter**, **nexttoward**, **pow**, **remainder**, **remquo**, **rint**, **round**, **scalbln**, **scalbn**, **sin**, **sinh**, **sqrt**, **tan**, **tanh**, **tgamma**, **trunc**, far too many as we can describe them in detail, here. Table 9 gives an overview over the functions that are provided.

Nowadays, the implementations of numerical functions should be of high quality, efficient and with well controlled numerical precision. Although any of these functions could be implemented by a programmer with sufficient numerical knowledge, you should not try to replace or circumvent these functions. Many of them are not just implemented as C functions but can use processor specific instructions. E.g processors may have fast approximations of **sqrt** or **sin** functions, or implement a *floating point multiply add*, **fma**, in one low level instruction. In particular, there are good chances that such low level instructions are used for all functions that inspect or modify floating point internals, such as **carg**, **creal**, **fabs**, **frexp**, **ldexp**, **llround**, **lround**, **nearbyint**, **rint**, **round**, **scalbn**, **trunc**. So replacing them or re-implementing them by handcrafted code is usually a bad idea.

6.2. Input, output and file manipulation. We have already seen some of the IO functions that come with the header file stdio.h, namely **puts** and **printf**. Where the second lets you format output in some convenient fashion, the first is more basic, it just outputs a string (its argument) and an end-of-line character. **#include** <stdio.h>

[Exs 37] Write a preprocessor condition that tests if **int** has two's complement sign representation.

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

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

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

```
int putchar(int c);
int puts(char const s[static 1]);
```

The type `int` as parameter for **putchar** is just a historical accident that shouldn't hurt you much. With this functions we could actually reimplement **puts** ourselves:

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

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

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

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

Here the ⋆ in the **FILE**⋆ type again indicates that this is a pointer type, so we couldn't go into details. The only thing that we need for now is Rule 1.4.2.4; a pointer can be tested if it is null, and so we will be able to test if a stream is valid or not.

More than the fact that this a pointer, the identifier **FILE** itself is a so-called ***opaque type***^{C}, for which don't know more than is provided by the functional interfaces that we will see in this section. The facts that it is implemented as a macro and the misuse of the name "FILE" for a "stream" is a reminder that this is one of the historic interfaces that predate standardization.

Rule 1.6.2.1 *Opaque types are specified through functional interfaces.*

Rule 1.6.2.2 *Don't rely on implementation details of opaque types.*

If we don't do anything special, there are two streams that are available for output to us: **stdout** and **stderr**. We already have used **stdout** implicitly, this is what **putchar** and **puts** use under the hood, and this stream is usually connected to the terminal. **stderr** is similar, it also is linked to the terminal by default, with perhaps slightly different properties. In any case these two are closely related. The purpose to have two of them is to be able to distinguish "usual" output (**stdout**) from "urgent" one (**stderr**).

We can rewrite the former functions in terms of the more general ones:

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

Observe that **fputs** differs from **puts** in that it doesn't append an end-of-line character to the string.

Rule 1.6.2.3 ***puts** and **fputs** differ in their end of line handling.*

6.2.2. *Files and streams.* If we want to write some output to real files, we have to attach the files to our program execution by means of the function **fopen**.

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

This can be used as simple as here:

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

This ***opens a file***^{C} called `"mylog.txt"` in the file system and provides access to it through the variable `logfile`. The mode argument `"a"` has the file opened for appending, that is the contents of the file is preserved, if it exists, and writing begins at the current end of that file.

There are multiple reasons why opening a file might not succeed, *e.g* the file system might be full or the process might not have the permission to write at the indicated place. In accordance with Rule 1.6.0.8 we check for such conditions and exit the program if such a condition is present.

As we have seen above, the function **perror** is used to give a diagnostic of the error that occurred. It is equivalent to something like the following.

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

This "some-diagnostic" might (but hasn't to) contain more information that helps the user of the program to deal with the error.

Annex K

There are also bounds-checking replacements **fopen_s** and **freopen_s** that ensure that the arguments that are passed are valid pointers. Here **errno_t** is a type that comes with `stdlib.h` and that encodes error returns. The **`restrict`** keyword that also newly appears only applies to pointer types and is out of our scope for the moment.

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

TABLE 10. Modes and modifiers for **fopen** and **freopen**. One of the first three must start the mode string, optionally followed by some of the other three. See Table 11 for all valid combinations.

mode	memo		file status after **fopen**
`'a'`	append	w	file unmodified, position at end
`'w'`	write	w	content of file wiped out, if any
`'r'`	read	r	file unmodified, position at start
modifier	memo		additional property
`'+'`	update	rw	open file for reading and writing
`'b'`	binary		view as binary file, otherwise text file
`'x'`	exclusive		create file for writing iff it does not yet exist

```
                    char const filename[restrict], char const
                        mode[restrict]
                    FILE* restrict stream);
```

There are different modes to open a file, `"a"` is only one of several possibilities. Table 10 contains an overview of the different characters that may appear in that string. We have three different base modes that regulate what happens to a pre-existing file, if any, and where the stream is positioned. In addition, we have three modifiers that can be appended to these. Table 11 has a complete list of the possible combinations.

