An introduction to C

Table of Contents

An introduction to C

An introduction to C − version 2004−05−03

Copyright (c) 1998−2004 by Tom Torfs
tom . torfs@imec . be (preferred), tom . torfs@chello . be

Any suggestions for additions, improvements and/or error corrections are welcome at any of the addresses mentioned above.

All the source code in this document should be distributed together with it as plain text files. If you don't have them, you can obtain them, and the latest version of this document, at:

http://www.geocities.com/tom_torfs/c.html

0. Contents

1. Overview and rationale

This document is intended to give people who are interested in learning C, whether they already know another programming language or not, a quick introduction to the language. It does not pretend to be complete, but it should get you familiar with most concepts of the C language. It is not intended to replace a good introductory book on the subject; on the contrary, it is probably best used together with such a book; if you haven't programmed before some of the explanations in this introduction may be a bit short and getting more detailed information from a book would often be useful.

This introduction discusses the standard C language as defined by the International Standards Organization [*], also commonly referred to as "ANSI C" (the American National Standards Institute had standardized the language before ISO). It does not discuss platform/compiler specific extensions such as accessing a screen in either text or graphics mode, interacting with external devices, direct access to memory, using operating system features etc.

[*] ANSI/ISO 9899−1990 aka C89/C90; normative addenda are not discussed here; new C99 features may be referred to occasionally; the implementation is assumed to be hosted

Rationale

It has been − rightfully − reported that the material in this tutorial may not be very easy reading for a complete beginner to the C language, mainly because I try to be as complete as possible even in early stages of this tutorial.
This has two reasons: first, this tutorial can also be useful as reference material, e.g. to look up what printf() specifier can be used for what purpose, etc.
The second, more fundamental reason, is that the principle objective of this tutorial was to prefer accuracy over simplicity. In other words, I feel there are already enough tutorials and introductory books that don't adhere to the standard in several cases to keep things simple. This has the consequence that some people who learned C from such a tutorial/book will have a hard time adjusting their programming habits to write conforming code whenever possible. Since this tutorial is partly intended as a "cure" for those people, it requires some degree of accuracy which also means that it may not read very fluently if you're completely new to C. This only confirms the above statement that this tutorial is not meant to be used on its own, but rather as a complement to a good introductory book.

When reading this tutorial, it is OK to skip certain descriptions and listings which you feel are not required yet. Examples may include the complete list of printf() specifiers, standard types, etc.

2. Hello, world!

2.1. The Hello, world! program

```
/* prog2−1.c: Hello, world! */
```

```
#include <stdio.h>

int main(void)
{
   puts("Hello, world!");
   return 0;
}
```

If you compile and run this small program [*], you'll see that it writes the message Hello, world! onto your screen (or printer, or whatever output device you're using) and then ends the program successfully.

[*] see your compiler documentation / online help for instructions

Also, keep in mind that you are free to choose your own formatting style. E.g. the above could have been written as: (the reasons it hasn't should be obvious)

```
/* prog2-1.c: Hello, world! */
#include <stdio.h>
int main(void) {puts("Hello, world!"); return 0;}
```

2.2. Comments

Now we'll analyze this program:

```
/* prog1.c: Hello, world! */
```

Everything that is inside /* and */ is considered a comment and will be ignored by the compiler. You must not include comments within other comments, so something like this is not allowed: [*].

```
/* this is a /* comment */ inside a comment, which is wrong! */
```

[*] your compiler may allow this, but it's not standard C

You may also encounter comments after //. These are C++ comments and while they will be allowed in the next revision of the C standard, you shouldn't use them yet. They are more limited than the flexible /* */ comments anyway.

2.3. #include

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

In C, all lines that begin with # are directives for the preprocessor, which means that all these directives will be processed before the code is actually compiled. One of the most commonly used preprocessor directives is **#include**. It has two forms. The first form is:

```
#include <header>
```

This form should only be used to include a standard header, such as stdio.h. This gives you access to all the goodies in the header. In this program we are going to use puts() (see below), and to be able to do that we need to [*] include the standard header stdio.h (which stands for standard input/output header).

[*] in some cases you may get away with not including the right standard header, but in other cases it may cause nasty things to happen (ranging from a simply malfunctioning program to a core dump or even a system crash: this is called undefined behaviour,

in other words anything might happen); basically you should always include the right headers.

The following standard headers are available: (see 16. Overview of the standard library for a detailed discussion of them)

Commonly used headers:

```
stdio.h    input/output
stdlib.h   general utilities
string.h   string handling
time.h     date and time
math.h     mathematics
ctype.h    character handling
```

Standard definitions:

```
stddef.h   standard type/macro definitions
limits.h   implementation limits
float.h    floating−point limits
errno.h    error handling
assert.h   diagnostics
```

More advanced headers:

```
stdarg.h   variable arguments
setjmp.h   non−local jumps
signal.h   signal handling
locale.h   localization
```

The second form is:

```
#include "file"
```

This directive will be replaced with the contents of file. Usually this form is used for non−standard headers that are part of your program (see 15. Modular approach).

2.4. Functions and types

Now onto the next line: (the { and } braces belong to this line as well)

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

This is a function definition. A function is a part of a program that can be called by other parts of the program. A function definition always has the following form:

```
type name(parameters)
{
   /* the function code goes here */
}
```

The function will return a value of type 'type' to the caller. C supports the following types: (note that the ranges indicated are minimum ranges, they may (and will!) be larger so you shouldn't rely on them having a certain size)

Integer types: (non−fractional numbers)

```
signed char    minimum range: −127..+127
unsigned char  minimum range: 0..255
signed short   minimum range: −32767..+32767
unsigned short minimum range: 0..65535
signed int     minimum range: −32767..+32767
unsigned int   minimum range: 0..65535
signed long    minimum range: −2147483647..+2147483647
unsigned long  minimum range: 0..4294967295
```

The type **char** may be equivalent to either **signed char** or **unsigned char** (that depends on your compiler), but it is always a separate type from either of these. Also notice that in C there is no difference between storing characters or their corresponding numerical values in a variable, so there is also no need for a function to convert between a character and its numerical value or vice versa (this is different from languages like Pascal or BASIC).
For the other integer types, if you omit **signed** or **unsigned** the default will be **signed**, so e.g. **int** and **signed int** are equivalent.
The type **int** must be greater than or equal to the type **short**, and smaller than or equal to the type **long**. This depends on your compiler and operating system. If you simply need to store some values which are not enormously large it's often a good idea to use the type **int**; it usually is the size the processor can deal with the easiest, and therefore the fastest.
For the next revision of the C standard, an even larger type '**long long**' is proposed. You shouldn't use it yet for now. If you need that large values, you may want to use floating−point types.

Floating point types: (fractional numbers)

```
float       minimum range: +/− 1E−37..1E+37  minimum precision: 6 digits
double      minimum range: +/− 1E−37..1E+37  minimum precision: 10 digits
long double minimum range: +/− 1E−37..1E+37  minimum precision: 10 digits
```

With several compilers **double** and **long double** are equivalent. That combined with the fact that most standard mathematical functions work with type **double**, is a good reason to always use the type **double** if you have to work with fractional numbers.

Special−purpose types: (don't worry too much about these yet)
Commonly used ones:

```
size_t        unsigned type used for storing the sizes of objects in bytes
time_t        used to store results of the time() function
clock_t       used to store results of the clock() function
FILE          used for accessing a stream (usually a file or device)
```

Less commonly used ones:

```
ptrdiff_t     signed type of the difference between 2 pointers
div_t         used to store results of the div() function
ldiv_t        used to store results of ldiv() function
fpos_t        used to hold file position information
```

More advanced ones:

```
va_list      used in variable argument handling
wchar_t      wide character type (used for extended character sets)
sig_atomic_t used in signal handlers
```

```
jmp_buf      used for non-local jumps
```

If the function does not return anything you can use the pseudo−type **void** as the return value type (this is not allowed for main(), see below).

The function name follows the rules for all names (formally: identifiers): it must consist entirely of letters (uppercase and lowercase are different! [*]), digits and underscores (_), but may not begin with a digit.

[*] during linking, case differences in external names might be ignored; so it's often not a good idea to use 2 variable/function names that only differ in their case (this would also reduce clarity)

The parameter list may be **void** in which case the function won't take any parameters. Otherwise it should be a list of the following form:

```
type1 name1, type2 name2, type3 name3     etc.
```

The possible types are the same as for the return value, and the names follow the same rules as those for the function name. An example:

```
double myfunction(int foo, long bar)
{
}
```

Is a function called 'myfunction' that accepts a parameter of type **int** and a parameter of type **long** and returns a value of type **double**

The parameters' values will be filled in by the callers. The above function might be called as follows:

```
myfunction(7,10000);
```

In this case the return value is ignored. If it's needed we may assign it to a variable as follows: (see 3. Using variables)

```
somevariable = myfunction(7,10000);
```

A function that takes no parameters (which has **void** as its parameter list) would be called as follows:

```
myfunction();
```

Notice that unlike in e.g. Pascal you can't do something like: [*]

```
myfunction;             /* wrong! */
```

[*] This will probably compile, but it does something entirely different than you think (basically, it does nothing).

Here's an example of a situation where using a function provides advantages, apart from the readability advantage that using a function instead of lumping all code together always provides.

```
   do_something();
   check_error_condition();

   do_something_else();
   check_error_condition();
   etc.
```

Suppose the do_something() and do_something_else() functions perform some useful tasks. However, it is possible that an error occurs while performing these tasks, and if that happens these functions set some sort of error condition to a specific value. By looking at this value, the calling code can then print an appropriate error message and/or take appropriate actions. But if the code to do this would have to be copied everywhere a function like do_something() is used, the program would get enormously bloated, not to mention the fact that if something has to be changed in this error handling code, it may have to be changed in hundreds of places. Now, if this code is put in its own function, say check_error_condition(), all that has to be done in all these places is a call to this function. The code stays compact and readable, and if something has to be changed in the error handling code, it only has to be changed in one place.

main() is a very special function: it is the function that is called at the beginning of our program's execution. Because it's so special we are not free to choose its return value or parameters. Only two forms are allowed:

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

Which you should use if you don't need access to command line arguments.

```
int main(int argc, char *argv[])
{
}
```

Which allows access to the command line arguments (see 10.5. Example: using command line arguments).

In some source code you may find other definitions of main(), such as returning **void** or taking 3 parameters etc. These may work on certain compilers but they are not standard so you should avoid them.

The actual function code (called the function body) goes inside the { and } braces. In this case the first line of the function body is:

```
   puts("Hello, world!");
```

The puts() function is declared in the standard header stdio.h as follows:

```
int puts(const char *s);
```

This is not a function definition, because it is not followed by { and } braces but by a semicolon, which in C means 'end of statement'. Instead, it is a function declaration (also known as 'prototype') which informs the compiler of the return value and parameter types of this function. The actual function definition can be found somewhere in your compiler's libraries, but you needn't worry about that because the declaration provides all the information you need in order to be able to call this function.

The return value of the puts() function is an **int**, and its meaning is rather straightforward: it returns EOF (which is a negative value defined in stdio.h) if an error occurs, or a nonnegative value if it's successful. We ignore this return value in our small program, but in 'real' code it's often a good idea to check these return values, and if an error occurs deal with it properly.

But the parameter, that's something else. To understand it, we must first know what puts() does: it writes a string (= text) to the standard output device (usually the screen, but it may be another device or a file) and then moves on to the next line. So obviously this '**const char** *s' parameter must accept the string we want to print.

If you look back to the types listed above, you will notice that there is no string type among them. That is because C doesn't have a special string type. A string in C is just a series of chars, and the end of the string is indicated by a **char** with value 0. Such a series is called an array. But instead of passing the array to the puts() function, a pointer to it is passed. The pointer is indicated by the * and the **const** means that the puts() function will not modify the string we pass it. If this seems like Chinese for now (provided you don't live in China, of course), don't worry: this will be more thoroughly explained later in this document (see 10. Arrays, strings and pointers). All you need to know for now is that the above declaration means that puts() will take a string as parameter, and that that string will not be modified.

In our program, we pass puts() the parameter "Hello, world!". The double quotes indicate that this is a string literal (which basically comes down to a string that may not be modified [*]). Since puts() accepts a string that it will not modify, this works out perfectly. If we would have attempted to call puts() with e.g. a number as parameter, the compiler would most likely have complained.

[*] Attempting to modify a string literal will cause undefined behaviour. Some compilers/operating systems may allow string literals to be modified, but this will fail miserably on others (or even the same ones with different compiler options set), so should definitely never be done.

You must be careful to use double quotes ("), which indicate a string literal, because single quotes (') indicate a character literal (e.g. 'A') and these are not interchangeable. [*]

[*] Another peculiarity is that character literals in C are not of type **char**, but of type **int**. This is different from C++.

String and character literals may contain special escape codes to perform some implementation−defined actions:

```
\a alert (causes an audible or visual alert when printed)
\b backspace
\f formfeed (may clear the screen on some systems, don't rely on this)
\n newline (goes to the beginning of the next line; will be automatically
            translated to/from \r\n or another (if any) end−of−line
            sequence on systems where this is necessary)
\r carriage return (goes to the beginning of the current line)
\t horizontal tab
\v vertical tab
\\ backslash (be careful when using backslashes in literals!)
\' single quote (useful inside character literals)
\" double quote (useful inside string literals)
\? question mark (useful to avoid unwanted trigraph translation when two
                  question marks are followed by certain characters)
\<octal digits> character value in octal
\x<hex digits>  character value in hexadecimal
```

2.5. return

The next, and final, line of our small program is:

```
   return 0;
```

return exits a function and unless the function has return type **void** it must be followed by a value corresponding to its return type.

As we mentioned above, main() always returns an **int** value. In the case of main(), this value indicates to the operating system whether the program ended successfully or because of an error. Success is indicated by returning the value 0. Also, the constants EXIT_SUCCESS and EXIT_FAILURE may be used; they are defined in the standard header stdlib.h. [*]

[*] Usually, other return values than 0, EXIT_SUCCESS or EXIT_FAILURE are allowed as well for the return value of main(), but this is not standard so you should avoid it unless you have a specific reason to do so, provided you're aware this may restrict the range of operating systems your program will work on.

Note that the above is specific to the main() function; other functions that you write are of course not subject to these restrictions, and may take any parameters and return any value they like. [*]

[*] Some other functions, like the compare function for qsort() or the function passed to atexit() have to have a certain form as well, but those few exceptions are of a more advanced nature.

3. Using variables

3.1. The 42 program

```
/* prog3-1.c: 42 */

#include <stdio.h>

int main(void)
{
   int answer = 42;

   printf("And the answer is: %d\n", answer);
   return 0;
}
```

When compiled and run, this program should write the following onto your output device:

```
And the answer is: 42
```

3.2. Defining variables

The first new thing that stands out is the first line of the body of main():

```
   int answer = 42;
```

This line defines a variable named 'answer' of type **int** and initializes it with the value 42. This might also have been written as:

```
   int answer;         /* define uninitialized variable 'answer' */

   /* and after all variable definitions: */

   answer = 42;        /* assigns value 42 to variable 'answer' */
```

Variables may be defined at the start of a block (a block is the piece of code between the braces { and }), usually this is at the start of a function body, but it may also be at the start of another type of block (unlike in C++ where variables may be defined anywhere in the code).

Variables that are defined at the beginning of a block default to the '**auto**' status. This means that they only exist during the execution of the block (in our case the function). When the function execution begins, the variables will be created but their contents will be undefined (unless they're explicitly initialized like in our example). When the function returns, the variables will be destroyed. The definition could also have been written as:

```
   auto int answer = 42;
```

Since the definition with or without the **auto** keyword is completely equivalent, the **auto** keyword is obviously rather redundant.

However, sometimes this is not what you want. Suppose you want a function to keep count of how many times it is called. If the variable would be destroyed every time the function returns, this would not be possible. Therefore it is possible to give the variable what is called **static** duration, which means it will stay intact during the whole execution of the program. For example:

```
   static int answer = 42;
```

This initializes the variable answer to 42 at the beginning of the program execution. From then on the value will remain untouched; the variable will not be re−initialized if the function is called multiple times!

Sometimes it is not sufficient that the variable be accessible from one function only (and it might not be convenient to pass the value via a parameter to all other functions that need it), but you need access to the variable from all the functions in the entire source file [*] (but not from other source files).

[*] The term the standard uses is "translation unit". On most implementations such a translation unit is simply a text file containing source code, often marked with a ".c" extension or something similar. Therefore in the rest of this document the term "source file" will be used where "translation unit" is meant.

This can also done with the **static** keyword (which may be a bit confusing), but by putting the definition outside all functions. For example:

```
#include <stdio.h>

static int answer = 42; /* will be accessible from entire source file */

int main(void)
{
   printf("And the answer is: %d\n", answer);
   return 0;
}
```

And there are also cases where a variable needs to be accessible from the entire program, which may consist of several source files (see 15. Modular approach). This is called a global variable and should be avoided when it is not required. This is also done by putting the definition outside all functions, but without using the **static** keyword:

```
#include <stdio.h>

int answer = 42; /* will be accessible from entire program! */
int main(void)
{
   printf("And the answer is: %d\n", answer);
   return 0;
}
```

There is also the **extern** keyword, which is used for accessing global variables in other modules. This is explained in 15. Modular approach).

There are also a few qualifiers that you can add to variable definitions. The most important of them is **const**. A variable that is defined as **const** may not be modified (it will take up space, however, unlike some of the constants in e.g. Pascal; see 13. Preprocessor macros/conditionals if you need an equivalent for those). For example:

```
#include <stdio.h>

int main(void)
{
   const int value = 42; /* constant, initialized integer variable */

   value = 100;    /* wrong! - will cause compiler error */

   return 0;
}
```

Then there are two more modifiers that are less commonly used:

The **volatile** modifier requires the compiler to actually access the variable everytime it is read; i.o.w. it may not optimize the variable by putting it in a register or so. This is mainly used for multithreading and interrupt processing purposes etc., things you needn't worry about yet.

The **register** modifier requests the compiler to optimize the variable into a register. This is only possible with **auto** variables and in many cases the compiler can better select the variables to optimize into registers itself, so this keyword is obsolescent. The only direct consequence of making a variable **register** is that its address cannot be taken (see 10.4. Addresses and pointers).

3.3. printf()

Then there is the line:

```
   printf("And the answer is: %d\n", answer);
```

printf(), like puts(), is a function that is declared in stdio.h. It also prints text to the standard output device (often the screen). However, there are two important differences:

1) While puts() automatically adds a newline (\n) to the text, this must be done explicitly when using printf() [*]

[*] If the newline is omitted, the output may not show up. If you really want no newline, you must add fflush(stdout); afterwards. (see 11.4. Using the standard I/O streams)

2) While puts() can only print a fixed string, printf() can also be used to print variables etc. To be able to do that, printf() accepts a variable number of parameters (later we'll discuss how to write such functions yourself,

see 14. Variable argument functions). The first parameter is the format string. In this string % signs have a special meaning (so if you need a % sign to be printed using printf() you must double it, i.e. use %%): they are placeholders for variable values. These variables are passed as parameters after the format string. For every variable passed, there must be a corresponding format specifier (% placeholder) in the format string, of the right type and in the right order. These are the possible format specifiers:

```
%d   signed int variable, decimal representation (equivalent to %i)
%u   unsigned int variable, decimal representation
%x   unsigned int variable, lowercase hexadecimal representation
%X   unsigned int variable, uppercase hexadecimal representation
%o   unsigned int variable, octal representation
     (there are no format specifiers for binary representation)
%f   float/double, normal notation
%e   float/double, exponential notation (%E uses E instead of e)
%g   float/double, notation %f or %e chosen depending on value (%E if %G)
%c   character (passed as int), text representation
%s   string (see 10. Arrays, strings and pointers)
%p   pointer (see 10. Arrays, strings and pointers)
%n   number of characters written upto now will be written to int
     that the corresponding argument points to
```

You can change the type of the printed variable by inserting one of the following characters between the % sign and the type character (for example: %ld for **long int** instead of an **int**).

```
h      for d,i,o,u,x,X: short int instead of int
       (the short int will be promoted to int when passed anyway)
       for n: store result in short int instead of int
l      for d,i,o,u,x,X: long int instead of int
       for n: store result in long int instead of int
       Do NOT use for e,E,f,F,g,G for e.g. printing doubles.
L      for e,E,f,F,g,G: long double instead of float/double
```

There are some flags and modifiers that can be put between the % and the type character:

```
-      left alignment, pad on the right with spaces (default=right alignment)
+      print plus sign if positive (default=only print minus sign if negative)
       (for signed numbers only)
space print space if positive (default=only print minus sign if negative)
       (for signed numbers only)
0      pad with zeros instead of with spaces (for numbers only)
#      "alternate form": - o: 0 will be prepended to a non-zero result
                         - x/X: prepends 0x/0X to result
                         - f/F,e/E,g/G: decimal point even if no decimals
                         - g/G: trailing zeros are not removed
<nonzero decimal value> specify field width to which result will be padded
       (this can be used together with the 0 flag)
*      field width will be passed as int parameter before the actual argument
.<nonzero decimal value> specify precision (default for f/F,e/E = 6)
       (for s, precision will limit the number of printed characters)
.0     no decimal point is printed for f/F,e/E
.*     precision will be passed as int parameter before the actual argument
```

Here's an example:

```
printf("Record %lX: name = %s, age = %d, hourly wage = %.3f\n",
       record_num, name, age, hourly_wage);
```

Would print the hexadecimal record number, the name, age, and hourly wage with 3 digits precision, provided record_num is of type **unsigned long**, name is a string (see 10. Arrays, strings and pointers), age is an integer (a smaller type would also be OK since it would automatically be promoted to an integer when passed to printf()) and hourly_wage is a **double** (or **float**, for the same reason).

