The C programming language
— T.02 —

Summary:

- My first C program
- C language overview
- Preprocessor directives
- Comments
- Data types
- Declaration, definition, and scope of variables
- Assignments and expressions
- Statements
- Functions
- Standard library functions
- Coding style
- **Useful web sites**
- Exercises

[Remark: due to space and time limitations, we will omit many details.]

According to an IEEE spectrum survey, on a scale from 0 to 100, in 2022 the top five most popular programming languages were:

- Python 100.0 (100.0 in 2021)
- C 96.8 (94.7 in 2021)
- C++ 88.6 (92.4 in 2021)
- C# 87.0 (82.4 in 2021)
- Java 70.2 (95.4 in 2021)

(It is in first place in another popularity index.) Among popular programming languages, C is the most energy-efficient one.

Recommended bibliography for this lecture:

- **C in a Nutshell, A Desktop Quick Reference**, Peter Prinz and Tony Crawford, O'Reilly, 2006.
- **C, A Reference Manual**, Samuel P. Harbison III and Guy L. Steele Jr., fifth edition, Prentice Hall, 2002.
- **C Programming (A Comprehensive Look at the C Programming Language and Its Features)**, wiki book.
- **The C Programming Language**, Brian W. Kernighan and Dennis M. Ritchie, second edition, Prentice Hall, 1988.

My first C program

The “hello world” program:

```
 1 /*
 2 ** Hello world program
 3 */
 4
 5 #include <stdio.h>
 6
 7 int main(void)
 8 {
 9   puts("Hello world!");
10   return 0;
11 }
```

The line numbers on the left are not part of the code.

Explanation:

- Lines 1 to 3 are a comment. The comment starts with `/*` and ends with `*/`.
- Line 5 instructs the compiler to replace that line by the contents of the file named `stdio.h` (one of the files of the C compiler's standard libraries).
- Lines 7 to 11 declare and define a function named `main`, which is the entry point of the program (the entry point of the program is always called `main`); in this case `main` does not have any arguments and it returns an integer.
- Lines 8 to 11 constitute the body of the function.
- Line 8 starts a block of code; code blocks start with a `{`.
- Line 9 is a call to a function named `puts` (declared in `stdio.h`), belonging to the C standard library, that outputs its string argument to the terminal. Strings are text, and are delimited by `"`.
- Line 10 forces a return from the main function with a return value of 0 (since `main` is the entry point of the program, this actually specifies the error code of the entire program; 0 means all is well, non-zero means some error occurred).
- Line 11 ends a block of code; code blocks end with a `}`.

C language overview

Each C source code file may contain

- preprocessor directives

  Preprocessor directives tell the compiler to manipulate the source code in certain ways. For example, the `#include` directive tells the compiler to replace the entire include directive line by the text of the file whose name appears after the include directive. There are also directives that allow us to define replacement text for a given word and to do conditional compilation of code.

- comments

  A comment is a chunk of text that is ignored by the compiler

- declaration of new data types

  New data types agglomerate one or more existing data type into a new type, and give it a name that we can use from that point on to refer to that new type.

- declaration of variables and of functions

  A declaration of a variable is a description of the type and memory storage attributes of the variable. A declaration of a function is a description of the arguments and return value (if any) and their type of the function. It **does not** reserve space in memory for a variable and **no code** is produced for a function. After a declaration, the variable or function can be used in our code, even if its definition (see next item) is elsewhere in the code (it can even be missing in our code if it resides in a code library).

- definition of variables and of functions

  When the compiler encounters a definition it reserves space for a variable and generates code for a function. It is possible to declare and define a variable (or a function) at the same time. Variable and function names have to be unique.

Preprocessor directives (part 1)

The C preprocessor can be used to modify on the fly the text that is going to be fed to the C compiler. Each preprocessor directive must be placed on a line that begins with the character #. The most important of them are:

- `#include <filename>`
  `#include "filename"`
  This directive instructs the preprocessor to replace the directive by the entire text of the file whose name follows the include directive. The first form looks for files only in compiler directories (standard library header files). The second form looks for user files (in the current directory).
- `#define NAME substitution_text`
  `#define NAME(arg1,arg2,...) substitution_text`
  This directive defines a preprocessor macro named `NAME`. On subsequent lines, each time the text `NAME` appears in the source code it is replaced by the substitution text. In the first form, the macro does not have any arguments. In the second form it can have one or more arguments. If the name of an argument appears in the substitution text it gets replaced by the text that was placed in the argument when the macro was invoked. For example, the code fragment

  ```
  #define C       (int)
  #define X(i)    x[i]
  #define Y(i,j)  i * j
  C X(3) + Y(i,7);
  ```

  gets transformed into the code

  ```
  (int) x[3] + i * 7;
  ```

  Note, however, that recursively expanding the same macro name is automatically disabled by the C preprocessor, so no infinite expansions can occur.

Preprocessor directives (part 2)

- `#undef NAME`
  It is not possible to redefine a macro (with a different replacement text) without first removing its previous definition. This directive makes sure that a previous definition of the macro (if any) is removed.
- `#if EXPRESSION`
  `#elif EXPRESSION`
  `#else`
  `#endif`
  If the integer expression, which must use only constants known to the preprocessor (macros with replacement text that are integers), is non-zero, then the text in the lines following an `#if` directive and up to an `#elif`, an `#else` or an `#endif` directive gets fed to the compiler; `#elif` and `#else` directives are treated in the logical way. Symbols unknown to the preprocessor are replaced by zeros. For example, in the code fragment

```
#if N == 1
line1
#elif N == 2
line2
#else
line3
#endif
```

`line1` is passed to the compiler only if the macro `N` is defined and evaluates to `1`, `line2` is passed to the compiler only if the macro `N` is defined and evaluates to `2`, and `line3` is passed to the compiler if the macro `N` does not evaluate to either `1` or `2` (if it is not defined it evaluates to zero, and so it falls in this case).

Preprocessor directives (part 3)

The substitution text of a macro can contain zero or more occurrences of the special character #. This special character followed by a macro argument name tells the preprocessor to remove the # and to **stringify** the macro argument, that is, to convert the macro argument into a string. For example, in the following code

```
#define abc(x,y) # x "_" # y
abc(123,xyz)
```

the first line becomes empty (it is a macro definition), and the second line becomes `"123" "_" "xyz"`, which is the same as `"123_xyz"` (the C compiler concatenates consecutive constant strings — note that there does not exist any + sign between the strings!).

The substitution text of a macro can contain also zero or more occurrences of the special characters ##. These special characters fuse what is on their left size with what is on their right side (no space character in between). For example, in the following code

```
#define abc(x,y) x y
#define def(x,y) x ## y
abc(xyz,123)
def(xyz,123)
```

the first two lines become empty (macro definitions), the third line becomes `xyz 123` (note the space between `xyz` and `123`), and the fourth line becomes `xyz123`, which is a single token (for example, a variable or function name).

Comments

Comments are annotations placed in the source code of a program. A good comment explains a non-obvious thing, such as how some piece of code works, or what trade-offs (between, say, execution time and memory usage) were made in that part of the code and why. Comments are also usually used to indicate who wrote a part of a program, and to record significant changes in the source code.

In C there are two kinds of comments: single line comments, which begin with // and end at the end of the line, such as in

```
d = (d + 1) | 1; // if d is even and non-negative increment it by one, otherwise, increment it by 2
```

and comments that can span multiple lines, which begin with /∗ and end with **the first** ∗/, such as in

```
/*
** in the following loop d takes the values 2, 3, 5, 7, 9, 11, 13, 15, 17, ...,
** up to (and including) the square root of n
**
** we would have liked for d to be the prime numbers 2, 3, 5, 7, 11, 13, 17, ...,
** but that is much more difficult to achieve
**
** FIX ME: there is arithmetic overflow if n is a prime number close to the largest representable
** signed integer; for 32-bit integers this can be fixed by exiting the loop as soon as d > 46340
*/
for(d = 2;d * d <= n;d = (d + 1) | 1)
```

Comments are removed by the preprocessor. The preprocessor joins a line terminated by \ with the next line, so single line comments may actually span more than one line if they are terminated in that way.

A simple and fast way to force the compiler to ignore a large continuous piece of code, even it is has comments, is to put it between `#if 0` and `#endif` lines.

Data types (part 1a, integer data types)

The C language has the following fundamental integer data types (in non-decreasing order of size): `char`, `short`, `int`, `long` and `long long`. Each of these types can be either `signed` (the default with the possible exception of the `char` type) or `unsigned`. In the following code fragment we present an example of the simultaneous declaration, definition, and initialization of variables for each one of these types.

```
              char  c0 = 'A';  // by default signed on most compilers
  signed      char  c1 = 'B';  // make sure the type is signed
unsigned      char  c2 = 'C';

              short s0 = 1763;  // the same as signed short
unsigned      short s1 = 1728;

              int   i0 = -1373762;  // the same as signed int
unsigned      int   i1 = 8382382U;  // the trailing U signals that the integer constant is unsigned

              long  l0  = 82781762873L;  // the same as signed long and signed long int
unsigned      long  l1 = 38273827322UL;  //   the int is optional, so we do usually do not put it

         long long  L0  = 8278176284398473LL;  // the same as signed long long int
unsigned long long  L1 = 3827382733493498322ULL;  //   the int is optional
```

Unfortunately, the designers of the C language did not specify the size (number of bytes) of most of these types. So, an `int` may have two bytes if the compiler is producing code for a very old processor, and four bytes if it is targeting a modern processor. The types `int8_t`, `uint8_t`, `int16_t`, `uint16_t`, `int32_t`, `uint32_t`, `int64_t`, and `uint64_t`, defined in the header file `stdint.h`, should be used whenever a specific size is desired.

