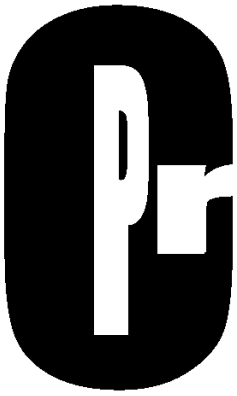
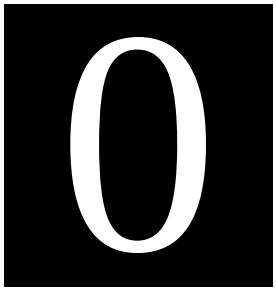


Programming in C
an introduction

PRC for E

Maarten Pennings

Version 0.5 (2007-12-31)



0 Preface

This book explains the C programming language. It assumes no programming knowledge. The reader is supposed to be somewhat familiar with binary arithmetic. Some knowledge of operating a computer is required in order to be able to use a compiler, which is needed for doing the exercises at the end of this book.

0.1 Position of this module (PRC for E)

This book was specifically written to support module 3 (PRC) for the trajectory SPR (system programmer) for Electronics students. The first module of SPR (ICM) presents sufficient background knowledge on signed and unsigned integers and their operations (addition, shifting, etc). The PRC module lays the groundworks for the last SPR module LLP (low level programming). In LLP the C programming language is used on a small microcontroller.

The standard PRC course (as opposed to the PRC for E that this book is written for) assumes previous programming experience. As a result, the standard PRC course does not explain what a variable, type, expression, statement or function is, it merely tells how to write them down in C. The PRC for E course does explain what they are, but as a consequence, it skips some of the more advanced topics. Most notably, PRC for E (this book) does *not* explain *pointers*.

Fortunately, pointers are not used in LLP. So PRC for E is sufficient in that sense.

However, those who want to program in C, should learn pointers. C without pointers is like driving a car in first gear only.

0.2 About the usage of English

We understand that for Dutch students, English is harder than Dutch. We therefore considered writing this book in Dutch. However, computer science is an English dominated profession. Most computer languages are English based (‘function’, ‘while’, ‘return’, ‘include’). Help files are English, most books are English, most forums on the internet are English. This book was assembled from various sources, all were in English.

So, we decided to write this book in English (but we did feel this justification was needed).

0.3 Structure

This book is tailored towards a module of 7 weeks. It consists of 6 chapters (chapters 1 to 6), one for each week. Chapter 7 (week 7) is for practicing an old examination. At the end of the book (chapter 8), there is an additional chapter with exercises, one set of exercises for each week (8.1-8.6). The class time will be two hours on the theory and two hours on the exercises per week.

The course PRC for E is concluded with a written examination of 100 minutes. The examination is an “open book examination”: it is allowed to bring any kind of written material to the examination (this book, exercise print-outs, personal notes) and even a calculator (but a plain one, not C programmable one).

In addition to passing the examination all exercises must be approved by the teacher.

0.4 References

http://computer.howstuffworks.com/c.htm – has nice visual demos
http://www.le.ac.uk/cc/tutorials/c/ – good examples
http://www.cprogramming.com/tutorial/c/lesson1.html
http://www.its.strath.ac.uk/courses/c/
http://cplus.about.com/od/beginnerctutoria1/l/blctut.htm
http://alpha.uhasselt.be/~gjb/MIT-C/slides/ – example programs (in dutch)

0.5 Table of contents

0.6 Revision history

0.0.1	2005 oct 10	Analysing existing PRC (1st meeting)	1 hr
0.0.2	2005 nov 14	Scoping PRC for E (2nd meeting)	1 hr
0.0.3	2005 nov 21+26	Inventorising existing books (visited bookshops, internet)	2 hr
0.0.4	2005 dec 24	Scanning through the websites Lennart de Graaf selected.	2 hr
0.0.5	2005 dec 28	Written chapter 'Program': what is C, what is programming language, edit-comp-link-exec.	2 hr
0.0.6	2005 dec 29	Added to chapter 'Program': using dev studio, print to screen, exercises for week 1	8 hr
0.0.7	2005 dec 30	Written chapter 'Expression': including exercises.	7 hr
0.0.8	2005 dec 31	Written chapter 'Statements': only theory, except the loops.	3 hr
0.0.9	2006 jan 1	Added to chapter 'Statements': loops and exercises.	3 hr
0.0.10	2006 jan 2	Written chapter 'Data types': arrays only theorie.	2 hr
0.0.11	2006 jan 4	Added to chapter 'Data types': structs, exercises.	2 hr
0.1	2006 jan 4	Written chapter 'Functions': theory and exercises. Mailed to Lennart and Agnes for review.	4 hr
0.1.1	2006 jan 29	Studied comments Lennart de Graaf on version 0.1	1 hr
0.1.2	2006 jan 30	Splitoff printf and scanf in new chapter 'Intermezzo on input and output' (comments Lennart)	2 hr
0.1.3	2006 jan 31	Added storage size and bit patterns (comments Lennart de Graaf)	3 hr
0.1.4	2006 feb 1	Added struct storage (comments Lennart de Graaf)	3 hr
0.2	2006 feb 2	Re-allocated exercises (now that chapters are reorganized), added chapter 'Preface', TOCs.	3 hr
0.2.1	2006 feb 4	Reviewed the whole document myself.	2 hr
0.2.2	2006 feb 5	Reworked my review comments	2 hr
0.2.3	2006 feb 5	Reworked comments of Marc Ridders	1 hr
0.3	2006 feb 5	Reworked review comments of Frans Meulenbroeks and published V0.3 for year 2005-2006.	0 hr
0.3.1	2006 dec 14	Reworked comments from last year (ch 0..3), added logo, added build figure, added program figure	4 hr
0.3.2	2006 dec 16	Reworked comments from last year (ch 4..5)	2 hr
0.3.3	2006 dec 17	Reworked comments from last year (ch 6..9) added header/body figure, examinations, mistakes	5 hr
0.4	2006 dec 17	Converted to pdf and published as V0.4 for year 2006-2007.	0 hr
0.5	2007 dec 31	Small corrections, added string layout, added passing arrays as param	2 hr
			67 hr

1 Program

This chapter explains the origin of the C programming language. It explains what a programming language is, what a program is, and that a programmer needs to edit, compile, and link a program before it can be executed. Near the end of this chapter we discuss the famous C program “Hello, world!”, so that we can start writing simple programs.

1.1 What is the C programming language?

The C programming language was developed at Bell Labs during the early 1970's. Quite unpredictably it derived from a computer language named B and from an earlier language BCPL. The earlier versions of C became known as *Bell Labs C* or *K&R C* after the authors of an earlier book, "The C Programming Language" by *Kernighan* and *Ritchie*. As the language further developed and standardized, a version know as *ANSI* (American National Standards Institute) C became dominant.[1]

As a programming language, C is rather like Pascal or Fortran. Values are stored in *variables*. Programs are structured by defining *functions*. Program flow is controlled using *loops*, *if*-statements and function *calls*. Input and output can be directed to the terminal or to files. Related data can be stored together in *arrays* or *structures*. Of the three languages, C allows the most precise control of input and output. C is also rather more terse than Fortran or Pascal. This can result in short efficient programs, where the programmer has made wise use of C's range of powerful operators. It also allows the programmer to produce programs which are impossible to understand. The C language also offers unparalleled pointer computation. Undisciplined use of pointers can lead to errors which are very hard to trace. This course does not deal with pointers (which actually means that a very important area of C is not covered).

