A brief C tutorial

Marc van der Sluys
`http://han.vandersluys.nl`

April 23, 2019

Contents

Code listings

0 Preface

This document is intended as either a quick introduction into basic C programming, or as a refreshment, before taking the **Operating systems** (OPS) course, which is taught in the second year of the curriculum *Embedded systems engineering* (ESE) at the HAN University of Applied Sciences in Arnhem,[1] the Netherlands. It is by no means intended as an exhaustive document that teaches the intricacies of C.

This document borrows heavily from *C Programming Language Cheat Sheet* [1] and uses the gcc C compiler [2] since it is freely available for everybody.

This document was typeset in LaTeX by the author, which allows the automatic inclusion of the C source files when generating the PDF file. Hence, the Code listings in the document are guaranteed to have been compiled, run and checked by the author. The most recent version of this document and an electronic version of the code examples in this document can be found online at `http://han.vandersluys.nl/?title=Publications`. They can be obtained, extracted and compiled by *e.g.*

```
$ wget han.vandersluys.nl/OPS/tarballs/C-tutorial.tar.gz
$ tar xfz C-tutorial.tar.gz
$ make
```

There are probably many aspects of the current document that can be improved. The reader is invited to communicate any suggestions to that extent to the author through `http://han.vandersluys.nl`.

Manuscript rev.22, git hash 94553a8 (2019-04-17 14:04), compiled on Tue 23 Apr 2019, 13:57 CEST.

1 C basics

1.1 A hello-world program

Code listing 1.1: hello.c: A hello-world program in C.

```c
#include <stdio.h>
int main() {
  printf("Hello, world!\n");
}
```

A C source file usually has the extension `.c`. Copy or type the four code lines above into a file called **`hello.c`** and save it. A C program has to be **compiled**, *i.e.* translated from human-readable text to machine-readable code, before it can be executed. Hence, we need a **compiler**. A good and freely available compiler that works on most systems is the **gcc compiler** [2].

On a typical command line (shell), you would type (the $ is the prompt, so don't type it):

```
$ gcc -Wall hello.c -o hello
```

The program `hello` can then be run from the current directory as

```
$ ./program
```

and should print to screen:

```
Hello, world!
```

No rocket surgery yet, but an important first step, so ensure that everything went well. An quick analysis of the code lines we used:

1. invoke extra C-library functions, in this case to print to screen (`printf()`— I/O);
2. start the `main()` function;
3. print the text to screen, followed by a newline (`\n`);

[1]HAN: Hogeschool van Arnhem en Nijmegen (university of applied sciences of Arnhem and Nijmegen).

4. end the `main()` function.

Every C program must have exactly one `main()` function. The code starts executing the first line of `main()` when the program is run, and then continues to the second. All code between these curly braces ({. . .}) belongs to the `main()` function — it is said to be in the **scope** of the `main()` function. We will see similar use of the curly braces to delimit a block of code later on. Most programs have more functions, which can be called from `main()` or each other (see Section 6).

Our simple program only has one true statement (the `printf()` line). Every C statement must end with a semicolon (;), as we see here. Also note that C is *case sensitive*; `printf()` would not work. Keywords like `printf()`, but also the names of variables, constants, etc., can only contain letters, numbers and underscores (_), and cannot start with a number.

1.2 Variables, data types, type casting and placeholders

1.2.1 Variables

Consider the program below. It uses the **integer variable** `sum`. An integer is a **data type** that can store whole numbers, like −1, 3 and 1023. The variable is *declared* an integer using the keyword `int` on line 3. Then the variable is assigned the value 12, and in the next line 5 is added. Line 6 prints the result. Compiling and running this program should yield:

```
The sum is 17.
```

Code listing 1.2: printSum.c: compute and print a sum.

```
1 #include <stdio.h>  // needed for printf()
2 int main() {
3     int sum;     // declare the variable sum as an integer
4     sum=12;      // initialise the variable
5     sum=sum+5;   // do operations with the variable
6     printf("The sum is %i.\n", sum);  // print the value of sum
7 }
```

Note that we now use `printf()` to print an integer as well as text. Since the number has to be converted to text, we need to specify how this should be done. In this case, we simply use `%i`, which just means 'integer' without further specification. However, for *e.g.* `float` and `double` variable types (numbers with a decimal point), we may want to specify the number of decimals. After the double quotes and comma, we specify the variable whose value we want to print.

1.2.2 Data types

The five basic built-in data types in C are:

`int`	integer
`char`	character
`long`	long integer
`float`	float number
`double`	long float

The general form for declaring a variable is `type name;`, as we saw in Code listing 1.2.