From these tables you see that a stream can not only be opened for writing but also for reading; we will see below how that can be done. To know which of the base modes opens for reading or writing just use your common sense. For `'a'` and `'w'` a file that is positioned at its end can't be read since there is nothing there, thus these open for writing. For `'r'` a file contents that is preserved and that is positioned at the beginning should not be overwritten accidentally, so this is for reading.

The modifiers are used less commonly in everyday's coding. "Update" mode with `'+'` should be used carefully. Reading and writing at the same time is not so easy and needs some special care. For `'b'` we will discuss the difference between text and binary streams in some more detail in Section 6.2, Lev. 2.

There are three other principal interfaces to handle streams, **freopen**, **fclose** and **fflush**.

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

The primary uses for **freopen** and **fclose** are straight forward: **freopen** can associate a given stream to a different file and eventually change the mode. This is particular useful to associate the standard streams to a file. *E.g* our little program from above could be rewritten as

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

TABLE 11. Mode strings for **fopen** and **freopen**. These are the valid combinations of the characters in Table 10.

`"a"`		create empty text file if necessary, open for writing at end-of-file
`"w"`		create empty text file or wipe out content, open for writing
`"r"`		open existing text file for reading
`"a+"`		create empty text file if necessary, open for reading and writing at end-of-file
`"w+"`		create empty text file or wipe out content, open for reading and writing
`"r+"`		open existing text file for reading and writing at beginning of file
`"ab"`	`"rb"`	same as above but for binary file instead of text file
`"wb"`	`"a+b"`	
`"ab+"`	`"r+b"`	
`"rb+"`	`"w+b"`	
`"wb+"`		
`"wx"`	`"w+x"`	same as above but error if file exists prior to call
`"wbx"`	`"w+bx"`	
`"wb+x"`		

6.2.3. *Text IO.* Output to text streams is usually ***buffered***^{C}, that is to make more efficient use of its resources the IO system can delay the physical write of to a stream for some time. If we close the stream with **fclose** all buffers are guaranteed to be ***flushed***^{C} to where it is supposed to go. The function **fflush** is needed in places where we want to see an output immediately on the terminal or where don't want to close the file, yet, but where we want to ensure that all contents that we have written has properly reached its destination. Listing 1.4 shows an example that writes 10 dots to **stdout** with a delay of approximately one second between all writes.[Exs 38]

The most common form of IO buffering for text files in ***line buffering***^{C}. In that mode, output is only physically written if the end of a text line is encoutered. So usually text that is written with **puts** would appear immediately on the terminal, **fputs** would wait until it meets an `'\n'` in the output. Another interesting thing about text streams and files is that there is no one-to-one correspondence between characters that are written in the program and bytes that land on the console device or in the file.

Rule 1.6.2.4 *Text input and output converts data.*

This is because internal and external representation of text characters are not necessarily the same. Unfortunately there are still many different character encodings around, the C library is in charge of doing the conversions correctly, if it may. Most notoriously the end-of-line encoding in files is platform depending:

Rule 1.6.2.5 *There are three commonly used conversion to encode end-of-line.*

C here gives us a very suitable abstraction in using `'\n'` for this, regardless of the platform. Another modification that you should be aware of when doing text IO is that white space that precedes the end of line may be suppressed. Therefore presence of such ***trailing white space***^{C} such as blank or tabulator characters can not be relied upon and should be avoided:

[Exs 38] Observe the behavior of the program by running it with 0, 1 and 2 command line arguments.

LISTING 1.4. 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, 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);
}
```

Rule 1.6.2.6 *Text lines should not contain trailing white space.*

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

```
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.

6.2.4. *Formatted output.* We have already seen how we can use **printf** for formatted output. The function **fprintf** is very similar to that, only that it has an additional parameter that allows to specify the stream to which the output is written:

```
int printf(char const format[static 1], ...);
int fprintf(FILE* stream, char const format[static 1], ...)
    ;
```

The syntax with the three dots `...` indicates that these functions may receive an arbitrary number of items that are to be printed. An important constraint is that this number must correspond exactly to the `'%'` specifiers, otherwise the behavior is undefined:

Rule 1.6.2.7 *Parameters of* **printf** *must exactly correspond to the format specifiers.*

TABLE 12. Format specifications for **printf** and similar functions, with the general syntax `"%[FF][WW][.PP][LL]SS"`

`FF`	flags	special form of conversion
`WW`	field width	minimum width
`PP`	precision	
`LL`	modifier	select width of type
`SS`	specifier	select conversion