C was originally closely coupled to the Unix operating system. One of the positive aspects of knowing C is that many other computer languages have been derived from it. There is an object oriented member in the C language family (C++), the new Microsoft flagship C# is based on it, and even the Sun language Java is heavily inspired by C’s syntax and semantics. So knowing C helps in starting with many other languages. However, those languages may differ considerably from C on a conceptual level, so understanding C helps, but it is not a free ticket.

Although C is relatively old, it is still one of the most used programming languages.[2] For example the Linux kernel and nearly all tools around it are written in C. Most embedded systems are written in C

1.2 What is a programming language anyhow?

C is a computer programming language. This means that we can use C to create a list of instructions for a computer to follow. C is one of thousands of programming languages currently in use. The word *language* suggests letters, words, sentences that follow rules to be correct. And indeed, the C programming language, like all computer languages, is very strict and unforgiving about errors (unlike humon longuoges where we con reploce oll o’s by o’s without losing our reoders, they do get irritoted though ☺). The aspect of forming correct “sentences” is called the *syntax* of a language.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  printf("Hello, world!\n");
  return 0;
}
```

[1] Unfortunately, many compilers claim to be ANSI compliant but aren't fully or they have specific extensions.

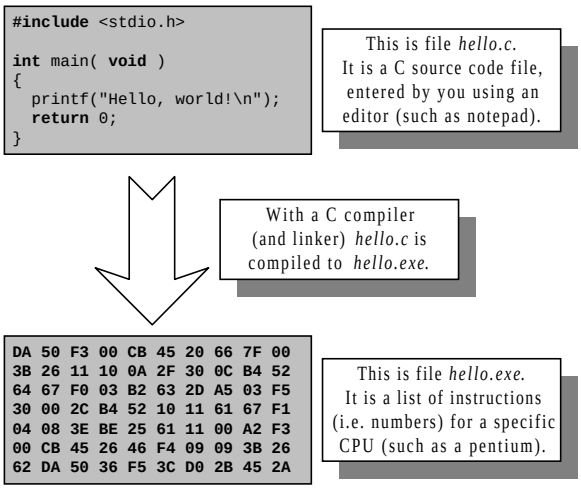
[2] Maybe, it is even *the* most popular language (see e.g. http://www.dedasys.com/articles/language_popularity.html).

Above, we see an example of a C program. There is an impressive amount of syntactic details that must be correct (we can not leave out any of the characters " * ; { [(< , # or the program will not work). We will be learning in this book how to write such programs, and we will learn what they *mean*. The latter is known as the *semantics* of a language.

> *The following English sentence has several syntax errors:* sUGAR arre sweet.
> *The following is syntactically correct (and has a sensible meaning or semantics):* Sugar is sweet.
> *The following is syntactically correct but its sematics are dubious:* Sugar is bitter.

By the way, in addition to the strict syntax in C there is also a notion of style ("coding conventions"). For example, all programmers *agree* that the lines between { and } should be *indented*, i.e. started with a couple of spaces. But this is a matter of taste, not part of the C syntax. Being taste, the result is that there are groups of people fighting over which coding convention is best (should indentation be with 2 or 4 spaces? Where should the { be, etc). Don't fight, chose a style and stick to it.[3]

C is a so-called a *compiled* language[4]. This means that once we write our C program, we must run it through a C compiler to turn our program into an executable that the computer can run (execute). The C program is the human-readable form, while the executable that comes out of the compiler is the machine-readable and executable form (i.e. a list of CPU instructions). What this means is that to write and run a C program, we must have access to a C compiler.



1.3 The edit-compile-link-execute cycle

Developing a program in a compiled language such as C requires at least four steps:

1. editing (or writing) the program
2. compiling it
3. linking it
4. executing it

1.3.1 Editing

We write a computer program with words and symbols that are understandable to human beings. This is the editing part of the development cycle. We type the program directly into a window on the screen and save the resulting text as a separate file. This is often referred to as the *source* file (we can read it with the *type* command in a DOS box[5] or the *cat* command in Unix). The custom is that the text of a C program is stored in a file with the extension *.c*.

---

[3] The author uses the "Philips Consumer Electronics coding conventions" because that's engraved in his mind.

[4] Another big class of languages are the *interpreted* languages (e.g. JavaScript on web pages).

[5] Where DOS box is mentioned, we not only mean *command.com* but also the command interpreter *cmd.exe* that comes with Windows NT/XP.

1.3.2 Compiling

We cannot directly execute the source file. To run on a computer system, the source file must be translated into "binary numbers" (instructions) understandable to the computer's CPU[6] (Central Processing Unit, for example, the 80x86 microprocessor). This process produces an intermediate object file – with the extension *.obj* or *.o* (which stands for object file).

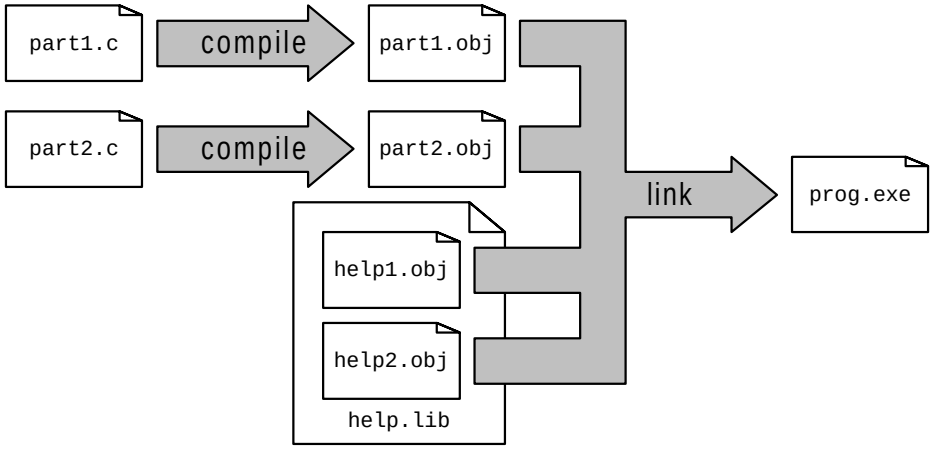
1.3.3 Linking

The first question that comes to most peoples minds is Why is linking necessary? The answer to this is that most programs are assembled from multiple source files each implementing parts of the resulting application. These parts need to be "linked" together.

Quite often, there is a set of stable parts implemented in several source files. These source files are compiled once and packaged together in a so called library (there is no magic in a library; it is much like zipping several *.obj* files into one *.lib* file). A normal way of working is to compile our *.c* files into *.obj* files and link those, together with the *.obj* files in (one or more) *.lib* files.

When we think this complex structure is only applicable for large applications, and not for our simple hello-program, we're unfortunately wrong. The reason is that many compiled languages come with *standard* library routines. Theses routines are written by the manufacturer of the compiler to perform a variety of tasks, from input/output to complicated mathematical functions (for example, most CPUs don't have an instruction to do a *sin* – sinus of an angle – rather the compiler adds a routine that computes the sinus using multiplication, addition etc). In the case of C the standard output function (the *printf* to put text on the screen) is contained in a library (*stdio*) so even the most basic program will require a library function and hence a linking step.

After linking the file extension is *.exe* (which stands for executable files). On unix, executables usually don't have an extension, and in embedded systems the resulting executable could be a *.hex* file (an actual memory image)[7].



1.3.4 Executing

Thus the text editor produces *.c* source files, which go to the compiler, which produces *.obj* object files, which go to the linker, which produces an *.exe* executable file. We can then run the *.exe* file as we can run any other application: simply by typing their names in the DOS box or by using a double-click in Windows Explorer. Note, that an *.exe* file is not only a list of instructions for the CPU; it usually starts with a header which instructs the loader of the operating system where and how to load the instructions (e.g. relocatable segments, dll's).

---

[6] This also means that when you want to run your C program on another CPU, you have to use another C compiler. In pactice you quite often also have to addapt your source files ("porting").

[7] For example, the Keil compiler for the 8051 CPU (used in LLP) generates hex files.

1.4 Building

The term *building* is used for compiling and linking together. Compilers and linkers are always delivered together, and most compilers also link. Or should we say that there is a small third program that actually first starts the compiler and then the linker? Very confusingly the term C-compiler is also used for the compiler and linker together.

That's why it suffices to type