Now you should understand how this program works, so we can move on to new territory...

4. Doing calculations

4.1. The calc program

```
/* prog4−1.c: calc */

#include <stdio.h>
#include <math.h>

int main(void)
{
   double a, pi;
   int b;

   a = 500 / 40; /* Fraction is thrown away (integer math) */

   printf("a = %.3f\n", a);

   a = 500.0 / 40; /* Fraction is NOT thrown away (floating point math) */

   printf("a = %.3f\n", a);

   a++;

   printf("a now = %.3f\n", a);

   b = (int) a;
   b ^= 5;

   printf("b = %d\n", b);

   pi = 4 * atan(1.0);
   printf("pi ~= %.10f\n", pi);

   return 0;
}
```

The output of this program is: [*]

```
a = 12.000
a = 12.500
a now = 13.500
b = 8
pi ~= 3.1415926536
```

[*] If you get "undefined symbol" or similar errors, you have to enable floating−point support in your compiler (see manual / online help)

4.2. Operators and typecasts

The first lines of the body of main() should be obvious:

```
   double a, pi;
   int b;
```

2 uninitialized double−precision floating−point variables, called a and pi, and an unitialized integer variable called b are defined. The next line contains something new:

```
   a = 500 / 40; /* Fraction is thrown away (integer math) */
```

We've encountered the assignment operator = before. The / is the division operator. So what this code does is dividing the value 500 by the value 40 [*] and storing the result in the variable a.

[*] Because both operands are constants, the compiler will most likely optimize this by calculating the result at compile−time.

The following operators are available (in order of precedence):

Precedence	Operator	Explanation
1. Highest	()	Function call (see 2.4. Functions and types)
	[]	Array subscript (see 10. Arrays, strings and pointers)
	−>	Indirect member selector (see 6. Structs, unions and bit−fields)
	.	Direct member selector (see 6. Structs, unions and bit−fields)
2. Unary	!	Logical negation: nonzero value −> 0, zero −> 1
	~	Bitwise complement: all bits inverted
	+	Unary plus
	−	Unary minus
	++	Pre/post−increment (see below)
	−−	Pre/post−decrement (see below)
	&	Address (see 10. Arrays, strings and pointers)
	*	Indirection (see 10. Arrays, strings and pointers)
	sizeof	Returns size of operand in bytes; two forms: 1) **sizeof**(type) 2) **sizeof** expression
3. Multi−plicative	*	Multiply
	/	Divide
	%	Remainder (only works for integers)
4. Additive	+	Binary plus
	−	Binary minus
5. Shift	<<	Shift bits left, e.g. 5 << 1 = 10
	>>	Shift bits right, e.g. 6 >> 1 = 3
6. Relational	<	Less than
	<=	Less than or equal to
	>	Greater than
	>=	Greater than or equal to (not =>)
7. Equality	==	Equal to (not =)
	!=	Not equal to
8. Bitwise AND	&	Bitwise AND, e.g. 5 & 3 == 1
9. Bitwise XOR	^	Bitwise XOR, e.g. 5 ^ 3 == 6
10. Bitwise OR	|	Bitwise OR, e.g. 5 | 3 == 7
11. Logical AND	&&	Logical AND

```
12. Logical OR  ||          Logical OR
13. Conditional ?:          Conditional operator
                            (see 7.4 Using the ?: operator)
14. Assignment  =           Simple assignment
                *=          Assign product (see below)
                /=          Assign quotient
                %=          Assign remainder
                +=          Assign sum
                -=          Assign difference
                &=          Assign bitwise AND
                ^=          Assign bitwise XOR
                |=          Assign bitwise OR
                <<=         Assign left shift
                >>=         Assign right shift
15. Comma       ,           Separates expressions
```

The precedence can be overruled by using parentheses. E.g. 5+4*3 would give 17, while (5+4)*3 would give 27.

The unary (2), conditional (13) and assignment (14) operators associate from right to left, the others from left to right. E.g. 4/2*3 groups as (4/2)*3, not 4/(2*3), but e.g. a = b = 7 groups as a = (b = 7), so you can assign a value to multiple variables in one line, but this reduces the clarity, at least in my opinion.

The left operand of the assignment operators must be what is called an "lvalue", i.o.w. something to which a value can be assigned. Of course you can't put a constant or an expression on the left side of an assignment operator, because they are not variables that have a location in which a value can be stored. For the same reason it cannot be, for example, the name of an array or string (see 10. Arrays, strings and pointers).

Don't worry if you're a bit overwhelmed by this list. It's normal if you don't understand most of them yet; the more advanced ones will be explained later in this document (see the forward references); the prefix/postfix and special assignment operators are explained below; the mathematical operators should be obvious, the bitwise operators too if you know something about binary arithmetic; just watch out for their precedence: they have higher precedence than the logical operators but lower precedence than the relational operators. For example:

```
a!=b&c
```

You may think the above expression performs a bitwise AND of b and c, and then compares this value to a for inequality. However, since the bitwise & operator has lower precedence than the equality operators, in fact a is compared to b for inequality, and the result of this comparison (1 or 0, see below) is then bitwise ANDed with c.

The logical AND/OR, relational and equality operators are usually used in conditions, such as those in an **if** statement (see Conditionals (**if**/**else**)). The relational and equality operators return 1 if the relationship or equality is true, 0 if it isn't. The logical AND/OR operators have the following logic:

logical AND: (&&)

```
left operand  right operand  result
zero          zero           0
zero          nonzero        0
nonzero       zero           0
nonzero       nonzero        1
```

logical OR: (||)

```
   left operand  right operand  result
   zero          zero           0
   zero          nonzero        1
   nonzero       zero           1
   nonzero       nonzero        1
```

Onto the next line of code:

```
   printf("a = %.1f\n", a);
```

This line outputs the value of a, which is 12.0. But 500/40 is 12.5, and since a **double** is used one could reasonably expect the decimal digits to be correct. The reason for the wrong result is that the constants 500 and 40 are both interpreted as integer values by the compiler. At least one of the constants should be explicitly made a floating−point constant by writing it as e.g. 500.0. That's what's done in the next line of code:

```
   a = 500.0 / 40; /* Fraction is NOT thrown away (floating point math) */

   printf("a = %.1f\n", a);
```

And this time the printf() will print the correct result.
Omitting this .0 (or just .) is a mistake beginners often make. The same goes for e.g. multiplying two **int** values into a **long**: if the result doesn't fit in an **int**, at least one of the constants must be explicitly made a **long**, e.g. 1000L*1000 instead of simply 1000*1000. There's also the U suffix to indicate an **unsigned** constant. There are more, but you'll rarely need any others.

```
   a++;
```

This is an example of the postfix increment operator. This is a special operator because it has the side effect of actually changing the value of the variable. The ++ (double plus) operator adds 1 to the value, the −− (double minus) operator substracts one from the value. Whether the operator is placed before (prefix) or after (postfix) the variable determines whether the value used in the expression is the one after resp. before the modification. An example will probably make this clear: (assume a and b are integer variables)

```
a = 5;
b = a++;
/* b will now be 5, a will be 6 */

a = 5;
b = ++a;
/* b will now be 6, a will be 6 */
```

You can't use something like:

```
a = a++;                      /* wrong! use a++; or a = a + 1; instead */
```

Because this code changes the variable a twice in the same line: in the assignment, and with the ++ operator. This has something to do with the so−called sequence points. For more information, see question 3.8 in the c.l.c FAQ, the URL can be found in: 17.2. Other interesting C−related online material However, in our example program we don't use the result of the expression, so the only effect is that a will be incremented by one, and will contain the value 13.5.

```
   b = (int)a;
```

This line assigns the value of a to b. But there is something special: the **int** between parentheses. This is called a typecast (or conversion). It basically tells the compiler "turn this **double** value into an integer value" [*]. If you would simply write b = a; this would automatically happen, but the compiler would probably give a warning because an **int** can't hold all the information that a **double** can (for example, the fractional digits we just calculated will be lost). Typecasting can be useful, but also dangerous, so you should avoid it when you can.

[*] Note that a typecasts does NOT reinterpret the bit pattern of a variable for the new type, but converts the value of the variable.

```
   b ^= 5;
```

This line uses a special assignment operator. It is actually a shorthand notation for:

```
   b = b ^ 5;   /* bitwise XOR of b with 5 */
```

Such a "shorthand" assignment operator exists for most operators (see table above).

```
   printf("b = %d\n",b);
```

This will output the value of b. Since the value of a that was assigned to b was 13 (1101b), the bitwise XOR operation with 5 (0101b) results in the value 8 (1000b) for b (you'll also need to know binary to understand this).

4.3. Using the functions from math.h

```
   pi = 4 * atan(1.0);
   printf("pi ~= %.10f\n", pi);
```

This piece of code calculates and prints an approximation for pi. It uses a function from the standard header math.h to do this: atan(), which calculates the arc tangent in radians. The arc tangent of 1 (written as 1.0 to make clear this is a floating−point constant) is pi/4, so by multiplying by 4 we get an approximation of pi. There are many useful functions like this in math.h (see 16. Overview of the standard library) They operate on doubles. For example, there is a pow() function in case you were worried that there was no power operator in the above table.

5. Enums and typedefs

5.1. The colours program

```
/* prog5−1.c: colours */

#include <stdio.h>

enum colours {RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET,
              NUMBER_OF_COLOURS};

typedef enum colours colour_t;

int main(void)
{
   colour_t sky, forest;
   printf("There are %d colours in the enum\n", NUMBER_OF_COLOURS);

   sky = BLUE;
   forest = GREEN;

   printf("sky = %d\n", (int)sky);
   printf("forest = %d\n", (int)forest);

   return 0;
}
```

The output of this program is:

```
There are 7 colours in the enum
sky = 4
forest = 3
```

5.2. Defining and using enums and typedefs

```
enum colours {RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET,
              NUMBER_OF_COLOURS};
```

This is the definition of an enumeration type (**enum**). Every one of the word constants (here colour names) will be assigned a numeric value, counting up and starting at 0 [*]. So the constants will have the following values:

[*] You can override this default numeric value by adding an explicit =value after the constant; the counting for subsequent constants will then continue from there.

word constant	numeric value
RED	0
ORANGE	1
YELLOW	2
GREEN	3
BLUE	4
INDIGO	5
VIOLET	6
NUMBER_OF_COLOURS	7

The reason that the NUMBER_OF_COLOURS is at the end of the **enum** is that because of the counting system, by adding a constant like this at the end of the **enum**, this constant will always correspond to the number of defined constants before it [*].

[*] Of course this will not be valid when different values are explicitly assigned to the constants as mentioned in the above note.

```
typedef enum colours colour_t;
```

If we wanted to define a variable of the **enum** type defined above, we would normally have to do that like this:

```
enum colours somevariable;
```

However, sometimes we may prefer to make the name "fit in" better with the rest of the types, so that we could simply use it like **int** etc. without the **enum** required. That's what the **typedef** keyword is for. If we add

typedef before what would otherwise be a variable definition, instead of creating a variable with a certain name we are creating a new type with that name.

In our example this means that everywhere where we use colour_t this is compiled as if there was an **enum** colours instead. So:

```
   colour_t sky, forest;
```

Is equivalent to:

```
enum colours sky, forest;
```

Onto the next line of code:

```
   printf("There are %d colours in the enum\n", NUMBER_OF_COLOURS);
```

C has a rather relaxed way of dealing with types (this is called weak typing). This is especially noticeable when it comes to enums: they are equivalent [*] with **int** variables.

[*] That doesn't necessarily mean that enums and ints will be exactly the same type, but they are compatible types

So we could also have defined the variables like this:

```
int sky, forest;
```

And still have used the assignment of **enum** constants below.

Another consequence of this equivalence is that there is no special printf() format specifier to print an **enum** value. We have to use the format specifier for an integer: %d. An enum constant needn't even be cast as it is guaranteed to be of type int.

```
   printf("There are %d colours in the enum\n", NUMBER_OF_COLOURS);
```

This also means that when we print an **enum** value, a number will be output, not the symbolic name.

```
   sky = BLUE;
   forest = GREEN;
```

Here we assign values to the variables using our defined constants. These statements are equivalent to:

```
   sky = 4;
   forest = 3;
```

But obviously that's much less clear, and when the **enum**'s definition is modified this code may no longer be equivalent to the above.

```
   printf("sky = %d\n", (int)sky);
   printf("forest = %d\n", (int)forest);
```

The values of the **enum** variables are printed, again as integers. [*] However, in this case we have typecast the **enum** to **int** first. This is because, although unlikely, the compiler may have chosen another (larger) type for the enum itself (not the constants).

[*] If you would really like the symbolic names to be printed, you'll have to write code to that yourself. You may want to use either an array of strings (see 10.2. Arrays and strings) or a **switch**() statement (see **8. switch/case/default**).

6. Structs, unions and bit−fields

6.1. The cars program

```
/* prog6−1.c: cars */

enum brands {VOLKSWAGEN, FORD, MERCEDES, TOYOTA, PEUGEOT};

enum colours {RED, GREEN, BLUE, GREY, BLACK, WHITE};

struct car {enum brands brand;
            enum colours colour;
            int second_hand;
            union {unsigned long mileage; int months_guarantee;} carinfo;
            unsigned diesel:1, airbag:1, airconditioning:1;
            };

int main(void)
{
   struct car mycar = {VOLKSWAGEN, GREY, 1, 50000, 0, 0, 0};
   struct car yourcar;

   yourcar.brand = MERCEDES;
   yourcar.colour = BLACK;
   yourcar.second_hand = 0;
   yourcar.carinfo.months_guarantee = 12;
   yourcar.diesel = 1;
   yourcar.airbag = 1;
   yourcar.airconditioning = 1;

   return 0;
}
```

This program has no output.

6.2. Using structs, unions and bit−fields

These lines should look familiar: (see 5. Enums and typedefs)

```
enum brands {VOLKSWAGEN, FORD, MERCEDES, TOYOTA, PEUGEOT};

enum colours {RED, GREEN, BLUE, GREY, BLACK, WHITE};
```

But the next few lines contain several new things:

```
struct car {enum brands brand;
            enum colours colour;
            int second_hand;
            union {unsigned long mileage; int months_guarantee;} carinfo;
            unsigned diesel:1, airbag:1, airconditioning:1;
            };
```

OK, this is the definition of a structure type. By writing something of this kind:

```
struct structure_tag {type1 member1; type2 member2; /* etc. */ };
```

You define a structure type which from this point on you can use to define structure variables, e.g.: [*]

```
struct structure_tag mystructure;  /* don't forget the struct keyword */
```

[*] This could be written in one line as **struct** structure_tag {type1 member1; type2 member2;} mystructure; (in which case the structure_tag is optional)

A structure is a collection of different variables (called members) into a single type. It is usually used to describe all properties of a certain item in a single variable. In our cars example we could store information like the brand, colour etc. in a different variable for every car. But it is a lot more clear to concentrate all the information about one specific car into a type of its own, analogous to e.g. Pascal records. That's why a structure is used.

The first two members are familiar enums:

```
            enum brands brand;
            enum colours colour;
```

The next variable:

```
            int second_hand;
```

Is used to store whether the car is a second hand car or a new car. In the former case a 1 is stored in this member variable, in the latter case a 0.

But the information that we need to store about a second hand car needn't be the same as that for a new car. For example, we only need to store the mileage information for second hand cars and guarantee information for new cars. So sometimes it may be a good idea to not have unneeded information take up space in our structures. That's why unions are available:

```
            union {unsigned long mileage; int months_guarantee;} carinfo;
```

This is a **union** definition inside our **struct** definition (you can put structs within structs etc.). The **union** definition looks very similar to a **struct** definition. There is one very important difference though: in a **struct** space is allocated for every member. In a **union** enough space is allocated so that the largest member can be stored (and therefore the other members as well). But only one member can hold a value at a time; by assigning a value to one member of a **union** the values of the other members of a **union** are destroyed. You can only use the value of the **union** member that a value was last assigned to. [*]

[*] Many systems allow reading other members than the one last assigned to, but this is not standard.

Because of this, we have to keep track of whether the mileage member or the months_guarantee member contains a valid value. This can be determined by looking at the second_hand variable. If that is 1, the mileage member contains a valid value; if that is 0, the months_guarantee member contains a valid value.

```
            unsigned diesel:1, airbag:1, airconditioning:1;
```

Sometimes member variables in a structure only need to store a few values, often only 1 or 0 which would fit in a single bit. To avoid having to allocate an entire **int** or **char** for every such variable, you can use bit−fields. The above definition will allocate an **unsigned int** and define three variables (diesel, airbag and airconditioning) of each 1 single bit wide. [*]

[*] Bit−fields should always be of type **signed int** or **unsigned int**. If you use plain **int**, it is implementation−defined whether they'll be **signed** or **unsigned**. Many systems support other types, but this is not standard.

OK, now we move into the body of main():

```
   struct car mycar = {VOLKSWAGEN, GREY, 1, 50000, 0, 0, 0};
   struct car yourcar;
```

As mentioned above, these are definitions of two **struct** car variables, called mycar and yourcar. What's special is that mycar is initialized. This is comparable to how regular variables are initialized, except for the form of the initializer. For a **struct** this should be between { and } braces, and initializer values for the members should be separated by commas. For a **union** (such as our carinfo member, which may contain either mileage or months_guarantee information) it is always the first member that is initialized, so in our case the 50000 will be used to initialize mileage, not months_guarantee. Bit−fields are initialized like normal member variables.

```
   yourcar.brand = MERCEDES;
   yourcar.colour = BLACK;
   yourcar.second_hand = 0;
   yourcar.carinfo.months_guarantee = 12;
   yourcar.diesel = 1;
   yourcar.airbag = 1;
   yourcar.airconditioning = 1;
```

Each member of yourcar is initialized manually. A member of a structure is accessed by appending .membername to the structure variable name. The same goes for a **union**, that's why we have to use yourcar.carinfo.months_guarantee in the fourth line instead of simply yourcar.months_guarantee.

The reason yourcar can't be initialized in the same way as mycar is that, as mentioned above, we can only initialize the first member variable of a **union** in that way, not the others (in this case the months_guarantee member variable).

7. Conditionals (if/else)

7.1. The yesno program

```
/* prog7−1.c: yesno */

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

int main(void)
{
   int yesno;

   srand((unsigned)time(NULL));
   yesno = rand() % 2;

   if (yesno==0)
      printf("No\n");
   else
      printf("Yes\n");

   return 0;
}
```

This program outputs either:

```
No
```

or

```
Yes
```

7.2. Obtaining random numbers

Although this chapter is about using conditionals, we're first going to see something else that's often useful: obtaining random numbers.

However, the oversize calculator that your computer really is cannot return really random numbers [*]. Instead, it uses something called pseudo−random numbers: they are numbers that are calculated using a mathematical algorithm, and which are therefore not really random at all, but which have the appearance of being random.

[*] Some specialized hardware may be used to measure random factors from e.g. the environment, but this falls way outside of the scope of the standard C language.

Obviously, if a program uses a certain mathematical algorithm to obtain these numbers, it will produce the very same numbers every time it runs. Not very random. That's why we must first initialize the random number generator with a value that is different nearly every time the program is run. A good value for that seems to be the current date and time.

The current date and time is returned when we call the time() function with NULL as parameter (see the section on time() in 16. Overview of the standard library if you want to know what this parameter is for). The value it returns is a time_t, and we don't really know what that is (it can be different on every system). All that interests us here is that the value will probably be different every time the program is run. [*]

[*] On some systems without a clock time() may return the same value (time_t)−1 every time, and on those this approach will of course fail.

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

The standard header stdlib.h contains the declarations for the srand() and rand() functions, and the standard header time.h contains the declaration for the time() function.

```
   srand((unsigned)time(NULL));
```

The srand() function initializes the random number generator. It takes an **unsigned** integer value as parameter. As explained above, we use the return value of time(NULL) as the initializer (often called the seed of the random number generator), which is this unknown type time_t. We explicitly typecast (see 4.2. Operators and typecasts) this type to an **unsigned** integer [*] to suppress possible compiler complaints (since we know what we're doing and don't really care if for example we may lose some of the time information in the conversion; all that matters is that the value should be different every time the program is run, when possible).