TABLE 13. Format specifiers for **printf** and similar functions

`'d'` or `'i'`	decimal	signed integer
`'u'`	decimal	unsigned integer
`'o'`	octal	unsigned integer
`'x'` or `'X'`	hexadecimal	unsigned integer
`'e'` or `'E'`	`[-]d.ddd e±dd`, "scientific"	floating point
`'f'` or `'F'`	`[-]d.ddd`	floating point
`'g'` or `'G'`	generic `e` or `f`	floating point
`'a'` or `'A'`	`[-]0xh.hhhh p±d`, hexadecimal	floating point
`'%'`	`'%'` character	no argument is converted
`'c'`	character	integer
`'s'`	characters	string
`'p'`	address	**`void`**`*` pointer

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

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

With the syntax `%[FF][WW][.PP][LL]SS`, a complete format specification can be composed of 5 different parts, flags, width, precision, modifiers and specifier. See Table 12 for details.

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

The modifier part is important to specify the exact type of the corresponding argument. Table 14 gives the codes for the types that we have encountered so far.

The flag can change the output variant, such as prefixing with signs (`"%+d"`), `0x` for hexadecimal conversion (`"%#X"`), `0` for octal (`"%#o"`), or padding with `0`. See Table 15.

If we know that the numbers that we write will be read back in from a file, later, the forms `"%+d"` for signed types, `"%#X"` for unsigned types and `"%a"` for floating point are the most appropriate. They guarantee that the string to number conversions will detect the correct form and that the storage in file will be without loss of information.

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

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

Rule 1.6.2.8 *Use* `"%+d"`, `"%#X"` *and* `"%a"` *for conversions that have to be read, later.*

Annex K

The optional interfaces **printf_s** and **fprintf_s** check that the stream, format and any string arguments are valid pointers. They **don't** check if the expressions in the list correspond to correct format specifiers.

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

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

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

This improves the diagnostic output by adding the file name to the output string. **fprintf_s** is used to check the validity of the stream, the format and the argument string. This function may mix the output to the two streams if they are both connected to the same terminal.

6.2.5. *Unformatted input of text.* Unformatted input is best done with **fgetc** for a single character and **fgets** for a string. There is one standard stream that is always defined that usually connects to terminal input: **stdin**.

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

Annex K

In addition, there are also **getchar** and **gets_s** that read from **stdin** but they don't add much to the above interfaces that are more generic.

```
char* gets_s(char s[static 1], rsize_t n);
```

Historically, in the same spirit as **puts** specializes **fputs**, prior version of the C standard had a **gets** interface. This has been removed because it was inherently unsafe.

Rule 1.6.2.9 *Don't use* **gets**.

The following listing shows a function that has an equivalent functionality as **fgets**.

LISTING 1.5. Implementing **fgets** in terms of **fgetc**

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

Again, such an example code is not meant to replace the function, but to illustrate properties of the functions in question, here the error handling strategy.

Rule 1.6.2.10 **fgetc** *returns* **int** *to be capable to encode a special error status,* **EOF**, *in addition to all valid characters.*

Also, the detection of a return of **EOF** alone is not sufficient to conclude that the end of the stream has been reached. We have to call **feof** to test if a stream's position has reached its end-of-file marker.

Rule 1.6.2.11 *End of file can only be detected* after *a failed read.*

Listing 1.6 presents an example that uses both, input and output functions.

LISTING 1.6. A program to concatenate text files

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

enum { buf_max = 32, };

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

Here, this presents a small implementation of `cat` that reads a number of files that are given on the command line, and dumps the contents to **stdout**.[Exs 39][Exs 40][Exs 41][Exs 42]

6.3. String processing and conversion. String processing in C has to deal with the fact that the source and execution environments may have different encodings. It is therefore crucial to have interfaces that work independent of the encoding. The most important tools are given by the language itself: integer character constants such as `'a'` or `'\n'` and string literals such as `"hello:\tx"` should always to the right thing on your platform. But handling such constants can become cumbersome if you have to deal with character classes.

Therefore the C library provides functions or macros that deals with the most commonly used classes through the header `ctype.h`. It has classifiers **isalnum**, **isalpha**, **isblank**, **iscntrl**, **isdigit**, **isgraph**, **islower**, **isprint**, **ispunct**, **isspace**, **isupper**, **isxdigit**, and conversions **toupper** and **tolower**. For historic reasons all of these take their arguments as **int** and also return **int**. See Table 16 for an overview over the classifiers. The functions **toupper** and **tolower** convert alphabetic characters to the corresponding case and leave all other characters as they are.

#include `<ctype.h>`

In that table we see all character control codes as defined by the C standard: `'\a'` (bell), `'\b'` (backspace), `'\f'` (form feed), `'\n'` (new line), `'\r'` (carriage return), `'\t'` (horizontal tabulator), and `'\v'` (vertical tabulator).