1.2.3 Type casting

In some cases, the type of a variable should be changed. In Code listing 1.2 we saw that the sum of two integers is itself an `int`. However, when we *divide* two integers, the result may be unexpected:

Code listing 1.3: typeCast.c: use type casting to convert an `int` to a `double`.

```c
#include <stdio.h>

int main() {
  int sum=17, count=5;
  double mean;

  mean = sum/count;
  printf("Value of mean: %f\n", mean);

  mean = (double)sum/count;
  printf("Value of mean: %f\n", mean);
}
```

The output is

```
Value of mean:  3.000000
Value of mean:  3.400000
```

Clearly, the first answer is wrong. The problem is that the result of the division is an `int`, which is then stored in a `double`. In the second try, we **type cast** or simply **cast** the `int sum` to a `double` by placing `(double)` *before* the variable we want to cast. Hence, the general format of a type cast is

```
(type)var
```

Casting between strings and numbers is more tricky, since only one of them is numeric. To convert a string to an `int` or `double` use a function like `strtoi()` `strtol()` or `strtod()`. To print a (formatted) number into a string, you can use `sprintf()`.

1.2.4 Placeholders for formatted I/O

The code `%i` used in the program is in fact the **placeholder** for an integer value. Placeholders to print data (*e.g.* using `printf()`) are a.o.:

`%d` or `%i`	signed decimal integer;
`%f` or `%lf`	decimal (long/double) floating point;
`%e` or `%E`	scientific notation (using `e` or `E`);
`%c`	character;
`%s`	string of characters;
`%p`	pointer;
`%%`	print a %.

Apart from indicating the data type, we can *e.g.* specify the (minimum) length that should be reserved for a value (*e.g.* to line up values in columns), whether a + is added to positive numbers, and how many decimals should be printed:

Code listing 1.4: printf.c: print formatted values.

```c
#include <stdio.h>
int main() {
    int ivar=5;
    double dvar=3.1415;

    printf("ivar has value %i\n", ivar);
    printf("ivar has value %4i\n", ivar);
    printf("ivar has value %04i\n", ivar);
    printf("ivar has value %+i\n", ivar);

    printf("dvar has value %f\n", dvar);
    printf("dvar has value %0.2f\n", dvar);
    printf("dvar has value %8.2f\n", dvar);
}
```

This produces

```
ivar has value 5
ivar has value    5
ivar has value 0005
ivar has value +5
dvar has value 3.141500
dvar has value 3.14
dvar has value     3.14
```

1.3 Control characters (escape sequences)

We have already seen how to print a newline, using the **escape character** "\". Below are some other useful control characters.

\n	new line	\'	apostrophe	\0	null
\t	horizontal tab	\"	quotation mark	\\	backslash

1.4 Command-line arguments

The arguments that we pass to our program from the command prompt are called **command-line arguments**. They can be passed to `main()` using its (nearly) full prototype:

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

The variable `argc` is the *argument counter*. In fact, it includes the program name as an argument. The string array `*argv[]` contains the words (strings[2]) on the command line, including the program name. Hence `argv[0]` contains the name of the program, while `argv[argc-1]` contains the last argument.

For example, consider the code below:

Code listing 1.5: cl-arguments.c: receive and print command-line arguments.

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf("Number of arguments: %i\n", argc-1);
  for(int iArg=0; iArg<argc; iArg++) {
    printf("Argument %i: %s\n", iArg, argv[iArg]);
  }
  return 0;
}
```

After compiling this code to create the program `cl-arguments`, and running it as

```
$ ./cl-arguments Hello 123
```

the output will be

```
Number of arguments:  2
Argument 0:  ./cl-arguments
Argument 1:  Hello
Argument 2:  123
```

[2]In fact, `char *argv[]` indicates that `argv` is an array of strings, and since a string in C is an array of characters (see Section 5), `argv` is an array of (arrays of characters).

1.5 Receiving input values from the keyboard

The `scanf()` function is used to receive input from the keyboard. The prototype of the `scanf()` function is:

```
scanf("formatString", &var1, &var2, ...);
```

The `formatString` contains placeholders for variables that we intend to receive from the keyboard. An ampersand (`&`) comes is prefixed to the variables where we want to store the values, except character strings. You should not insert any additional characters in the format string other than placeholders and some special characters. An extra space or other undesired character may cause unexpected results. The following example receives multiple variables from the keyboard.