```
gcc  part1.c  part2.c  -o myapplication.exe
```

in Unix.

There are many C compilers under Windows, including *gcc*. On windows, a likely candidate is the Microsoft C compiler integrated with Microsofts Developer Studio. Developer Studio is a so-called *IDE* or integrated development environment. This means that the whole edit-compile-link-execute cycle takes place within one environment. This environment not only guides the *compile* and *link* phases, but it even includes an *editor* and an environment for *executing* the compiled application. The latter is very convenient for *debugging*; running a program under "supervision" with the aim to find errors (or "bugs" as they are commonly referred to).

See the exercises for an introduction in using Microsoft Developer Studio.

1.5 Overview of the "Hello, world!" program

Let us start with the famous "Hello, world!" program that every book on C starts with.

```
#include <stdio.h>

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

Let's walk through this program to see what the different lines are doing. Of course the real details follow in later chapters.

The "Hello, world!" program starts with *#include <stdio.h>*. This line includes the "standard I/O library"[8] into our program. The standard I/O library lets us read input from the keyboard (called "standard in"), write output to the screen (called "standard out"), process text files stored on the disk, and so on. It is an extremely useful library. C has a large number of standard libraries like *stdio*, including libraries to manipulate date and time, math libraries, string libraries. A library is simply a package of code that someone else has written to make our life easier, in this case the *stdio* library comes with the compiler.

The line *int main(void)* declares the main function[9]. Every C program must have exactly one function named *main* somewhere in the code. At run time, that is, when the program is executing, execution starts at the first line of the *main* function.[10] Typically, C programs contain many functions in addition to main. In one of the last chapters of this book we learn how to write functions ourselves.

In C, the { and } symbols mark the beginning and end of a block of code. A block of code contains several statements. In this case, the block of code makes up the so-called body of the *main* function. It contains only two statements (*printf* and *return*).

The first statement, *printf* , sends output to *standard out* (e.g. the screen). The portion in quotes is called a string (a series of characters). For *printf*, the string describes the data to be printed. The string contains *literal characters* such as 'H', 'e', 'l', etc. It may also contain *escape characters*, for example carriage returns (\n).

The *main* function must return a number (an *int* or integer) to its caller (the operating systems loader) to signal succesful (or erroneous) termination. This is a requirement from ANSI C, and can be seen from the header: ***int***

---

[8] Actually, it doesn't include the *library* itself, but the *header* file describing the library.

[9] According to the ANSI C specification there is only one alternative: *int main(int argc, char *argv[])*. Unfortunately, many compilers allow many other alternatives like *int main( )*. Don't use these if you care for portable code or standards.

[10] That is, after the *loader* of the operating system has loaded the executable, it *calls* the main function.

main(void). The last line: *return 0;* causes the function to return, passing an code of 0 (no error) to the operating system.

By the way, the C compiler is rather indifferent about *whitespace* (not to be confused with *syntax* for which it is very picky). Whitespace is the computer-science term for layout (spaces, tabs, new lines), so the above program could also be rendered as:

```
#         include        <stdio.h>
int

main(
    void ) { printf
(  "Hello, world!\n")
            ;return 0
    ;}
```

but we wouldn't have many friends. On the contrary, source code should follow strict coding conventions to enhance readability, and comments should be added where needed. The old syntax for comments is to put them between /* and */ but the latest ANSI C also allows comments from // till the end of the line.

```
#include <stdio.h>

// It is good practice to start every function with an explication of what it does.
int main( void )
{
  printf("Hello, world!\n");
  return /* This comment is placed very strangly, but syntacticly ok */ 0;
}
```

1.6 Functions

Functions are one of the most important concepts in many programming languages, and C is no exception to this rule. Even in the simple program discussed in this chapter, there are already two functions! The first we encountered was *main*. Every C program must have a *main*; it defines which statements to execute upon startup of the program.

The second function was a bit hidden. But the first statement of main is actually a *call* to the *function* with the name *printf*. The function *printf* is defined (its body is implemented) in the standard I/O library (that's why we need the *#include* on the first line).

We can recognize a function *definition* from its body in braces { }:

```
…bla… SomeFunctionName( …bla… )
{
  some statements
}
```

and we recognize a function *call* from its actual arguments in parenthesis ( ):

```
…
SomeFunctionName( …arguments… );
…
```

Functions define a series of statements to perform. A function groups these statements and gives them a name (the function name). One could say that functions are like black-boxes that can perform some trick. The trick is performed by *calling* the function.

The first line of the definition (*…bla… SomeFunctionName(…bla…)*) defines the *interface* or *header* of the function: its name, what goes in and what comes out.

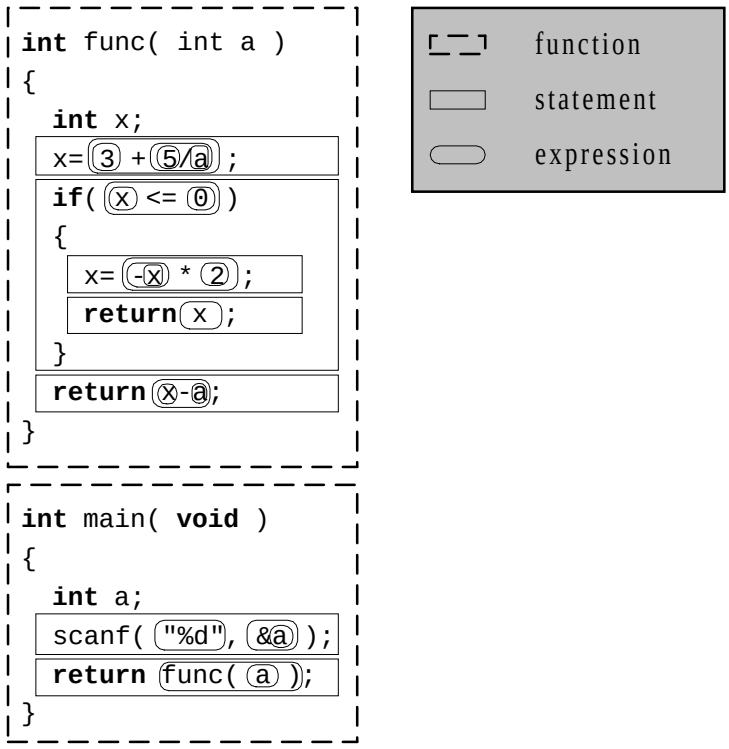
A well known example of a function available in C is the *sin* function. It has some clever (mathematical) trick to compute the sinus of a number. It is defined once (in the math library), and each time we need it, we simply call it. For this we need to know its name (*sin*) and that a number goes in (*sin(3.14)*) and a number comes out ("is returned"): *0.001592652*. Usually, the C compiler comes with manuals (Unix: manpages; Windows: help files – F1) that explain all functions in all libraries that come with the compiler.

We may write our own functions. We will see how to do that in Chapter 6.

1.7 Rest of this book

This chapter gave a outside-in overview of what a program in C looks like. The rest of this book will give an inside-out detaling of C. We start with the smallest items: expressions, which are used in bigger expressions, which are used in statements, which are used in bigger statements, which are used in functions, which are used in programs.