Data types (part 1b, integer data types)

Let $b_0, b_1, b_2, \ldots, b_{n-2}, b_{n-1}$ be the bits on an n-bit integer B, b_0 being the least significant bit and b_{n-1} being the most significant bit. On virtually all contemporary processors, for an **unsigned integer data type** the value of B is given by

$$B = b_0 \times 2^0 + b_1 \times 2^1 + b_2 \times 2^2 + \cdots + b_{n-2} \times 2^{n-2} + b_{n-1} \times 2^{n-1} = \sum_{i=0}^{n-1} b_i 2^i.$$

Therefore, to increase the number of bits of an unsigned integer the new bits get a value of 0 since doing that does not change the value represented by the bits. To reduce the number of bits, in C the most significant bits are simply discarded (no error is raised if the result does not represent the original unsigned integer; it is assumed that the programmer knows that she/he is doing).

For a **signed integer data type** the value of B is given by (two's complement!)

$$\begin{aligned} B &= b_0 \times 2^0 + b_1 \times 2^1 + b_2 \times 2^2 + \cdots + b_{n-2} \times 2^{n-2} - b_{n-1} \times 2^{n-1} \\ &= -b_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i = -b_{n-1} 2^n + \sum_{i=0}^{n-1} b_i 2^i. \end{aligned}$$

This last equality, that expresses B with $n+1$ bits in which $b_n = b_{n-1}$, shows that to increase the number of bits of a signed integer the new bits get the value of the most significant bit of the original signed integer, since doing that does not change the value represented by the bits. As for unsigned integers, to reduce the number of bits, in C the most significant bits are simply discarded (again, no error is raised if the result does not represent the original signed integer).

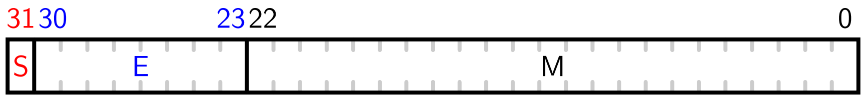
To convert from a n-bit signed to an n-bit unsigned integer, and vice versa, the C compiler simply does nothing (does not change any bit). Again, no error is raised if the result does not represent the original integer.

Data types (part 2, floating point)

The C language has two fundamental floating point data types: `float` (single precision, 4 bytes, about 7 significant decimal digits) and `double` (double precision, 8 bytes, about 16 significant decimal digits). In the following code fragment we present an example of the simultaneous declaration, definition, and initialization of variables of each one of these types.

```
 float f =  1.23e3f;  // the same as 1230.0f (f denotes a 4-byte floating point constant)
double d = -1.23e6;   // the same as -1230000.0 (no f, so a 8-byte floating point constant)
```

All contemporary processors represent floating point numbers using the IEEE 754 standard. For example, a single precision float number is encoded in 32 bits according to the following format:

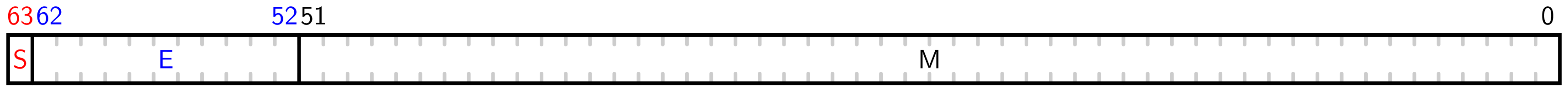


One bit encodes the sign (S) of the number, 8 bits encode an exponent (E), and 23 bits encode a so-called mantissa (M). In this format, assuming that both E and M are unsigned integers with the appropriate number of bits, a non-zero real number x is given by

$$x = (-1)^S \times \left(1 + \frac{M}{2^{23}}\right) \times 2^{E-127}.$$

The real number 0.0 is encoded by all zeros; 0 and 255 are special values of E, for which the previous formula is not valid.

The double precision floating point format is similar, but the number of bits used to represent the exponent and the mantissa are larger (11 bits for the exponent and 52 bits for the mantissa):



Data types (part 3, pointer data types)

The C language has only one more fundamental data type: a pointer. A pointer is an unsigned integer that represents the memory address where a variable of a given type is stored. One can, using the appropriate syntax (see below) read or write the contents of the memory location whose address is stored in the pointer. (Note that, to the processor, the pointer itself is an integer variable.) Given that it is possible to do arbitrary modifications to a pointer, it is possible in C to manipulate the contents of arbitrary memory locations (dangerous, but powerful!). Pointers are declared and used by putting a `*` before the pointer name. To get the address of a variable put an `&` before the variable name. For example, in the code fragment

```
float f = 1.0f,*pf = &f;
*pf = 2.0;
```

the single precision floating point variable `f` gets the value 1.0 at the end of the first line and the value of 2.0 at the end of the second. As long as `pf` is not modified, it is possible to change the contents of `f` through the pointer. Incrementing (decrementing) a pointer makes it point to the next (previous) adjacent variable in memory of the same type. If that memory location does not hold a variable of that type, changing the memory through the pointer leads to all kinds of problems. Assigning `NULL` to a pointer is the standard way in C to say that the pointer points to nothing.

We also have pointers to functions: they store the starting address of a function.

Data types (part 4, literal values)

Literal values are the numeric constants we put in our programs. Integer literals can be encoded in several ways:

- as a character, as in `'3'`, `'\''`, `'"'`,`'\n'`, or `'\t'`
- as a decimal integer, as in `123` or `-17`
- as an hexadecimal integer (base 16), as in `0x01F3`
- as an octal integer (base 8), as in `0173`. Constants beginning with with a 0 **are specified in octal**!
- as an binary integer (base 2), as in `0b10101` (`gcc` compiler).

With the exception of character encodings, append `U` at the end to mark an unsigned integer, append `L` to mark a `long` integer, and append `LL` to mark a `long long` integer. All of these can also be in lower case.

Floating points constants can have an optional sign (plus or minus), can have zero or more digits before an optional decimal point and zero or more digits after the decimal point (in all, at least one digit must be present). It can also have an optional exponent part, composed by the letter e (upper or lower case), followed by a decimal signed integer. For example, `-1.2`, `.2e-13`, `+12.`, `1e-8` are all valid floating point literals. By default, floating point literals are in double precision; append `f` to get a single precision literal, as in `1.23e4f`.

String literals, enclosed by double quotation marks, as in `"12\"9348"`, are of type `const char *`, i.e., they cannot be overwritten. Note, however, that in the first line of the following code

```
char str[] = "292348"; // a char array with 7 elements (why?); str[0] = 'T' is ok
char *pstr = "2983";   // a string literal; *pstr = '4' gives a runtime error
```

the string is used to initialize the array, and so it is not a string literal.

Data types (part 5, arrays)

It is possible to extend the fundamental data types in two ways: using arrays and creating new data types.

An array is just a contiguous group of variables of the same type. For example, the following code

```
double d[10],D[10][10];
```

declares an (unidimensional) array `d` of 10 double precision floating point numbers and declares a bi-dimensional array (a matrix) `D` of 10 by 10 (i.e., 100) double precision floating point numbers. Accessing the array elements is done using square brackets. Indices start at 0. Usually, **no run-time tests** are performed to verify if the index being used to access an array element has a valid value. Using an out-of-range value does not result in any compiler error but will usually lead to a hard to discover run-time error.

It is important to realize that C does not allow you to manipulate an entire array as a single entity. This is so because an array name is a pointer to its first element. In the code

```
double d[10],*pd = d;
```

This implies that `i[d]` is the same as `d[i]`, but no one sane writes an array access in this twisted way.

it is possible to perform accesses to the array using either `d[i]`, which is the same as `*(d+i)`, or `pd[i]`, which is the same as `*(pd+i)`. Both are equivalent, as long as `pd` is not modified. On the other hand, in the code

```
double d[10],*pd = &d[9];
```

an access to `pd[i]` is equivalent to an access to `d[i+9]`, i.e., `pd[-9]` is the same as `d[0]` (assuming that `pd` has not been modified).

In C a string is an array of characters, terminated by a 0. For example, the code

```
char str[20] = "AB"; // same as char str[20] = { 'A','B',0 }; or char str[20] = { 65,66,0 };
```

defines a string that can hold up to 19 characters (space **must** be reserved for the 0 terminator), initialized with the two character string `"AB"`.

Data types (part 6a, strings and character sets)

As mentioned in the previous page, a string is an array of characters terminated by a byte with value 0 (`'\0'`). The actual text of the string depends on the character set used. For example, in the ASCII character set, a byte with a (unsigned) value smaller than 32 is a control character, and a byte with a value between 32 and 126 represent letters, numbers, and various symbols; values larger than 127 are undefined (127 is the delete symbol, usually not represented graphically).