[*] On some rare systems this may cause an overflow error.

```
   yesno = rand() % 2;
```

The rand() function returns a random number between 0 and RAND_MAX (the value of RAND_MAX is defined in stdlib.h and may be different on every system). You can't force rand() to return random numbers in a certain range. Instead, we convert its return value so that it fits in this range. The most obvious method to do this is the one we use above: using the remainder operator. For example, rand() % 10 will always give a value in the range 0..9. [*] So in our case yesno will contain either the value 0 or 1.

[*] This method is not very good, but used here for its simplicity. A better method (taken from the c.l.c FAQ) would be to use something like: (**int**)((**double**)rand() / ((**double**)RAND_MAX + 1) * N)
For more information, see question 13.16 in the c.l.c FAQ, the URL can be found in: 17.2. Other interesting C−related online material

7.3. Using if/else

```
   if (yesno==0)
      printf("No\n");
   else
      printf("Yes\n");
```

In the preceding code, yesno was set randomly to either 0 or 1. This code now prints "No" when the value of yesno was 0, and "Yes" if it was any other value (which would be 1 in this case). The **if**() statement works as follows:

```
if (condition)
   statement_that_is_executed_if_the_condition_is_true;
else
   statement_that_is_executed_if_the_condition_is_false;
```

Note that, unlike in Pascal, there must be a semicolon after the code in the **if** part of the statement.

If the code in the **if** or **else** part of the statement consists of more than one statement, a block must be used:

```
if (condition)
{
   possibly_more_than_one_statement;
}
else
   still_one_statement;
```

When a block is used, no semicolon may be put after the closing }.

You can chain **if** statements as following:

```
if (condition)
   some_code;
```

```
else if (another_condition)
   some_other_code;
else
   even_different_code;
```

7.4. Using the ?: operator

The ?: operator may be used as an alternative to an **if**−statement, with the difference that the ?: operator yields a value. For example:

```
a = c + (b<4 ? 1 : 2);
```

Is equivalent to:

```
if (b<4)
   a = c + 1;
else
   a = c + 2;
```

So, the expression:

```
   condition ? expr1 : expr2
```

Will give expr1 if condition is true and expr2 if condition is false.

You have to be careful with predecence: the ?: operator has very low precedence (only assignment and comma operators have lower precedence). If you're not sure, it's safest to use enough parentheses.

8. switch/case/default

8.1. The animals program

```
/* prog8−1.c: animals */

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

enum animal {CAT, DOG, COW, SNAIL, SPIDER, NUMBER_OF_ANIMALS};

void animalsound(enum animal animalnumber)
{
   switch(animalnumber)
   {
   case CAT:
      printf("Miaow!\n");
      break;
   case DOG:
      printf("Woof!\n");
      break;
   case COW:
      printf("Moo!\n");
      break;
   default:
      printf("Animal makes no audible sound.\n");
      break;
   }
}

void animalfood(enum animal animalnumber)
{
   switch(animalnumber)
   {
   case COW:
   case SNAIL:
      printf("Eats plants.\n");
      break;
   case CAT:
   case SPIDER:
      printf("Eats beasts.\n");
      break;
   }
}

int main(void)
{
   enum animal myanimal, youranimal;

   srand((unsigned)time(NULL));

   myanimal = rand() % NUMBER_OF_ANIMALS;

   animalsound(myanimal);
   animalfood(myanimal);

   youranimal = rand() % NUMBER_OF_ANIMALS;

   animalsound(youranimal);
   animalfood(youranimal);

   return 0;
}
```

This program's output varies. It might be something like this:

```
Moo!
Eats plants.
Animal makes no audible sound.
Eats beasts.
```

8.2. Using the switch() statement

```
enum animal {CAT, DOG, COW, SNAIL, SPIDER, NUMBER_OF_ANIMALS};
```

This is yet another familiar **enum**. Again, NUMBER_OF_ANIMALS will be set to the number of defined animals.

```
void animalsound(enum animal animalnumber)
{
}
```

This time we put part of our program in a separate function. This way the same code can be used several times without having to be completely written out every time (see also 2.4. Functions and types).

In this case, the animalsound() function will print out the corresponding sound for a given animal of our above **enum** type.

```
   switch(animalnumber)
   {
   case CAT:
      printf("Miaow!\n");
      break;
   case DOG:
      printf("Woof!\n");
      break;
   case COW:
      printf("Moo!\n");
      break;
   default:
      printf("Animal makes no audible sound.\n");
      break;
   }
```

This is the new part in our program. A **switch** statement executes certain code depending on the value of the variable that is switched upon (the variable between the **switch**() parentheses, in this case animalnumber). For every different possible case a "**case** value:" is present, followed by the code that is to be executed in case the switched variable is equal to the specified value.

Every case should end with a **break**; statement, because otherwise the code simply keeps continuing into the next **case** statement. This is known as fallthrough, and has both its advantages and disadvantages. Once you're used to it it shouldn't pose any problems, you just have to remember to put the **break**; in there. See below for a useful example of fallthrough.

A special case is the **default**: case. If a **default**: case is present, its code is executed when none of the other cases match. In our case the animals SNAIL and SPIDER don't have explicit case labels, and for them the **default**: case of "Animal makes no audible sound." will be executed.

The above code would be equivalent to: (apart from the fact that the animalnumber is evaluated multiple times in the **if** form)

```
if (animalnumber==CAT)
   printf("Miaow!\n");
else if (animalnumber==DOG)
   printf("Woof!\n");
else if (animalnumber==COW)
   printf("Moo!\n");
else
   printf("Animal makes no audible sound.\n");
```

But is obviously clearer and easier to expand.

Another difference between the **if**() and **switch**() form is that in the **if**() form non−integral types like **double** may be used. However, floating point variables may not be able to store an exact value, so you should not compare them with the equality operator (==). Instead you should limit the absolute value of the difference between the two to a certain percentage of the values, something like:

```
if (fabs(x-y) < x * 0.00001)
   /* x and y may be considered equal */ ;
```

The following functions prints out the type of food the specified animal number will consume:

```
void animalfood(enum animal animalnumber)
{
}
```

It also uses a **switch**() statement:

```
   switch(animalnumber)
   {
   case COW:
   case SNAIL:
      printf("Eats plants.\n");
      break;
   case CAT:
   case SPIDER:
      printf("Eats beasts.\n");
      break;
   }
```

However, there are some differences here. This is where a useful example of fallthrough is demonstrated: the COW and CAT case labels are not ended with a **break**; statement so they will continue right into the following case labels (SNAIL and SPIDER, respectively).

Also, there is no **default**: statement, which means that for the animals for which no explicit **case** is present (such as DOG here), no code will be executed, and therefore nothing will be printed.

```
   enum animal myanimal, youranimal;

   srand((unsigned)time(NULL));

   myanimal = rand() % NUMBER_OF_ANIMALS;

   animalsound(myanimal);
   animalfood(myanimal);

   youranimal = rand() % NUMBER_OF_ANIMALS;

   animalsound(youranimal);
   animalfood(youranimal);
```

The rest of the program is equivalent to the code in the yesno program, except that there are NUMBER_OF_ANIMALS possible values instead of 2, and our own functions animalsound() and animalfood() are called to handle the generated random animal numbers.

9. Loops (do,while,for)

9.1. The loops program

```
/* prog9-1.c: loops */

#include <stdio.h>

int main(void)
{
   int i;

   i = 1;
   while (i<4)
   {
      printf("%d\n",i);
      i++;
   }

   printf("\n");

   i = 1;
   do
   {
      printf("%d\n",i);
      i++;
   }
   while (i<4);

   printf("\n");

   for (i=15; i>=10; i--)
      printf("%d\n",i);

   return 0;
}
```

This program should output:

```
1
2
3

1
2
3

15
14
13
12
11
10
```

9.2. Using loops

```
   i = 1;
   while (i<4)
   {
   }
```

The first line assigns 1 to the i variable. What follows is a **while**() loop. A **while** loop has this form:

```
while (condition)
   statement;
```

If the code inside the loop consists of multiple statements, a block must be used, just like with **if** statements:

```
while (condition)
{
   code;
}
```

Be careful not to put a semicolon after the closing parentheses of the **while**(condition).

The code will be repeatedly executed (looped) for as long as the condition is true. Obviously the code somehow needs to do something which may change this condition, or the loop will execute forever.

In our case the code in the loop body is:

```
      printf("%d\n",i);
      i++;
```

Which prints out the current value of i, and then increments its value by 1. This way the condition i<4 in the **while** loop will no longer be true as soon as i reaches 4. Since i was originally set to 1, the output will become:

```
1
2
3
```

Next, a printf() call to insert a blank line:

```
   printf("\n");
```

Now, the next loop:

```
   i = 1;
   do
   {
      printf("%d\n",i);
      i++;
   }
   while (i<4);
```

This seems quite similar, and indeed its output will be identical in this case. Then what's the difference ? Well, in a **while**() loop the condition is at the start, which means that if i was, for example, 5 at the beginning, the loop would never have executed and nothing would have been output.

However, in a **do−while**() loop the condition is at the end, which means that the loop will always be executed at least once, even if the condition was false to begin with. So in the above example where i was 5 at the beginning, this loop would still have output 5 before ending.

```
   for (i=15; i>=10; i−−)
      printf("%d\n",i);
```

This is a **for** loop. A **for** loop is a sort of shorthand form for a loop, which consists of the following parts:
− an initialization statement
− a loop condition
− a modification statement

A **for** loop has the following form:

```
for (initialization; condition; modification)
   code;
```

Or, if multiple statements need to be put inside the loop:

```
for (initialization; condition; modification)
{
   code;
}
```

And is equivalent to: [*]

```
initialization;
while (condition)
{
   code;
   modification;
}
```

[*] Not 100% equivalent, because the **continue** statement will work differently in the code using **while**() than that using **for**()

You can put more than one expression in the initialization, condition and/or modification part by separating them by the comma operator (,).

In our case:

```
   for (i=15; i>=10; i--)
      printf("%d\n",i);
```

The initialization statement assigns 15 to the variable i.

The loop condition is i>=10.

The modification decrements i by 1.

So this loop will start with i=15, checks that condition i>=10 is fulfilled, print out the value 15, decrement i by 1, check the condition again, etc. until the condition is no longer fulfilled.

This will therefore produce the following output:

```
15
14
13
12
11
10
```

Which is, of course, the same as that would have been produced by the equivalent **while**() loop:

```
i = 15;
while (i>=10)
{
   printf("%d\n",i);
   i--;
}
```

Sometimes you will encounter something like this:

```
for (;;)
{
   code;
}
```

Or:

```
while (1)
{
   code;
}
```

These are infinite loops. They never end until they are aborted by, for example, a **break**; statement. The **for**(;;) version will work because when the condition is not filled in, it will default to "always true". The **while**(1) version will work because the condition 1 is always true.

9.3. Using break,continue,goto

We have already encountered the **break**; statement inside a **switch**() block, where it aborts the execution of the **case** statements.

It works in a similar way for loop statements, such as **while**(), **do−while**() or **for**(). It aborts the execution of the loop without checking the condition. It's a way of saying "exit now, no questions asked". You should try to avoid this when it is not really necessary, because it may cause the program's execution flow to be difficult to follow. Also be aware that a **break**; statement can only exit the innermost loop or **switch**() block, not any surrounding ones.

The **continue**; statement is a slightly more 'civilised' version of the **break**; statement. It says "abort this round of the loop execution, jump immediately to the condition evaluation". If the condition is still fulfilled, the loop will continue executing at the next round. If it is no longer fulfilled, the loop will be aborted. For example, suppose you want to process all elements i,j of a matrix except those on the main diagonal line (those where i==j), you could do something like this: (assume i and j are integer variables)

```
for (i=0; i<10; i++)
{
   for (j=0; j<10; j++)
   {
      if (i==j)
         continue;             /* skip elements on main diagonal line */
      /* code to process element i,j */
   }
}
```

The **goto** statement can be used to jump to anywhere in the current function [*], but it's best to use it only to jump out of blocks, never into blocks, and jumping past variable initializations is usually a bad idea.

[*] To jump between functions − if you really must! − you should use setjmp()/longjmp() (see 16. Overview of the standard library)

You have to prefix the statement you wish to jump to by a label as follows:

```
mylabel: printf("some statement\n");
```

The statement may also be an empty statement, like this:

```
mylabel: ;
```

You can then immediately jump to this statement from anywhere in the function by using:

```
   goto mylabel;
```

You have to be very careful about using **goto**. Unthoughtful use can make your code look like a bowl of spaghetti, i.e. very difficult to follow. A good use for it might be to jump out of a set of nested loops when an error condition is encountered, for example: (assume i, j and k are integer variables)

```
for (i=1; i<100; i++)
{
   for (j=1; j<100; j++)
   {
      for (k=1; k<100; k++)
      {
         if (some_error_condition)
            goto abort_for_loops;
      }
   }
}
abort_for_loops: ;
```

10. Arrays, strings and pointers

10.1. The strings program

```
/* prog10−1.c: strings */

#include <stdio.h>
#include <string.h>

int main(void)
{
   char s[20];

   strcpy(s, "strings");

   printf("s = %s\n", s);
   printf("s[3] = %c\n", s[3]);
   printf("s[7] = %d\n", s[7]);
   printf("strlen(s) = %lu\n",(unsigned long)strlen(s));
   strcat(s, " program");
   printf("s now = %s\n", s);
   printf("strlen(s) = %lu\n",(unsigned long)strlen(s));

   return 0;
}
```

The output of this program is:

```
s = strings
s[3] = i
s[7] = 0
strlen(s) = 7
s now = strings program
strlen(s) = 15
```

10.2. Arrays and strings

Because we are going to use string handling functions in this program, a new header file is included:

```
#include <string.h>
```

This header file declares several useful functions for dealing with strings, such as strcpy(), strlen() and strcat() which we use in this program (see below).

As usual, at the beginning of main() we find a variable definition:

```
   char s[20];
```

The new thing here is the [20]. This is a definition of an array. An array is simply a contiguous series of variables of the same type. In this case we define s to be an array of 20 chars. The number inside the brackets [ ] must be a constant. [*]

[*] Variable length arrays are proposed for the new C standard

Multi−dimensional arrays (e.g. one that has rows and columns) can be defined as, for example, **int** multiarray[100][50]; etc.

You can have arrays of any type of variable. However arrays of **char** are often used for a special purpose: strings. A string is basically just text. Some languages have direct support for a 'string' type, but C doesn't. A string is an array of **char** with a special feature: the end of the string is marked by a **char** with value 0. That means we need to allocate one additional byte to store this 0 terminator. The consequence for our example is that s is able to hold a string of at most 19 characters long, followed by the 0 terminator byte.

```
   strcpy(s, "strings");
```

The string s we've defined above doesn't contain any useful contents yet. As we've seen before (see 2. Hello World!) a string literal between double quotes (") can be used as contents for the string. You might be tempted to do something like this:

```
   s = "strings";          /* wrong! */
```

However, this doesn't work. C doesn't support assigning arrays (including strings) in this way. You must actually copy all the characters in the string literal into the string you wish to initialize. Fortunately there is a standard function to this: strcpy(), declared in string.h.

To copy string1 to string2 you would write:

```
   strcpy(string2, string1);    /* notice the order of the parameters! */
```

When you fill in the string variable (array of **char**) s for string2 and the string literal "strings" for string1 (not vice versa, as we've seen before you can't write to string literals, see 2. Hello World!), we have the code from our program.

```
   printf("s = %s\n", s);
   printf("s[3] = %c\n", s[3]);
   printf("s[7] = %d\n", s[7]);
```

The first line prints out the entire text contained in string variable s, by using printf() format specifier %s.

The second line prints out the character (element) with index 3 in the array. Such an element is written as arrayname[indexnumber]. You have to be careful with the index numbers, though: in C all counting starts from 0. So the 1st element really is element number 0, the 2nd element is element number 1, etc. This also means that s[20] would be the 21st element of our array, and since the array is only 20 elements long, this value is out of range and therefore not valid!

In our case the string variable contains the text "strings" followed by the 0 byte. Since s[3] is the 4th element of the array, thus the 4th character of the string, an 'i' is printed (the %c printf() format specifier is used to print a character).

The third line prints s[7], which is the 8th character of the string. Since the text is only 7 characters long, the 8th character is our 0 terminator byte. That's why it's printed using %d (integer value) instead of %c (the 0 terminator usually doesn't have a character representation).

```
   printf("strlen(s) = %lu\n",(unsigned long)strlen(s));
```

The strlen() function returns the length of a string (the number of characters it contains, the terminating 0 not included). It returns a size_t, which is some **unsigned** integer type capable of holding the size of an object. Since there is no special printf() format specifier for size_t we typecast it to an **unsigned long** (see 4.2. Operators and typecasts) and use the %lu format specifier.

```
   strcat(s, " program");
```

The strcat() function is used to append text to an already existing string variable. To append string1 to string2 you would write:

```
   strcat(string2, string1);   /* notice order of parameters */
```

You must make sure the string (character array) is large enough to hold all characters + the 0 terminator byte after the append!

In our case we add the string literal " program" to our already existing string "strings". The resulting string will be 7 + 8 = 15 characters long, plus the 0 terminator this requires 16 elements. Since our array is 20 elements long, there is no problem.

```
   printf("s now = %s\n", s);
   printf("strlen(s) = %lu\n",(unsigned long)strlen(s));
```

Finally the new text contained in s is printed. This will of course be "strings program". The new string length is also printed. This will of course be larger than the previous value.

10.3. The pointers program

```
/* prog10−2.c: pointers */

#include <stdio.h>

int main(void)
{
   int a[] = {1,2,3,5,7,11,13,17,19};
   int *p;

   printf("a[4] = %d\n", a[4]);

   printf("the address of the array = %p\n", (void *)a);
   printf("the address of element nr. 0 of the array = %p\n", (void *)&a[0]);
   printf("the address of element nr. 4 of the array = %p\n", (void *)&a[4]);

   p = a;

   printf("the value of the pointer = %p\n", (void *)p);
   printf("the value the pointer points at = %d\n", *p);

   p += 4;

   printf("the value of the pointer now = %p\n", (void *)p);
   printf("the value the pointer now points at = %d\n", *p);

   return 0;
}
```

The output of this program may be something like the following: (the address values and even their notations will most likely be entirely different on your system)

```
a[4] = 7
the address of the array = 00030760
the address of element nr. 0 of the array = 00030760
the address of element nr. 4 of the array = 00030770
the value of the pointer = 00030760
the value the pointer points at = 1
the value of the pointer now = 00030770
the value the pointer now points at = 7
```

10.4. Addresses and pointers

OK, now we're going to discuss the often feared pointers. Don't worry if the above program seems terribly difficult to you − your first encounter with pointers may be a bit confusing, but it shouldn't be that difficult to understand the concept and how to use them.

What's a pointer ? Well, every variable has to be stored somewhere in your computer's memory. The place where they are stored is called the address of the variable. Such an address is some sort of position value (its

exact representation depends on your system, and shouldn't really matter).
Such an address value can of course be stored in another variable (which has its own, different, address). That's called a pointer. A pointer contains the position in memory (address) of another variable, so in effect 'points' to this variable. Now let's analyze the program line by line and explain things on the way:

```
   int a[] = {1,2,3,5,7,11,13,17,19};
```

This is a definition of an array (see previous section). However, it's different from the one we used in our strings example in 3 ways:
1) it is not an array of **char**, but an array of **int**
2) the array is initialized with values; the initialization is similar to that of a **struct** variable (see 6.1. The cars program). We have several values (one for every element of the array), separated by a comma and surrounded by braces { }.
3) there is no number inside the brackets [ ]; this means we let the compiler count the number of elements we initialize and allocate just enough memory to be able to store all these elements (in our case the compiler will count 9 initialized elements, and will allocate a 9 elements long array).

```
   int *p;
```

Another variable definition: the * indicates "is a pointer to". So we've defined p as a pointer to **int**, which means that p can hold the address value of an **int** variable. [*]

[*] If you need a generic pointer, not to any specific type of variable, you should use a pointer to **void** (**void** *). However, you can't do the sort of calculations with these sort of pointers that we're doing in this program.

```
   printf("a[4] = %d\n", a[4]);
```

This prints the value of element number 4 (the 5th element) of the array, which is 7. Nothing new here.

```
   printf("the address of the array = %p\n", (void *)a);
```

We've seen before that by using something of the form arrayname[indexnumber] we can access an element of an array. But what happens if we simply use the arrayname itself (in this case a) ? That will return the address of the first element of the array (element number 0). That's also why assigning arrays simply using the assignment operator or passing the entire contents of an array as a parameter is not possible.

OK, so we have the address of the array's first element, which is the same value as the address of the array (the type is different; the former is a pointer to **int**, the latter a pointer to an array of **int**). Now how do we print it ? That's something special. The printf format specifier %p prints the value of a pointer (thus an address). What it prints exactly will be different on every system, but the concept will be the same.