The figure below shows a program with its nested items.



2

2 Intermezzo on input and output

In order to be able to do some practical work, it is very convenient to know how to put characters on the screen. We've seen that *printf* takes care of this, but we need to understand a bit more about *printf* before we can effectively use it in our programs. To make our programs even more attractive; we'll have a look at *scanf*, which allows us to input something into our program from the keyboard.

Before we start with *printf*, we introduce strings; before we touch on *scanf*, we introduce variables.

2.1 Introduction into strings

As we will see later, the C programming language knows about different kinds of values (this is commonly referred to as *types*). The two most prominent ones are numbers and strings. We have seen examples of both. The statement

```
return 0;
```

contains the number (to be more precise, the *integer*) 0, and the statement

```
printf("Hello, world!\n");
```

contains the string *"Hello, world!\n"*.

The notion of a string of characters is common in nearly all programming languages, but its notation is not trivial. The author of this book once worked on a university, which had a 'no bike parking' sign near the entrance reading:

"no bike parking"

Note the quotes, which were really present – presumably the guy ordering the sign wrote a letter saying:

Please make a sign with the text "no bike parking".

It makes sense to have quotes to separate the text of the letter from the text of the sign, because otherwise the following letter gives an ambigous order to the sign manufacturer:

Please make a sign with the text no bike parking before 12am.

Which one would it be:

Please make a sign with the text "no bike parking" before 12am.
Please make a sign with the text "no bike parking before 12am".

On the other hand, it is not enough to agree that all quotes should be stripped, because then it would be impossible to order a sign with

no "wrecks" please

2.2 Strings

A string is a sequence of characters enclosed in " (double quotes). The C compiler stores all characters of the string (but not the enclosing quotes), and it always appends a 0 (null) for technical reasons beyond the scope of this book. So,

- "APE" is a string consisting of 3 characters 'A', 'P', and 'E'
  | A | P | E | null |
  that is, ascii codes 65, 80, and 69
  | 65 | 80 | 69 | 0 |

- "mary" is a string consisting of 4 characters.
  m | a | r | y | null
- "He said 'hi'." consists of H, e, space, s, a, i, d, space, single quote, h, i, single quote, dot.
  H | e | space | s | a | i | d | space | ' | h | i | ' | . | null

But suppose we want to have

```
He said
--hi
```

on the screen on two lines. How would we do that? Unfortunately, C does not allow a linefeed embedded within a string:

```
printf("He said
--hi");
```

There is this convention that ascii code 10 causes a linefeed. And C has the notion of *escape sequences* to embed those “unprintable” characters in a string. This is done by using the escape character \ followed by x (for hexadecimal) followed by hexadecimal digits. So

```
printf("He said\x0A--hi");
```

does the job. It maps to the following sequence in memory

H | e | space | s | a | i | d | linefeed | - | - | h | i | null

or, when using ASCII notation

72 | 101 | 32 | 115 | 97 | 105 | 100 | 10 | 45 | 45 | 104 | 105 | 0

It goes without saying that

```
printf("\x41\x42\x43");
```

prints

```
ABC
```

since hex 41 is the ascii code for A (and 42 is B and 43 is C).

However, since a linefeed is so popular, and \x0A so unreadable there is another shorthand: \n (n for newline). We have already seen \n in the hello world example. This means that

```
printf("He said\n--hi");
```

also does the job.

The table below lists other popular shorthands

```
\\       to embed a backslash in a string (an escaped backslash)
\"       to embed a double quote in a string
\'       to embed a single quote in a string (or in a character, see next chapter)
\b       to embed a backspace in a string
\n       to embed a linefeed in a string
\t       to embed a tab in a string
\0       to embed a null character in a string (don't do this!)
\xnn     to embed hexadecimal character nn in the string
```

Recall that these escape sequences are part of the syntax of the C programming language; they have nothing to do with *printf*.

2.3 Printf

The job of *printf* is to put characters on the screen (actually, to standard-out). When calling *printf*, we must pass at least one string, the so-called *format string*. The format string holds a “template” of what to print.

Let's look at some variations to understand *printf* better. Here is the simplest *printf* statement:

```
printf("Hello");
```

This call to *printf* has a format string that tells *printf* to send the string *"Hello"* (that is, the characters H, e l, l, and o) to standard out. Contrast it with this:

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

The second version sends the string *"Hello\n"*, that is H, e, l, l, o and a linefeed (ascii 10) to standard out.

```
printf("H\ne\nl\nl\no\n");
```

Prints Hello “vertically”.

```
H
e
l
l
o
```

The following line shows how to output an integer using *printf*.

```
printf("%d", 10);
```

The *%d* is a placeholder (for integers to be printed decimal) that will be replaced by 10 when the *printf* statement is executed. Of course it would be more logical to say

```
printf("10");
```

but we can also say

```
printf("%d", 5*(100-32)/9 );
```

Often, we will want to embed the calculated number within some other words. One way to accomplish that is like this:

```
printf("The temperature is ");
printf("%d", 5*(100-32)/9 );
printf(" degrees celsius\n");
```

An easier way is to say this:

```
printf( "The temperature is %d degrees celsius\n", 5*(100-32)/9 );
```

We can also use multiple *%d* placeholders in one *printf* statement:

```
printf( "Note: %d degrees fahrenheit is %d degrees celsius.\n", 100, 5*(100-32)/9 );
```

This prints

```
Note: 100 degrees fahrenheit is 37 degrees celsius.
```

It is now hopefully clear why the format string is a *template*, it is after all, a string with *%d* holes in it that still needs to be filled out.

In addition to *%d* for decimal printing we can also use *%x* for hexadecimal printing (or *%X* for uppercase hexadecimal). There is also *%c* for character printing (treating the number as an ascii value). Thus

```
printf("[%d,%x,%X,%c]", 75, 75, 75, 75 );
```

prints

```
[75,4b,4B,K]
```

The *%f* is for printing floating point numbers, and there is also a placeholder for string (*%s*):

```
printf("He said '%s' and %f!\n", "Hi", 2.5);
```

which prints

```
He said 'Hi' and 2.500000!
```

This is sometimes handy because there are several extra tricks with *printf*, for example padding.

```
printf("A123456789\nB%5s\nC%6d\n", "ape", 54 );
```

pads the ape string with spaces until it is 5 characters long and the 54 integer with spaces until it is 6 long:

```
A123456789
B  ape
C    54
```

For more details on *printf* tricks look in the developer studio help file or internet.

> In the *printf* statement, it is extremely important that the number of placeholders in the format string corresponds exactly with the number and type of the values following it. For example, if the format string contains three *%d* operators, then it must be followed by exactly three integers.
> Failing to do so might crash the program.

We have seen that the C language uses the \ as escape character in strings (in order to get some special characters embedded in a string). In order to embed the string escape character \ itself in a string it needs to be escaped: \\. Similarly the *printf* routine has chosen to use the % as escape character in format strings (in order to get holes embedded in the format string). In order to embed the format string escape character % itself in a format string it needs to be escaped: %%. So

```
printf("There is a reduction of %d%%.\n", 25 );
```

which prints

```
There is a reduction of 25%.
```

2.4 Variables

As a programmer, we will frequently want our program to "remember" values (for later use). For example, if our program requests a value from the user, or if it calculates a value, we will want to remember it somewhere so we can use it later. The way our program remembers things is by using *variables*. For example:

```
int b;
```

This line says “I want to create some room called *b* that is able to hold one integer number”. A variable has a *name* (in this case *b*) and a *data type* or *type* for short (in this case *int*, an integer). We can store a value in *b* by saying something like:

```
b= 5;
```

We can use the value in *b* by saying something like:

```
printf( "%d", b );
```

But we can also use *b* in an expression:

```
printf( "The square of %d is %d", b, b*b );
```

Or, when there is a second integer variable *a*, we can assign variable *a* a value using an expression in which variable *b* occurs:

```
a= 2*(b+1)
```

When *b* is 5, *(b+1)* equals 6, so *2*(b+1)* equals 12. As a result, variable *a* is set to 12 by this assignment.