The following table presents some control characters (under GNU/Linux, use the command

```
man ascii
```

on a terminal to get the complete list).

value	name	meaning	escape sequence
0	NUL	null character (end of string)	`\0`
7	BEL	terminal bell	`\a`
8	BS	backspace	`\b`
9	HT	horizontal tab	`\t`
10	LF	new line	`\n`
12	FF	form feed (new page)	`\f`
13	CR	carriage return	`\r`
27	ESC	escape	`\e`

Try also

```
man console_codes
```

and, on a GNU/Linux terminal,

```
echo -e "\e[5;32mHello\e[0m"
```

Data types (part 6b, strings and character sets)

The following table presents the so-called printable ASCII characters (range 32 to 126, i.e., 0x20 to 0x7E). The encoding (the byte values) are presented in hexadecimal (sum of the value of the first row with the value of the first column), because that makes the way the letters and numbers are organized in the ASCII code crystal clear.

	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
0x20		!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
0x30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x60	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

For example, to get the character corresponding to a single decimal digit stored in an integer variable named `digit` all we need to do is write

```c
char c = (char)(0x30 + digit); // '0' + digit
```

To encode letters with accents it is possible to use the (now deprecated) so-called iso-latin character set, which encodes them using byte values in the range 160 to 255. (Under GNU/Linux, use the command

```
man iso_8859-1
```

on a terminal to get the complete list of characters that can be encoded in this way.)

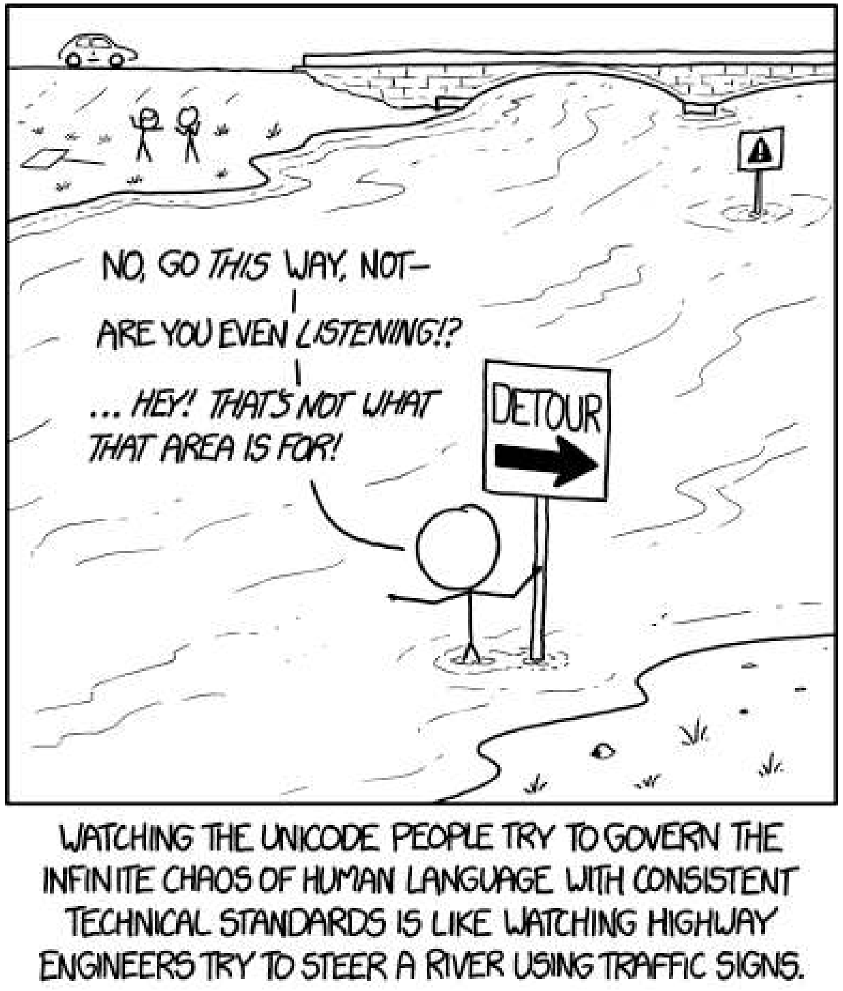
Data types (part 6c, strings and character sets)

Nowadays the preferred encoding is the ISO 10646 Universal Character Set (UCS), which can encode any character in any written language. In UCS, each character is encoded in a 31-bit integer (the often mentioned unicode is a 20-bit subset of the UCS). To avoid wasting a lot of memory space, it is common to store the UCS/unicode code using the so-called `utf-8` encoding (use "`man utf8`" to get more details about this). In the `uft-8` encoding of UCS, each ASCII character is encoded in only one byte, and letters with accents are encoded in two bytes. The following small table gives some examples of the iso-latin and utf-8 encodings.

character	iso-latin	unicode	utf-8 bytes	unicode in a C string
á	0xE1	0x000E1	0xC3,0xA1	`"\u00E1"`
é	0xE9	0x000E9	0xC3,0xA9	`"\u00E9"`
í	0xED	0x000E9	0xC3,0xAD	`"\u00ED"`
ó	0xF3	0x000F3	0xC3,0xB3	`"\u00F3"`
ú	0xFA	0x000FA	0xC3,0xBA	`"\u00FA"`
ã	0xE3	0x000E3	0xC3,0xA3	`"\u00E3"`
ç	0xE7	0x000E7	0xC3,0xA7	`"\u00E7"`

Irrespective of the encoding used, a string always terminates with a single byte with the value 0, and that value cannot occur anywhere inside the string.

Unicode (spoiler)


NO, GO THIS WAY. NOT—
ARE YOU EVEN LISTENING!?
... HEY! THAT'S NOT WHAT THAT AREA IS FOR!
DETOUR
WATCHING THE UNICODE PEOPLE TRY TO GOVERN THE INFINITE CHAOS OF HUMAN LANGUAGE WITH CONSISTENT TECHNICAL STANDARDS IS LIKE WATCHING HIGHWAY ENGINEERS TRY TO STEER A RIVER USING TRAFFIC SIGNS.

I'm excited about the proposal to add a "brontosaurus" emoji codepoint because it has the potential to bring together a half-dozen different groups of pedantic people into a single glorious internet argument.

Data types (part 7, pointer arithmetic)

A pointer is the address of a memory location. In C, adding an integer to a pointer **does not**, in general, change the address by that amount: it changes the address by that amount **times** the size in bytes of the type that the pointer points to. Things are done in this way to make working with pointers easier to the programmer, in particular when dealing with arrays. For example, in the code

```
int a[100];
int *pa = &a[30]; // same as int *pa = a + 30;
int *pA = &a[-2]; // same as int *pA = a - 2;
```

the pointer `pa` points to the element with index 30 of the array `a`. The address of this element is the sum of the address of the beginning of the array (`a`, or, what is the same, `&a[0]`) with the number of bytes required to store 30 integers (120 bytes if each integer occupies 4 bytes). In C we only need to add 30 to the pointer to get the correct address; the multiplication by the size of the type pointed to is done automatically by the compiler.

There are two exceptions to the above rule of pointer arithmetic:

- Adding an integer to a pointer to a function is meaningless, because the size of a function (the number of bytes of its code) is not constant and is not known at compile time. So, if one attempts to do this the compiler generates an error.
- Adding a constant to a pointer to `void` (the `void` type will be formally introduced soon), adds that many bytes to the pointer (so `sizeof(void)` is 1 and not 0, as would be more natural). This is done in order to make life easier to the programmer, since manipulating pointers to `void` is sometimes useful.

[**Homework:** study carefully how pointer arithmetic in done in C.]

Data types (part 8, structures and unions)

A structure is a contiguous group of variables of the same or different types, each with its own name. It is declared using the keyword `struct`. In the following code,

```
struct dot
{
  double x;
  double y;
  int color;
  struct dot *next;
};
```

a new data type, named `struct dot`, is declared. It has 4 fields named `x`, `y`, `color`, and `next`, respectively of types `double`, `double`, `int`, and pointer to a dot structure. (One can also put arrays, and even other structures, inside a structure.) The following code fragment gives an example of how structure fields are accessed:

```
struct dot d;   // a dot structure (RESERVES SPACE FOR THE STRUCTURE)
struct dot *pd; // a pointer to a dot structure (DOES NOT RESERVE SPACE FOR THE STRUCTURE)
pd = &d;        // set the pointer to point to the structure (now it can be safely used)
d.x = 3;        // set the x field to 3
pd->color = 5;  // set the color field to 5
d.next = NULL;  // set the next field to NULL (a special address that points to nothing)
```

Unlike arrays, the name of the structure represents the entire structure (it is **not** a pointer to its position in memory; to get that use an &). The abbreviation `struct struct_name;` tells the compiler that a structure named `struct_name` will be fully specified later on.

Unions are like structures, except that all its fields are superimposed in memory. (Believe it or not that is sometimes useful.) Only one field should be in use at any given time.