But %p is used to print a generic pointer as seen in the note above: a **void** * instead of the **int** * that the address of the array will be. Therefore we have to typecast (see 4.2. Operators and typecasts) the pointer value (address) to a **void** *. [*]

[*] On many systems simply passing an **int** * to the printf() format specifier %p will work fine; but the standard requires this conversion.

```
   printf("the address of element nr. 0 of the array = %p\n", (void *)&a[0]);
   printf("the address of element nr. 4 of the array = %p\n", (void *)&a[4]);
```

These two lines are similar to the previous one: they print out an address using the %p format specifier, after typecasting that address value to a generic **void** *.

However, instead of using the arrayname a to get the address of the array's first element, an element index is used (0 resp. 4) between the [ ], and another operator is prepended: the address operator (&). Putting & before something takes the address of that something (when that something has an address of course; something like &5 will not work since the constant 5 doesn't have an address in memory; it is not an lvalue). So &variable takes the address of the variable, and &arrayname[indexnumber] takes the address of an element of an array.

In our case, the first line will always print the same address value as the previous one, because &a[0] (the address of element number 0, i.e. the first element of the array) will always be the same as just a (the address of the array's first element). But &a[4] (the address of element number 4, i.e. the fifth element of the array) will be a bit higher. [*]

[*] The example output printed above on my system can be explained as follows: an **int** is 4 bytes on my system (this may be different on yours), and my %p prints out addresses as hexadecimal values (that may also be different on yours), so 00030770 (&a[4]) is 16 (10h) bytes past 00030760 (&a[0] or just a).

```
   p = a;
```

What's this ? We assign an array of **int** (a) to a pointer to **int** (p). Like we've said in our strings example, assigning an array using the = operator does not copy the actual elements of the array (that's why we needed the strcpy() function for strings). We've seen above that the arrayname a on its own returns the address of the first element of the array. So it is that address (of type pointer to **int**) that we assign to p, which is a pointer to **int**. That's why it works. The result now is that p contains the address of array a's first element; we therefore say that p now points to a.

```
   printf("the value of the pointer = %p\n", (void *)p);
```

We print the address value stored in the pointer, which is of course the address of the array.

```
   printf("the value the pointer points at = %d\n", *p);
```

Ah, another operator. The * is called the dereference operator [*]. To dereference a pointer means to take the value of the variable to which the pointer is pointing. So *p means the value of the **int** that p is pointing at (i.o.w. whose address value p contains).

[*] As seen above, this * is also used for defining pointers. You could also read the definition **int** *p; as "p dereferenced is **int**".

Since p points at the first element of the array, the value of element number 0 of the array will be printed: 1.

```
   p += 4;
```

This is equivalent to p = p + 4; (see 4.2. Operators and typecasts). But how can you calculate with a pointer ? Well, the address value contained in the pointer variable can be modified using the operators + and −. However, there's one important difference to calculating with normal integral values: adding 4 to the pointer as in this example does not necessarily add 4 to the address value; instead it moves the pointer 4 elements of the type it points at further (so if your ints are 4 bytes like mine, this will add 4*4=16 to your address value). [*]

[*] You can't do this with the generic pointers to **void**, because the compiler doesn't know how many bytes the elements you're pointing at are.

So in this case p will now no longer point to element number 0 of the array, but to element number 4.

```
   printf("the value of the pointer now = %p\n", (void *)p);
   printf("the value the pointer now points at = %d\n", *p);
```

Which is verified by the output of these printf() calls.

Note: you can make pointers constants, too, just like regular variables; but watch out:

```
int *p;               /* p itself and what p points to may be modified */
const int *p;         /* p itself may be modified; what p points to may not */
int * const p;        /* p itself may not be modified; what p points to may */
const int * const p; /* p itself nor what it points to may be modified */
```

Something else to watch out for is this:

```
char *a,b;
```

This defines a **char** pointer (a) and a **char** (b), not 2 **char** pointers! On the other hand:

```
typedef char *char_pointer;
char_pointer a,b;
```

Does define 2 **char** pointers.

There is also a special pointer value, which may be used for all pointer types, that is used to indicate an invalid pointer value. This is the NULL pointer. NULL is defined in stdio.h, stdlib.h, stddef.h and in other headers. You should use NULL only for pointer values (to indicate an invalid pointer), not for other types of variables (e.g. not for an integral 0 value). You can use a plain 0 instead of NULL as well for this purpose, but most people prefer using NULL as a stylistic hint. [*]

[*] You shouldn't worry about what pointer value NULL is exactly. At the source code level 0 will evaluate to a NULL pointer value, but this does definitely not mean that the actual NULL pointer value must be a value with all bits 0.

Pointers are also useful in simulating "call by reference", which means that a function can change the value of its parameters and these changes will be passed on to the caller as well. With normal parameters (other than arrays) this is not possible in C, because they are always passed as "call by value", which means that the function can only access a copy of the actual values, and this will have no effect on the calling code.

By using a parameter of type "pointer to type" instead of just "type" this is possible. The calling code must then insert the address operator & before the variable passed as the parameter. The function can then access and modify the value using the dereference operator (*). This is also referred to as "pass by pointer" and you should not confuse it with the call by reference that is supported in C++, Pascal etc. because they don't require the calling code to explicitly insert an address operator.

An example may shed some light on this:

```
void swapintegers(int a, int b)
{
   int temp;

   temp = a;
   a = b;
   b = temp;
}
```

You would expect this function to swap the two integer variables a and be that it is passed as parameters. It does that, but only within the function itself. If the caller calls this function, e.g. like this:

```
swapintegers(myvariable, yourvariable);
```

Then after the call, myvariable and yourvariable will not be swapped. The solution is to do it like this:

```
void swapintegers(int *a, int *b)
{
   int temp;

   temp = *a;
   *a = *b;
   *b = temp;
}
```

And in the calling code:

```
swapintegers(&myvariable, &yourvariable);
```

10.5. Example: using command line parameters

```
/* prog10-3.c: command line arguments */

#include <stdio.h>

int main(int argc, char *argv[])
{
   int i;

   printf("There are %d command line arguments\n", argc);

   for (i=0; i < argc; i++)
   {
      printf("Command line argument number %d = %s\n", i, argv[i]);
   }

   return 0;
}
```

The output on my system is, provided I invoke the program as "prog11 test 123": (but this may be entirely different on yours):

```
There are 3 command line arguments
Commandline argument number 0 = C:\CINTRO\PROG11.EXE
Commandline argument number 1 = test
Commandline argument number 2 = 123
```

The first thing that stands out is the other definition of main():

```
int main(int argc, char *argv[])
{
}
```

The first parameter (argc) is the number of command line arguments, and the second parameter (argv) is an array (because of the []) of pointers (because of the *) to **char**. So argv[indexnumber] is a pointer to **char**, which in this case points to a string (array of **char**). Strings are often passed as pointers to **char**, because just using the arrayname of the string will give you the address of the array as a pointer to **char**. This does not mean that strings (arrays of **char**) and pointers to **char** are the same, just that in some cases they have equivalent behaviour. This may be a bit confusing, but it is something you'll soon get familiar with.

On most systems (not all!) argc is usually at least 1. If that is the case argv[0] is either an empty string or the filename (with or without the full path) of the executable. If argc is at least 2 there are command line arguments. In that case argv[1] is the first command line argument, argv[2] the second one, etc. until argv[argc−1].

```
printf("There are %d command line arguments\n", argc);
```

This line simply prints out the number of arguments. As noted above, be aware that even if argc is 1 that still means there are no command line arguments other than the program name, argv[0], if supported. So if for example your program requires 1 command line argument, you should check for argc==2.

```
int i;

/* ... */

for (i=0; i < argc; i++)
{
   printf("Commandline argument number %d = %s\n", i, argv[i]);
}
```

This **for** loop goes through the argv array from index 0 to argc−1 using the integer variable i as counter. For every command line argument the number and content is printed. Notice that if argc==0 the loop is never executed.

Note: as said above, the definition:

```
char *argv[]
```

Defines an array of pointers to **char**. Because an array parameter is always passed as pointer, in this case this is equivalent to:

```
char **argv
```

Which is a pointer to a pointer to **char**, that can be used as an array of pointers to **char** (see 12. Dynamic memory allocation for another example of this handy similarity between pointers and arrays).

These sort of definitions and declarations can soon become confusing, especially when pointers to functions etc. start coming into play. A handy rule to understand complex definitions is the so−called right−to−left rule. You can read all about it in rtlftrul.txt in the Snippets collection (see 17.2. Other interesting online C−related material).

11. Using files

11.1. The fileio program

```
/* prog11−1.c: fileio */

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

int main(void)
{
   char s[20];
   char *p;
   FILE *f;

   strcpy(s,"file I/O test!");

   printf("%s\n",s);

   f = fopen("testfile","w");
   if (f==NULL)
   {
      printf("Error: unable to open testfile for writing\n");
      return EXIT_FAILURE;
   }

   fputs(s,f);

   fclose(f);

   strcpy(s,"overwritten");

   printf("%s\n",s);

   f = fopen("testfile","r");
   if (f==NULL)
   {
      printf("Error: unable to open testfile for reading\n");
      return EXIT_FAILURE;
   }

   if (fgets(s, sizeof s, f) != NULL)
   {
      p = strchr(s,'\n');
      if (p!=NULL)
         *p = '\0';

      printf("%s\n",s);
   }

   fclose(f);

   if (remove("testfile") != 0)
   {
      printf("Error: unable to remove testfile\n");
      return EXIT_FAILURE;
   }
   return EXIT_SUCCESS;
}
```

This program's output should be:

```
file I/O test!
overwritten
file I/O test!
```

11.2. Using disk files

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

We need stdio.h not only for printf(), but also because it declares all file I/O functions that we're going to use here.

We need stdlib.h because we're no longer going to simply use **return** 0; but we're going to return a success/failure status to the operating system and EXIT_SUCCESS and EXIT_FAILURE are defined in stdlib.h (see 2.5. **return**).

We need string.h because we're going to work with strings, and need the functions strcpy() and strchr().

```
   char s[20];
   char *p;
   FILE *f;
```

The first line defines an array of 20 characters, which we'll use to hold a string of upto 19 characters plus 0 terminator byte.

The second line defines a pointer to **char** that will be used to store the return value of the strchr() function (see below).

The third line defines a pointer to FILE. Such a pointer to FILE is used by almost all the file I/O programs in C. It is also referred to as a stream. This pointer is the only form the FILE type is used as; you may not define variables of type FILE by themselves.

```
   strcpy(s,"file I/O test!");

   printf("%s\n",s);
```

The text "file I/O test!" is copied into s and printed.

```
   f = fopen("testfile","w");
```

OK, this is new. The fopen() function opens a file and returns a pointer to FILE that will be used by all other file I/O functions you'll use to access this file.

The first parameter to fopen() is the filename (a string). The second parameter is also a string, and can be any of the following:

```
"w"    open for writing (existing file will be overwritten)
```

```
"r"    open for reading (file must already exist)
"a"    open for appending (file may or may not exist)
"w+"   open for writing and reading (existing file will be overwritten)
"r+"   open for reading and updating (file must already exist)
"a+"   open for appending and reading (file may or may not exist)
```

Append mode ("a") will cause all writes to go to the end of the file, regardless of any calls to fseek(). By default the files will be opened in text mode (which means that '\n' will be translated to your operating system's end−of−line indicator, other sorts of translations may be performed, EOF characters may be specially treated etc.). For binary files (files that must appear on disk exactly as you write to them) you should add a "b" after the above strings. [*]

[*] Some compilers also support a "t" for text files, but since this is non−standard and not required (since it is the default) you shouldn't use it.

```
   if (f==NULL)
   {
      printf("Error: unable to open testfile for writing\n");
      return EXIT_FAILURE;
   }
```

As mentioned above, fopen() returns a pointer to FILE that will be used by the other file I/O functions. However, in case an error occurred (fopen() was unable to open the file for some reason) the returned pointer to file will be NULL (the invalid pointer value, see 10.4. Addresses and pointers).

If that happens, this program prints an error message and returns to the operating system with the EXIT_FAILURE status.

```
   fputs(s,f);
```

The fputs() function writes a string to a file. It is similar to the puts() function (see 2.4. Functions and types), but there are two differences:
− a second parameter is taken: a pointer to FILE corresponding to the file the string should be written to
− unlike puts(), fputs() does not automatically append a '\n' character

```
   fclose(f);
```

The fclose() function closes the file associated with a pointer to FILE, which has previously been opened by fopen(). After the fclose() you may not use the pointer to FILE as a parameter to any of the file I/O functions anymore (until a new file handle returned by e.g. fopen() has been stored in it, of course).

```
   strcpy(s,"overwritten");

   printf("%s\n",s);
```

This code copies a new text into s, thereby overwriting the previous text, and prints this new text.

```
   f = fopen("testfile","r");
   if (f==NULL)
   {
      printf("Error: unable to open testfile for reading\n");
      return EXIT_FAILURE;
   }
```

This code opens the previously written "testfile" for reading. The only difference with the above code is that the "w" has been replaced with "r".

```
   if (fgets(s, sizeof s, f) != NULL)
   {
   }
```

The fgets() function reads a string from a file. The middle parameter is the size of the string. We use the **sizeof** operator for this purpose, which will in this case return 20, the size of the s array. Be aware that this way of using **sizeof** will only work for arrays, not for e.g. a pointer to dynamically allocated memory (see 12. Dynamic memory allocation). It is important that you use the correct value here, because if the value is too large, too many characters can be read so that the array is overflowed, with unpredictable consequences (e.g. a core dump or system crash). The fgets() function will return NULL if an error occurred, hence the above **if**() statement.

```
      p = strchr(s,'\n');
      if (p!=NULL)
         *p = '\0';
```

The fgets() function does not automatically remove the '\n' character that terminates a line. That's why it is manually removed by the above code. The return value of strchr(s,'\n') will be a pointer to the first occurence of the character '\n' in the string, or NULL if none is found. So we store this return value in the **char** pointer p, and if it is not NULL we set the newline character it is pointing at to the '\0' terminating character ('\0' is an octal character code constant (see 2.4. Functions and types), and is therefore equivalent to 0. It is often used instead of plain 0 to make clear that it is used as a character constant), thereby effectively terminating the string in such a way that the newline character is stripped off.

```
      printf("%s\n",s);
```

The just read string is printed. If all went well, this should print the original text again.

```
   fclose(f);
```

Again, the file is closed.

```
   if (remove("testfile") != 0)
   {
      printf("Error: unable to remove testfile\n");
      return EXIT_FAILURE;
   }
```

The file we used to store our string is no longer needed, so we remove it using the remove() function. The remove() function will return 0 if it is successful, and nonzero if an error occurs (hence the above **if**() statement).

```
   return EXIT_SUCCESS;
```

If we got here, all was successful, so we return to the operating system with status EXIT_SUCCESS.

Note: fgets() and fputs() are used for reading/writing strings, usually from/to text files. When interfacing with binary files, often non−string data should be read/written. That's what the fread()/fwrite() functions are for (see 16. Overview of the standard library) [*].

[*] When reading/writing a specified file format you may be tempted to use fread()/fwrite() to read/write structures from/to a file. However, there may (and will!) often be padding bytes inbetween member variables and at the end of the structure. So in that case you'll have to resort to either reading/writing every member variable separately, or if portability isn't your greatest concern your system may support a way of turning off this padding.

For reading/writing single characters the fgetc()/fputc() functions are available (see 16. Overview of the standard library) [*]. There are also the equivalent getc()/putc() functions which may be implemented as a macro (see 13. Preprocessor macros/conditionals).

[*] When using these functions you should store the return value in a variable of type **int**, not of type **char**, because they can return EOF if an error occurred.

11.3. The interact program

```
/* prog11-2.c: interact */

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

int main(void)
{
   char name[20];
   char buffer[10];
   char *p;
   int age, ok;

   printf("Enter your name: ");
   fflush(stdout);

   if (fgets(name, sizeof name, stdin) == NULL)
   {
      printf("Error reading name from stdin\n");
      return EXIT_FAILURE;
   }

   p = strchr(name,'\n');
   if (p!=NULL)
      *p = '\0';

   fprintf(stdout, "Welcome, %s!\n", name);

   do
   {
      printf("Enter your age [1−150]: ");
      fflush(stdout);
      if (fgets(buffer, sizeof buffer, stdin) != NULL
          && sscanf(buffer, "%d", &age) == 1
          && age>=1 && age<=150)
      {
         ok = 1;
      }
      else
      {
         ok = 0;
         printf("Invalid age! Enter again...\n");
      }
   }
   while (!ok);
   printf("Your age now is %d. In 10 years your age will be %d.\n",
          age, age+10);

   return EXIT_SUCCESS;
}
```

The output of this program should be something like this:

```
Enter your name: John Doe
Welcome, John Doe!
Enter your age [1−150]: 32
Your age now is 32. In 10 years your age will be 42.
```

11.4. Using the standard I/O streams

```
   char name[20];
   char buffer[10];
   char *p;
   int age, ok;
```

Here we define an array of 20 characters, which will be used to hold the name string, and an array of 10 characters, which will be used as an input buffer for the age, another **char** pointer which will be used for the return value of strchr(), and two integer variables: one for the age, and one that will be used as a flag variable (0 for not OK, 1 for OK).

```
   printf("Enter your name: ");
   fflush(stdout);
```

As seen in 3.3. printf(), if we don't end the text with a newline (which we don't do here because we want to stay on the same line), we must add an explicit fflush(stdout). fflush() makes sure all output is sent through (it flushes the buffers). stdout is the standard output stream, which is a pointer to FILE that corresponds to the output device that is also used by printf() etc. [*]

[*] fflush() should only be used for output streams, because when used on input streams it produces undefined behaviour; on some systems it clears any pending input characters but this is definitely not standardized behaviour

```
   if (fgets(name, sizeof name, stdin) == NULL)
   {
      printf("Error reading name from stdin\n");
      return EXIT_FAILURE;
   }

   p = strchr(name,'\n');
   if (p!=NULL)
      *p = '\0';
```

An fgets() [*] with error checking, followed by the stripping of the newline (if any), very similar to what we did in the fileio program.

[*] If the user attempts to enter more characters then allowed, the excess characters will be left on stdin to be read later. This may be undesirable, but isn't dealt with here to keep the code simple.

The only difference is that this time we don't use a pointer to FILE which we opened ourselves, but the stdin standard input stream, which is a pointer to FILE that is already opened for us. The following are available:

```
stdin    standard input stream (often keyboard)
stdout   standard output stream (often screen)
stderr   standard error stream (often screen)
```

[*] Some systems support more, such as e.g. stdprn for a standard printer stream, but this is not standard.

You may notice that for obtaining input from the stdin stream, instead of fgets() some people use the gets() function. However, this function does not accept a size parameter, so it can't control the input to fit within the array and will therefore happily overwrite any memory past the array thereby causing undefined behaviour (core dump, system crash, ...). Concerning the gets() function I have only one advice: do NOT use it. At all.

```
   fprintf(stdout, "Welcome, %s!\n", name);
```

The fprintf() function is equivalent to the printf() function, except that its first parameter is a pointer to FILE where the output will be written to. In this case stdout is used, which is the default for printf() anyway, so this is equivalent to:

```
   printf("Welcome, %s!\n", name);
```

Onto the next construct:

```
   do
   {
   }
   while (!ok);
```

The code inside this loop is repeated until the ok variable becomes non−zero at the end of the loop.

```
      printf("Enter your age [1−150]: ");
      fflush(stdout);
```

Another prompt, with appropriate fflush(stdout).

```
      if (fgets(buffer, sizeof buffer, stdin) != NULL
          && sscanf(buffer, "%d", &age) == 1
          && age>=1 && age<=150)
      {
         ok = 1;
      }
      else
      {
         ok = 0;
         printf("Invalid age! Enter again...\n");
      }
```

First another string is read from stdin using fgets() and the return value checked (this time we don't bother to remove the newline because we are going to convert this buffer to an integral variable anyway).

printf() has an opposite: the scanf() function. It also uses a format string like printf(), and is passed a variable number of arguments consisting of variables that correspond to a specific format specifier in the format string. But because the variables will be used to store an input result in, they must be prepended with the & address operator, so that their address is passed instead of their value (this is not required for arrays, such as strings, because they are always passed via their address anyway).

The format specifiers are mostly the same as those for printf() (see 3.3. printf). However, because an address is passed instead of a value there is no more **float** −> **double**, **char** −> **int** or **short** −> **int** promotion, so the following new format specifiers or modifiers are added/changed:

```
Format specifiers:
%i    unlike %d, allows for entering hex (0x...) or octal constants (0...)
%n    number of characters read upto now will be stored in signed int
%c    exact number of characters specified by width specifier will be read
      (when no width specifier is present, 1 character will be read)
%[character list]   longest string consisting of the characters in the
                    character list will be read
%[^character list]  longest string consisting of characters other than
                    those in the character list will be read
Modifiers:
h     for d,u,x,o: short int instead of int
      for n: result will be stored in short int
l     for d,u,x,o: long int instead of int
      for n: result will be stored in long int
      for f,e,g: double instead of float
      (note that this is different from printf(), where %lf etc.
       may not be used for printing a double)
*     assignment suppression
```

But the scanf() function itself is not very reliable. If the user enters wrong data, it can leave things on the input stream etc. which may be very annoying.