Let us now look at a program that shows variables in action

```
#include <stdio.h>

int main( void )
{
  int a;
  int b;
  int c;
  a= 5;
  b= 7;
  c= a + b;
  printf( "%d + %d = %d\n", a, b, c );
  return 0;
}
```

The first three lines of the *main* function *declare* three integer variables named *a*, *b* and *c*. The next two lines initialize the variable named *a* to the value *5* and *b* to *7*.

The next line adds *a* and *b* and assigns the result to *c*. What happens is that the computer adds the value in *a* (*5*) to the value in *b* (7) to form the result *12*, and then places that new value (*12*) into the variable *c*. We say: the variable *c* is assigned the value *12*. For this reason, the = in this line is called the *assignment operator*. It is pronounced as "becomes" as in "*c* becomes *a* plus *b*" (don't say "is" as in "*c* is *a* plus *b*").

The *printf* statement then prints the line *5 + 7 = 12*. The *%d*-s in the *printf* statement act as placeholders for values. There are three *%d* placeholders, and at the end of the *printf* line there are the three variable names: *a*, *b* and *c*. *Printf* matches up the first *%d* with *a* and substitutes 5 there. It matches the second *%d* with *b* and substitutes *7*. It matches the third *%d* with *c* and substitutes *12*. Then it prints the complete line to the screen: 5 + 7 = 12. The +, the = and the spacing are a part of the format line and get embedded automatically between the *%d* operators as specified by the programmer.

As an alternative, we could save variable *c* and do the calculation in *printf*:

```
#include <stdio.h>

int main( void )
{
  int a;
  int b;
  a= 5;
  b= 7;
  printf( "%d + %d = %d\n", a, b, a+b );
  return 0;
}
```

We wrap up this chapter with some rules on variables.

Before a variable can be used in a function, it must be *declared*. Declaration of a variables in a function[11] must occur as the first lines[12].

```
int main( void )
{
  int a;   // Declaration
  a= 5;    // Statement
  float b; // Declaration error: can not have a declaration after a statement
  …
```

A declaration consists of a type[13] followed by a variable name. Variable names must start with a lower- or uppercase letter (or underscore) and may be followed by any number of lower- or uppercase letters, digits or underscore. A variable name may not be a C keyword. These are good varibale names: *ape*, *Nut*, *i*, *Tax2005*, *shoot2kill*, *printf*, *main*, *number_of_columns*. These are illegal as variable name: *2005Tax* (can not start with number), *shoot 2 kill* (can not have spaces), *for* (is a C keyword). Variables names are case sensitive so *ape* and *Ape* are different variable names.

It is allowed to add an initializer to a declaration:

---

[11] It is also possible to declare variables *outside* a function; then the variable can be used by *all* functions. Such a variable is called a *global* variable. Global variables are considered bad style, but sometimes they are necessary.

[12] The C++ language, which is an extension of C, does allow a declaration anywhere in a block of code.

[13] As we will see later, not just a type but a *type expression*.

```
int a= 1;
int b= a+a;
```

It is allowed to group the declaration of several variables of the same type (but some consider it bad style because it mixes badly with type expressions and initializers):

```
int a, b, c;
```

2.5 Scanf

Until now, our programs could only perform output. Let's now delve into data input. The *stdio* library has a function *scanf* – the input counterpart of *printf*.

The *scanf* function allows us to accept input from standard-in, which for us is generally the keyboard. The *scanf* function can do a lot of different things, but it is generally unreliable unless used in the simplest ways. It is unreliable because it does not handle human errors very well. But for simple programs it is good enough and easy-to-use.

The simplest application of *scanf* looks like this:

```
scanf( "%d", &i );
```

The program will read an integer value that the user enters on the keyboard (*%d* is for integers, as it is in *printf*, so *i* must be declared as an *int*) and place that value into variable *i*.

The *scanf* function uses the same placeholders as *printf*:

- *int* uses *%d*
- *float* uses *%f*
- *char* uses *%c*
- character strings (not discussed) use *%s*

We must put & in front of the variable used in *scanf*. The reason for this is pointers[14] (which is outside the scope of this book). It is easy to forget the & sign, and when we forget it, our program will almost always crash when we run it.

In general, it is best to use *scanf* as shown here – to read a single value from the keyboard. Use multiple calls to scanf to read multiple values. In any real program, we would use the *gets* or *fgets* functions instead to read text a line at a time. Then we will "parse" the line to read its values. With this approach we can detect errors in the input (entering 5o instead of 50) and handle them as we see fit. But this is beyond the scope of this book.

But in other words, never do

```
scanf( "%d %d", &i, &j );  // WRONG
```

The *printf* and *scanf* functions will take a bit of practice to be completely understood, but once mastered they are extremely useful.

The fragment below shows a simple calculator, well, extremely inconvenient adder is a better description.

[14] A *scanf("%d",i)* call will make *scanf* read an integer (that's what the first argument *"%d"* tells it to do) and store that at a memory location specified by the second argument. If variable *i* happens to have the value, say, 0, the read integer will be stored at location 0. This is most likely wrong. Instead, we would like the read integer to be stored at the location reserved for variable *i*. Suppose that *i* is located at address 0400, then *scanf* should have been given the address 0400 as second argument. The C compiler has a trick for this: *&i* returns the address of *i*, that would be 0400 in this example. So *scanf("%d",&i)* does the trick.

```c
#include <stdio.h>

int main( void )
{
  int a;
  int b;
  int c;
  printf("Enter the first value:");
  scanf("%d", &a );
  printf("Enter the second value:");
  scanf("%d", &b );
  c = a + b;
  printf("%d + %d = %d\n", a, b, c);
  return 0;
}
```

3

3 Expressions

C programs consist of functions (explained in chapter 6) which consist of statements (explained in chapter 4) most of which contain expressions. Expressions denote computations, the activity that gave computers their name. Expressions are explained in this chapter.

3.1 Introduction

C is a programming language, for programming a computer, and computers compute. The formal word for a single computation recipe like *2+5* or *2*(b+1)* is *expression*. Expressions compute to a value: *2+5* computes to *7* and *2*(b+1)* computes to 12 in case *b* equals 5 (since 2 times 6 (5 plus 1) equals 12).

Let us examine that last expression. Assuming we have a variable *b* we can form an expression using *b*:

```
2*(b+1)
```

The * and + are called *operators*. These two happen to be *binary* operators because they have *two* operands (the + has *b* and *1*, the * has *2* and *(b+1)*). In C, there are also *unary* operators (like - in -*x*) and there is even a ternary operator.

The operands of an expression are either numbers (officially: literals), variables, function calls or sub-expressions. The following expression shows them all:

```
1 + a + cos(1) + (3*5)
```

It should be noted that each expression has a so-called *type*, that subtly changes the meaning of an expression. For example, 7 and 7.0 both represent the number seven. But the former is an integer and the latter is a floating point. This results in a different meaning for / the divide operator:

```
7   / 2         // computes to integer 3
7.0 / 2         // computes to float 3.5
```

In the former expression, we divide two integers, resulting in an *integer* result, hence the 3 instead of 3.5.