```
float a;
int n;
scanf("%i%f", &n, &a);
```

Note that `scanf()` does not do error checking. The programmer is responsible for validating input data (type, range, etc.) and preventing errors.

2 Decision control structures

C has two major decision-making instructions — the if/if–else construct and the switch construct.

2.1 The if statement

C uses the keyword `if` to implement the decision control instruction. The general form of the `if` statement looks like:

```
// for a single-line statement:
if(condition) statement;

// for a multiple-line statement:
if(condition) {
    block of statements;
}
```

Here `condition` can be any **logical expression**, often a comparison. For example

```
if(day < 0 || day > 31) printf("Day number should be between 1 and 31!\n");
```

The most common comparison operators are:

less than	<	equal to	==
less than or equal to	<=	not equal to	!=
greater than	>	logical and	&&
greater than or equal to	>=	logical or	||

2.2 The if–else statement

The `if` statement by itself will execute a single statement, or a group of statements, when the condition following `if` evaluates to true. The `else` statement can be used to execute code if the condition evaluates to false:

```
if(condition) {
    block of statements if condition is true;
} else {
    block of statements if condition is false;
}
```

2.3 Nested if constructs

If we write an entire `if-else` construct within either the body of the if statement or the body of an else statement. This is called *nesting* of `if` constructs. This is demonstrated by

```
if(condition1)
    statement;
else {
    if(condition2)
        statement;
    else {
        block of statements;
    }
}
```

2.4 The if/else if/else structure

The **`else-if`** construct is the most general way of writing a multi-way decision:

```
if(condition1)
    statement;
else if(condition2)
    statement;
else if(condition3)
    statement;
else if(condition4) {
    block of statements;
}
else
    statement;
```

The `condition` s are evaluated in order; if a condition is true, the `statement` or `block of statements` associated with it is executed, and this terminates the whole chain. The last `else` part handles the *default case* if none of the other conditions is satisfied. Hence, there can only be one `else` case. A working code example is shown in Code listing 2.1.

Code listing 2.1: if-else.c: demonstrate the if–else if–else construct.

```c
#include <stdio.h>
int main() {
  int var=2;

  if(var==1) printf("var equals unity\n");
  else if(var==2 || var==3) printf("var equals two or three\n");
  else printf("var is less than one or more than three\n");
}
```

2.5 The switch–case construct

The **switch-case** construct is a multi-way decision that tests whether a condition matches one of a number of constant integer values, and branches accordingly. The switch statement that allows us to make a decision from the number of choices is called a switch, or more correctly a switch-case-default, since these three keywords go together to make up the switch statement.

```
switch(expression) {
    case constant-expression1:
        block of statements;
        break;
    case constant-expression2:
        block of statements;
        break;
    ...
    default:
        block of statements;
}
```

The **default** section is entered if none of the earlier cases provide a match. Note that the **break** statement is needed if we want at most one match — in particular, without `break`, the **default** case will always be entered. A code example is shown below.

Code listing 2.2: switch-case.c: demonstrate the switch–case construct.

```c
#include <stdio.h>
int main() {
  int var=2;

  switch(var) {
  case 1:
    printf("var equals unity\n");
    break;
  case 2:
  case 3:
    printf("var equals two or three\n");
    break;
  default:
    printf("var is less than one or more than three\n");
  }
}
```

3 Loop control structures

Sometimes we want some part of our code to be executed more than once. We can either repeat the code in our program or use loops instead. It is obvious that if for example we need to execute some part of code for a hundred times it is not practical to repeat the code. Alternatively we can use our repeating code inside a loop. There are three methods to repeat a part of a program: `for`, `while` and `do-while` loops.

3.1 For loops

A **for** loop is the most straightforward example of a loop structure. It uses a control statement, which determines how many times the loop will run, and a command section. The command section is either a single command or a block of commands.

```
// for a single-line statement:
for(control statement) statement;

// for a multi-line statement:
for(control statement) {
    block of statement
}
```

The control statement itself has three parts:

```
for ( initialisation; test condition; run every time command )
```

1. The initialisation is performed only once, at the first iteration. The loop control-variable is initialised here.
2. The loop will continue to run as long as the test condition is true. If the condition becomes false, the loop will terminate.
3. Run every time command section will be performed in every loop cycle. We use this part to reach the final condition for terminating the loop, for example by increasing the loop counter by one. Thus, if the counter reaches the specified number of cycles, the loop condition becomes invalid and the `for` loop will terminate.

Code listing 3.1 shows a simple `for` loop, which prints `012345678`. Note that **`iVar++`** increases the variable `iVar` by one at every iteration. See Code listing 3.5 for another code example using a for loop.

Code listing 3.1: for.c: a simple for loop.

```c
#include <stdio.h>
int main() {
  for(int iVar=0; iVar<9; iVar++) printf("%i", iVar);
}
```

3.2 While loops

A **`while`** loop is constructed from a condition and a single command or a block of commands that must be executed at every iteration:

```
// for a single-line statement:
while(condition) statement;

// for a multi-line statement:
while(condition) {
    block of statements;
}
```

The statements within the `while` loop will keep on getting executed as long as the condition being tested remains true. In contrast to a for loop, the condition in a while loop is typically changed somewhere in the block of statements, often by an *external* event. When the condition becomes false, the control passes to the first statement that follows the body of the while loop.

Code listing 3.2 shows a simple `while` loop, which keeps asking for a number and printing the result, until you enter a negative number. Note that *external* information causes the loop to end. See also Code listing 3.4.

Code listing 3.2: while.c: a simple while loop.