A better approach is to read a string using the fgets() function, and then process the input from this string using the sscanf() function. The sscanf() function works like the scanf() function, except that it takes as its first parameter the string where it has to read its input from. So in our case:

```
      sscanf(buffer, "%d", &age)
```

Reads a decimal integer (%d) from the string buffer and stores it in the integer variable age (notice the address operator!).

However, we have to check whether all of this was successful because the user might also have entered something like "foo bar" instead of a valid age. Hence the code:

```
          sscanf(buffer, "%d", &age) == 1
```

sscanf(), like scanf(), returns the number of variables it has been able to assign to properly. In our case this should be 1 if all went well.

```
          && age>=1 && age<=150
```

Furthermore we check that the entered age is between 1 and 150, otherwise we reject it as well.

We use a special feature of the logical operators && (and ||) here: they are the only operators that are guaranteed to execute from left to right, so we can be sure that by the time the sscanf() is reached the fgets() has been executed and by the time we check the value of age, the sscanf() has been executed. You may not rely on this with other operators (except the comma operator and the ?: operator). The && and || operators also have something called short−circuiting: if the left operand of the && operator is false, the right operand is guaranteed not to be evaluated. Similarly, if the left operand of the || operator is true, the right operand is guaranteed not to be evaluated.

```
      if (fgets(buffer, sizeof buffer, stdin) != NULL
          && sscanf(buffer, "%d", &age) == 1
          && age>=1 && age<=150)
      {
         ok = 1;
      }
      else
      {
         ok = 0;
         printf("Invalid age! Enter again...\n");
      }
```

So if all these conditions are met, we set ok to 1 which will cause the loop to be ended. If they are not, we set ok to 0 which will cause the loop to be executed again, and we write an error message to inform the user that he has to re−input.

```
   printf("Your age now is %d. In 10 years your age will be %d.\n",
          age, age+10);
```

To make use of the fact that we now have the user's age in integer format, we do a little calculation with it.

Another function that is often used to read input from stdin is getchar(). It reads a single character, but does require the user to press enter. To avoid this, compiler/operating system specific functions have to be used. You have to be careful when using it similar to this:

```
if ((c=getchar()) != EOF)
   /* code */ ;
```

In the above code, the variable c must be of type **int**, not char! The getchar() function returns an **int**, which may be an **unsigned** character value if a character was successfully read, or EOF (a negative value) if an error occurred.

12. Dynamic memory allocation

12.1. The persons program

```
/* prog12−1.c: persons */

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

struct person {char name[30];   /* full name */
               unsigned birthday,    /* 1..31 */
                   birthmonth,  /* 1..12 */
                   birthyear;   /* 4 digits */
              };

int main(void)
{
   struct person *personlist;
   unsigned number_of_persons, num;
   char buffer[30];
   char *p;
   int year, month, day;
   int ok;

   do
   {
      printf("Enter the number of persons: ");
      fflush(stdout);
      if (fgets(buffer, sizeof buffer, stdin) != NULL
          && sscanf(buffer, "%u", &number_of_persons) == 1)
      {
         ok = 1;
         if (number_of_persons>0)
         {
            personlist = malloc(number_of_persons * sizeof(struct person));
            if (personlist==NULL)
            {
               printf("Not enough memory to store that many persons!\n");
               ok = 0;
            }
         }
      }
      else
      {
         ok = 0;
         printf("Invalid number! Enter again...\n");
      }
   }
   while (!ok);

   if (number_of_persons==0)
   {
      printf("OK, perhaps another time!\n");
      return 0;
   }

   for (num=0; num<number_of_persons; num++)
   {
      printf("\nEnter the information for person #%u:\n", num);
      printf("Name: ");
      fflush(stdout);
      if (fgets(buffer, sizeof buffer, stdin) == NULL)
      {
         printf("Error reading from stdin; input aborted\n");
         number_of_persons = num;
         break;
      }
      p = strchr(buffer,'\n');
      if (p!=NULL)
         *p = '\0';
      if (strlen(buffer)==0)
      {
         printf("Input stopped\n");
         number_of_persons = num;
         break;
      }
      strcpy(personlist[num].name, buffer);
      do
      {
         printf("Birthday [YYYY-MM-DD]: ");
         fflush(stdout);
         if (fgets(buffer, sizeof buffer, stdin) != NULL
             && sscanf(buffer, "%d-%d-%d", &year, &month, &day) == 3
             && year>=1000 && year<=9999
             && month>=1 && month<=12
             && day>=1 && day<=31)
         {
            ok = 1;
         }
         else
         {
            ok = 0;
            printf("Invalid birthday! Enter again...\n");
         }
      }
      while (!ok);
      personlist[num].birthyear = year;
      personlist[num].birthmonth = month;
      personlist[num].birthday = day;
   }

   printf("\nOK, thank you.\n");

   printf("\nYou entered the following data:\n");

   printf("\n%-10s%-30s%s\n", "Number", "Name", "Birthday");

   for (num=0; num<number_of_persons; num++)
   {
      printf("%-10u%-30s%04d-%02d-%02d\n",
                num,
                personlist[num].name,
                personlist[num].birthyear,
                personlist[num].birthmonth,
                personlist[num].birthday);
   }

   free(personlist);

   return 0;
}
```

An example output of this program is:

```
Enter the number of persons: 10

Enter the information for person #0:
Name: John Doe
Birthday [YYYY-MM-DD]: 70-5-31
Invalid birthday! Enter again...
Birthday [YYYY-MM-DD]: 1970-5-31

Enter the information for person #1:
Name: Foo Bar
Birthday [YYYY-MM-DD]: 1948-1-1

Enter the information for person #2:
Name:
Input stopped

OK, thank you.

You entered the following data:
```

```
Number    Name                          Birthday
0         John Doe                      1970−05−31
1         Foo Bar                       1948−01−01
```

12.2. Dynamically allocating memory

Well, that seems to be a reasonably complex program, doesn't it ? It's sort of a mini−database, and combines our recently acquired knowledge of structures, strings, pointers and using the standard input/output streams together with a completely new feature: dynamic memory allocation. As usual, we'll explain things step by step:

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

As usual, stdio.h is needed for our input/output, and string.h for the string functions we need, like strchr() and strcpy(). However, stdlib.h is also needed because it contains the declarations for the malloc() and free() functions, which we'll use for our dynamic memory allocation.

```
struct person {char name[30];   /* full name */
               unsigned birthday,    /* 1..31 */
                   birthmonth,  /* 1..12 */
                   birthyear;   /* 4 digits */
              };
```

As seen in 6. Structs, unions and bit−fields, this is a definition of a structure type. It consists of an array of 30 chars, and 3 **unsigned** variables. The comments indicate what each member variable will hold:
− the name array will contain the person's full name as a string
− the birthday, birthmonth and birthyear member variables will contain the person's birthdate; the exact values used are documented so that there can be no confusion over the representation of the date

Don't forget that what we've defined here is just a structure type, we don't have any actual variables capable of holding any information yet.

```
   struct person *personlist;
   unsigned number_of_persons, num;
   char buffer[30];
   char *p;
   int year, month, day;
   int ok;
```

Now, to hold that information we could have used an array, for example:

```
   struct person personlist[100];
```

But this way we would limit the number of persons to 100, and someday someone will need more. If we would use a very large number, say 10000, we might seriously reduce the chance of the array being too small, but in most cases we'd also be wasting a lot of space (and it is even possible that your compiler doesn't support very large arrays like that).

How do we solve this problem ? Instead of determining the number of persons in our program's source code, we let the user determine it at runtime, dynamically. That's why we'll need dynamic memory allocation. To be able to do that we must define a pointer (see 10.4. Adresses and pointers) to our type (in this case **struct** person). This pointer doesn't yet point anywhere valid and no memory is allocated yet, so you shouldn't make the mistake of attempting to use the pointer before any memory has been allocated for it.

That explains the following definition of a pointer to **struct** person:

```
   struct person *personlist;
```

The other variable definitions are:

```
   unsigned number_of_persons, num;
   char buffer[30];
   char *p;
   int year, month, day;
   int ok;
```

Here we define two **unsigned** integers, number_of_persons and num, of which the former will hold − as its name suggests − the number of persons, and the latter will be used as a counter. The buffer array will, as in previous examples, be used to buffer the user's input before processing, and the **char** pointer will be used to hold the return value of strchr(). The year, month and day variables will be used to temporarily hold a person's birthday, and the ok variable will again be used as a flag variable (1 for ok, 0 for not ok).

```
   do
   {
   }
   while (!ok);
```

As we've done before, this loop will repeatedly prompt the user for input until the entered data is valid (which will be indicated by setting the flag value ok to 1).

```
      printf("Enter the number of persons: ");
      fflush(stdout);
      if (fgets(buffer, sizeof buffer, stdin) != NULL
          && sscanf(buffer, "%u", &number_of_persons) == 1)
      {
         ok = 1;
      }
      else
      {
         ok = 0;
         printf("Invalid number! Enter again...\n");
      }
```

This is also analogous to our previous examples. Again, an explicit fflush(stdout); is needed because we don't end our prompt with a newline. The only difference in the sscanf() call is that the %u format specifier is used instead of the %d format specifier because number_of_persons is an **unsigned** integer instead of a **signed** integer.

However, after the ok=1; statement something new follows:

```
         if (number_of_persons > 0)
         {
            personlist = malloc(number_of_persons * sizeof(struct person));
            if (personlist==NULL)
            {
               printf("Not enough memory to store that many persons!\n");
            ok = 0;
         }
      }
```

If the number of persons entered is greater than 0, memory is allocated for our personlist pointer [*]. The malloc() function is used for that.

[*] We explicitly check for the 0 case, because according to the standard, the behaviour of malloc(0) is implementation−defined.

The parameter of malloc() is the number of bytes that should be allocated. In our case we use:

```
   malloc(number_of_persons * sizeof(struct person))
```

The **sizeof** operator (as seen in 4.2. Operators and typecasts) will give the size of its operand in bytes (the form with parentheses is used for types, the parentheses are not required for variables). Therefore **sizeof**(**struct** person) will give the number of bytes required to store 1 person structure [*]. By multiplying this value by the entered number of persons, we get the total number of bytes required to store all person structures.

[*] Another, slightly better method would be to use **sizeof** *personlist instead of **sizeof**(**struct** person), because this will remain correct even if we change the type that personlist points to to something other than **struct** person.

The malloc() function returns a generic pointer to **void** which points to the allocated memory. By assigning this value to the personlist pointer, our pointer now points to actual usable memory:

```
         personlist = malloc(number_of_persons * sizeof(struct person));
```

However, it is quite possible for the malloc() function to fail, usually because there is not enough memory available. In that case the returned pointer will be the NULL pointer (invalid pointer value). So we must explicitly check for this case after every call to malloc():

```
         if (personlist==NULL)
         {
            printf("Not enough memory to store that many persons!\n");
            ok = 0;
         }
```

In our case, if the malloc() fails, we just reset our flag variable to 0 (not ok) and the user will be asked to enter a new value.

```
   if (number_of_persons==0)
   {
      printf("OK, perhaps another time!\n");
      return 0;
   }
```

If the entered number of persons was 0, no memory has been allocated in our above code, and we return to the operating system.

```
   for (num=0; num<number_of_persons; num++)
   {
   }
```

We loop using the num variable as counter from 0 to number_of_persons−1. We don't count from 1 to number_of_persons because, as seen before, in C array indexes etc. start counting from 0, and it's best to maintain consistency throughout the program.

```
      printf("\nEnter the information for person #%u:\n", num);
      printf("Name: ");
      fflush(stdout);
      if (fgets(buffer, sizeof buffer, stdin) == NULL)
      {
         printf("Error reading from stdin; input aborted\n");
         number_of_persons = num;
         break;
      }
      p = strchr(buffer,'\n');
      if (p!=NULL)
         *p = '\0';
      if (strlen(buffer)==0)
      {
         printf("Input stopped\n");
         number_of_persons = num;
         break;
      }
      strcpy(personlist[num].name, buffer);
```

Most of this should also look familiar. After the input has been entered into the buffer array and checked that no error has occurred, and the newline has been removed, we check whether buffer is an empty string (i.o.w. the user has pressed enter without entering any data). If that is the case, we stop our input, and adjust the number_of_persons variable accordingly (we don't need the original value of number_of_persons, namely the total number of persons for which memory has been allocated, further in this program; otherwise we would have to use a separate variable for the number of persons for which data has been entered) and exit the **for**() loop with a **break**; statement [*]. If the fgets() function returned NULL an analogous logic is followed.

[*]See 9.3. Using **break**,**continue**,**goto**. This is an example of where **break**; can be useful.

After we verified that the input is valid (i.o.w. not an empty string), we copy the string in buffer to the name member variable of the current structure in the personlist. The n−th element (in this case structure) in the memory that personlist points to is accessed like this: (here as well, n starts counting at 0)

```
   personlist[n]
```

In other words, just like if personlist was an array of **struct** instead of a pointer to struct. This is a very handy similarity between arrays and pointers, however you must not let it confuse you into thinking that arrays and pointers are the same.

The member selector operator (.) is then used to access a member variable of this structure, just like if it were a simple **struct** variable, so that:

```
   personlist[num].name
```

Gives us the name member variable of structure number num (which starts counting at 0) in the memory pointed to by personlist, which is used as the target string of our strcpy() function:

```
      strcpy(personlist[num].name, buffer);

      do
      {
         printf("Birthday [YYYY−MM−DD]: ");
         fflush(stdout);
         if (fgets(buffer, sizeof buffer, stdin) != NULL
             && sscanf(buffer, "%d−%d−%d", &year, &month, &day) == 3
             && year>=1000 && year<=9999
             && month>=1 && month<=12
             && day>=1 && day<=31)
         {
            ok = 1;
         }
         else
         {
            ok = 0;
            printf("Invalid birthday! Enter again...\n");
         }
      }
      while (!ok);
```

Another input loop which will keep repeating until the entered data is valid. The sscanf() function this time scans for 3 variables instead of 1 (that's also why we compare its return value against 3 instead of 1). In the **if** condition, which again makes use of the fact that the && operator always evaluates from left to right, after the input has been read without errors and properly processed by the sscanf() function, the year, month and day values are checked for validity [*], and if all this went well, the ok flag variable is set to 1.

[*] Invalid dates, such as the 31st of February, still go undetected. If you wish to detect these cases also, keep in mind that February has 28 days except in leap years when it has 29. All years that are divisible by 4 but not by 100, unless they're also divisible by 400, are leap years.

```
      personlist[num].birthyear = year;
      personlist[num].birthmonth = month;
      personlist[num].birthday = day;
```

The just entered and verified birthday values are assigned to the corresponding member variable in our current person structure. The addressing is completely analogous to that of the name member variable above.

```
   printf("\nOK, thank you.\n");

   printf("\nYou entered the following data:\n");

   printf("\n%−10s%−30s%s\n", "Number", "Name", "Birthday");
```

Input is ended, and we're going to print out the contents of all entered person structures. To write out the title of this table we use a somewhat peculiar printf() call. The %−10s format specifier will print a string, left−aligned into a 10 character wide field. The value of this string is a literal. We might also have manually aligned this title, but this is more obvious and easier to change. The %−30s format specifier is completely analogous. See 3.3. printf for details.

```
   for (num=0; num<number_of_persons; num++)
   {
      printf("%−10u%−30s%04d−%02d−%02d\n",
                num,
                personlist[num].name,
                personlist[num].birthyear,
                personlist[num].birthmonth,
                personlist[num].birthday);
   }
```

We loop through all entered person structures again with a **for**() loop from 0 to number_of_persons−1 and print out the contents of every structure.

The printf() format string aligns the data in the same way as was done in our title bar (except of course that some of the strings are replaced by numerical data). The %04d format specifier means to align the integer variable to the right in a field of 4 characters wide, and fill up at the left with 0s. The %02d format specifiers are analogous. Again, see 3.3. printf for details.

```
   free(personlist);
```

Now that we no longer need the memory we allocated, we must free it by calling the free() function with our previously allocated personlist pointer as parameter. [*]

[*] While the standard guarantees that free(NULL); will work fine, you may not pass any pointers to free() other than those received from malloc(), or nasty things may happen (undefined behaviour).

Note: instead of malloc() you can also use calloc() to dynamically allocate memory. The calloc() function takes two parameters, which it multiplies to get the actual number of bytes that should be allocated [*], and initializes the allocated memory to all−bits−0 (however, this will only correctly initialize integral variables; floating−point variables are not guaranteed to be set to 0.0 and pointers are not guaranteed to be set to NULL).

[*] The result of the multiplication should not be larger than what would normally fit in either, so you can't rely on this feature to get around compiler limits (such as the 64K limit on 16−bit compilers, for example).

Note 2: you can use the realloc() function to resize an allocated block of memory while preserving its contents (see 16. Overview of the standard library). However you must watch out for these:
− the realloc() function may actually move the memory block
− if the realloc() function returns NULL, the original memory block is still allocated and unchanged

13. Preprocessor macros/conditionals

13.1. The preproc program

```
/* prog13−1.c: preproc */

#include <stdio.h>
#include <assert.h>

/* uncomment the line below for a test of the USEDUTCH macro */
/* #define USEDUTCH */

/* uncomment the line below for a test of the assert() macro */
/* #define TEST_ASSERT */

#define PROGVERSION 1

#define MAX(a,b) ((a) > (b) ? (a) : (b))

int main(void)
{
   int q = 5, r = 3;
#ifdef USEDUTCH

   printf("Dit is versie %d van het preproc programma\n", PROGVERSION);

   printf("De grootste waarde van %d en %d is %d\n", q, r, MAX(q,r));

   printf("Bestand %s, lijn %ld, compilatiedatum %s\n",
          __FILE__, (long)__LINE__, __DATE__);

#else

   printf("This is version %d of the preproc program\n", PROGVERSION);

   printf("The greatest value of %d and %d is %d\n", q, r, MAX(q,r));

   printf("File %s, line %ld, compilation date %s\n",
          __FILE__, (long)__LINE__, __DATE__);

#endif

#if defined(TEST_ASSERT)
   q++;
#endif

   assert(q==5);

   return 0;
}
```

This program's output should be something like this:

```
This is version 1 of the preproc program
The greatest value of 5 and 3 is 5
File prog13−1.c, line 36, compilation date Aug 05 1998
```

When both the **#define** USEDUTCH and **#define** TEST_ASSERT lines are uncommented, the output may resemble this:

```
Dit is versie 1 van het preproc programma
De grootste waarde van 5 en 3 is 5
Bestand prog13−1.c, lijn 27, compilatiedatum Aug 05 1998
Assertion failed: a==5, file prog13−1.c, line 44
ABNORMAL PROGRAM TERMINATION
```

13.2. Using preprocessor macros/conditionals

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

The assert.h header is included because it defines the assert() macro (see 13.3. Using the assert() macro).

```
/* uncomment the line below for a test of the USEDUTCH macro */
/* #define USEDUTCH */

/* uncomment the line below for a test of the assert() macro */
/* #define TEST_ASSERT */
```

We've already seen in 2.3. **#include** that all lines that begin with # are directives for the preprocessor. After the **#include** preprocessor directive, the **#define** preprocessor directive is the most commonly used one.

When the above lines are uncommented (the surrounding */* and */* removed), they will define preprocessor macros named USEDUTCH and TEST_ASSERT. In this case, these macros have only names, no corresponding values. This sort of macro is usually used to turn on or off a certain feature in a program at compile−time. In our case the USEDUTCH macro will, when defined, make the program switch to using Dutch instead of English, and the TEST_ASSERT macro will cause the assert() macro to be triggered (see below).

```
#define PROGVERSION 1
```

This is also a preprocessor macro definition, but this time the macro not only has a name but also a corresponding value (1). Everywhere in the program (outside of a string literal) where the PROGVERSION symbol appears, it will be replaced by the value 1 during preprocessing (right before the actual compilation).