When using variables, we have to make the type explicit. The C language has several standard types for variables. To name a few:

- *int* integer (whole number) values (such as 3, 5721, or –55).
- *float* floating point values (such as 1.0, -1.5, 3.1415926535, or –2.11E+17).
- *char* single character values (such as 'a', 'Z', '3' or '+').

Before we stop with this introduction, let's have a look at a simple expression that sometimes bewilders novices. It is in the commented line in the program below:

```
#include <stdio.h>

int main( void )
{
  int x;
  x= 5;
  printf( "x=%d\n", x );
  x= x+1;                  // increment variable x by one
  printf( "x=%d\n", x );
  return 0;
}
```

This prints

```
x=5
x=6
```

Recalling the advice on pronunciation, the commented line reads "*x* becomes *x* plus 1". And *x* was 5 so it becomes *5+1* that is, *6*. Incrementing a variable by one occurs so often that C has a shorthand notation for it: *x+=1* or even *x++* (both are explained in a section below).

3.2 Types

The C programming language comes with integer numbers and floating pont numbers and it knows about characters (but barely).

3.2.1 Integers

The type *int* stores integer numbers like *0, 1, 2, 3*, but also *-1, -2, -3*. One of the problems in C (and many other languages for that matter) is the question: What is the biggest (and smallest) number fitting in an *int*? The ANSI specification leaves this largely to the C compiler. It does specify that an *int* should be stored in at least 16 bits. So, on every CPU and every compiler, all numbers in the range -32768..32767 fit into an *int*. On a modern[15] PC (and a modern compiler) an *int* is 32 bits (-2147483648..2147483647), but the Keil compiler we use for the 8051 CPU in LLP will have 16 bits *int*'s.

The C programming language has the modifiers *short* and *long* that modify the number of bits used for *int*.

```
short int a;
int       b;
long int  c;
short     d; // same as short int
long      e; // same as long int
```

ANSI not only specifies that *int* is a least 16 bits wide, it also specifies that *short* has a size smaller or equal than that of *int* and that *long* has a size greater or equal than that of *int*. This is really a pain when writing a program that must run on multiple CPUs. As a rule of thumb:

	short	**int**	**long**
PC (pentium and bigger)	16	32	32
8051	16	16	32

All of the above are *signed* integral numbers. There are two other modifiers: *unsigned* and the superfluous *signed*.

```
unsigned int       a;
int                b; // is signed
signed int         c; // same as int
unsigned short int d0;
short unsigned int d1; // same as unsigned short int
unsigned short     d2; // same as unsigned short int
short unsigned     d3; // same as unsigned short int
// similar 4 cases for long
```

The signed and unsigned modifiers do not change the size of the variable (so if an *int* is 16 bits, a *signed int* and an *unsigned int* are also 16 bits).

In chapter 5 we discuss in more depth how integers are actually stored (size, bit patterns, signed and unsigned).

3.2.2 Floating point numbers

The type *float* stores floating point numbers like 0, 1.5e-30, 1.5, 15.0, 150.0, 1.5e+30 and their negative counterparts. Next to *float*, there is *double* and *long double*. Typically, these use the 4 respectively 8 and 10 byte (32, 64 and 80 bits) representation of the IEEE 754 standard discussed in ICM.

```
float f= -3.14e+20;
```

[15] A word like "modern" is of course dangerous. We are talking standard computers around 2000 (plus or minus 10 years?).

In chapter 5 we discuss in more depth how *float*'s are stored.

3.2.3 Characters

The C programming language also features the type *char*. This type stores characters like 'A', 'B', 'C', …, '0', '1', '2', '3', …, 'a', 'b', 'c', …, '+', '=', '(', …, but also '\n', '\t', '\'', … . Note that a *char* is a single character enclosed in single quotes (as opposed to a string, which is a series of characters enclosed in double quotes).

The funny aspect is that characters in C are actually numbers. So the type *char* stores integer numbers, and 'A' is a funny way of writing 65 (the ascii value of A). So the code below is perfectly legal C:

```
int  i;
char c;
c= 'A';
i= c+2;
printf( "i=%c i=%d c=%c c=%d", i, i, c, c );
```

and this prints

```
i=C i=67 c=A c=65
```

On most CPU's *char* is a byte (8 bits). This makes it a very often used data type in low level software that communicates bytes with hardware. Unfortunately ANSI C does not specify whether *char* is signed or unsigned, so if it matters to us, we must add these modifiers. For example, when using the Keil compiler with the 8051 CPU in LLP, we often see

```
unsigned char portvalue;
```

3.2.4 Strings

Characters have one special feature in the C language: there is a special notation for a series of characters, the so-called *strings*. This has already been introduced in the previous chapter. Since pointers are outside the scope of this book, and a string is also a pointer to a series of characters, this book will largely ignore strings (we will only use it in *printf*).

Keep in mind that

- "A" is a string of length 1 and 'A' is a character;
- "AB" is a string of two characters and 'AB' is a syntax error because single quotes must enclose a single character;
- "\n" is a string of length 1 (with the linefeed character) and '\n' is a single character;
- "" is an empty string and ' ' a syntax error because single quotes must enclose a single character.

3.2.5 Void

We have seen another type in the examples: *void*.[16] This type can not hold any kind of value. It is mainly used in functions to specify that the function computes nothing, as in ***void*** *f(int x)*, or that a functon has no arguments, as in *int main(**void**)*.

Void is also used extensively with pointers, which are not covered in this book.

3.3 Literals and constants

A C program will typically feature *literals* (sometimes referred to as *constants*, but that's less precise) of the different types. We have seen several examples already, but there are some new ones:

- 0, 1, 2, 3, –1, -2, -3 for integral types
- 'A', 'B', 'C', '0', '1', '2', '3', 'a', 'b', 'c', '+', '=', '(', '\n', '\t', '\'' also for integral types (but typically *char*)
- 0x0, 0x9, 0xA, 0xB, 0xAA, 0xF3F8, 0xffff also for integral types (hexadecimal)

[16] Void means empty (*leeg* in Dutch).

- 0, 1.5e-30, 1.5, 15.0, 150.0, 1.5e+30, -1.5E30 for floating point types
- "", " ", "Ape nut mary", "He said \"Hey, it's my book!\"\n" for string types.

Most programmers agree that it is ok to have literals like 0 and 1 in our code. But when there is a fragment like

```
f= e * 2.20371;
```

most programmers will frown on us and ask "what is this magic constant". In such a case, give the magic constant a name (making it less magic) by converting it into an explicit constant.

```
#define  GuildersPerEuro  2.20371
…
int main( void )
{
  …
  f= e * GuildersPerEuro;
  …
}
```

Observe that there is no semicolon at the end of the *#define* line.[17]

3.4 Operators

A literal is a very simple expression. So is a constant or a variable. These are actually the basis for more complex expressions. More complex expressions are formed by applying an operator to one or more less complex expressions.

For example

```
3.95
GuildersPerEuro
e
f
```

might be a floating point literal, a constant respectively a variable and a variable and as such (basic) floating point expressions. These can be combined using operators into more complex floating point expressions.

```
e * GuildersPerEuro
3.95 + f
```

And these can be combined (using operators) into a yet more complex floating point expression.

```
(e * GuildersPerEuro) - (3.95 + f)
```

The *, + and – are examples of operators. This section introduces several of C's operators.

3.4.1 Arithmetic operators

The *arithmetic operators* are the ones typically taught at school.

```
printf("%d", 14+4); // addition           prints 18
printf("%d", 14-4); // subtraction        prints 10
printf("%d", 14*4); // multiplication     prints 56
printf("%d", 14/4); // (integer) division prints 3
printf("%d", 14%4); // remainder (modulo) prints 2
printf("%d",   -4); // negation           prints -4
```

Note that - is not only a binary operator (14-4) but also a unary operator (-4).