[Exs 39] Under what circumstances this program finishes with success or failure return codes?
[Exs 40] Surprisingly this program even works for files with lines that have more than 31 characters. Why?
[Exs 41] Have the program read from **stdin** if no command line argument is given.
[Exs 42] Have the program precedes all output lines with line numbers if the first command line argument is `"-n"`.

TABLE 16. Character classifiers. The third column indicates if C implementations may extend these classes with platform specific characters, such as 'ä' as a lower case character or '€' as punctuation.

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

The following function hexatridecimal uses some of these functions to provide a numerical value for all alphanumerical characters base 36. This is analogous to hexadecimal constants, only that all other letters give a value in base 36, too. [Exs 43] [Exs 44] [Exs 45]

```c
#include <string.h>

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

Besides the function **strtod**, the C library has **strtoul**, **strtol**, **strtoumax**, **strtoimax**, **strtoull**, **strtoll**, **strtold**, and **strtof** to convert a string to a numerical value. Here the

[Exs 43] The second **return** of hexatridecimal makes an assumption about the relation between a and 'A'. Which?

[Exs 44] Describe an error scenario in case that this assumption is not fulfilled.

[Exs 45] Fix this bug.

characters at the end of the names correspond to the type, `u` for **`unsigned`**, `l` (the letter "el") for **`long`**, `d` for **`double`**, `f` for float, and `[i|u]max` to **intmax_t** and **uintmax_t**.

The interfaces with an integral return type all have 3 parameters, such as e.g **strtoul**

```
unsigned long int strtoul(char const nptr[restrict],
                          char** restrict endptr,
                          int base);
```

which interprets a string `nptr` as a number given in base `base`. Interesting values for base are `0`, `8`, `10` and `16`. The later three correspond to octal, decimal and hexadecimal encoding, respectively. The first, `0`, is a combination of these three, where the base is choosing according to the usually roles for the interpretation of text as numbers: `"23"` is decimal, `"007"` is octal and `"0x7"` is hexadecimal.

More precisely, the string is interpreted as potentially consisting of four different parts: white space, a sign, the number and some remaining data.

The second parameter can in fact be used to obtain the position of the remaining data, but this is still too involved for us. For the moment, it suffices to pass a `0` for that parameter to ensure that all works well. A convenient combination of parameters is often **strtoul**`(S, 0, 0)`, which will try to interpret `S` as a string, regardless of the input format.

The three functions that provide floating point values work similar, only that the number of function parameters is limited to two.

In the following we will demonstrate how the such functions can be implemented from more basic primitives. Let us first look into `Strtoul_inner`. It is the core of a **strtoul** implementation that uses `hexatridecimal` in a loop to compute a large integer from a string.

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

In case that the string represents a number that is too big for an **`unsigned long`**, this function returns **ULONG_MAX** and sets **errno** to `ERANGE`.

Now `Strtoul` give a functional implementation of **strtoul**, as far as it can be done without pointers:

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

It wraps `strtoul_inner` and previously does the adjustments that are needed: it skips white space, looks for an optional sign, adjusts the base in case the `base` parameter was, `0`, skips an eventual `0` or `0x` prefix. Observe also that in case that a minus sign has been provided it does the correct negation of the result in terms of **`unsigned long`** arithmetic.[Exs 46]

To skip the spaces, `Strtoul` uses **strspn**, one of the string search functions that are provided by `string.h`. This functions returns the length of the initial sequence in the first parameter that entirely consists of any character of the second parameter. The function **strcspn** ("c" for "complement") works similarly, only that it looks for an initial sequence of characters **not** present in the second argument.

`#include` `<string.h>`

This header provides at lot more memory or string search functions: **memchr**, **strchr**, **strpbrk strrchr**, **strstr**, and **strtok**. But to use them we would need pointers, so we can't handle them, yet.

6.4. Time. The first class of "times" can be classified as calendar times, times with a granularity and range as it would typically appear in a human calendar, as for appointments, birthdays and so on. Here are some of the functional interfaces that deal with times and that are all provided by the `time.h` header:

`#include` `<time.h>`

```
time_t time(time_t *t);
double difftime(time_t time1, time_t time0);
time_t mktime(struct tm tm[1]);
size_t strftime(char s[static 1], size_t max,
                char const format[static 1],
                struct tm const tm[static 1]);
```

[Exs 46] Implement a function `find_prefix` as needed by `Strtoul`.