This sort of macro is usually used to avoid hardcoding certain "magic numbers", usually for one of these two reasons:
− using a symbolic constant makes the program clearer, e.g. PI may be more obvious than just 3.1415 and is also less likely to contain typing errors
− the value to which the macro corresponds may change (e.g. like in our example, where it is the program version) and by using a macro the value needs to be changed only in one place, instead of in a lot of places possibly spread throughout the source code

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
```

This is yet another preprocessor macro definition, and a more advanced form. The (a,b) are parameters just like if the macro were a function that's called. Everywhere in the program where the MAX(a,b) macro is used, it will be replaced by the following code:

```
   ((a) > (b) ? (a) : (b))
```

In which the a and b parameters will be replaced by whatever was inside the MAX() parentheses (see below for an example).

This sort of macro is often used to implement a sort of "fake functions", where the code is actually inserted in the program instead of called like a function. However, there are two very important considerations here:
− watch out for precedence: safest is to put parentheses around the entire macro (in case the macro is used next to a higher precedence operator) and around every parameter (in case the actual parameter value that the macro is "called" with contains lower precedence operators), as is done in the above example.
− watch out for multiple evaluation of the parameters: in our above example the a and b parameters are evaluated twice, once in the comparison and once in the actual value returned. This may become a problem if the parameters passed to the macro do not just have values, but also side−effects, like e.g. a function call which would in this case be called twice instead of once, or the ++ or −− operators which would in this case increment or decrement the value twice instead of once [*]. If there is any risk of such problems, it's probably a good idea to make your function−like macro a real function; using a macro for optimization purposes is usually a bad idea.

[*] Actually, that isn't even guaranteed. Using operators like ++ and −−, which modify the same variable twice in the same expression (and not separated by a so−called sequence point, but I'm not going to explain all that here; see question 3.8 in the c.l.c FAQ, the URL can be found in 17.2. Other interesting C−related online material) gives undefined behaviour according to the standard, which means anything may happen.

Here are some examples of what might happen if you ignore the above warnings:
− not putting parentheses around macro parameters:

```
#define SQR(x) (x*x)
/* SQR(a+2) yields (a+2*a+2) */
```

− not putting parentheses around the entire macro value:

```
#define INC(x) (x)+1
/* INC(i)*3 yields (i)+1*3 */
```

− not being careful with multiple evaluation of macro parameters:

```
#define CHECK_BOUNDS(value,min,max) ((value)>=(min) && (value)<=(max))

/* the ask_for_integer() function asks the user to enter a number
   and returns this number as an integer */
int ask_for_integer(void);

/* CHECK_BOUNDS(ask_for_integer(),1,100) will ask the user to
   enter the number twice */
```

In a macro value, the operator '#' may be used to expand a macro parameter to the textual appearance of the actual argument in the form of a string literal, exactly as supplied in the calling source. The parameter must follow the '#' immediately. For example:

```
#define TEXT(m) #m
TEXT(300)
```

will expand to a string "300" (including the quotation marks). (Quotation marks and backslashes in the macro argument will be escaped with a backslash when generating the string literal.)
This feature is useful if you want to export parts of the source to the program output. It is mostly used for debugging or logging purposes.

In a macro value, the operator '##' may be used to concatenate a macro parameter's invocation argument with adjacent text. This can be used to generate identifiers. For example:

```
#define GEN_ID(a,b) a##b
int GEN_ID(i,9);
```

expands to:

```
int i9;
```

Any macro may be undefined using the #undef preprocessor directive. For example, the following code will undefine the macro TEST if it is defined, or have no effect otherwise:

```
#undef TEST
```

Now onto another preprocessor construct:

```
#ifdef USEDUTCH

#else
```

```
#endif
```

The code between the **#ifdef** and the **#else** lines will only be compiled if the USEDUTCH macro has been defined, and the code between the **#else** and **#endif** lines will only be compiled if that macro has not been defined.

The difference with a normal **if** statement is that this is evaluated by the preprocessor at compile−time. So while **if** code that may never be executed would probably still be in the executable, it will not be when a preprocessor **#ifdef** is used.

See below for a more advanced form of preprocessor conditionals.

```
   printf("This is version %d of the preproc program\n", PROGVERSION);

   printf("The greatest value of %d and %d is %d\n", q, r, MAX(q,r));

   printf("File %s, line %ld, compilation date %s\n",
          __FILE__, (long)__LINE__, __DATE__);
```

(or the equivalent version in Dutch)

The first line prints the program version, as defined in our PROGVERSION macro. Since this is an integer constant (in our case 1) the %d printf format specifier is used.

The second line uses the MAX(a,b) macro. The MAX(q,r) will be replaced by:

```
   ((q) > (r) ? (q) : (r))
```

Which, as seen in 7.4. Using the ?: operator, will give the value of q in case q is greater than r, or the value of r otherwise.

This means that the third number that is printed will always be the greatest value of the two.

The third line uses some macros that are pre−defined for you by the compiler. The __FILE__ macro expands to a string containing the current filename, the __LINE__ macro expands to a decimal constant containing the current line number in the file, and the __DATE__ macro expands to a string containing the compilation date. The __LINE__ macro is cast to a **long** because we use the %ld format specifier and we don't know of what type the line number constant will be (in a very long program it might be too large to fit in an integer). The following such macros are available:

```
__FILE__   source filename (string literal)
__LINE__   line number (decimal constant)
__DATE__   compilation date (string literal)
__TIME__   compilation time (string literal)
__STDC__   set to 1 for standard−conforming mode (decimal constant)
```

On to the next lines of code:

```
#if defined(TEST_ASSERT)
   q++;
#endif
```

This is equivalent to:

```
#ifdef TEST_ASSERT
   q++;
#endif
```

Which means that the code to increment the q variable by 1 will only be compiled when the TEST_ASSERT macro is defined (see below for the effect this increment has).

The **#if** preprocessor directive is a lot more flexible than **#ifdef**. It allows for expressions combined using operators, such as e.g.:

```
#if defined(MYCOMPILER_VERSION) && (MYCOMPILER_VERSION>=2)
   /* some code specific to MYCOMPILER versions greater than 2 */
#endif
```

Which means the code between the **#if** and the **#endif** will only be compiled if the MYCOMPILER_VERSION macro is defined, and the value of this macro is greater than or equal to 2.

You can only use things that are known at preprocessing−time in an **#if** statement. For example, you can't check for a typedef using **#if defined**(mytypedef) and you can't check for certain sizes using something like **#if sizeof**(**int**)>=4. As an alternative for the latter, you can check the value of e.g. UINT_MAX from limits.h.

There's also the **#elif** preprocessor directive, which works similar to an **else if** () construct.

The **#error** directive can be used to force a diagnostic message from the compiler. The **#pragma** directive can be used to control compiler specific features. The **#line** directive can be used to change the current line number value. See your compiler manual / online help if you want to know more details.

13.3. Using the assert() macro

```
   assert(q==5);
```

The assert() macro, defined in assert.h, can be used to make absolutely sure a certain condition is fulfilled, intended for debugging purposes.

What the assert() macro does depends on whether the NDEBUG macro is defined when assert.h is included or not. If the NDEBUG (no debugging) macro is defined at that time, the assert() macro will do nothing (this is intended for release code, to get rid of unnecessary debugging code).

If the NDEBUG macro isn't defined when assert.h is included, the assert() macro checks the condition that is passed to it as parameter. If this condition is true, nothing happens. If this condition is false, however, the program prints an error message containing the source file, line number and failed condition, and aborts the program abnormally.

Obviously this is not very elegant, but it is only intended for debugging purposes. The conditions you check with it should be conditions that absolutely have to be fulfilled, unless some serious error has occurred in the program's internal logic. It should never be used to trap errors that are caused by external factors such as user input or not−enough−memory errors etc.

In our example, the assert() macro makes sure the q variable equals 5. It should, because we assigned that value to it in its definition. However, when the TEST_ASSERT macro is defined some code above has incremented the value of q by 1, so that it no longer equals 5 but 6, and this will of course trigger the assert() macro.

14. Variable argument functions

14.1. The vararg program

```
/* prog14-1.c: vararg */

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

int sumintegers(int numbers, ...)
{
   va_list vl;
   int i;
   int sum;

   va_start(vl,numbers);

   sum = 0;
   for (i=0; i<numbers; i++)
      sum += va_arg(vl,int);

   va_end(vl);

   return sum;
}

int main(void)
{
   int a,b;

   printf("Sum #1 = %d\n",
          sumintegers(10,1,2,3,5,7,9,11,13,17,19));

   a = 100;
   b = 200;

   printf("Sum #2 = %d\n",
          sumintegers(2,a,b));

   return 0;
}
```

The output of this program should be:

```
Sum #1 = 87
Sum #2 = 300
```

14.2. Using variable argument functions

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

The standard header stdarg.h is needed for variable argument functions.

```
int sumintegers(int numbers, ...)
{
}
```

This seems like a normal function, except for the three dots. This is called an ellipsis and means that a variable number of arguments may follow. An ellipsis must be the last parameter of a function and it may not be on its own, i.o.w. there should always be at least one normal, fixed parameter, before the variable argument list can begin.

```
   va_list vl;
   int i;
   int sum;
```

The vl variable of type va_list is required for processing the variable argument list. The i variable is just a counter and the sum variable will hold the calculated sum.

```
   va_start(vl,numbers);
```

The va_start() function initializes the variable argument list. This call is required before any of the variable arguments can be accessed. The first parameter must be the abovementioned va_list variable and the second parameter must be the last fixed parameter of the function (i.o.w. the parameter right before the ellipsis). That's also a reason why the ellipsis can never be on its own, and at least one fixed parameter is required.

```
   sum = 0;
   for (i=0; i<numbers; i++)
      sum += va_arg(vl,int);
```

This **for**() loop simply adds up a number of integers. The fixed numbers parameter is used to pass the number of integers that will follow in the variable argument list. The va_arg() function returns the next variable argument. The first parameter to va_arg() should again be the va_list variable and the second parameter should be the type of the argument (in our case **int**).

We use the fixed numbers parameter here to determine how many variable arguments there are. Another commonly used method is to use some sort of "terminator" value, e.g. the value 0 would end the list. However, keep in mind that at least 1 fixed parameter is required.

```
   va_end(vl);
```

The variable argument list processing is closed with the va_end() function, again with the va_list variable as parameter. This is required.

```
   return sum;
```

And the calculated sum is returned to the caller.

```
   int a,b;

   printf("Sum #1 = %d\n",
          sumintegers(10,1,2,3,5,7,9,11,13,17,19));

   a = 100;
   b = 200;

   printf("Sum #2 = %d\n",
          sumintegers(2,a,b));
```

This code uses the sumintegers() function twice: once to calculate the sum of 10 constant numbers, and once to calculate the sum of 2 variables. The disadvantage to this approach is that we must manually count the number of arguments, a disadvantage that would not be present if we had used the terminator−value approach mentioned above.

14.3. Example: a printf() encapsulation

```
/* prog14−2.c: myprintf */

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

void myprintf(const char *text, const char *fmt, ...)
{
   va_list vl;

   printf("myprintf(): %s: ", text);

   va_start(vl,fmt);
   vprintf(fmt,vl);
   va_end(vl);

   printf("\n");
}

int main(void)
{
   int a = 5;

   myprintf("Test #1", "a = %d", a);

   myprintf("Test #2", "File name %s, line number %ld",
                       __FILE__, (long)__LINE__);

   return 0;
}
```

The output should be something like:

```
myprintf(): Test #1: a = 5
myprintf(): Test #2: File name prog14−2.c, line number 26
```

One of the most commonly used variable argument functions is printf(). To determine the number and type of its arguments it uses a format string with format specifiers (see 3.3. printf).

Sometimes it may be useful to encapsulate the printf() function in a function of your own, so that you can add some text of your own, or write a copy of the text to a logfile, etc. That's what the above program does.

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

Of course, the stdarg.h header is needed here as well.

```
void myprintf(const char *text, const char *fmt, ...)
{
}
```

Our myprintf() function will accept two fixed parameters of type pointer to constant **char** (i.o.w. strings that will not be modified, so for example a string literal would be a suitable value for these) followed by a variable argument list. The first parameter accepts some text which will be printed before the normal printf() output and the second parameter is the format string for the printf() function.

```
   va_list vl;
```

The vl variable is familiar.

```
   printf("myprintf(): %s: ", text);
```

Some text is printed, to show that we have encapsulated the printf() function.

```
   va_start(vl,fmt);
```

The variable argument list is initialized. The last fixed parameter before the variable argument list is the format string.

```
   vprintf(fmt,vl);
```

The vprintf() function is similar to the printf() function. However, the variable values corresponding to the format specifiers in the format string are no longer passed as separate parameters after the format string, but instead a va_list variable is accepted, so that our variable argument list can be used instead.

```
   va_end(vl);
```

The variable argument list processing is closed. This is the only thing we can do with the va_list after the vprintf() call (or a call to any function that takes a va_list parameter). If we wanted to use the va_list variable after the call, we would have to close processing with va_end(), restart it with va_start(), do what has to be done, and close it again with va_end().

```
   printf("\n");
```

A newline is appended to the previous output.

```
   int a = 5;

   myprintf("Test #1", "a = %d", a);

   myprintf("Test #2", "File name %s, line number %ld",
                       __FILE__, (long)__LINE__);
```

A small test of our myprintf() function, by printing the value of a variable, and the value of the __FILE__ and __LINE__ macros (see 13. Preprocessor macros/conditionals).

15. Modular approach

15.1. The module program

This program consists of three files:

```
/* prog15−1.c: module 1 */

#include <stdio.h>

#include "prog15−2.h"

int main(void)
{
   myfunction(25);

   printf("This is %s. The value of myvariable is %d\n",
          __FILE__, myvariable);

   return 0;
}
```

```
/* prog15−2.c: module 2 */

#include <stdio.h>

int myvariable = 42;

void myfunction(int number)
{
   printf("This is %s. The sum of number and myvariable is %d.\n",
          __FILE__, number+myvariable);
}
```

```
/* prog15−2.h: module 2 header file */

extern int myvariable;

void myfunction(int number);
```

The output of this program should be something like this:

```
This is prog15−2.c. The sum of number and myvariable is 67.
This is prog15−1.c. The value of myvariable is 42
```

To compile this program you must compile and link both modules. Depending on your system, this may be done by putting both prog15−1.c and prog15−2.c on your command line or by inserting them in your make− or projectfile. If during linking you get "unresolved external myvariable" or similar error messages, you are not linking in the correct modules. See your compiler manual for more information.

15.2. Using different modules

When you start writing larger programs, you'll soon notice that if you keep all your functions in one module (one .c file), this file soon becomes very difficult to overview. Not to mention the fact that if one thing changes in this file the entire program needs to be recompiled.

By splitting up your program in separate modules, each combining the functions that are related to one specific task, the program will become much easier to maintain and only those modules where something has changed need to be recompiled.

Let's start by analyzing the prog15−1.c module:

```
#include "prog15−2.h"
```

As mentioned in 2.3. **#include**, this form with the double quotes instead of the angle brackets is used to include header files that are part of your program. In this case the prog15−2.h header declares the variables and functions in the prog15−2.c module that we'll be using in our prog15−1.c module. The contents of this header file are discussed below.

```
int main(void)
{
   myfunction(25);

   printf("This is %s. The value of myvariable is %d\n",
          __FILE__, myvariable);

   return 0;
}
```

The prog15−1.c module contains the main() function (only 1 module in a program may contain a main() function). This code uses a function (myfunction) and a variable (myvariable) that are not defined in this module. They are declared in the prog15−2.h header file and defined in the prog15−2.c module, which we'll analyze now:

```
int myvariable = 42;

void myfunction(int number)
{
   printf("This is %s. The sum of number and myvariable is %d.\n",
          __FILE__, number+myvariable);
}
```

As we've seen in 3.2. Defining variables, since myvariable is not inside a function and it isn't defined using the **static** keyword, this is a global variable. That means it can be accessed from all modules, in our case this means the prog15−1.c module can access it.

Similarly, since the myfunction() is not defined as **static**, it can also be accessed from all modules. If you don't want this, you should define it like this:

```
static void myfunction(int number)
{
}
```

And now the header file that allows the prog15−1.c module to access the global variables and functions from the prog15−2.c module:

```
extern int myvariable;

void myfunction(int number);
```

The first line looks like the definition of an integer variable called myvariable. The **extern** makes it a declaration of a variable that is defined in another module (in this case prog15−2.c). The **extern** is like saying "you can assume the variable is there and that it is of this type". If it turns out afterwards that the variable isn't really there, your linker will most likely object violently.

The second line looks like the definition of the myfunction() function. The only difference is that where normally the function definition would be between { and }, there's now a semicolon. This is called a prototype. Similar to the above case, it is like saying "you can assume this function is there and that it takes these parameters". In fact, we could also have written:

```
extern void myfunction(int number);
```

Unlike for variables, for function prototypes the **extern** is optional.

Apart from variable and function declarations (externs and prototypes), a header file is also often used to define macros, typedefs, enums, structs and unions. However, it should not be used for variable or function definitions, because this way the variable resp. function would be defined in every module where the header is included, and this will cause linker errors (unless they are defined as static, but even then it's usually a bad idea).

16. Overview of the standard library

16.1. Introduction to the standard library

This chapter provides an overview of the C standard library, on an alphabetical header by header basis. The explanations are kept short to keep the size of this document within reasonable limits. If you need more help concerning certain functions, your compiler's manual or online help may be of assistance. You may also find what you're looking for in Dinkumware's C reference, see: 17.2. Other interesting C−related online material

This overview is sorted on a strictly alphabetical basis (case and underscores are ignored).

All the library functions may be implemented as macros. If you really need a function version (for example, if the argument has side effects), you should #undef the macro first (see 13. Preprocessor macros/conditionals).

All the identifiers (variable and function names etc.) that are defined in the standard library are reserved, which means that you shouldn't use them in your own programs. Identifiers that start with an underscore are reserved in most contexts as well. [*]

[*] An exhaustive discussion of which identifiers are reserved in which contexts falls outside the scope of an introductory document about C programming. More information can be found in: 17.2. Other interesting C−related online material

In addition, the following identifiers are reserved for possible extensions to the library:

```
Macros:
− Macros that start with E followed by a digit or uppercase letter
− Macros that start with LC_ followed by an uppercase letter
− Macros that start with SIG or SIG_ followed by an uppercase letter
Function names:
− Function names that start with is or to followed by a lowercase letter
− Function names that start with str, mem or wcs followed by a lowercase
  letter
− The function names declared in math.h with the suffix f or l
```

The following header files that have been added to the original ANSI/ISO standard by normative addenda etc. or the new C99 standard are not included here: (your compiler may not support them yet, either)

```
complex.h  complex arithmetic
fenv.h     floating-point environment
inttypes.h integer types
iso646.h   alternative spellings for operators
stdbool.h  standard boolean type
tgmath.h   type-generic mathematics
wchar.h    wide-character utilities
wctype.h   wide-character classification
```

16.2. assert.h (diagnostics)

Macros

```
void assert(int expression);
```

If NDEBUG macro is defined when assert.h is included, does nothing. Otherwise, checks whether expression is true. If it isn't, an error message containing the failed condition, the filename and line number is printed and the abort() function is called (see also 13.2. Using the assert() macro).
assert() is always defined as a macro and may not be #undef'ed.

16.3. ctype.h (character handling)

Be careful that the values you pass to the functions declared in ctype.h are inside the range of an **unsigned char** or EOF. This may bite you if you use an implementation where the type **char** defaults to **signed**.

Functions

```
int isalnum(int c);
```

Returns nonzero if c is an alphanumeric character (letter or digit).

```
int isalpha(int c);
```

Returns nonzero if c is a letter.

```
int iscntrl(int c);
```

Returns nonzero if c is a control character.

```
int isdigit(int c);
```

Returns nonzero if c is a decimal digit.

```
int isgraph(int c);
```

Returns nonzero if c is any printing character except space.

```
int islower(int c);
```

Returns nonzero if c is a lowercase letter.

```
int isprint(int c);
```

Returns nonzero if c is a printing character including space.

```
int ispunct(int c);
```

Returns nonzero if c is any printing character other than a letter, a digit or space.

```
int isspace(int c);
```

Returns nonzero if c is a whitespace character (space, formfeed, newline, carriage return, horizontal or vertical tab).

```
int isupper(int c);
```

Returns nonzero if c is an uppercase letter.

```
int isxdigit(int c);
```

Returns nonzero if c is a hexadecimal digit.

```
int tolower(int c);
```

If c is an uppercase letter, returns the corresponding lowercase letter. Otherwise, returns c unchanged.

```
int toupper(int c);
```

If c is a lowercase letter, returns the corresponding uppercase letter. Otherwise, returns c unchanged.

16.4. errno.h (error handling)

Macros

```
EDOM                          /* domain error */
ERANGE                        /* range error */
```

Integer constants representing certain error conditions. Your implementation may define more error condition macros, starting with an uppercase E and followed by an uppercase letter or digit.