As soon as both operands of / are integer, the operator performs an integer division (pronounced as "div"). So 14/4 equals 3, but 14.0/4.0, 14.0/4 and 14/4.0 all equal 3.5 (one operand is float, so the division becomes a float division). In case of an integer division, the remainder is lost. There is a special operator % (pronounced as "mod") to compute the remainder: 14%4 equals 2 because 4 times 14/4 (that is, 4 times 3) equals 12 so we have a remainder of 2 (14-12).

[17] Many C coding conventions suggest to use all capitals for constants: GUILDERSPEREURO, which is quite unreadable, so that often underscores get added GUILDERS_PER_EURO.

These operators are applicable to integral types (including *char*) and floating types (except %). It is *not* possible to add (concatenate or glue together) strings in C with +.

3.4.2 Relational operators

The C programming language does not have a separate data type for *true* and *false*. In many other languages this type is known as Bool or Boolean. C simply uses *int* instead and the values *1* and *0* for *true* respectively *false*. With this knowledge the outcome of the following *relational operators* should be clear.

```
printf("%d", 5<3 ); // less than           prints 0 (false)
printf("%d", 5<=3); // less or equal       prints 0 (false)
printf("%d", 5==3); // equal              prints 0 (false)
printf("%d", 5>=3); // greater or equal    prints 1 (true)
printf("%d", 5>3 ); // greater than       prints 1 (true)
printf("%d", 5!=3); // unequal            prints 1 (true)
```

Observe that equality is denoted with = = (where most other languages use a single =) and inequality is denoted with != (where several other languages use < >).

These operators are applicable to integral types (including *char*) and floating types. It is *not* possible to compare strings using these operators.

3.4.3 Logical operator

The C programming language might not have a separate datatype for Booleans, it does have operators for them, the so called *logical operators*. However, since *int* is used for Booleans, a decision had to be made on what the meaning is of any value other than 0 or 1. ANSI decided that any value other than 0 means true. Knowing this, the following should be clear.

```
printf("%d", 1&&1 ); // logical and  prints 1 (true)
printf("%d", 1&&0 ); // logical and  prints 0 (false)
printf("%d", 0&&0 ); // logical and  prints 0 (false)
printf("%d", 5&&3 ); // logical and  prints 1 (true) 5 and 3 are both seen as true
printf("%d", 1||1 ); // logical or   prints 1 (true)
printf("%d", 1||0 ); // logical or   prints 1 (true)
printf("%d", 0||0 ); // logical or   prints 0 (false)
printf("%d", 5||3 ); // logical or   prints 1 (true) 5 and 3 are both seen as true
printf("%d", 0||3 ); // logical or   prints 1 (true) 3 is seen as true
printf("%d", !0   ); // logical not  prints 1 (true)
printf("%d", !1   ); // logical not  prints 0 (false)
printf("%d", !3   ); // logical not  prints 0 (false) 3 is seen as true
```

Note that ! is an example of a unary operator.

“Normally”, logical operators are used to assemble *relational* sub-expressions, as in the following examples:

```
(0<=x) && (x<10)
(x<0)  || (x>=10)
!(x==0)
```

3.4.4 Bitwise operators

The ALU of most CPUs is capable of bitwise manipulation. The C programming language has operators for them. The following *bitwise operators* are only applicable to integral types (including *char*). Recall that 5 is binary 101 and 3 is binary 11.

```
printf("%d", 5&3 ); // bitwise and   prints 1  (0000000000000001)
printf("%d", 5|3 ); // bitwise or    prints 7  (0000000000000111)
printf("%d", 5^3 ); // bitwise xor   prints 6  (0000000000000110)
printf("%d", ~3  ); // 1-complement  prints -4 (1111111111111100)
printf("%d", 5<<3); // upshift       prints 40 (0000000000101000) 000 shifted in
printf("%d", 5>>2); // downshift     prints 1  (0000000000000001) 01 shifted out
```

The acronym *xor* means ‘exclusive or’. Also recall that 1-complement is a bit-flip (officially known as *inverse*).

The result of the bitwise operators *depends on the size of the int* (in the example above 16 bits is assumed).

The upshift always shifts in 0. The downshift shifts in 0 for unsigned integral types. For a negative signed integer it might shift in a 1 but also a 0, this is very inconveniently left unspecified by ANSI. If we shift as many bits or more as the size of the (left) operand, the behavior of the shift is also unspecified.

Note that ~ is an example of a unary operator.

Do not confuse || (logical or) with | (bitwise or):

```
printf("%d", 5||3 ); // logical xor   prints 1 (true; 5 and 3 are both seen as true)
printf("%d", 5 |3 ); // bitwise or    prints 7 (bitwise or of 0101 and 0011 is 0111)
```

3.4.5 Operators with side-effects – part 1

There is one class of operators in C which should be used wisely. These are operators that not only use the *value* of its operand, but actually *change* the operand. Of course, this is only possible if the operand is a variable. This probably sounds very cryptic. Let's look at an example.

As noted earlier, one of the most common things in C is incrementing a variable by one. This is so common, that there is a short hand for that:

```
int i=3;
printf("%d\n",i);
i++; // here is the increment-by-one shorthand
printf("%d\n",i);
```

This prints

```
3
4
```

The trick is that *i*++ not only increments *i* by one but also has a value, namely the old value of *i*. So, in C it is perfectly legal to write

```
int i=3;
int j=0;
printf("i=%d j=%d\n",i,j);
j= 5 * i++;
printf("i=%d j=%d\n",i,j);
```

This prints

```
i=3 j=0
i=4 j=15
```

As we see, *i* is incremented by one (from 3 to 4) and *j* is set to the 5 times the old value of *i* (3).

In addition to the post-increment (*i*++) there is also a pre-increment operator (++*i*). The expression ++*i* increments *i* by one and has as value the *new* value of *i*. So,

```
int i=3;
int j=0;
printf("i=%d j=%d\n",i,j);
j= 5 * ++i;  // changed from post- to pre-incement
printf("i=%d j=%d\n",i,j);
```

This prints

```
i=3 j=0
i=4 j=20
```

As we see, *i* is incremented by one (from 3 to 4) and *j* is set to the 5 times the new value of *i* (4).

In addition, there are also pre- and post-decrement operators (- -*i* and *i*- -). They decrement their operand and have as value the new respectively old value of the operand.

We should use these operators sparingly. It's best to only use ++ and - - in isolation.

3.4.6 Operators with side-effects – part 2

It gets worse. There are several other operators with side effects: =, +=, -=, *= etc. These *assignment operator* assign, add, substract, respectively multiply the variable on the left with the value on the right as a side-effect, but also have the outcome as value. So

```
int i=3;
int j=2;
i+= 4*j;
printf("i=%d j=%d\n",i,j);
```

prints

```
i=11 j=2
```

and

```
int i=3;
int j=2;
j= (i+= 4*j);
printf("i=%d j=%d\n",i,j);
```

prints

```
i=11 j=11
```

since *i+=4*j* not only sets *i* to 11 but has a value of 11, which is assigned to *j*.

In isolation the assignment-operators make sense, but not when cascaded[18]. The only sensible application of cascading probably is

```
i= j= 0
```

which sets both *i* and *j* to zero.

Why do we have to know this? Because of the single most-often made error in C. Consider

```
int a=5;
int iszero;
iszero= a==0;
printf("iszero=%d a=%d\n",iszero);
```

This computes (in *iszero*) whether *a* equals 0. It prints *iszero=0* (false) because *a* was 5 and it prints *a=5* because it is unmodified.