Data types (part 8, example of the use of a union)

In the following code we examine how the floating point number 10.0 is stored in memory.

```
#include <stdio.h>
int main(void)
{
  union
  {
    double d;
    unsigned char c[8];
  }
  t;

  t.d = 10.0;               // store a double
  for(int i = 0;i < 8;i++)
    printf(" %02X",t.c[i]); // print its individual bytes
  printf("\n");
  return 0;
}
```

On a little-endian processor (like all Intel and AMD processors) the least significant byte is stored in the lowest address, and so the program's output will be

```
 00 00 00 00 00 00 24 40
```

On a big-endian processor (like the IMB PowerPC processor) the least significant byte is stored in the highest address, and so the program's output will be

```
 40 24 00 00 00 00 00 00
```

This may be relevant when raw data is to be exchanged between two computer systems. If they have different endianesses, extra work has to be done to invert the order of the bytes (in each data field).

Data types (part 9, enum and void)

It is possible to create a data type which has a discrete set of constant integer values, each one with its own name. Such a type is an enumerated type. For example, the code

```
enum color { black,red,green,blue = 4 };
```

declares an enumerated type called `enum color`, that can have 4 values: `black`, `red`, `green`, and `blue`, respectively with numerical values 0, 1, 2, and 4. (These numerical values are how the names are internally represented in the program; the program code should use the names.) The following code defines a variable of this type and initializes it with the value `red`:

```
enum color dot_color = red;
```

An enumerated data type is an integer data type. The compiler is free to choose the number of bytes needed to internally represent its values.

There is one final data type that can be used in programs: `void`. This data type cannot have a value. It can be used to tell the compiler that a function does not return anything, or that a function does not have arguments, as exemplified in the following code:

```
void f(int x); // returns nothing, has an integer argument
int g(void);   // returns an integer, does not have arguments
void h(void);  // returns nothing, does not have arguments
```

A pointer to `void` is a popular way to declare a pointer to something which has a type that is not explicitly given. Of course it is not possible to dereference a pointer to void, i.e., access the memory it points to. One has to first cast it (see next slide) to a pointer to a specific type.

Data types (part 10, casts)

It is possible to tell the compiler to explicitly convert one data type to another compatible type. Such conversions, called casts, are implicitly done by the compiler in arithmetic expressions. For example, if `i` if an `int` and `d` is a `double`, the result of the expression `i + d`, which is the sum of the integer variable `i` with the double precision floating point variable `d` is a double precision floating point number. Before doing the addition, `i` is converted (cast) to the double precision floating point number, and only then is the addition performed. To make the conversion explicit, use `(double)i + d`. [My personal opinion is that mixing different data types in an expression without explicit type conversions is a bad programming practice.]

In C, to explicitly convert `x` to the data type `T` one writes `(T)x`. In C it is only possible to convert from one numeric type to another (that the compiler knows about), because the language does not offer any mechanism to tell the compiler how to perform more complex conversions. For example, it is not possible to convert from an integer to a structure. You will need to write your own function, and call it explicitly, in order to do that.

It is possible to cast a pointer to a data type to a pointer to another data type. That is a very dangerous thing to do (you **must** know what you are doing!) unless one of the two is a pointer to `void`. Indeed, a pointer to `void` is the standard way to go when one desires to perform some action on a memory region without needing to known what it contains (for example, reading or writing it to a file):

```
void write_data(void *ptr,int size);     // function to write size bytes starting at address ptr
double array[100];
write_data((void *)array,sizeof(array)); // write the 100 doubles
```

Since a pointer is represented by an unsigned integer (with a number of bits compatible with the processor architecture the program is being compiled to) it is also possible to convert a pointer to a sufficiently wide integer (we recommend using for this purpose the `size_t` data type), and vice versa, but that is not recommended (and not needed in any sane and portable program).

Data types (part 11, typedef)

In C it is also possible to give another name to an existing data type using the `typedef` keyword. For example, the following code fragment declares a data type named `u64` that is supposed to be a 64-bit unsigned integer:

```
#ifdef IS_A_32_BIT_CPU
typedef unsigned long long u64; // A 64-bit data type on a 32-bit CPU
#endif
#ifdef IS_A_64_BIT_CPU
typedef unsigned long u64;      // A 64-bit data type on a 64-bit CPU
#endif
```

The rest of our code can now use the type `u64`. Switching from a 32-bit to a 64-bit CPU requires only **two** very small changes in the code (and a recompilation), namely, undefining the symbol `IS_A_32_BIT_CPU` and defining the symbol `IS_A_64_BIT_CPU`. (This can be done without modifying the code by defining the appropriate symbol on the command line that invokes the compiler.)

A `typedef` can also be used in conjunction with the declaration of a `struct`:

```
typedef struct dot // "struct dot" is now known to exist
{
  double x,y; int color;
  struct dot *next; // the type dot is not yet known, so we have to use struct dot, which is known
}
dot; // the name of the new type is dot; it is the same as struct dot
```

We recommend that type names end in `_t` just to distinguish them from valuable and function names. That was not done above just to illustrate that that is not mandatory.

Data types (part 12, sizeof)

The number of bytes used by any variable of type `T` is given by `sizeof(T)`. The parenthesis are optional (to avoid confusion due to the precedence of operators, please always use parenthesis). For example, in the code

```
int i = sizeof(dot); // same as i = sizeof dot;
```

the variable `i` will be initialized with the number of bytes required to store a `dot` in memory.

The number of bytes used by a specific variable named, say, `var`, is given by `sizeof(var)`; again, the parenthesis are optional. The `sizeof` operator does the right thing when the variable is an array (it returns the number of bytes needed to store the entire array), despite the fact that in other places the array name is actually a pointer to its first element.

The argument of the `sizeof` operator may also be an expression. For example, `sizeof(1 + 2)` is the number of bytes needed to store the result of the expression `1 + 2`. Here the parenthesis **must** be used; `sizeof(1 + 2)` is not the same as `sizeof 1 + 2` (why?).

The `sizeof` operator returns an integer with type `size_t`, which is usually a 32-bit data type in 32-bit processors and a 64-bit data type in 64-bit processors (it is usually the number of bytes needed to store a pointer variable). Thus, to avoid an implicit cast, the first example above should have been written as follows:

```
int i = (int)sizeof(dot);
```

[**Homework:** since `a[i]` is converted by the compiler into `*(a + i)` before generating code, what should the type of `i` be on i) 32-bit processors, ii) 64-bit processors?]

Declaration, definition, and scope of variables (part 1)

Variables can be defined either outside or inside a function body. Variables defined outside a function body, called global variables, can be used by more than one function. Variables defined inside a function body, called local variables, can only be used directly by that function; they can, however, be made accessible to other functions **called** by that function via pointers (it is a serious mistake to return the address of a local variable).

In the case of variables declared or defined **outside** a function body there are several cases to consider:

- Declaration of a variable that may, or may not, be defined in the same source code file, as in
  ```
  extern int global_var;
  ```
  The `extern` keyword tells the compiler that the variable may be defined elsewhere.
- Declaration of a variable that is defined in the same source code file, as in
  ```
  int global_var;
  ```
  Without initialization, this is actually a declaration (it will be transformed into a definition if a definition is not found elsewhere in the source code; in that case the memory location where the variable resides is initialized with all zeros).
- Definition of a variable that is defined in the same source code file, as in
  ```
  int global_var = 0;
  ```
- Declaration and definition of a global variable that is visible only to the functions of the same source code file, as in
  ```
  static int global_var;
  ```
  The same line of code in a different source code file gives rise to a **different** variable (it is considered very bad practice to use variables with the same name in this way).

Declarations remains in effect until the end of the source code file.

Declaration, definition, and scope of variables (part 2)

A code block (a compound statement) begins with { and ends with }. A function body has one outermost code block. Inside this outermost code block there may exist more, possibly nested, code blocks.

In the case of variables declared or defined **inside** a code block there are also several cases to consider:

- Declaration of a variable that may, or may not, be defined in the same source code file, as in

  ```
  extern int global_var;
  ```

- Definition of a variable that lives only inside the code block, as in

  ```
  int local_var_1;       // uninitialized
  int local_var_2 = 123; // initialized
  ```

  Without initialization, the variable initially has an unspecified value.
  The variable is created whenever the program enters the code block and is destroyed when it leaves the code block.

- Definition of a persistent variable, visible only on the code block, that retains its value even outside the code block, as in

  ```
  static int local_var_1;       // initialized
                                //   with zeros
  static int local_var_2 = 123; // initialized
  ```

  Each invocation of the function uses the **same** variable.

In all cases, the declaration/definition is forgotten at the end of the code block. It is considered bad practice to give the same name to a local and a global variable.

The C99 language standard allows declaration of variables anywhere inside a code block. In previous versions of the C language standard, variables could only be declared at the beginning of a code block.

Declaration, definition, and scope of variables (part 3)

Each data type may be modified by the use of so-called **qualifiers**. A qualifier may be used by the programmer in a declaration or definition to tell the compiler what operations or optimizations it is allowed to do when copying that variable to or from memory. Of the three qualifiers that the C language currently knows of, `const`, `volatile`, and `restrict`, only the `const` qualifier will be described here. [**Homework:** find what the other two are for.] An object qualified as `const` is constant; the program cannot modify it. In the following code

```
const double pi = 3.14159265358979323846;
```

the value of `pi` can be used in an expression as if it were a `double` variable, but it cannot be changed.