```
errno
```

Integer variable that contains an error number or 0 for no error. Several library functions may set this value to nonzero if an error occurs. It is never automatically reset to 0.

16.5. float.h (floating−point limits)

Macros

```
DBL_DIG
```

Number of decimal digits of precision in a **double**. Minimum 10.

`DBL_EPSILON`

Minimum positive **double** x such that 1.0 + x != 1.0. Maximum 1E−9.

`DBL_MANT_DIG`

Number of base−FLT_RADIX digits in the **double** mantissa.

`DBL_MAX`

Maximum representable finite **double**. Minimum 1E+37.

`DBL_MAX_EXP`

Maximum positive integer for which FLT_RADIX**FLT_MIN_EXP−1 is a representable, finite **double**.

`DBL_MAX_10_EXP`

Maximum positive integer for which 10**FLT_MIN_EXP is a representable, finite **double**. Minimum +37.

`DBL_MIN`

Minimum normalized **double**. Maximum 1E−37.

`DBL_MIN_EXP`

Minimum negative integer for which FLT_RADIX**FLT_MIN_EXP−1 is a normalized **double**.

`DBL_MIN_10_EXP`

Minimum negative integer for which 10**FLT_MIN_EXP is a normalized **double**. Maximum −37.

`FLT_DIG`

Number of decimal digits of precision in a **float**. Minimum 6.

`FLT_EPSILON`

Minimum positive **float** x such that 1.0 + x != 1.0. Maximum 1E−5.

`FLT_MANT_DIG`

Number of base−FLT_RADIX digits in the **float** mantissa.

`FLT_MAX`

Maximum representable finite **float**. Minimum 1E+37.

`FLT_MAX_EXP`

Maximum positive integer for which FLT_RADIX**FLT_MIN_EXP−1 is a representable, finite **float**.

`FLT_MAX_10_EXP`

Maximum positive integer for which 10**FLT_MIN_EXP is a representable, finite **float**. Minimum +37.

`FLT_MIN`

Minimum normalized **float**. Maximum 1E−37.

`FLT_MIN_EXP`

Minimum negative integer for which FLT_RADIX**FLT_MIN_EXP−1 is a normalized **float**.

`FLT_MIN_10_EXP`

Minimum negative integer for which 10**FLT_MIN_EXP is a normalized **float**. Maximum −37.

`FLT_RADIX`

Radix of floating−point exponent representation.

`LDBL_DIG`

Number of decimal digits of precision in a **long double**. Minimum 10.

`LDBL_EPSILON`

Minimum positive **long double** x such that 1.0 + x != 1.0. Maximum 1E−9.

`LDBL_MANT_DIG`

Number of base−FLT_RADIX digits in the **long double** mantissa.

`LDBL_MAX`

Maximum representable finite **long double**. Minimum 1E+37.

`LDBL_MAX_EXP`

Maximum positive integer for which FLT_RADIX**FLT_MIN_EXP−1 is a representable, finite **long double**.

`LDBL_MAX_10_EXP`

Maximum positive integer for which 10**FLT_MIN_EXP is a representable, finite **long double**. Minimum +37.

`LDBL_MIN`

Minimum normalized **long double**. Maximum 1E−37.

`LDBL_MIN_EXP`

Minimum negative integer for which FLT_RADIX**FLT_MIN_EXP−1 is a normalized **long double**.

`LDBL_MIN_10_EXP`

Minimum negative integer for which 10**FLT_MIN_EXP is a normalized **long double**. Maximum −37.

16.6. limits.h (implementation limits)

`CHAR_BIT`

Number of bits in a byte (smallest object that is not a bit−field). Minimum 8.

`CHAR_MAX`

Maximum value a **char** can represent. Same as SCHAR_MAX if **char** defaults to **signed**, same as UCHAR_MAX if **char** defaults to **unsigned**.

`CHAR_MIN`

Minimum value a **char** can represent. Same as SCHAR_MIN if **char** defaults to **signed**, same as UCHAR_MIN if **char** defaults to **unsigned**.

`INT_MAX`

Maximum value an **int** can represent. Minimum +32767.

`INT_MIN`

Minimum value an **int** can represent. Maximum −32767.

`LONG_MAX`

Maximum value a **long int** can represent. Minimum +2147483647.

`LONG_MIN`

Minimum value a **long int** can represent. Maximum −2147483647.

`MB_LEN_MAX`

Maximum number of bytes in a multibyte character.

`SCHAR_MAX`

Maximum value a **signed char** can represent. Minimum +127.

`SCHAR_MIN`

Minimum value a **signed char** can represent. Maximum −127.

`SHRT_MAX`

Maximum value a **short int** can represent. Minimum +32767.

`SHRT_MIN`

Minimum value a **short int** can represent. Maximum −32767.

`UCHAR_MAX`

Maximum value an **unsigned char** can represent. Minimum 255.

`UINT_MAX`

Maximum value an **unsigned int** can represent. Minimum 65535.

`ULONG_MAX`

Maximum value an **unsigned long int** can represent. Minimum 4294967295.

`USHRT_MAX`

Maximum value an **unsigned short int** can represent. Minimum 65535.

16.7. locale.h (localization)

Types

```
struct lconv {
   char *decimal_point;       /* decimal point character ["."] */
   char *thousands_sep;       /* digit group separating character [""] */
   char *grouping;            /* size of digit groups [""] */
   char *int_curr_symbol;     /* international currency symbol [""] */
   char *currency_symbol;     /* local currency symbol [""] */
   char *mon_decimal_point;   /* monetary decimal point [""] */
   char *mon_thousands_sep;   /* monetary digit group separator [""] */
   char *mon_grouping;        /* monetary digit group size [""] */
   char *positive_sign;       /* monetary nonnegative sign [""] */
   char *negative_sign;       /* monetary negative sign [""] */
   char int_frac_digits;      /* int. monetary decimal digits [CHAR_MAX] */
   char frac_digits;          /* monetary decimal digits [CHAR_MAX] */
   char p_cs_precedes;        /* 1 if currency symbol precedes nonnegative
                                 value [CHAR_MAX] */
   char p_sep_by_space;       /* 1 if currency symbol is separated by space
                                 from nonnegative value [CHAR_MAX] */
   char n_cs_precedes;        /* 1 if currency symbol precedes negative
                                 value [CHAR_MAX] */
   char n_sep_by_space;       /* 1 if currency symbol is separated by space
                                 from negative value [CHAR_MAX] */
   char p_sign_posn;          /* indicates positioning of positive sign
                                 for nonnegative values [CHAR_MAX] */
   char n_sign_posn;          /* indicates positioning of negative sign
                                 for negative values [CHAR_MAX] */
  };
```

Structure which contains at least the above specified members related to the formatting of numeric values [*]. The values between the square brackets [ ] are those for the C locale. See the explanation of localeconv() for more information.

[*] The order of the members may be different

Macros

```
LC_ALL                        /* all categories */
```

```
LC_COLLATE                    /* strcoll() and strxfrm() */
LC_CTYPE                      /* character handling */
LC_MONETARY                   /* localeconv() monetary formatting */
LC_NUMERIC                    /* decimal point */
LC_TIME                       /* strftime() */
```

Constant integers used by the setlocale() function. Your implementation may define additional macro definitions, starting with LC_ followed by an uppercase letter.

`NULL`

Explained in 16.12. stddef.h.

Functions

```
char *setlocale(int category, const char *locale);
```

Selects the specified locale for the specified category of the program (one of the above specified LC_ macros). For locale, use "C" for the minimal environment for C translation (this is also the default at program startup) or "" for the implementation's native locale.

```
struct lconv *localeconv(void);
```

Returns a pointer to a lconv structure describing the formatting of numeric quantities in the current locale. The members of this structure can point to an empty string or have values CHAR_MAX, which indicates that the information is not available in the current locale.

The structure may not be modified. Subsequent calls to the localeconv() function may overwrite its contents.

16.8. math.h (mathematics)

Macros

`HUGE_VAL`

Positive **double** constant, which indicates that the result of a mathematical function is too large to fit in a **double**.

Functions

```
double acos(double x);
```

Returns the principal value of the arc cosine of x in radians. Will cause a domain error for arguments outside the range [−1, +1].

```
double asin(double x);
```

Returns the principal value of the arc sine of x in radians. Will cause a domain error for arguments outside the range [−1, +1].

```
double atan(double x);
```

Returns the principal value of the arc tangent of x in radians.

```
double atan2(double y, double x);
```

Returns the principal value of the arc tangent of y/x in radians, using the signs of the arguments to determine the quadrant. May cause a domain error if both arguments are 0.

```
double ceil(double x);
```

Returns the smallest integral value not less than x.

```
double cos(double x);
```

Returns the cosine of x (measured in radians). Accuracy may be lost if x is of large magnitude.

```
double cosh(double x);
```

Returns the hyperbolic cosine of x. Will cause a range error if x is too large in magnitude.

```
double exp(double x);
```

Returns e raised to the power x. Will cause a range error if x is too large in magnitude.

```
double fabs(double x);
```

Returns the absolute value of x.

```
double floor(double x);
```

Returns the largest integral value not greater than x.

```
double fmod(double x, double y);
```

Returns the remainder of x/y. If y is nonzero, the result has the same sign as x. May cause a domain error if y is 0.

```
double frexp(double value, int *exp);
```

Breaks value into a normalized fraction (returned; magnitude in interval [1/2, 1[or zero) and an integral power of 2 (stored in the integer pointed to by exp) so that the returned value multiplied by x raised to the power *exp equals value.

```
double ldexp(double x, int exp);
```

Returns x multiplied by 2 raised to the power exp. May cause a range error.

```
double log(double x);
```

Returns the natural logarithm of x. Will cause a domain error if the argument is negative. May cause a range error if the argument is 0.

```
double log10(double x);
```

Returns the base−10 logarithm of x. Will cause a domain error if the argument is negative. May cause a range error if the argument is 0.

```
double modf(double value, double *iptr);
```

Breaks value into a fractional part (returned) and an integral part (stored in the **double** pointed to by iptr), both with the same sign as value.

```
double pow(double x, double y);
```

Returns x raised to the power y. Will cause a domain error if x is negative and y is not an integer. May cause a domain error if x is 0 and y is negative or 0. May cause a range error.

```
double sin(double x);
```

Returns the sine of x (measured in radians). Accuracy may be lost if x is of large magnitude.

```
double sinh(double x);
```

Returns the hyperbolic sine of x. Will cause a range error if x is too large in magnitude.

```
double sqrt(double x);
```

Returns the nonnegative square root of x. Will cause a domain error if the argument is negative.

```
double tan(double x);
```

Returns the tangent of x (measured in radians). Accuracy may be lost if x is of large magnitude.

```
double tanh(double x);
```

Returns the hyperbolic tangent of x.

16.9. setjmp.h (non−local jumps)

Types

```
jmp_buf
```

A type that is used for holding the information necessary for performing non−local jumps.

Macros

```
int setjmp(jmp_buf env);
```

Sets up for a longjmp() to this position using the specified jmp_buf variable. The return value will be 0 when the setjmp() function is originally called. When a longjmp() is performed using the jmp_buf variable, the code will continue at this position with a return value different from 0.

The setjmp() macro may only be used as an expression on its own, or compared with an integral constant, or the logical ! operator.

Functions

```
void longjmp(jmp_buf env, int val);
```

The longjmp() function performs a non−local jump to the setjmp() macro that was last used in combination with the specified jmp_buf variable. This setjmp() macro will then return with the value specified in val. However, if val is set to 0, setjmp() will return with the value 1.

16.10. signal.h (signal handling)

Types

```
sig_atomic_t
```

An integral type that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.

Macros

```
SIGABRT                     /* abnormal program termination */
SIGFPE                      /* arithmetic error */
SIGILL                      /* illegal instruction */
SIGINT                      /* interactive attention signal */
SIGSEGV                     /* invalid access to storage */
SIGTERM                     /* termination request received */
```

These are integer constants representing the signal numbers corresponding to the specified error conditions. Your implementation need not generate any of these signals (unless explicitly raised), and additional signals starting with SIG followed by an uppercase letter may be defined.

```
SIG_DFL                     /* default handling */
SIG_IGN                     /* ignore signal */
```

These macros can be used as the second parameter of the signal() function. Your implementation may define more of these macros starting with SIG_ followed by an uppercase letter.

```
SIG_ERR
```

This macro corresponds to the return value of signal() in case an error occurs.

Functions

```
void (*signal(int sig, void (*func)(int)))(int);
```

Determines how the specified signal number sig (one of the above listed SIG macros) is handled. The second parameter, func, may be either SIG_DFL, which means default handling for that signal, SIG_IGN, which means that signal will be ignored, or a pointer to a function that is called when the signal occurs, a so−called signal handler. Before this handler is called, the signal will be reset to the default handler.

The signal handler must be a function returning **void** and taking one parameter of type **int**, the signal number. It may terminate by means of a return statement (unless the signal occurring was SIGFPE or another form of exception) or a call to abort(), exit() or longjmp(). Unless the signal was invoked by the raise() or abort() function, the signal handler may not use any other standard library functions or access any **static** data other than assigning a value to a **static** variable of type **volatile** sig_atomic_t.

If the request is successful, the signal() function will return the address of the previous signal handler for this signal. Otherwise it will return SIG_ERR and set errno.

```
int raise(int sig);
```

Causes the specified signal number to occur. Returns 0 if successful, nonzero otherwise.

16.11. stdarg.h (variable arguments)

See also 14. Variable argument functions.

Types

```
va_list
```

This type is used by the va_start(), va_arg() and va_end() macros to hold information about the variable argument list.

Macros

```
void va_start(va_list ap,  parmN);
```

The va_start() macro initializes the va_list variable for use by the va_arg() and va_end() macros; parmN is the name of the parameter right before the ellipsis (...).

```
type va_arg(va_list ap,  type);
```

Returns the next variable argument of the specified type. This type must correspond to the type of the parameter actually passed after the default promotions of **short** to **int**, **float** to **double**, etc. The va_list variable must have been initialized by the va_start macro before.

```
void va_end(va_list ap);
```

The va_end() macro must be applied to the va_list variable before the function returns.

16.12. stddef.h (standard type/macro definitions)

Types

```
ptrdiff_t
```

Signed integral type which is the result of substracting two pointers.

```
size_t
```

Unsigned integral type used for holding the size of an object (also the type of the result of the **sizeof** operator).

```
wchar_t
```

Wide character type; in any extended character set the value 0 is reserved for the null character.

Macros

```
NULL
```

Pointer constant used for a pointer that points nowhere.

```
offsetof(type,  member−designator)
```

Returns the offset (of type size_t) in bytes from the beginning of the structure type to its member variable specified by member−designator.

16.13. stdio.h (input/output)

Types

```
FILE
```

A type used for interfacing with streams (files). See also 11. Using files.

```
fpos_t
```

A type capable of holding every possible file position. Used by fgetpos() and fsetpos().

```
size_t
```

Explained in 16.12. stddef.h.

Macros

```
BUFSIZ
```

Integer constant; the size of the buffer used by the setbuf() function. Minimum 256.

```
EOF
```

Negative integer constant which several file I/O functions (e.g. fgetc()) return to indicate end−of−file condition.

```
FOPEN_MAX
```

Integer constant; the minimum number of files that can be open at the same time. This is at least 8 (of which 3 are reserved for the standard I/O streams; your compiler may even reserve more for other non−standard pre−opened I/O streams).

```
FILENAME_MAX
```

Integer constant; the maximum length for a filename (includes path etc.) that is guaranteed to be usable. This doesn't mean that longer filenames may not work as well.

```
_IOFBF                      /* full buffering */
_IOLBF                      /* line buffering */
_IONBF                      /* no buffering */
```

Integer constants used by the setvbuf() function.

```
L_tmpnam
```

Integer constant; the maximum length of the temporary filename generated by the tmpnam() function.

```
NULL
```

Explained in 16.12. stddef.h.

```
SEEK_CUR                     /* seek from current position */
SEEK_END                     /* seek from end of file */
SEEK_SET                     /* seek from beginning of file */
```

Integer constants used by the fseek() function.

```
stderr
stdin
stdout
```

Pointers to FILE representing the standard input, output and error streams. See also 11.4. Using the standard I/O streams.

```
TMP_MAX
```

Integer constant; minimum number of unique temporary filenames the tmpnam() function can generate. At least 25.

Functions

```
void clearerr(FILE *stream);
```

Clears the end−of−file and error indicators for the specified stream.

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

Closes the file associated with the stream; all buffered data will be written. The stream may not be used afterwards except for assigning a new value to it. Returns 0 if successful, EOF otherwise.

```
int feof(FILE *stream);
```

Returns nonzero if an attempt has been made to read past the end of the file of the specified stream [*], returns 0 otherwise.

[*] This is different from the way EOF works in some other languages, where it can be used to check if end−of−file has been reached.

```
int ferror(FILE *stream);
```

Returns nonzero if the error indicator is set for the specified stream, returns 0 otherwise.

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

Writes all buffered data from the stream to the file. The stream must be an output stream or an update stream which was last written to. If stream is null all buffered data for all such streams will be written to the file. Returns 0 if successful, EOF otherwise.

```
int fgetc(FILE *stream);
```

Returns the next character from the specified stream as an **unsigned char** converted to **int**, or EOF on error.

```
int fgetpos(FILE *stream, fpos_t *pos);
```

Stores the current file position for the specified stream into the fpos_t variable pointed to by pos. This variable can later be used by the fsetpos() function to reposition the stream to its current position. Returns 0 if successful or nonzero otherwise.

```
char *fgets(char *s, int n, FILE *stream);
```

Reads upto n−1 characters or a newline (whichever comes first) from the specified stream and stores it in the array pointed to by s. A 0 terminator character is appended. Any newline character is not removed. Returns s if successful, or NULL otherwise.

See also 11.2. Using disk files.

```
FILE *fopen(const char *filename, const char *mode);
```

The fopen() function is discussed in 11.2. Using disk files.

```
int fprintf(FILE *stream, const char *format, ...);
```

The fprintf() function is analogous to the printf() function, except that it writes its output to the specified stream.

```
int fputc(int c, FILE *stream);
```

Writes the character specified by c (converted to **unsigned char**) to the specified stream at the current file position, and adjusts this position appropriately. In append mode, the character is always appended to the end of the file. Returns the character written if successful or EOF otherwise.

```
int fputs(const char *s, FILE *stream);
```

Writes the string pointed to by s to the specified stream (not including the terminating 0 character). Returns a nonnegative value if successful, or EOF otherwise.

```
size_t fread(void *ptr, size_t size, size_t nmemb,
             FILE *stream);
```

Reads upto nmemb elements of the specified size [*] from the specified stream into the array pointed to by ptr. The file position will be adjusted appropriately. Returns the number of members (not bytes!) successfully read, which may be less than nmemb if an error such as end−of−file occurred.

[*] The result of the multiplication size * nmemb should fit in a size_t, so you can't rely on this feature to get around compiler limits (such as the 64K limit on 16−bit compilers, for example).

```
FILE *freopen(const char *filename, const char *mode,
              FILE *stream);
```

Opens the file specified by filename in mode mode (see fopen() for details) and associates the specified stream with this file. Any file previously associated with stream is closed. Returns stream when successful or NULL otherwise.

```
int fscanf(FILE *stream, const char *format, ...);
```

The fscanf() function is analogous to the scanf() function, except that it reads its input from the specified stream.

```
int fseek(FILE *stream, long int offset, int whence);
```

Sets the file position for the specified stream to the specified offset, which is counted in bytes from the beginning of the file if whence equals SEEK_SET, the current file position if whence equals SEEK_CUR or the end of the file if whence equals SEEK_END. For text files, if whence is SEEK_SET offset must be either 0 or a value returned by an earlier call to ftell() on this stream; if whence is SEEK_CUR or SEEK_END offset must be 0. Returns nonzero if the request cannot be satisfied. If the return value is 0, a call to ftell() may be used to make sure the file position has been successfully updated.

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

Sets the file position for the specified stream to the position stored in the pos variable by an earlier call to fgetpos() on the same stream. Returns 0 if successful, nonzero otherwise.

```
long int ftell(FILE *stream);
```

Returns the current file position for the specified stream. For binary files this is the number of characters from the beginning of the file. For text files, this number is only useful as a parameter to the fseek() function. Returns −1L if unsuccessful.

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
```

Writes upto nmemb elements of the specified size [*] to the specified stream from the array pointed to by ptr. The file position will be adjusted appropriately. Returns the number of members (not bytes!) successfully written, which may be less than nmemb if an error such as end−of−file occurred.

[*] The result of the multiplication size * nmemb should fit in a size_t, so you can't rely on this feature to get around compiler limits (such as the 64K limit on 16−bit compilers, for example).

```
int getc(FILE *stream);
```

The getc() function is equivalent to fgetc() except that it may be a macro that can evaluate its argument more than once.

```
int getchar(void);
```

The getchar() function is equivalent to getc(stdin).

```
char *gets(char *s);
```

Reads characters from stdin into the array pointed to by s, until a newline character is found or an error occurs. Any newline character is removed, and a 0 character is appended. Returns s if successful, NULL otherwise.