```c
#include <stdio.h>
int main() {
  int num=999;
  while(num>0) {
    printf("Enter a number: ");
    scanf("%i", &num);
    printf("You entered the number %i\n", num);
  }
}
```

3.3 Do-while loops

The while and for loops test the termination condition at the top. By contrast, the third loop in C, the do-while, tests at the bottom after making each pass through the loop body. Hence, the body is always executed at least once. The syntax of the do-while structure is

```
do {
    statements;
} while(condition);
```

The `statements` are executed, after which `condition` is evaluated. If it is true, `statements` are executed again, until `condition` is false. A do-while loop is typically used to ensure that the `statements` within the loop are executed at least once.

Code listing 3.3 shows a simple `do-while` loop, which does the same as Code listing 3.2. However, note that in this case, we can initialise `num` as a negative number.

Code listing 3.3: do-while.c: a simple do-while loop.

```c
#include <stdio.h>
int main() {
  int num=-1;
  do {
    printf("Enter a number: ");
    scanf("%i", &num);
    printf("You entered the number %i\n", num);
  } while(num>0);
}
```

3.4 Break and continue statements

As well as in the switch-case structure, the **`break`** statement can be used inside a loop to terminate it if a specific condition is met. In the code example below, the program indefinitely (`while(1)` is *always* true!) asks for a number and prints it, until the entered number is negative:

Code listing 3.4: break.c: jump out of an infinite `while` loop using **`break`**.

```c
#include <stdio.h>
int main() {
  int num=0;

  while(1) {
    printf("Enter a number: ");
    scanf("%i", &num);
    if(num<0) break;
    printf("You entered the number %i\n", num);
  }
  printf("You entered the negative number %i - exiting.\n", num);
}
```

The **`continue`** statement can also be used in loops. When executed, it skips the rest of current iteration of the loop and returns to the top of the loop for the next iteration. The code example below prints the numbers 0 through 8, except 5:

Code listing 3.5: continue.c: skip an iteration in a `for` loop using **`continue`**.

```c
#include <stdio.h>
int main() {
  for(int i=0;i<9;i++) {
    if(i==5) continue;
    printf("%i\n", i);
  }
}
```

4 Arrays

Arrays are structures that hold multiple variables of the same data type. The first element in the array is labelled 0, so the last element has label 1 less than the size of the array ($n-1$). Before using an array its type and dimension must be declared. Note that internally, C treats an array as a *pointer* (see Section 7) that holds the memory address of the first element of the array. Hence element `[0]` refers to that address, element `[1]` to the next address, etc.

4.1 Array declaration

Like other variables we need to declare the type of an array. In addition, we use square brackets to define its size. The variable `array` declared below can contain 30 integers:

```c
int array[30];
```

The examples below indicate how to initialise an array while declaring it:

```c
int num[6] = {2, 4, 12, 5, 45, 5;}
int n[] = {2, 4, 12, 5, 45, 5;}
float press[] = {12.3, 34.2 -23.4, -11.3};
```

4.2 Accessing elements of an array

Once an array is declared, the individual elements of the array can be referred to by specifying an index between square brackets:

```c
printf("%i", num[1]);
num[2] = 13;
```

Remember that `num[1]` and `num[2]` are the *second* and *third* elements of `num`, since counting starts at 0.

4.3 Array example

The program below computes the average grade for a test made by 10 students:

Code listing 4.1: array.c: using an array to compute the mean.

```c
#include <stdio.h>
int main() {
  int avg, i, sum=0;
  int grades[10];  // array declaration

  for(i=0; i<10; i++) {
    printf("Enter grade %i: ", i+1);
    scanf("%i", &grades[i]);  // store data in array
  }

  for(i=0; i<10; i++)
    sum = sum + grades[i];  // use data from array
  avg = sum / 10;

  printf( "\nAverage grade = %i\n", avg );
}
```

5 Strings

5.1 Basic use of strings

Strings are like arrays of characters. Each ‘array element’ contains one character of the string. As a consequence, just like in the case of arrays, C internally treats a string as a `char` *pointer* that points to the memory address of the first character. C uses the subsequent memory addresses to find the other characters of the string, until a null character (‘\0’) is found, which marks the end of the string.

A code example reading and printing a string is:

Code listing 5.1: string.c: using a string.