For a pointer type, a qualifier to the right of the asterisk qualifies the pointer itself; a qualifier to the left of the asterisk qualifies the type of object it points to. For example, in the following code some things are allowed and some are not (read the comments).

```
int i;                     // an integer
const int c = 3;           // an integer constant
const int *p;              // a (variable) pointer to a constant integer
                           // p = &i; is allowed, but i cannot be changed via the pointer
                           // *p = 1; is not allowed
                           // p = &c; is allowed
int * const q = &i;        // constant pointer (it must always point to i)
                           // *q = 1; is allowed
int * const r = &c;        // not allowed
const int * const z = &c; // allowed
```

The `const` qualifier is particularly useful to qualify function arguments that are pointers (many functions receive a pointer to a memory region that will not be modified by the function, in which case a `const` will help the compiler produce better code).

Declaration, definition, and scope of variables (part 4)

An example where variables are shadowed (to detect shadowed variables, use the `gcc` compilation flag `-Wshadow`):

```c
 1 // Contents of the file a.c
 2 //
 3
 4 extern int x;     // the global variable x defined in b.c (line 24); this variable is known for the rest of the file
 5 int y = 1;        // the global variable y (visible to all files that declare it as extern)
 6 static int z = 3; // a local (to the file) variable z
 7
 8 int f(int x)      // the global variable x becomes shadowed by the argument to the function
 9 {
10   int t = z + x;  // use of the variable z defined in line 6 and use of the argument to the function
11   {
12     int z = 2;    // this local variable shadows the variable defined in line 6
13     t *= z;       // use of the variable defined in line 12
14     for(int z = 1;z <= 10;z++)  // new variable, also named z, shadows the variable defined in line 12
15       t += z * z; // use of the variable defined in line 14
16     t -= z;       // use of the variable defined in line 12
17   }
18   return t;
19 }
20
21 // Contents of the file b.c
22 //
23
24 int x = 1;        // the global variable x (visible to all files that declare it as extern)
25 static int z;     // a local (to the file) variable z; it is DIFFERENT from the use defined in line 6
26
27 int g(void)
28 {
29   extern int y;   // the global variable y defined in a.c (line 5); this variable is only known inside this code block
30   return y;       // use of the variable defined in line 5
31 }
```

Assignments and expressions (part 1, assignments)

Assignments take the form `LHS = RHS;`, where `LHS` is the so-called lvalue (left value, or, more accurately, location value) and where `RHS` is an expression. The lvalue represents the place where the value of the `RHS` expression will be stored. The following code presents examples of legal and illegal `LHS` lvalues:

```
int a;           a = 3;     // legal
                 a + 1 = 7; // illegal (what is the location of a+1?)
int *pa;         pa = &a;   // legal (the pointer itself has a location)
                 *pa = 7;   // legal
int A[10];       A[3] = a;  // legal
const int c = 3; c = 4;     // illegal (c has a location, but it is not writable)
```

When a pointer is used on the `LHS` to specify a write address, the `LHS` may also modify the pointer if the `++` or `--` operators are used. For example, the following code

```
*++pa = 3; // same as *(++pa) = 3; increment the pointer, and use its new value as the write address
*pa-- = 3; // same as *(pa--) = 3; use the pointer as the write address, and then decrement the pointer
```

is legal, while the code

```
pa++ = &a; // nonsense (are we really trying to store &a in pa and then attempting to increment pa?)
```

is not.

An assignment is in itself an expression, with a value equal to that of the `RHS` (after conversion to the type of the `LHS`). So, it is possible to put several assignments in a chain, as in the following code:

```
int i,j,k;
i = j = k = 3; // same as i = (j = (k = 3));
```

Assignments and expressions (part 2, expressions)

As expression is a sequence of constants, variables, and function calls intertwined with operators that combine them. An expression may perform a mathematical calculation, in which case either we will be interested in recording its value by saving it in a variable (as assignment), or we may be interested to test its value with the purpose of deciding what the program should do next (a conditional jump). An expression may also be useful because of its side effects, as when a function that returns nothing (`void`) is called to do something. The type of an expression is the type of the value that is the result of the expression; it may be `void` if the expression has no value (as in a call to a function that does not return anything). The following are examples of expressions (`i` is an `int`, `d` is a `double` and `exit` is a function than has one `int` argument and that does not return anything):

```
i                    // int
i + '0'              // int
(i << 3) ^ (i & 7)   // int   (the parentheses force i<<3 and i&7 to be evaluated first)
(double)i * d        // double, same as i * d
exit(1)              // void
i && (d == 3.0)      // int
"abc"                // const char *
"abc"[i]             // char
i = 3                // int
d = i = 5            // double
i = d = 2.5          // int
```

The binary arithmetic operators perform an automatic type conversion whenever their two arguments are not of the same type: the argument having the type with smaller range of values is converted to the other. For example, a `char` is converted to an `int` (`char + int` becomes `int + int`), a `signed int` is converted to an `unsigned int` (`signed int - unsigned int` becomes `unsigned int - unsigned int`), and an `int` is converted to a `double` (`int * double` becomes `double * double`).

Assignments and expressions (part 3a, operators)

C has the following operators, in decreasing order of priority:

1. **postfix operators, left to right associativity**

`[]`	array access
`()`	function call
`.`	structure field
`->`	structure field, from a pointer
`++`	post increment; use value, then increment
`--`	post decrement; use value, then decrement

2. **unary operators, right to left associativity**

`++`	pre increment; increment, then use value
`--`	pre decrement; decrement, then use value
`!`	logic negation (0 gives 1, non-zero gives 0)
`~`	bitwise negation
`+`	does nothing (+1 is just 1)
`-`	arithmetic negation
`*`	pointer dereference
`&`	address of

3. **cast operator, right to left associativity**

`(T)`	type conversion; T is a data type

4. **multiplicative operators, left to right associativity**

`*`	multiplication
`/`	division
`%`	remainder

5. **additive operators, left to right associativity**

`+`	addition
`-`	subtraction

6. **shift operators, left to right associativity**

`<<`	shift left
`>>`	shift right

Note: since these are usually used to perform multiplications and divisions by powers of 2 they should have been given a higher priority than addition and subtraction!

7. **relational operators, left to right associativity**

`<`	less than
`<=`	less than or equal to
`>`	larger than
`>=`	larger than or equal to

Note: 1 when true, 0 when false

8. **equality operators, left to right associativity**

`==`	equal to
`!=`	different from

Note: 1 when true, 0 when false

9. **bitwise and, left to right associativity**

`&`	bitwise and

Left to right associativity: the thing on the left is done first. **Homework:** what is the value of `(0 == 0 == 2)`?

Assignments and expressions (part 3b, operators)

10. **bitwise exclusive or, left to right associativity**

`^`	bitwise exclusive or

11. **bitwise or, left to right associativity**

`|`	bitwise or

12. **logical and, left to right associativity**

`&&`	logical and

Note: if the argument on the left is zero, the result is 0 and the argument on the right **is not** evaluated; otherwise the result is 0 if the argument on the right is zero and is 1 if not.

13. **logical or, left to right associativity**

`||`	logical or

Note: if the argument on the left is nonzero, the result is 1 and the argument on the right **is not** evaluated; otherwise the result is 1 if the argument on the right is nonzero and is 0 if not.

14. **conditional operator, right to left associativity**

`?:`	`a ? b : c`	evaluates to `b` if `a` is nonzero and evaluates to `c` if `a` is zero

15. **assignment operators, right to left associativity**

`=`	simple assignment
`+=`	compound assignment (add)
`-=`	compound assignment (subtract)
`*=`	compound assignment (multiply)
`/=`	compound assignment (divide)
`%=`	compound assignment (remainder)
`&=`	compound assignment (bitwise and)
`^=`	compound assignment (bitwise exclusive or)
`|=`	compound assignment (bitwise or)
`<<=`	compound assignment (left shift)
`>>=`	compound assignment (right shift)

Note: the compound assignment `a op= b` where `op` is one of the operators above is equivalent to `a = a op (b)`.

16. **comma operator, right to left associativity**

`,`	discard an expression; start a new one

When in doubt about the priority of an operator, use parentheses! Right to left associativity means that things on the right have precedence over things on the left. For example, `a = b = 3;` means `a = (b = 3);`. Left to right associativity is just the opposite. Expressions with side-effects that affect the same variable, like `j = i++ + ++i;`, are ill-defined because different compilers may choose different orders of evaluation.