WARNING: do not use this function; it does not provide any method of limiting the input to fit inside the array bounds.

```
void perror(const char *s);
```

Writes the string pointed to by s (if s is not NULL) to stdout, followed by an appropriate error message depending on the value of errno (the same message as returned by strerror()).

```
int printf(const char *format, ...);
```

The printf() function is discussed in 3.3. printf().

```
int putc(int c, FILE *stream);
```

The getc() function is equivalent to fputc() except that it may be a macro that can evaluate its argument more than once.

```
int putchar(int c);
```

The putchar() function is equivalent to fputc(c, stdout).

```
int puts(const char *s);
```

Writes the string pointed to by s to stdout (not including the terminating 0 character) and appends a newline character. Returns a nonnegative value if successful, or EOF otherwise.

```
int remove(const char *filename);
```

Removes the specified file. Whether an open file can be removed depends on your system. Returns 0 when successful, nonzero otherwise.

```
int rename(const char *old, const char *new);
```

Changes the name of the specified file from the string pointed to by old to that pointed to by new. Whether a file by the latter name is allowed to exist already depends on your system. Returns 0 when successful, nonzero otherwise.

```
void rewind(FILE *stream);
```

Sets the file position for the specified stream to the beginning of the file and clears the error indicator for the stream.

```
int scanf(const char *format, ...);
```

The scanf() function is discussed in 11.4. Using the standard I/O streams.

```
void setbuf(FILE *stream, char *buf);
```

If buf is not NULL, enables full buffering for the stream, using a buffer of size BUFSIZ pointed to by buf. If buf is NULL, disables all buffering for the stream. Should be called after the file has been opened but before any other operations are performed on the stream.

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

The mode parameter determines the buffering mode for the stream: _IOFBF for full buffering, _IOLBF for line buffering and _IONBF for no buffering. The size parameter is the size of the buffer used. If buf is not

NULL, the array it points to will be used as buffer. Otherwise, a buffer will be dynamically allocated. Returns 0 on success or nonzero otherwise. Should be called after the file has been opened but before any other operations are performed on the stream.

```
int sprintf(char *s, const char *format, ...);
```

The sprintf() function is analogous to the printf() function, except that it writes its output into the specified string.

```
int sscanf(const char *s, const char *format, ...);
```

The sscanf() function is analogous to the scanf() function, except that it reads its input from the specified string.

```
FILE *tmpfile(void);
```

Creates a temporary file and opens it in "w+b" mode. The file will automatically be removed when closed or at program termination. Returns a valid FILE pointer (stream) when successful, or NULL otherwise.

```
char *tmpnam(char *s);
```

Generates a valid, non−existing temporary file name. Every call will generate a different filename, up to TMP_MAX times. If the parameter s is the NULL pointer, the filename will be stored in an internal **static** object and a pointer to it returned; this also means that subsequent calls to tmpnam() may overwrite its contents. If the parameter is not NULL, it must point to an array of at least L_tmpnam characters; in that case the return value will be the same as the parameter.

```
int ungetc(int c, FILE *stream);
```

Pushes the character specified by c (converted to **unsigned char**) back onto the specified input stream. The pushed back characters will be returned upon subsequent reads on this stream, in the reverse order they were pushed back in. At least 1 character can be pushed back. A successfull call to fseek(), fsetpos() or rewind() on this stream will discard the pushed back characters. Returns the pushed back character if successful, or EOF otherwise.

```
int vfprintf(FILE *stream, const char *format, va_list arg);
```

The vfprintf() function is analogous to the vprintf() function, except that it writes its output to the specified stream.

```
int vprintf(const char *format, va_list arg);
```

The vprintf() function is discussed in 14.3. Example: a printf() encapsulation.

```
int vsprintf(char *s, const char *format, va_list arg);
```

The vsprintf() function is analogous to the vprintf() function, except that it writes its output into the specified string.

16.14. stdlib.h (general utilities)

Types

```
typedef struct {
   int quot;                 /* quotient */
   int rem;                  /* remainder */
  } div_t;
```

Structure returned by the div() function. [*]

```
typedef struct {
   long quot;                /* quotient */
   long rem;                 /* remainder */
  } ldiv_t;
```

Structure returned by the ldiv() function. [*]

[*] The order of the members may be different

```
size_t
```

Explained in 16.12. stddef.h.

```
wchar_t
```

Explained in 16.12. stddef.h.

Macros

```
EXIT_FAILURE                 /* unsuccessful termination status */
EXIT_SUCCESS                 /* successful termination status */
```

Integer constants for use as argument to the exit() function or as the return value of main().

```
MB_CUR_MAX
```

Integer constant representing the maximum number of bytes in a multibyte character in the current locale. Never greater than the value of MB_LEN_MAX (see 16.6. limits.h).

```
NULL
```

Explained in 16.12. stddef.h.

```
RAND_MAX
```

Integer constant representing the maximum value returned by rand(). Minimum 32767.

Functions

```
void abort(void);
```

Causes an abnormal program termination. Open files may or may not be closed and temporary files may or may not be removed.

```
int abs(int j);
```

Returns the absolute value of the integer j.

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

Registers the function pointed to by func to be called at normal program termination, in the reverse order of their registration. At least 32 such functions can be registered. Returns 0 if successful, nonzero otherwise.

```
double atof(const char *nptr);
```

Returns the decimal number in the specified string converted to **double** representation. Need not set errno on error. See strtod() for a function with better error handling.

```
int atoi(const char *nptr);
```

Returns the decimal number in the specified string converted to integer representation. Need not set errno on error. See strtol() for a function with better error handling.

```
long int atol(const char *nptr);
```

Returns the decimal number in the specified string converted to **long int** representation. Need not set errno on error. See strtol() for a function with better error handling.

```
void *bsearch(const void *key, const void *base,
         size_t nmemb, size_t size,
         int (*compar)(const void *, const void *));
```

Searches an array of nmemb elements of the specified size, pointed to by base and sorted in ascending order by the relevant criterion, for an element that matches the specified key.
The function pointed to by compar is called with as parameters pointers to the key resp. an array element. It must return a strictly negative value if the key comes before the array element, a strictly positive value if the key comes after the array element, or zero if the key matches the array element.
Returns a pointer to a matching array element, or NULL if none is found.

```
void *calloc(size_t nmemb, size_t size);
```

Allocates memory for an array of nmemb elements of the specified size and initializes it to all−bits−0. Returns a pointer to the start of the allocated memory if successful, or NULL otherwise. The result of requesting 0 bytes may be either a pointer to 0 bytes of memory or NULL.

```
div_t div(int numer, int denom);
```

Divides numer by denom. Returns the quotient and remainder in the quot resp. rem member variables of the returned div_t structure (see above).

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

Causes normal program termination. The functions registered by atexit() are called in the reverse order they were registered in, open files are closed and temporary files are removed. The macros EXIT_SUCCESS and EXIT_FAILURE can be used to indicate successful resp. unsuccessful termination.

```
void free(void *ptr);
```

Deallocates the memory pointed to by ptr, which should be a pointer returned by an earlier call to malloc(), calloc() or realloc(). The pointer may not be used afterwards except for assigning a new value to it.

```
char *getenv(const char *name);
```

Returns a pointer to a string representing the value of the environment variable specified by name, or NULL if the specified environment variable can't be found. The returned string is read−only and may be overwritten by subsequent calls to getenv().

```
long int labs(long int j);
```

Returns the absolute value of the **long int** j.

```
ldiv_t ldiv(long int numer, long int denom);
```

Divides numer by denom. Returns the quotient and remainder in the quot resp. rem member variables of the returned ldiv_t structure (see above).

```
void *malloc(size_t size);
```

Allocates size bytes of memory and returns a pointer to the start of the allocated memory if successful, or NULL otherwise. The allocated memory is not initialized and has to be written to before it may be read. The result of requesting 0 bytes may be either a pointer to 0 bytes of memory or NULL.

```
int mblen(const char *s, size_t n);
```

If s is NULL, returns nonzero if multibyte characters have state− dependent encodings, 0 otherwise. If s is not NULL, returns the number of bytes that comprise the multibyte character pointed to by s of max. n bytes, or −1 if it is not a valid multibyte character, or 0 if it is the null character.

```
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

Converts a 0−terminated sequence of multibyte characters into a sequence of upto n corresponding wide character codes. Returns the number of written wide character codes or (size_t)−1 if an invalid multibyte character is encountered.

```
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

The mbtowc() function is analogous to the mblen() function, except that, unless pwc is NULL, it also stores the wide−character code that corresponds to the multibyte character in the wchar_t pointed to by pwc (provided the multibyte character is valid, of course).

```
int rand(void);
```

Returns a pseudo−random integer in the range 0..RAND_MAX.

```
void *realloc(void *ptr, size_t size);
```

Changes the size of the allocated memory for the specified pointer to the specified size. The contents of the memory will be left intact. Returns a pointer to the newly allocated memory, which may be different from the original pointer. If the returned pointer is NULL, the original memory will still be allocated.

```
void srand(unsigned int seed);
```

Initializes a new sequence of pseudo−random numbers with the specified seed. At program startup the seed is 1, so if no call to srand() is made the rand() function will return the same pseudo−random number sequence every time the program is run.

```
double strtod(const char *nptr, char **endptr);
```

Returns the decimal number in the string pointed to by nptr, converted to **double** representation. Leading whitespace is skipped. Unless endptr is NULL, the pointer pointed to by endptr is set to the first character that can't be converted anymore. If no conversion was possible, 0 is returned. If the value is too large, +/− HUGE_VAL is returned; if the value is too small, 0 is returned. In both these cases errno will be set to ERANGE.

```
long int strtol(const char *nptr, char **endptr, int base);
```

Returns the base−n number (base must be in the range 2..36) in the string pointed to by nptr, converted to **long int** representation. Leading whitespace is skipped. Unless endptr is NULL, the pointer pointed to by endptr is set to the first character that can't be converted anymore. If no conversion was possible, 0 is returned. If the value is too large, LONG_MAX or LONG_MIN is returned and errno will be set to ERANGE.

```
unsigned long int strtoul(const char *nptr, char **endptr,
                          int base);
```

The strtoul() function is analogous to strtol(), except that if the value is too large ULONG_MAX will be returned.

```
int system(const char *string);
```

Lets the command processor execute the specified string. If string is NULL, returns nonzero if a command processor exists, 0 otherwise. If string is non−NULL, the returned value depends on your system.

```
void qsort(void *base, size_t nmemb, size_t size,
         int (*compar)(const void *, const void *));
```

Sorts in ascending order an array of nmemb elements of the specified size, pointed to by base.
The function pointed to by compar is called with as parameters pointers to two array elements. It must return a strictly negative value if the first element comes before the second, a strictly positive value if the first element comes after the second, or zero if the two elements match (in which case their sorting order is unspecified).

```
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

Converts a 0−terminated sequence of wide character codes into a sequence of upto n bytes of multibyte characters. Returns the number of written bytes or (size_t)−1 if a code not corresponding to a valid multibyte character is encountered.

```
int wctomb(char *s, wchar_t wchar);
```

If s is NULL, returns nonzero if multibyte characters have state− dependent encodings, 0 otherwise. If s is not NULL, returns the number of bytes that comprise the multibyte character corresponding to the specified wide character code (less than or equal to MB_CUR_MAX) and stores the multibyte character in the array pointed to by s, or returns −1 if the wide character code does not correspond to a valid multibyte character.

16.15. string.h (string handling)

Types

`size_t`

Explained in 16.12. stddef.h.

Macros

`NULL`

Explained in 16.12. stddef.h.

Functions

```
void *memchr(const void *s, int c, size_t n);
```

Returns a pointer to the first occurence of the character c (converted to **unsigned char**) in the first n bytes of memory pointed to by s, or NULL if none is found.

```
int memcmp(const void *s1, const void *s2, size_t n);
```

Compares the first n characters of the memory pointed to by s1 with the first n characters of the memory pointed to by s2. Returns a strictly negative value if s1 comes before s2, a strictly positive value if s1 comes after s2, or 0 if they match.

```
void *memcpy(void *s1, const void *s2, size_t n);
```

Copies n characters from the memory pointed to by s2 to the memory pointed to by s1. The areas of memory should not overlap. Returns the value of s1.

```
void *memmove(void *s1, const void *s2, size_t n);
```

Analogous to memcpy(), but supports overlapping memory areas.

```
void *memset(void *s, int c, size_t n);
```

Initializes the first n bytes of the memory pointed to by s to the value c (converted to **unsigned char**).

```
char *strcat(char *s1, const char *s2);
```

Appends a copy of the string pointed to by s2 (including the 0 terminator byte) to the end of the string pointed to by s1. The areas of memory should not overlap. Returns the value of s1.

```
char *strchr(const char *s, int c);
```

Returns a pointer to the first occurence of the character c (converted to **char**) in the string pointed to by s, or NULL if none is found.

```
int strcmp(const char *s1, const char *s2);
```

Compares the string pointed to by s1 with the string pointed to by s2. Returns a strictly negative value if s1 comes before s2, a strictly positive value if s1 comes after s2, or 0 if they match.

```
int strcoll(const char *s1, const char *s2);
```

Analogous to strcmp(), but interprets the strings according to the LC_COLLATE category of the current locale.

```
char *strcpy(char *s1, const char *s2);
```

Copies the string pointed to by s2 (upto and including the 0 terminator byte) into the array pointed to by s1. The areas of memory should not overlap. Returns the value of s1.

```
size_t strcspn(const char *s1, const char *s2);
```

Returns the maximum length of the substring in the beginning of the string pointed to by s1 which does not contain any characters from the string pointed to by s2.

```
char *strerror(int errnum);
```

Returns a pointer to an error message string corresponding to the error number specified in errnum. The string is read−only, but may be modified by subsequent calls to the strerror() function.

```
size_t strlen(const char *s);
```

Returns the length of the string pointed to by s (excluding the 0 terminating character).

```
char *strncat(char *s1, const char *s2, size_t n);
```

Similar to strcat(), but appends no more than n characters. The areas of memory should not overlap. Returns the value of s1.
This function may produce an unterminated "string" if the limit of n characters is reached.

```
int strncmp(const char *s1, const char *s2, size_t n);
```

Similar to strcmp(), but compares no more than n characters.

```
char *strncpy(char *s1, const char *s2, size_t n);
```

Similar to strcpy(), but copies no more than n characters, and if the string is shorter than n characters s2 is padded with 0 characters. The areas of memory should not overlap. Returns the value of s1.
This function may produce an unterminated "string" if the limit of n characters is reached.

```
char *strpbrk(const char *s1, const char *s2);
```

Returns a pointer to the first occurence in the string pointed to by s1 of any character from the string pointed to by s2, or NULL if none was found.

```
char *strrchr(const char *s, int c);
```

Returns a pointer to the last occurence of the character c (converted to **char**) in the string pointed to by s, or NULL if none is found.

```
size_t strspn(const char *s1, const char *s2);
```

Returns the maximum length of the substring in the beginning of the string pointed to by s1 which contains only characters from the string pointed to by s2.

```
char *strstr(const char *s1, const char *s2);
```

Returns a pointer to the first occurence of the substring pointed to by s2 in the string pointed to by s1, or NULL if none is found.

```
char *strtok(char *s1, const char *s2);
```

The first call to strtok() should pass the string to be processed as parameter s1 (note that this string will be modified by the strtok() function) and a string consisting of delimiting characters as parameter s2. The function will return a pointer to the first substring (token) that consists entirely of characters not in s2, with the first occurence of a delimiting character from s2 being overwritten with the 0 character.
Subsequent calls to strtok() should pass NULL for s1, which will cause strtok() to automatically continue with the next token. The string of delimiting characters s2 may be different each time.
The strtok() function will return NULL as soon as no more tokens can be found.

```
size_t strxfrm(char *s1, const char *s2, size_t n);
```

Transforms the string pointed to by s2 into the array pointed to by s1, with no more than n characters being written. Two transformed strings will compare identical using the strcmp() function as the original strings would using the strcoll() function. Returns the length of the transformed string, not including the terminating 0 character.

16.16. time.h (date and time)

Types

```
clock_t
```

Arithmetic type returned by the clock() function.

```
size_t
```

Explained in 16.12. stddef.h.

```
time_t
```

Arithmetic type used to hold a time.

```
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 1 [0..365] */
   int tm_isdst;            /*  daylight saving time flag */
  };
```

Structure used to hold a broken−down calendar time. The value of tm_isdst is strictly positive if daylight saving time is in effect, zero if it is not in effect, and negative if it is unknown.

Macros

```
CLOCKS_PER_SEC
```

Number of units per second of the value returned by clock(); replaces the earlier CLK_TCK.

```
NULL
```

Explained in 16.12. stddef.h.

Functions

```
char *asctime(const struct tm *timeptr);
```

Converts the broken−down calendar time in the structure pointed to by timeptr to a string of the form:

```
   Sun Sep 16 01:03:52 1973\n\0
```

Where the weekday name is one of:

```
   Sun, Mon, Tue, Wed, Thu, Fri, Sat
```

And the month name is one of:

```
   Jan, Feb, Mar, Apr, May, Jun,
   Jul, Aug, Sep, Oct, Nov, Dec
```

A pointer to the string is returned. This string may be modified by subsequent calls to asctime() or ctime().

```
clock_t clock(void);
```

Returns an approximation to the processor time used since a certain, unspecified starting point. By dividing the difference between two calls to clock() by CLOCKS_PER_SEC, the processor time used can be calculated in seconds. If the processor time is not available (clock_t)−1 is returned.

```
char *ctime(const time_t *timer);
```

A call to the ctime() function is equivalent to asctime(localtime(timer)).

```
double difftime(time_t time1, time_t time0);
```

Returns the difference in seconds between two calendar times (time1 − time0) as a **double**.

```
struct tm *gmtime(const time_t *timer);
```

Returns a pointer to the broken−down UTC time corresponding to the specified time_t value, or NULL if UTC is not available. The broken−down time structure may be modified by subsequent calls to gmtime(), localtime() or ctime().

```
struct tm *localtime(const time_t *timer);
```

Returns a pointer to the broken−down local time corresponding to the specified time_t value. The broken−down time structure may be modified by subsequent calls to localtime(), gmtime() or ctime().

```
time_t mktime(struct tm *timeptr);
```

Converts a broken−down local calendar time (see above for the structure definition) in the structure pointed to by timeptr to a time encoded as a value of type time_t.
Before the conversion, the tm_wday and tm_yday member variables need not be set and the other member variables need not be restricted to the above specified ranges. After the conversion, the tm_wday and tm_yday member variables will be filled in and the other member variables will be adjusted to the specified ranges.
Returns the encoded time value, or (time_t)−1 if the calendar time can't be represented.

```
size_t strftime(char *s, size_t maxsize,
                const char *format, const struct tm *timeptr);
```

The strftime() function works in similar way as the sprintf() function, except that it is used exclusively for formatting a broken−down time structure, pointed to by timeptr.
No more than maxsize characters are written in the array pointed to by s. The number of characters written (including the 0 character) is returned, unless the output doesn't fit in maxsize characters, in that case 0 is returned.
Again, the % character indicates a format specifier, so to print a % sign, you must use %%. The following format specifiers are supported:

```
%a  locale's abbreviated weekday name
%A  locale's full weekday name
%b  locale's abbreviated month name
%B  locale's full month name
%c  locale's appropriate date and time representation
%d  day of the month as a decimal number [01..31]
%H  hour (24−hour clock) as a decimal number [00..23]
%I  hour (12−hour clock) as a decimal number [01..12]
%j  day of the year as a decimal number [001..366]
%m  month as a decimal number [01..12]
%M  minute as a decimal number [00..59]
%p  locale's equivalent of AM/PM
%S  second as a decimal number [00..60]
%U  week number of the year (the first Sunday as the first day
    of week 1) as a decimal number [00..53]
%w  weekday (where Sunday is 0) as a decimal number [0..6]
%W  week number of the year (the first Monday as the first day
    of week 1) as a decimal number [00..53]
%x  locale's appropriate date representation
%X  locale's appropriate time representation
%y  last 2 digits of the year as a decimal number [00..99]
%Y  whole year as a decimal number (e.g. 1998)
%Z  timezone name or abbreviation, or no characters if the
    timezone can't be determined
```

```
time_t time(time_t *timer);
```

Returns the current calendar time as an encoded time_t value. If timer is not NULL, the value will be stored in the time_t pointed to by timer as well.

17. Epilogue

17.1. Credits

Thanks to the following people for providing useful suggestions, corrections, additions: (in alphabetical order by surname)

```
Nick Austin
Aaron Crane
Dann Corbit
Lawrence Kirby
Bernd Luevelsmeyer
Tom Nolette
Sunil Rao
Robin Sheppard
Jan Schuhr
David Tait
George White
```

17.2. Other interesting C−related online material

The comp.lang.c answers to frequently asked questions:

http://www.eskimo.com/~scs/C−faq/top.html

Bob Stout's Snippets collection:

http://www.snippets.org/

Dinkumware's C reference:

http://www.dinkumware.com/refxc.html

Overview of reserved identifiers:

http://www.mindspring.com/~brahms/c−predef.htm

Free C/C++ interpreter:

http://www.softintegration.com/

Links to other C resources:

http://www.lysator.liu.se/c/