```
int timespec_get(struct timespec ts[static 1], int base);
```

The first simply provides us with a timestamp of type **time_t** of the current time. The simplest form to use the return value of **time**(0). As we have already seen, two such times that we have taken at different moments in the program execution can then be use to express a time difference by means of **difftime**.

Let's start to explain what all this is doing from the human perspective. As we already have seen, `struct` **tm** structures a calendar time mainly as you would expect. It has hierarchical date fields such as **tm_year** for the year, **tm_mon** for the month and so on, down to the granularity of a second. They have one pitfall, though, how the different fields are counted. All but one start with `0`, e.g **tm_mon** set to `0` stands for January and **tm_wday** `0` stands for Sunday.

Unfortunately, there are exceptions:

- **tm_mday** starts counting days in the month at 1.
- **tm_year** must add 1900 to get the year in the Gregorian calendar. Years represent in that way should lie between Gregorian years 0 and 9999.
- **tm_sec** is in the range from 0 to 60, including. The later is for the rare occasion of leap seconds.

There are three supplemental date fields that are used to supply additional information to a time value in a `struct` **tm**.

- **tm_wday** for the week day,
- **tm_yday** for the day in the year, and
- **tm_isdst** a flag that informs if a date is considered being in DST of the local time zone or not.

The consistency of all these fields can be enforced with the function **mktime**. It can be seen to operate in three steps

(1) The hierarchical date fields are normalized to their respective ranges.
(2) **tm_wday** and **tm_yday** are set to the corresponding values.
(3) If `tm_isday` has a negative value, this value is modified to 1 if the date falls into DST for the local platform, and to `0` otherwise.

mktime also serves an extra purpose. It returns the time as a **time_t**. **time_t** is thought to represent the same calendar times as `struct` **tm**, but is defined to be an arithmetic type, more suited to compute with them. It operates on a linear time scale. A **time_t** value of `0`. the beginning of **time_t** is called ***epoch***^{C} in the C jargon. Often this corresponds to the beginning of Jan 1, 1970.

The granularity of **time_t** is usually to the second, but nothing guarantees that. Sometimes processor hardware has special registers for clocks that obey a different granularity. **difftime** translates the difference between two **time_t** values into seconds that are represented as a double value.

Annex K

Others of the traditional functions that manipulate time in C are a bit dangerous, they operate on global state, and we will not treat them here. So these interfaces have been reviewed in Annex K to a `_s` form:

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

The following picture shows how all these functions interact:

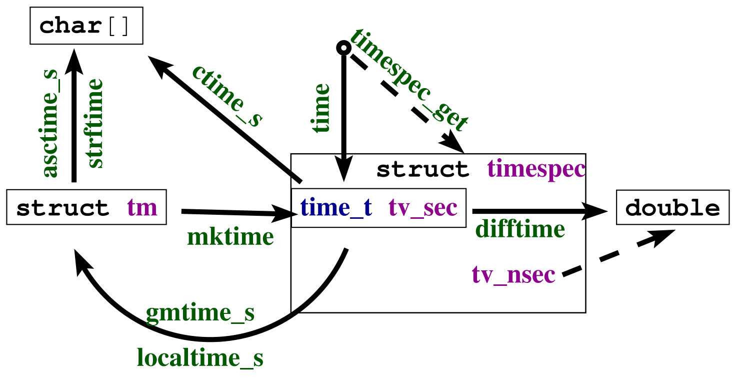


FIGURE 1. Time conversion functions

Two functions for the inverse operation from **time_t** into **struct tm** come into view:

- **localtime_s** stores the broken down local time
- **gmtime_s** stores the broken time, expressed as universal time, UTC.

As indicated, they differ in the time zone that they assume for the conversion. Under normal circumstances **localtime_s** and **mktime** should be inverse to each other, **gmtime_s** has no direct counterpart for the inverse direction.

Textual representations of calendar times are also available. **asctime_s** stores the date in a fixed format, independent of any locale, language (it uses English abbreviations) or platform dependency. The format is a string of the form

`"Www␣Mmm␣DD␣HH:MM:SS␣YYYY\n"`

strftime is more flexible and allows to compose a textual representation with format specifiers.

It works similar to the **printf** family, but has special `%`-codes for dates and times, see Table 17. Here "locale" indicates that different environment settings, such as preferred language or time zone may influence the output. How to access and eventually set these will be explained in Section 6.5. **strftime** receives three arrays: a `char[max]` array that is to be filled with the result string, another string that holds the format, and a `struct tm const[1]` that holds the time to be represented. The reason for passing in an array for the time will only become apparent when we know more about pointers.

The opaque type **time_t** and with that **time** only has a granularity of seconds.

If we need more precision than that, **struct timespec** and function **timespec_get** can be used. With that we have an additional field **tv_nsec** that provides nanosecond precision. The second argument `base` only has one value defined by the C standard, **TIME_UTC**. You should expect a call to **timespec_get** with that value to be consistent with calls to **time**. They both refer to Earth's reference time. Specific platforms may provide additional values for `base` that would specify a "clock" that is different from that "walltime" clock. An example for such a clock could be relative to the planetary or other

TABLE 17. **strftime** format specifiers. Those marked as "locale" may differ dynamically according to locale runtime settings, see Section 6.5. Those marked with ISO 8601 are specified by that standard.

spec	meaning	locale	ISO 8601
`"%S"`	second (`"00"` to `"60"`)		
`"%M"`	minute (`"00"` to `"59"`)		
`"%H"`	hour (`"00"` to `"23"`).		
`"%I"`	hour (`"01"` to `"12"`).		
`"%e"`	day of the month (`"␣1"` to `"31"`)		
`"%d"`	day of the month (`"01"` to `"31"`)		
`"%m"`	month (`"01"` to `"12"`)		
`"%B"`	full month name	X	
`"%b"`	abbreviated month name	X	
`"%h"`	equivalent to `"%b"`	X	
`"%Y"`	year		
`"%y"`	year (`"00"` to `"99"`)		
`"%C"`	century number (year/100)		
`"%G"`	week-based year, same as `"%Y"`, except if the ISO week number belongs another year		X
`"%g"`	like `"%G"`, (`"00"` to `"99"`)		X
`"%u"`	weekday (`"1"` to `"7"`), Monday being `"1"`		
`"%w"`	weekday (`"0"` to `"6"`, Sunday being `"0"`		
`"%A"`	full weekday name	X	
`"%a"`	abbreviated weekday name	X	
`"%j"`	day of year (`"001"` to `"366"`)		
`"%U"`	week number in the year (`"00"` to `"53"`), starting at Sunday		
`"%W"`	week number in the year (`"00"` to `"53"`), starting at Monday		
`"%V"`	week number in the year (`"01"` to `"53"`), starting with first 4 days in the new year		X
`"%Z"`	timezone name	X	
`"%z"`	`"+hhmm"` or `"-hhmm"`, the hour and minute offset from UTC		
`"%n"`	newline		
`"%t"`	horizontal tabulator		
`"%%"`	literal `"%"`		
`"%x"`	date	X	
`"%D"`	equivalent to `"%m/%d/%y"`		
`"%F"`	equivalent to `"%Y-%m-%d"`		X
`"%X"`	time	X	
`"%p"`	either `"AM"` or `"PM"`, noon is `"PM"`, midnight is `"AM"`	X	
`"%r"`	equivalent to `"%I:%M:%S␣%p"`.	X	
`"%R"`	equivalent to `"%H:%M"`		
`"%T"`	equivalent to `"%H:%M:%S"`		X
`"%c"`	preferred date and time representation	X	

physical system that your computer system is involved with.[47] Relativity or other time adjustments could be avoided by using a so-called "monotonic clock" that would only

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

be refering to the startup time of the system. A “cpu clock”, could refer to the time the program execution had been attributed processing resources.

For the later, there is an additional interface that is provided by the C standard library.