Statements (part 1, expression, compound, go to, and return statements)

Statements come in many guises:

- **expression statements**

As expression statement is an expression, possibly empty, followed by a semicolon. The following are valid expression statements:

```
;
i = 2;
i = 3, j = 4;
exit(1);
```

- **compound statements (block statements)**

A compound statement (what we called before a code block) starts with a {, is followed by zero of more declarations or definitions of variables and zero of more declarations of functions, is then followed by zero or more statements, and is terminated by a }. The following code is a valid compound statement:

```
{
  int i = 3 + x;       // x declared elsewhere
  {
    int j = i * i + y; // y declared elsewhere
    k += i + j;        // k declared elsewhere
  }
}
```

- **go to statements** (by some considered harmful)

The go to statement causes an unconditional jump to another statement in the same function. It should be used with **care** and **parsimony**, if at all. The destination of the jump is specified by the name of a label, as in

```
goto x_marks_the_spot;
```

The label itself is a name followed by a colon, as in

```
x_marks_the_spot:
```

The break and continue statements, to be presented below, are disciplined (and disguised) go to statements.

- **return statements**

Return statements are used to end the execution of the current function. It has the form

```
return expression;
```

The expression must be missing if the function does not return anything (declared as returning a `void`). The value of the expression is returned to the caller of the function.

Statements (part 2, if statements)

- **if statements**

An if statement has two possible forms: either

```
if(expression)
  statement_t   // to be executed if the
                //   expression is non-zero
```

or

```
if(expression)
  statement_t   // to be executed if the
                //   expression is non-zero
else
  statement_f   // to be executed if the
                //   expression is zero
```

Care must be taken if several if statements are nested and the else part is present in some of them. For example, in the following code

```
if(i >= 0)
  if(i > 0)
    j = 1;
  else
    j = 0;
else
  j = -1;
```

the first else belongs to the second if and the second else belongs to the first if, as suggested by the indentation of the code. (**Advice:** always indent correctly your code.) This is so because the `statement_t` of the first if is `if(i > 0) j = 1; else j = 0;`. If in doubt use curly braces to transform a statement into a compound statement:

```
if(i >= 0)
{
  if(i > 0)
    j = 1;
  else
    j = 0;
}
else
  j = -1;
```

- **loop statements**

There are three kinds of loop statements: for, while, and do-while statements. There is one way to quickly get out of a loop: the break statement. There is one way to quickly jump to the next loop iteration: the continue statement.

Statements (part 3, for, while, do-while, break, and continue statements)

- **for statements**

A for statement has the following form:

```
  for(expression1;expression2;expression3)
    body_statement
```

It is equivalent to

```
  {
      expression1;
loop: if((expression2) == 0) goto end;
      body_statement
next: expression3;
      goto loop;
end:  ;
  }
```

- **while statements**

A while statement has the following form:

```
  while(expression)
    body_statement
```

It is equivalent to

```
  {
loop: if((expression) == 0) goto end;
      body_statement
next: goto loop;
end:  ;
  }
```

- **do-while statements**

A do-while statement has the following form:

```
  do
    body_statement
  while(expression);
```

It is equivalent to

```
  {
loop: body_statement
next: if((expression) != 0) goto loop;
end:  ;
  }
```

Each loop statement will get its own private label names. A break statement inside the body of a loop statement amounts to a `goto end;`, i.e., it forces an exit of the loop statement. A continue statement inside the body of a loop statement amounts to a `goto next;`, i.e., the remaining statements of the body of the loop are skipped.

Statements (part 4, switch statements)

- **labeled statements**

All statements can be labeled, i.e., they can start with a label. There are three forms of a labeled statement:

```
label_name:            statement
case const_expression: statement
default:               statement
```

The first form is used by go to statements. The other two are used by switch statements.

- **switch statements**

A switch statement has the form:

```
  switch(int_expression)
    body_statement
```

It is usual, but not necessary, for `body_statement` to be a compound statement. The switch statement works as follows. First, `int_expression` is evaluated. If its value matches the `const_expression` value of one of the case statements, the program jumps to that statement. If none of the cases match, and if there is a default label, the program jumps to the corresponding default statement. Otherwise, the program skips the entire switch statement. A break statement transfers execution to the end of the switch statement. The following code presents an example of a switch statement:

```
  switch(c)
  {
    case 't':
      k = 1;
      do_t();
      break;  // terminate the switch
    default:  // the default can be anywhere
      k = 2;
      break;  // terminate the switch
    case 'z': // no break; do next statements
    case 'x':
      do_x(); // no break; do next statements
    case 'X':
      k = 3;
      break;  // terminate the switch
  }
```

The following code is valid but a bit weird (the switch can be replaced by an if statement):

```
  for(i = j = 0;i < 10;i++)
    switch(i % 3)
      case 1: j += i;
```

Functions (part 1, prototypes)

A function definition is composed of two parts, a function header, which specifies the function name, its return type, and its arguments and their types, and a function body, which must take the form of a compound statement. A function declaration (a so-called function prototype) is just the function header, followed by a semicolon, and possibly preceded by the keyword `extern`.

It is not possible to define a function inside another function. Just like variables, it is possible to declare functions at the beginning of a compound statement.

The modern form of the header of a function (there exists an older form, but nowadays no one uses it) has the following form

```
qualifier type function_name( parameter_declarations )
```

The `qualifier` may be absent. The `parameter_declarations` may either be `void`, if the function does not take any arguments, or be composed by one or more individually declared arguments (type and name), separated by comas. The following are examples of function prototypes (function headers terminated by a semicolon):

```
int main(void);
int main(int argc,char **argv); // same as int main(int argc,char *argv[]);
extern double sqrt(double x);
static int F(int n);
inline static double sqrt_2(void);
```

(Actually, it is possible to omit the argument names in function prototypes, but we prefer to include them, as their name may shed some light about what they stand for.)

Functions (part 2, parameters)

The parameters of a function behave (in the function body) as if they were ordinary variables, i.e., as if they had been declared and initialized at the very beginning of the compound statement that constitutes the function body. They are passed to the function **by value**, i.e., a copy of each argument is made when the function is called. Thus, changes to the function arguments inside the function, which is allowed, are made on **the copy**. For example, the call `swap(x,y)` to the function

```
void swap(int x,int y)
{
  int tmp = x;
  x = y;
  y = tmp;
}
```

will exchange `x` with `y` only inside the `swap` function. To actually reflect the exchange outside of this function, in C one has to use pointers and pass to the (modified) function the addresses of what we want to exchange. In this case, the function call becomes `swap(&x,&y)` and the function becomes

```
void swap(int *x,int *y)
{
  int tmp = *x;
  *x = *y;
  *y = tmp;
}
```

Note that since the name of an array is a pointer to its first element, in the case of arrays what is passed to the function is a pointer. So, **arrays** are automatically passed **by reference**. No copy of the entire array is made. On the other hand, **structures** and unions are passed by **value**, so if you put an array inside a structure you can actually pass (in disguise) an array by value to a function.

Functions (part 3, qualifiers)

It is possible to declare that a function is `static`. This means that its name is only known to the current file being compiled. A function with the same name can be defined in another source code file. It is very bad practice to have static functions (or one non-static and the others static) with the same name in different source code files. If the source code of a program is distributed among several source code files, using a static function is a great method to hide it from the rest of the source code (for example, to make sure that it is not used inappropriately).

It is also possible to declare that a function is `inline`. This means that the compiler will try to replace all calls to the function by copies of its code. The program will be somewhat larger, as the function code may appears in several places, but it will also be faster, as there will be no function call overhead. If the function body is too large, the compiler may silently refuse to inline a function.

The function called `main` is the entry point of the program. It can be defined to either have no arguments, as in

```
int main(void);
```

or it can be defined to have two arguments, as in

```
int main(int argc,char **argv); // same as int main(int argc,char *argv[]);
```

In the second case, the first argument is the number of command line arguments with which the program was invoked and the second is an array of pointers to strings with the text of the arguments. For example:

```
#include <stdio.h>
int main(int argc,char **argv)
{                              // this program prints its arguments, one per line
  for(int i = 0;i < argc;i++)  // definition of a variable anywhere (a la C++) is a c99 feature
    printf("%s\n",argv[i]);    // compile this using "cc -Wall -O2 -std=c99 main.c"
  return 0;
}
```

Functions (part 4, variable number of arguments)

It is possible for a function to have a variable (i.e., optional) number of arguments. It is the responsibility of the programmer to determine how many arguments were actually provided in every function call. There exists a standard mechanism (c.f. `stdarg.h`) to fetch the value of the next variable argument given its type. The programmer has to know the type of the argument. For example, this can be provided by a previous argument (as done, for example, in the `printf` function; see next slide).