```
#include <stdio.h>
#include <string.h>
int main() {
    char name[20];
    printf("Enter your name: ");
    scanf("%s", name);
    printf("Hello %s, how are you?\n", name);
    printf("Your name is %i characters long!\n", (int)strlen(name));
}
```

Note that we have used the `strlen()` function which requires `string.h` (see Sect 5.2). The output is:

```
Enter your name: Brian
Hello Brian, how are you?
Your name is 5 characters long!
```

If the user enters “Brian”, the first element of the array `name` will contain ‘B’, the second ‘r’ and the last ‘\0’. Hence, we could create the same string by assigning character values to each element:

```
name[0]='B';
name[1]='r';
name[2]='i';
name[3]='a';
name[4]='n';
name[5]='\0';
```

5.2 Standard library string functions

The string functions of the C standard library can be used by including `string.h`:

strlen	return the length of a string;
strlwr	convert a string to lowercase;
strupr	convert a string to uppercase;
strcat	append one string at the end of another;
strncat	append first n characters of a string at the end of another;
strcpy	copy a string into another;
strncpy	copy first n characters of one string into another;
strcmp	compare two strings;
strncmp	compare first n characters of two strings;
strcmpi	compare two strings without regard to case.

See `man string.h` for a complete list of functions.

6 Functions

A **function** consists of a block of code that can be used anywhere in the program by calling the function name. We have already discussed the special case of the `main()` function (see Sect. 1.1). The example below shows how to declare, define and call a simple function:

Code listing 6.1: function.c: declaring, defining and using a simple function.

```c
#include <stdio.h>

// Function prototype:
void writeMessage();

int main() {
    writeMessage();  // Call the function writeMessage()
}

// Function Definition:
void writeMessage() {
    printf("Hello, this is a test\n");
}
```

In the example above, we first declare the prototype of the function **`writeMessage()`** to ensure the compiler knows its syntax when first encountered. The type **`void`** indicates that the function does not return a value (see Sect. 6.2.1). The program starts (as always) in the `main()` function, where we call the function `writeMessage()`. There the `printf()` statement is executed, after which the function **returns**. Note that functions can also call each other. A function that calls another function is called the **caller function** or **caller**, while the other function is called the **called function** or **callee**.

There are two reasons to use functions. Firstly, they allow us to easily reuse code by calling the same function from different locations in the program. Secondly, dividing up pieces of code that form logical entities into functions helps to structure our code.

6.1 Function arguments

C functions are able to accept input parameters in the form of variables. These input variables can then be used in function body. In the code below we call the function `printMessage()` with a string and an integer as parameters. The function then prints the desired message as often as we request:

Code listing 6.2: function-arg.c: call a function with arguments.

```c
#include <stdio.h>

// Function prototype:
void printMessage(char message[20], int count);
```

```c

int main() {
  printMessage("Hello  ", 4);
}

void printMessage(char message[20], int count) {
  for(int c=0;c<count;c++)
    printf(message);
}
```

The output is: `Hello Hello Hello Hello`

6.2 Function return values

In mathematics we generally expect a function to return a value. In C also, functions can provide a return value. The return value of a function can have any type, like a variable. The type of the return value is identical to the type of the function.

The code below calls the function `sum()` with two integers as arguments. The function computes the sum of the two values, which is also an integer, and returns it to the calling function (`main()`). In `main()`, the result (7) is printed:

Code listing 6.3: function-return.c: a function with a return value.

```c
#include <stdio.h>
int sum(int a, int b);  // prototype

int main() {
    int result=sum(3, 4);
    printf("%i\n", result);
}

int sum(int a, int b) {
    int c = a+b;
    return c;
}
```

For the function `main()`, which is of type `int`, it is custom to return the value 0 to indicate that no errors occurred:

```c
int main() {
    ...
    return 0;
}
```

6.2.1 Void function type

As we have encountered above, C uses the type **`void`** to indicate functions without return value. A function of this type does not assign a value to a `return` statement and cannot be called as *e.g.* `result = function();`.

```c
void printMessage() {
    printf("Message!\n");
}
```

6.3 Passing variables by value

There are two ways in which programming languages can pass a variable to a function. By *value* and by *reference*. C by default uses *pass by value*, where the *value* of a variable is passed, resulting in one-way traffic:

Code listing 6.4: passByValue.c: pass a variable by value.

```c
#include <stdio.h>
void change(int y);

int main() {
  int x=2;
  printf("X equals %i\n", x);
  change(x);
  printf("X equals %i\n", x);
}

void change(int y) {
  y=5;
  printf("Y equals %i\n", y);
}
```

In the code, one may expect to find the result `X equals 5` in the last output line. Instead, both lines read `X equals 2`, even though `Y equals 5`! The reason is that the *value* 2 is passed to the function `change()`, but nothing is passed back. The parameter `y` is an *input* parameter only.

6.4 Passing variables by reference

The second method of sending variables to a function is called *pass by reference*. This allows the function to modify the value of the parameter in the calling function:

Code listing 6.5: passByReference.c: pass a variable by reference.