```
clock_t clock(void);
```

For historical reasons, this introduces yet another type, **clock_t**. It is an arithmetic time that gives the processor time in **CLOCKS_PER_SEC** units per second.

Having three different interfaces, **time**, **timespec_get** and **clock** is a bit unfortunate. It would have been beneficial to provide predefined constants such as `TIME_PROCESS_TIME` or `TIME_THREAD_TIME` for other forms of clocks.

6.5. Runtime environment settings. A C program can have access to an ***environment list***^{C}: a list of name-value pairs of strings (often called ***environment variables***^{C}) that can transmit specific information from the runtime environment. There is a historic function **getenv** to access this list:

```
char* getenv(char const name[static 1]);
```

With our current knowledge, with this function we are only able to test if a `name` is present in the environment list:

```
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]);
```

If any, this function copies the value that corresponds to `name` from the environment into `value`, a `char[maxsize]`, provided that it fits. Printing such value can look as this:

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

As you can see, that after detecting if the environment variable exists, **getenv_s** can safely be called with the first argument set to `0`. Additionally, it is guaranteed that the

TABLE 18. Categories for the **setlocale** function

LC_COLLATE	string comparison through **strcoll** and **strxfrm**
LC_CTYPE	character classification and handling functions, see Section 6.3.
LC_MONETARY	monetary formatting information, **localeconv**
LC_NUMERIC	decimal-point character for formatted I/O, **localeconv**
LC_TIME	**strftime**, see Section 6.4
LC_ALL	all of the above

`value` target buffer will only be written, if the intended result fits in, there. The `len` parameter could be used to detect the real length that is needed, and dynamic buffer allocation could be used to print out even large values. We have to refer to higher levels for such usages.

The environment variables that will be available to programs depend heavily on the operating system. Commonly provided environment variables include `"HOME"` for the user's home directory, `"PATH"` for the collection of standard paths to executables, `"LANG"` or `"LC_ALL"` for the language setting.

The language or ***locale***^{C} setting is another important part of the execution environment that a program execution inherits. At startup, C forces the locale setting to a normalized value, called the `"C"` locale. It has basically American English choices for numbers or times and dates.

The function **setlocale** from `locale.h` can be used to set or inspect the current value. `#include <locale.h>`