A function has a variable number of arguments if its last argument (the optional part) is specified as .... It must have at least one standard argument. The following code illustrates how a variable number of arguments is specified and used (in this example all extra arguments all integers, with a special value to signal the end):

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

int sum(int terminator, ...)
{
  va_list a;                            // standard way to access the extra arguments
  int sum,n;

  va_start(a,terminator);               // the extra arguments start after the terminator argument
  for(sum = 0;;sum += n)
    if((n = va_arg(a,int)) == terminator)  // get the next int from the argument list
      break;
  va_end(a);
  return sum;
}

int main(void)
{
  printf("%d\n",sum(-1,3,4,7,3,-1));        // should print 17 (3+4+7+3)
  return 0;
}
```

Standard library functions

The C language comes equipped with a relatively large set of predefined functions, declared in so-called header files, and stored in library archives. Among them are functions to read and write data, declared in `stdio.h`, such as

```
int printf(const char *format, ...);                              // write formatted data
int scanf(const char *format, ...);                               // read formatted data
FILE *fopen(const char *path, const char *mode);                  // open a file
int fclose(FILE *fp);                                             // close a file
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);  // raw read
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream); // raw write
int fprintf(FILE *stream, const char *format, ...);               // formatted write
```

functions to allocate and free memory, and to terminate a program, declared in `stdlib.h`, such as

```
void *malloc(size_t size);                                        // allocate memory
void free(void *ptr);                                             // free memory
void *calloc(size_t nmemb, size_t size);                          // zero-allocate memory
void *realloc(void *ptr, size_t size);                            // resize an allocation
void exit(int status);                                            // terminate
```

and functions to compute transcendental mathematical function, declared in `math.h`, such as