```c
#include <stdio.h>
void change(int *y);

int main() {
  int x=2;
  printf("X equals %i\n", x);
  change(&x);
  printf("X equals %i\n", x);
  return 0;
}

void change(int *y) {
  *y=5;
  printf("Y equals %i\n", *y);
}
```

Note that the differences seem minor: an ampersand (`&`) when calling `change()` and an asterisk (`*`) in the function itself. What we actually have done using these symbols, is sending the *memory address* of the variable `x` to the function `change()` using the operator &, rather than its *value*. The data type of the parameter `y` of that function has been changed from `int` to `(int *)`, so that it can store the memory address. The second asterisk, in the line `*y=5;`, is the operator that refers to the *value* that is stored in the address that `y` contains. Hence, the value 5 is now stored at that address, this time *replacing* the value 2. After the function returns, `main()` prints the value of `x` a second time, retrieving the (updated) value stored at the memory address for that variable. C variables that contain the memory address of other variables are called *pointers* (see Section 7).

Note that in Sect. 6.3, we passed the *value* of the variable `x`, which was stored at a *local* memory address of the function `change()`. This address was different from the address in the calling function (*i.e.* `main()`). When passing by *reference*, we pass the *memory address* rather than the value, so that both functions refer to the same address and an updated value in the one also causes an updated value in the other function.

Also note that passing by reference is still one-way traffic: we pass the memory address from the caller to the called function, and nothing is passed back. However, because the memory address is the same, the `main()` function can retrieve the updated value for `x` from the memory.

Finally, since C treats arrays as pointers (see Section 4), arrays are *always* passed by reference and the `&` should be omitted (see also Code listing 7.4).

7 Pointers

A **pointer** is a variable that contains the memory address of another variable. With the address known, different functions can retrieve or update the data stored in it.

7.1 Pointer declaration in C

Before we can store the address in a pointer variable, we not only need to tell the compiler that the variable is a pointer, but also what type of variable is stored at the address. Hence, a pointer declaration looks like this:

```
type *name;
```

Here, the asterisk (`*`) indicates that the variable `name` is a pointer, and `type` indicates the type of variable that can be stored at the address. Hence, a pointer to an integer variable would be declared as:

```
int *intPtr;
```

7.2 Pointer operators

Consider the declaration

```
int var = 3;
```

This declaration tells the C compiler to:

1. reserve space in memory to hold an integer value,
2. associate the name `var` with this memory location, and
3. store the value 3 at this location.

We can print the memory address as follows:

Code listing 7.1: variableAddress.c: print the value and address of a variable.

```c
#include <stdio.h>
int main() {
    int var=3;
    printf("Value of var = %i\n", var);
    printf("Address of var = %p\n", &var);
}
```

Here the ampersand (`&`) is the operator that returns the *address* of the variable `var`. We print this address with `printf()` using the format specifier `%p` for pointers. The output of the program above could be:

```
Value of var = 3
Address of var = 0x7ffcbd537c5c
```

Note that while the value of `var` should always be 3, the *address* at which it is stored will be different for different runs.

The other pointer operator available in C is the asterisk (`*`), called the **dereference operator**. This operator specifies the *value* that is stored at the address of a given pointer. Hence, the program above, which uses the normal integer variable `var`, can be rewritten using the *pointer* variable `ptr`:

Code listing 7.2: pointerAddress.c: print the value/address of a pointer and the value that is stored there.

```c
#include <stdio.h>
int main() {
  int var=3;
  int *ptr = &var;  // store the address of var in ptr
  printf("Value stored at ptr = %i\n", *ptr);
  printf("Address of ptr = %p\n", ptr);
}
```

While the output of this program is similar to that of the previous example, note that this time we need no &-operator to print the address, since `ptr` already contains an address. Instead, we now need the `*`-operator to dereference the pointer in order to print its value.

7.3 Pointer examples

Here is a program that demonstrates the relation between variables, addresses and pointers:

Code listing 7.3: pointers.c: print the values and addresses of variables and pointers.