```
char* setlocale(int category, char const locale[static 1]);
```

Besides `"C"`, the C standard only prescribes the existence of one other valid value for `locale`, the empty string `""`. This can be used to set the effective locale to the systems default. The `category` argument can be used to address all or only parts of the language environment. Table 18 gives an overview over the possible values and the part of the C library they affect. Additional platform dependent categories may be available.

6.6. Program termination and assertions. We already have seen the simplest way of program termination: a regular return from **main**:

Rule 1.6.6.1 *Regular program termination should use* **`return`** *from* **main**.

Using the function **exit** from within **main** is kind of senseless, it can be as easily done with a **`return`**.

Rule 1.6.6.2 *Use* **exit** *from a function that may terminate the regular control flow.*

The C library has three other functions that terminate the program execution, in order of severity:

```
_Noreturn void quick_exit(int status);
_Noreturn void _Exit(int status);
_Noreturn void abort(void);
```

Now, **`return`** from **main** (or a call to **exit**) already provides the possibility to specify if the program execution is considered to be a success or not. Use the return value to specify that; as long as you have no other needs or you don't fully understand what these other functions do, don't use them, really don't.

Rule 1.6.6.3 *Don't use other functions for program termination than* **exit***, unless you have to inhibit the execution of library cleanups.*

Cleanup at program termination is important. The runtime system can flush and close files that are written or free other resources that the program occupied. This is a feature and should rarely be circumvented.

There even is a mechanism to install your own ***handlers***^{C} that are to be executed at program termination. Two functions can be used for that:

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

These have a syntax that we have not yet seen: ***function parameters***^{C}. E.g the first reads "function **atexit** that returns an `int` and that receives a function `func` as a parameter".[48]

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

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

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

This uses the function **atexit** to establish the **exit**-handler `sayGoodBye`. After normal termination of the program code this function will be executed and give a status of the execution. This might be a nice thing to impress your fellow co-worker if you are in need of some respect. More seriously, this is the ideal place to put all kind of cleanup code, such as freeing memory, or such as writing a termination time stamp to a logfile. Observe that the syntax for calling is **atexit** `(sayGoodBye)`, there are no `()` for `sayGoodBye` itself: here `sayGoodBye` is not called at that point, but only a reference to the function is passed to **atexit**.

Under rare circumstance you might want to circumvent these established **atexit** handlers. There is a second pair of functions, **quick_exit** and **at_quick_exit**, that can be used to establish an alternative list of termination handlers. Such alternative list may be useful in case that the normal execution of the handlers is too time consuming. Use with care.

The next function, **_Exit**, is even more sever than that: it inhibits both types of application specific handlers to be executed. The only thing that is still executed are the platform specific cleanups, such as file closure. Use this with even more care.

The last function, **abort**, is even more intrusive. Not only that it doesn't call the application handlers, it also inhibits the execution of some system cleanups. Use this with extreme care.

At the beginning of this section, we have already seen **`_Static_assert`** and **static_assert** that should be used to make compile time assertions. This can test for any form of compile time Boolean expression. There are two other identifiers that come from `assert.h` that can be used for runtime assertions: **assert** and **NDEBUG**. The first can be used to test for an expression that must hold at a certain moment. It may contain any Boolean expression and it may be dynamic. If the **NDEBUG** macro is not defined

`#include <assert.h>`

[48]In fact, in C such a notion of a function parameter `func` to a function **atexit** is equivalent to passing a ***function pointer***^{C}. In descriptions of such functions you will usually see that pointer variant. For us this distinction is not yet relevant and it is simpler to think of a function being passed by reference.

during compilation, every time execution passes by the call to this macro the expression is evaluated. The functions `gcd` and `gcd2` from Section 5.3 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 all of this is not something that should make it through into a production executable. From the discussion above we know that the use of **abort** is harmful, in general, and also an error message such as