```
double sqrt(double x);
double sin(double x);
double cos(double x);
```

Use the help system of your computer (`man` command on GNU/Linux), to get a full description of what a given function does. This web page, which has links to documents describing in full the GNU implementation of the C standard library functions, is also quite useful.

Your mother was a punch card reader and your father smelt of static C libraries.

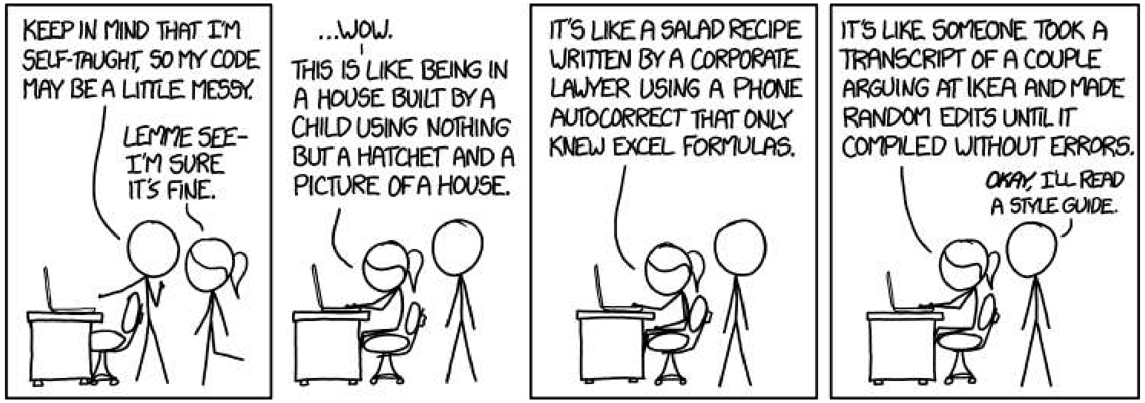
Coding style

Some advice:

- use a consistent coding style; in particular, always indent properly your code,
- do not put too much stuff in a single function,
- use reasonable function and variable names (are you a camelCase fan or a snake_case fan?),
- don't comment obvious code,
- explain, with a comment, each clever trick used in the program (see, for example, the explanation of what the expression `(d+1)|1` does in the `factor.c` program),
- try very hard not to write code what would be admired and envied in the international obfuscated C code contest,
- the C programming language, the representation of numbers, and the memory layout of data can be abused to produce plausible deniability bugs in a program, as illustrated in The Underhanded C Contest, so, don't do it, and
- try very hard not to be like the guy in the three xkcd cartoons on the next page.

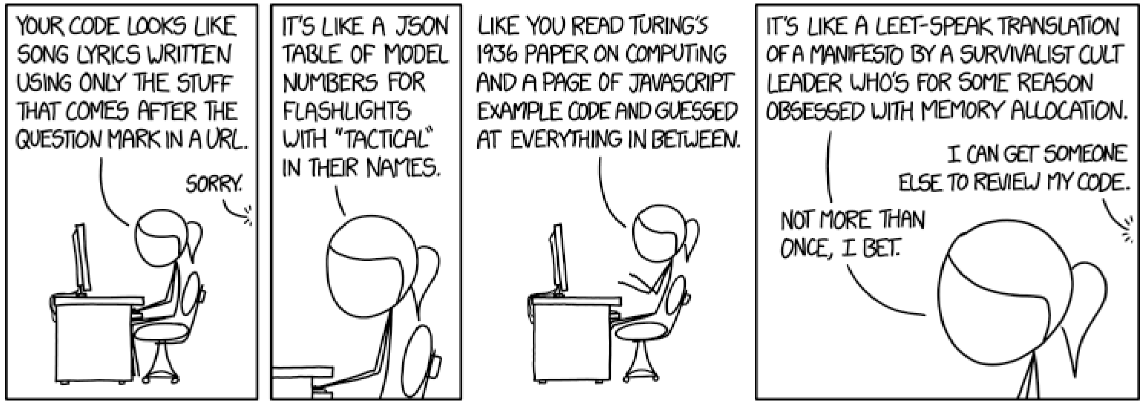
Coding style (xkcd cartoons)

Code Quality (spoiler)


KEEP IN MIND THAT I'M SELF-TAUGHT, SO MY CODE MAY BE A LITTLE MESSY.
LEMME SEE– I'M SURE IT'S FINE.
...WOW.
THIS IS LIKE BEING IN A HOUSE BUILT BY A CHILD USING NOTHING BUT A HATCHET AND A PICTURE OF A HOUSE.
IT'S LIKE A SALAD RECIPE WRITTEN BY A CORPORATE LAWYER USING A PHONE AUTOCORRECT THAT ONLY KNEW EXCEL FORMULAS.
IT'S LIKE SOMEONE TOOK A TRANSCRIPT OF A COUPLE ARGUING AT IKEA AND MADE RANDOM EDITS UNTIL IT COMPILED WITHOUT ERRORS.
OKAY, I'LL READ A STYLE GUIDE.

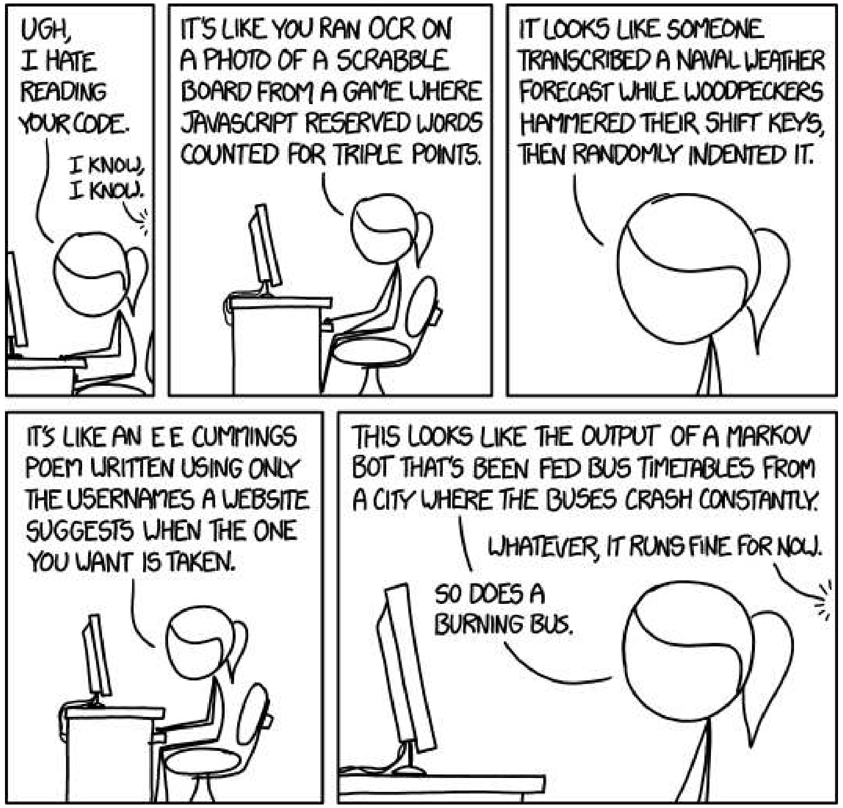
It's like you tried to define a formal grammar based on fragments of a raw database dump from the QuickBooks file of a company that's about to collapse in an accounting scandal.

Code Quality 3 (spoiler)


YOUR CODE LOOKS LIKE SONG LYRICS WRITTEN USING ONLY THE STUFF THAT COMES AFTER THE QUESTION MARK IN A URL.
SORRY.
IT'S LIKE A JSON TABLE OF MODEL NUMBERS FOR FLASHLIGHTS WITH "TACTICAL" IN THEIR NAMES.
LIKE YOU READ TURING'S 1936 PAPER ON COMPUTING AND A PAGE OF JAVASCRIPT EXAMPLE CODE AND GUESSED AT EVERYTHING IN BETWEEN.
IT'S LIKE A LEET-SPEAK TRANSLATION OF A MANIFESTO BY A SURVIVALIST CULT LEADER WHO'S FOR SOME REASON OBSESSED WITH MEMORY ALLOCATION.
I CAN GET SOMEONE ELSE TO REVIEW MY CODE.
NOT MORE THAN ONCE, I BET.

It's like a half-solved cryptogram where the solution is a piece of FORTH code written by someone who doesn't know FORTH.

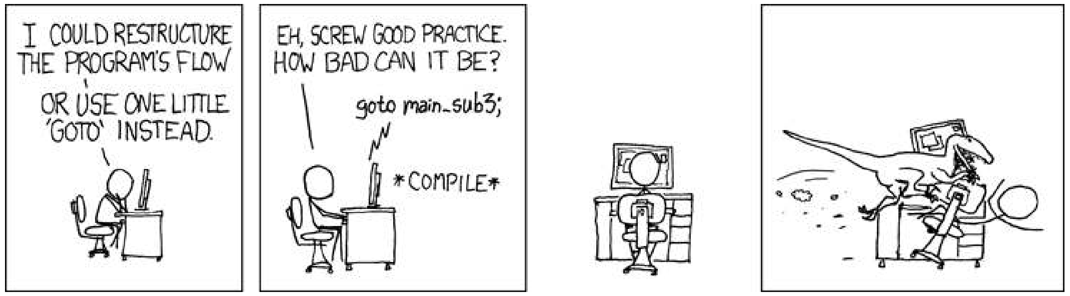
Code Quality 2 (spoiler)


UGH, I HATE READING YOUR CODE.
I KNOW, I KNOW.
IT'S LIKE YOU RAN OCR ON A PHOTO OF A SCRABBLE BOARD FROM A GAME WHERE JAVASCRIPT RESERVED WORDS COUNTED FOR TRIPLE POINTS.
IT LOOKS LIKE SOMEONE TRANSCRIBED A NAVAL WEATHER FORECAST WHILE WOODPECKERS HAMMERED THEIR SHIFT KEYS, THEN RANDOMLY INDENTED IT.
IT'S LIKE AN E E CUMMINGS POEM WRITTEN USING ONLY THE USERNAMES A WEBSITE SUGGESTS WHEN THE ONE YOU WANT IS TAKEN.
THIS LOOKS LIKE THE OUTPUT OF A MARKOV BOT THAT'S BEEN FED BUS TIMETABLES FROM A CITY WHERE THE BUSES CRASH CONSTANTLY.
WHATEVER, IT RUNS FINE FOR NOW.
SO DOES A BURNING BUS.

I honestly didn't think you could even USE emoji in variable names. Or that there were so many different crying ones.

Three more xkcd cartoons

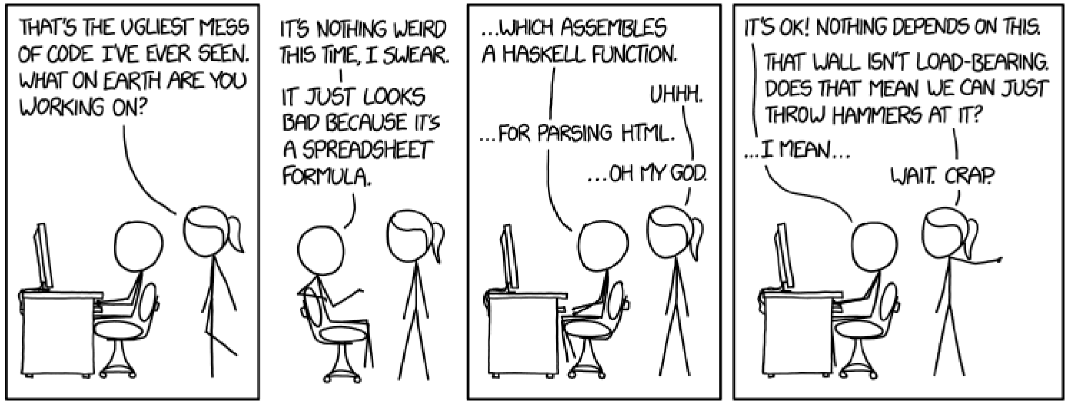
Goto (no spoiler)



I COULD RESTRUCTURE THE PROGRAM'S FLOW
OR USE ONE LITTLE 'GOTO' INSTEAD.
EH, SCREW GOOD PRACTICE. HOW BAD CAN IT BE?
goto main_sub3;
COMPILE

Neal Stephenson thinks it's cute to name his labels 'dengo'.

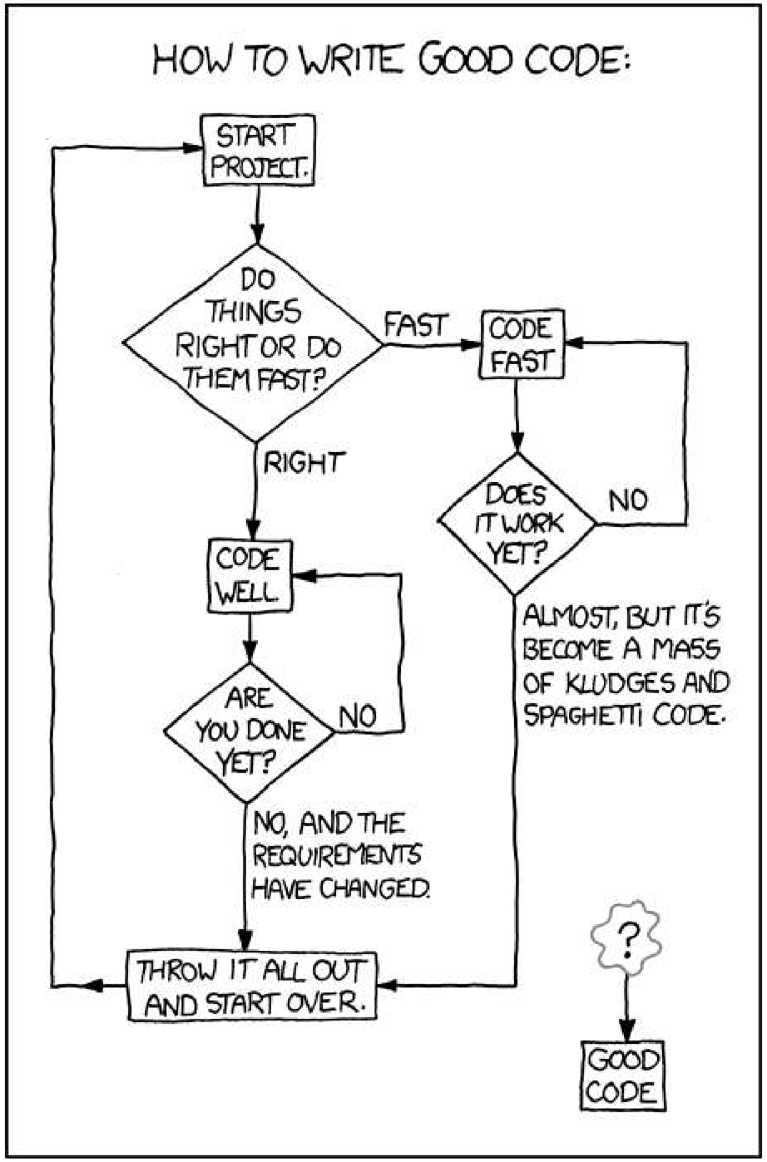
Bad code (spoiler)



THAT'S THE UGLIEST MESS OF CODE I'VE EVER SEEN. WHAT ON EARTH ARE YOU WORKING ON?
IT'S NOTHING WEIRD THIS TIME, I SWEAR.
IT JUST LOOKS BAD BECAUSE IT'S A SPREADSHEET FORMULA.
...WHICH ASSEMBLES A HASKELL FUNCTION.
UHHH.
...FOR PARSING HTML.
...OH MY GOD.
IT'S OK! NOTHING DEPENDS ON THIS.
THAT WALL ISN'T LOAD-BEARING. DOES THAT MEAN WE CAN JUST THROW HAMMERS AT IT?
...I MEAN...
WAIT. CRAP.

"Oh my God, why did you scotch-tape a bunch of hammers together?"
"It's ok! Nothing depends on this wall being destroyed efficiently."

Good code (spoiler)



HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
RIGHT
CODE FAST
DOES IT WORK YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
CODE WELL
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

You can either hang out in the Android Loop or the HURD loop.

Some useful web sites

Here is a list of some useful web sites related to C and C++ (in random order):

- MIT OpenCourseWare: for a second opinion (about C and C++)
- C tutor: for those wishing to learn C
- online compiler and debugger for C and C++: for those with some experience in C or C++
- Interactive C tutorial: for an interactive way of learning C
- project Euler: some programming problems
- Google Code Jam: more programming problems
- glibc manual: for those wishing to use the (GNU version of the) standard C library
- C standard: for those that what to know every detail
- common weakness enumeration, in particular, top 25 most dangerous software weaknesses: for those that want to become good programmers
- C reference: a gentle summary of the C programming language
- C tutorial: a tutorial that explain the C programming language using examples
- another C tutorial: another C tutorial
- C tutor: write and execute simple C code in the browser
- C++ reference: a gentle summary of the C++ programming language
- C++ tutorial: a tutorial that explain the C++ programming language using examples
- another C++ tutorial: another C++ tutorial
- C++ tutor: write and execute simple C++ code in the browser
- Google's C++ tutorial: Google's way of teaching C++
- Google's C++ style guide: guidelines abount how to write and format C++ code