```c
#include <stdio.h>
int main() {
    int var = 3;
    int *ptr = &var;

    printf("Value of var   = %i\n",var);
    printf("Value of var   = %i\n",*ptr);
    printf("Value of var   = %i\n",*(&var));

    printf("Address of var = %p\n",&var);
    printf("Address of var = %p\n",ptr);

    printf("Value of ptr   = %p\n",ptr);
    printf("Address of ptr = %p\n",&ptr);
}
```

The output of the above program should look like:

```
Value of var = 3
Value of var = 3
Value of var = 3
Address of var = 0x7fffe391af7c
Address of var = 0x7fffe391af7c
Value of ptr = 0x7fffe391af7c
Address of ptr = 0x7fffe391af80
```

Here, we notice that `*(&var)` equals `var`. This makes sense, since it is the value that is stored at the address of the variable `var`. Secondly, we notice that the *value* of `ptr` is the same as the address of `var`, which also makes sense, because this is exactly what `ptr` is. Finally, notice that, of course, `ptr` has a memory address of its own.

The example in Code listing 7.4 shows the passing of variables to functions using pointers. Note that since an array is a pointer, there is no need to use the `&` — arrays are always passed by reference (see Sect. 6.4). The notation `int *arr1` in the function prototype is equivalent to `int arr[]`. Note that only `var1` is passed by value and not updated by the function. Hence, the output is `99 20 11 12 21 22`.

Code listing 7.4: pointerFunction.c: passing variables to a function using pointers.

```c
#include <stdio.h>

void function(int var1, int *var2,  int *arr1, int arr2[]);

int main() {
  int var1=99, var2=99, arr1[2]={99,99}, arr2[2]={99,99};

  function(var1, &var2,  arr1, arr2);
  printf("%i  %i   %i %i   %i %i\n", var1,var2, arr1[0],arr1[1], arr2[0],arr2[1]);
}

void function(int var1, int *var2,  int *arr1, int arr2[]) {
  var1    = 10;
  *var2   = 2*var1;
  arr1[0] = 11;  arr1[1] = arr1[0]+1;
  arr2[0] = 21;  arr2[1] = arr2[0]+1;
}
```

8 Structures or structs

A structure or **struct** is a collection of (one or) multiple variables, possibly of different types, grouped together under a single name for convenient handling.

8.1 Declaring a structure

The general form of a structure declaration is given below (note the semicolon after the closing brace):

```
struct <structure name> {
    structure element 1;
    structure element 2;
    ...
    structure element n;
};
```

Once the new structure data type has been defined, one or more variables of the new type can be declared. For example, we can define the struct `book` and then declare the variables `b1`, `b2`, `b3` of type `book`:

```c
struct book {
    char title[10];
    float price;
    int pages;
};

struct book b1, b2, b3;
```

Like normal variables and arrays, struct variables can also be initialised when they are declared. The format is similar to that for defining arrays:

```c
struct book b1 = { "Basic", 130.00, 550 };
struct book b2 = { "Physics", 150.80, 800 };
```

Note that the *order* in which the elements of the struct are defined is important!

8.2 Accessing structure elements

C uses the **dot operator** (`.`) to access the *elements* of a struct variable. For example, the line

```
printf("The book %s has %i pages and costs e%f.", b2.title, b2.pages, b2.price);
```

should produce

```
The book Physics has 800 pages and costs e150.00.
```

However, if we are dealing with a **pointer** to a struct, we need to use `->` instead of the dot operator. Hence, instead of `&book.price` or `book.&price`, `book->price` is used, *e.g.*:

```
book->price = 99.95;
```

8.3 Struct example

The following code example illustrates the use of the data type `book`. Note that we use the function `setBookPrice()` to update the price of `book2`, so that we need to pass the struct by reference and need the `->` operator.

Code listing 8.1: struct.c: definition and use of a struct and its pointer.

```c
#include <stdio.h>

struct book {
  char title[10];
  float price;
  int pages;
};

void setBookPrice(struct book *bookI);
void printDetails(struct book bookI);

int main() {
  struct book book1 = { "Basic", 130.00, 550 };
  struct book book2 = { "Physics", 150.80, 800 };

  setBookPrice(&book2);  // Book2 is currently available at a discount

  printDetails(book1);
  printDetails(book2);
}

void setBookPrice(struct book *bookI) {
  bookI->price = 99.95;
}

void printDetails(struct book bookI) {
  printf("The book %s has %i pages and costs e%0.2f.\n",
         bookI.title, bookI.pages, bookI.price);
}
```

This should produce:

```
The book Basic has 550 pages and costs e130.00.
The book Physics has 800 pages and costs e99.95.
```

Note that variables can be *arrays* of structs. In the example above, we could have declared

```
struct book book[2];
```

9 Basic file access

9.1 File pointers

There are many ways to use files in C. The most straightforward use of files is via a **file pointer**:

```
FILE *fp;
```

Here, `fp` is a pointer to a file. The type `FILE` is defined in the header file `stdio.h`, which must be included in your program.

9.2 Opening a file

A file is opened using the `fopen()` function:

```
fp = fopen(filename, mode);
```

The `filename` and `mode` are both strings. The mode can be

r	read;
w	write, overwrite file if it exists;
a	write, but append instead of overwrite;
r+	read and write, do not destroy file if it exists;
w+	read and write, but overwrite file if it exists;
a+	read and write, but append instead of overwrite;

Hence,

```
fp = fopen("data.dat","a");
```

will open the file `data.dat` for writing, and any information written will be appended to the file if it already exists.

The function `fopen` returns `NULL` if the file could not be opened in the mode requested. The returned value should be checked before any attempt is made to access the file. The following code shows how the value returned by fopen might be checked. When the file cannot be opened a suitable error message is printed and the program halted. In most situations this would be inappropriate, instead the user should be given the chance of re-entering the file name.

Code listing 9.1: file.c: (try to) open a file.

```c
#include <stdio.h>
int main() {
  char filename[80] = "file.txt";
  FILE *fp;

  fp = fopen(filename,"r");
  if(fp == NULL) {
    perror(filename);
    return 1;  // Exit with error
  }

  // ... code that accesses the contents of the file ...
  return 0;  // Exit without error
}
```

The function `perror()` will provide a useful message for the last error that occurred, *e.g.*:

```
file.txt: No such file or directory.
```

Sequential file access can be performed with a.o. the following library functions:

`fopen(filename, mode)`	open a file;
`fclose(fp)`	close a file;
`fprintf(fp, formatstring, ...)`	write to a file;
`fscanf(fp, formatstring, ...)`	read from a file;
`getc(fp)`	get a character from a file;
`putc(c, fp)`	put a character in a file.

See the man pages for these functions for more details.

Appendices

A Programming exercises

Exercise 1

Write a C program with a for loop that iterates the variable `iLoop` from 0 to 5. In the body of the loop, use `if`, `if-else` and `else` statements as well as `printf()` to print the *word* that corresponds to that number. Hence, if `iLoop` equals 0, print ”zero” to screen, if it equals 1, print ”one”, etc. If the number is larger than three, print ”Larger than three” instead.

Exercise 2

The same as Exercise 1, but use a while loop instead of a for loop.

Exercise 3

In your main function, define a double variable `x` with the value `3.1415`. Use `printf()` to print `x` to screen, using only 3 decimals. Write the function `timesTwo()` that takes a double variable as argument, multiplies it with two and passes it back to the caller function using the **same argument** in the interface of the function. Do **not** use the `return` statement or `printf()` in `timesTwo()`!

In your main function, call the new function with `x` as argument and print `x` again to ensure that the value was really updated. Don't forget to define the prototype of the function `timesTwo()` before the main function and note that you must pass `x` by *reference*. What type does the function `timesTwo()` have if it has no return value?

Exercise 4

The same as Exercise 3, but in your main function add a variable `y` and set it to the same value as `x`. Print `x` and `y` to screen. Change your function `timesTwo()` so that `x` becomes an **input** argument only, and is *not* updated (*i.e.*, it is passed by *value*). Instead, return `x` at the end of your function, using the `return` statement. Note that you will need to change both the type of the function and of its argument.

In your main function, call `timesTwo()` with `x` as argument and store the return value in `y`. You will need to make two changes to this function call as well. Print `x` and `y` again, and verify that only `y` was changed.

Exercise 5

The same as Exercise 3 (*i.e.*, **no** `return` statement), but in your main function, create a **pointer** `px` and store the address of `x` in it. Call `timesTwo()` with `px` instead of `x`. Note that you will need to update the prototype of `timesTwo()` (as well as its call). Print the values of `x` and `px` before and after the call. What is the *placeholder* needed to print the value of a pointer? Note that you may need to cast `px` to `(void*)` when using it in `printf()` to remove compiler warnings.

Exercise 6

Copy your code from Exercise 5, and change the type of the argument of the function `timesTwo()` to `(void*)`, *i.e.* its prototype now becomes `void timesTwo(void *x)`. Make sure you type cast `px` to `(void*)` when you call the function from main (*i.e.*, *in* the call). In the function `timesTwo()`, receive the void pointer in the argument, cast it back to `(double*)` and store it in a different (pointer) variable. Then, update the *value* stored at the memory address stored in the double pointer. The result is still passed back to the main function through the argument in the function interface, **not** through a `return` statement. In your main function, print `x` and `px` before and after the call and verify that the *value* of `x` changed, but its *address* remained the same.

Bibliography

[1] **Choudhary, V.** *C Programming Language Cheat Sheet*, 2019. URL `https://developerinsider.co/c-programming-language-cheat-sheet/`. Visited 2019-03-22.

[2] **The GCC project**. *GCC, the GNU Compiler Collection*, 2018. URL `https://gcc.gnu.org/`.

Index