```
Terminal
0   assertion failed in file euclid.c, function gcd2(), line 6
```

is not very helpful for your customers. It *is* helpful during the debugging phase where it can lead you to spots where you make false assumptions about the value of some variables.

Rule 1.6.6.4 *Use as many* **assert** *as you may to confirm runtime properties.*

As mentioned **NDEBUG** inhibits the evaluation of the expression and the call to **abort**. Please use it to reduce the overhead.

Rule 1.6.6.5 *In production compilations, use* **NDEBUG** *to switch off all* **assert**.

APPENDIX A

TABLE 1. Scalar types used in this book

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

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

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

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

operator	nick	form	type	result	
		`o`	array*	pointer	array decay
		`o`	function	pointer	function decay
		`o`	other	value	evaluation
`=`		`o@a`	non-array	value	assignment
`+= -= *= /=`		`o@a`	arithmetic	value	arithmetic
`+= -=`		`o@a`	pointer	value	arithmetic
`%=`		`o@a`	integer	value	arithmetic
`++ --`		`@o o@`	real or pointer	value	arithmetic
`&= |= ^=`	**and_eq**	`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`	non-function	**size_t**	size

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

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

Reminders

1.2.5.1	`&&`, `||`, `?:` and `,` evaluate their first operand first.	26
1.2.5.2	Don't use the `,` operator.	26
1.2.5.3	Most operators don't sequence their operands.	26
1.2.5.4	Function calls don't sequence their argument expressions.	26
1.2.5.5	Functions that are called inside expressions should not have side effects.	26
1.3.0.6	All values are numbers or translate to such.	27
1.3.0.7	All value has a type that is statically determined.	28
1.3.0.8	A value's type determines its possible operations.	28
1.3.0.9	A value's type determines the results of all operations.	28
1.3.0.10	A type's binary representation determines the results of all operations.	28
1.3.0.11	A type's binary representation is observable.	28
1.3.0.12	Programs execute **as if** they were following the abstract state machine.	28
1.3.0.13	Type determines optimization opportunities.	29
1.3.1.1	Each of the 4 classes of base types has 3 distinct unpromoted types.	29
1.3.1.2	Use **size_t** for sizes, cardinalities or ordinal numbers.	30
1.3.1.3	Use **`unsigned`** for small quantities that can't be negative.	30
1.3.1.4	Use **`signed`** for small quantities that bear a sign.	30
1.3.1.5	Use **ptrdiff_t** for large differences that bear a sign.	30
1.3.1.6	Use **`double`** for floating point calculations.	30
1.3.1.7	Use **`double`** **complex** for complex calculations.	30
1.3.2.1	Numerical literals are never negative.	32
1.3.2.2	Decimal integer constants are signed.	32
1.3.2.3	A decimal integer constant has the first of the 3 signed types that fits it.	32
1.3.2.4	The same value can have different types.	32
1.3.2.5	Don't use octal or hexadecimal constants to express negative values.	32
1.3.2.6	Use decimal constants to express negative values.	32
1.3.2.7	Different literals can have the same value.	33
1.3.2.8	The effective value of a decimal floating point constant may be different from its literal value.	33
1.3.2.9	Literals have value, type and binary representation.	33
1.3.3.1	All variables should be initialized.	33
1.3.3.2	Use designated initializers for all aggregate data types.	34
1.3.3.3	`{0}` is a valid initializer for all types that are not VLA.	34
1.3.4.1	All constants with particular meaning must be named.	35
1.3.4.2	All constants with different meaning must be distinguished.	35
1.3.4.3	Enumeration constants have either an explicit or positional value.	35
1.3.4.4	Enumeration constants are of type **`signed int`**.	36
1.3.4.5	An integer constant expression doesn't evaluate any object.	36
1.3.4.6	Macro names are in all caps.	37
1.3.4.7	Don't hide a terminating semicolon inside a macro.	37
1.3.4.8	**I** is reserved for the imaginary unit.	38
1.3.5.1	The same value may have different binary representations.	38
1.3.5.2	Unsigned arithmetic wraps nicely.	38

Listings

Bibliography

JTC1/SC22/WG14, editor. *Programming languages - C*. Number ISO/IEC 9899. ISO, cor. 1:2012 edition, 2011. URL `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`.

Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

Brian W. Kernighan and Dennis M. Ritchie. The C programming language. *Encyclopedia of Computer Science*, 1980.

Brian W. Kernighan and Dennis M. Ritchie. The state of C. *BYTE*, 13(8):205–210, August 1988a.

Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988b.

Dennis M. Ritchie. Variable-size arrays in C. *Journal of C Language Translation*, 2(2): 81–86, September 1990.

Dennis M. Ritchie. The development of the C language. In Thomas J. Bergin, Jr. and Richard G. Gibson, Jr., editors, *Proceedings, ACM History of Programming Languages II*, Cambridge, MA, April 1993. URL `http://cm.bell-labs.com/cm/cs/who/dmr/chist.html`.

Dennis M. Ritchie, Brian W. Kernighan, and Michael E. Lesk. The C programming language. Comp. Sci. Tech. Rep. No. 31, Bell Laboratories, Murray Hill, New Jersey, October 1975. Superseded by Kernighan and Ritchie [1988b].

Dennis M. Ritchie, Steven C. Johnson, Michael E. Lesk, and Brian W. Kernighan. Unix time-sharing system: The C programming language. *Bell Sys. Tech. J.*, 57(6):1991–2019, 1978.

Index