However, many people make the mistake of writing

```
int a=5;
int iszero;
iszero= a=0; // assignment operator instead of relational operator
printf("iszero=%d a=%d\n",iszero);
```

This print still prints *iszero=0*, so it looks ok! However, it also prints *a=0*, because *a* is *assigned* the value 0 (and that is assigned to *iszero*).

> Be aware that = *assigns* the value on the right to the variable on the left whereas = = *compares* the left-hand value with the right-hand value. It is one of the most frequent errors in C to write = (becomes) where == (equals) is intended.[19]

3.4.7 Combining operators

A final warning is on evaluation order. When we write

```
i= 2 * 3+4;
```

---

[18] Dutch: *geschakeld*

[19] Actually, the most frequent error is writing *if(a=0)* instead of *if(a==0)* but the if-statement has not yet been explained.

the value of *i* becomes 10 (6 plus 4) not 14 (2 times 7). The reason for this is that multiplication has a higher precendence than addition.[20] But we can (and usually should) override that with parenthesis:

```
i= 2 * (3+4);
```

So precedence is about the order of evaluating *operators*.

What most people don't know is that ANSI C leaves unspecified the order of evaluating the *operands*.[21]

```
int i= 0;
int j= ++i + i;
```

The above might result in *j* getting the value 1 where most people would expect 2. They claim: first *i* is pre-incremented so the left hand side operand of + has value 1 and the right-hand side is also *i* which then has the value of 1, so *j* is set to 1+1. However since the order of computing the operands is free, a C compiler might decide to *first* compute the right-hand operand of + (the *i*, which has value 0) and then the left-hand side operand (the ++*i*, which increments *i* from 0 to 1 and has value 1). Finally it evaluates the operator (the + on 1 and 0) resulting in *j* being set to 1.

In the following more realistic examples it is not guaranteed that *f* is called before *g*.

```
int i= f(3) + g(5);
printf( "f=%d, g=%d\n", f(3), g(4) )
```

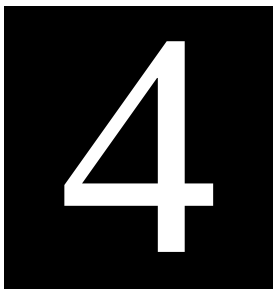
3.5 Outside the scope of this course

Several aspects of expressions are outside the scope of this book. For example

- Several operators (typecasts, pointer dereferencing, if-operator, comma-operator).
- The details of operator precedence.
- The details of evaluation order and sequence points.
- Typecoersions, sign extension, integer promotion.

[20] Some people (claim to) know the precedence of all operators. But is it wise to assume that the *maintainer* of your code also knows them? After all, there are nearly 50 opeartors in C. Furthermore, precendence is not intuitive in all cases. For example << (which is like multiplying) has lower precedence than + (whereas * has higher). The bitwise operators (&, ^, |) have lower precedence than the relational ones like <= (wheras + and – have higher).

[21] ANSI C does define the order of evaluating operands for some operators, the so-called *sequence points* (e.g. *&&* and ||).



4 Statements

This chapter discusses several incarnations of statements: assignments, function-calls, blocks, the if-statement, and two loop-statements: while and for.

4.1 Statements

An expression *computes* something, but it doesn't *do* anything (unless it is an expression with side effects). Statements on the other hands are the things that actually do something. We have already seen several examples of statements.

For example,

```
i= 5;
```

is a so-called *assignment statement*;

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

is a *function-call statement*, and

```
return 0;
```

is a so-called *return* statement.

We have also seen the *block statement*.

```
{
  … first declarations …
  … then a series of statements …
}
```

It was a bit disguised, but the *main* function has a block statement after its header (the *int main(void)* part). Observe that the block statement consists of an opening brace {, then zero or more declarations, then zero or more statements, and finally a closing brace }. Multiple statements can simply be put one after the other.

The block statement is actually rather important. The rest of this chapter shows several other statements such as *if*, *while* and *for*. All of these conditionally and/or repeatedly execute the a single statement. If that one statement is not enough, the block statement is used. It groups several statements into one.

4.2 If-statement

Sometimes, a program needs to take different steps when a certain condition holds. This is achieved with an *if-statement*, also known as selection statement, conditional statement, or branch statement. The condition is a "boolean" expression, i.e. an integer expression that is either 0 (false) or non-zero (true).

Here is a simple C program demonstrating an if-statement:

```
#include <stdio.h>

int main( void )
{
  int i;
  printf("Enter a value:");
  scanf("%d", &i );
  if( i<0 )
    printf("Warning: the value is negative\n");
  printf("Done\n");
  return 0;
}
```

This program accepts a number from the user. It then tests the number using an if-statement to see if it is less than 0. If it is, the program prints a warning message. Otherwise, the program prints no warning. The *i<0* portion of the program is the condition (a "boolean" expression). C evaluates this expression to decide whether or not to print the message. If the expression evaluates to "true" (non-zero), then C executes the single statement immediately following the *if* keyword (the so-called *then statement*). If the expression is "false" (zero), then C skips the then statement.

In either case, execution continues with the statement after the if/then; *Done* is always printed.

Note, the parenthesis after the *if* keyword are mandatory! So the following is an error.

```
if i<0        // Error: parenthesis missing
  printf("Warning: the value is negative\n");
```

The C programming language is case sensitive, so the if-statement must be written all lower case (and similar for all other keywords: *while*, *for*, *return*, …).

Coding conventions dictate that the then statement is indented. However, the C compiler doesn't care. So the following fragments all achieve the same (but especially the third one is frawned upon).

```
if( i<0 )
  printf("Warning: the value is negative\n");

if( i<0 ) printf("Warning: the value is negative\n");

if( i<0 )
printf("Warning: the value is negative\n");
```

We could even convert the then statement into a block statement "grouping" just the function call.

```
if( i<0 )
{
  printf("Warning: the value is negative\n");
}
```

Let's see how to deal with an if requiring multiple statements in the then part. The good approach is a block.

```
if( i<0 )
{
  printf("Some extra statement\n”);
  printf("Warning: the value is negative\n");
}
```

A common mistake is to indent both, but forgetting the braces.

```
if( i<0 )
  printf("Some extra statement\n”);
  printf("Warning: the value is negative\n"); // Indentation misleading or braces missing
```

Recall that C doesn't care about indentation (it's for human readability), it only cares about braces for grouping. So the above fragment will print

```
Some extra statement
Warning: the value is negative
```

when *i* is negative, and it will print

```
Warning: the value is negative
```

when *i* is positive (or zero).

Back to the if-statement. It has an optional *else* part. Here's slightly more complex example, that uses it:

```
#include <stdio.h>

int main( void )
{
  int i;
  printf("Enter a value:");
  scanf("%d", &i );
  if( i<0 )
    printf("The value is negative\n");
  else
    printf("The value is not negative\n");
  printf("Done\n");
  return 0;
}
```

The then statement as well as the *else statement* (the statement immediately following the *else*) could be any kind of statement, including an if-statement!

```
#include <stdio.h>

int main( void )
{
  int i;
  printf("Enter a value:");
  scanf("%d", &i );
  if( i<0 )
    printf("The value is negative\n");
  else
    if( i==0 )
      printf("The value is zero\n");
    else
      printf("The value is positive\n");
  printf("Done\n");
  return 0;
}
```

The two *if*-statements are said to be cascaded[22]. Cascading *if* 's are quite often not indented; rather they are written as follows (this is style, not syntax):

```
if( i<0 )
  printf("The value is negative\n");
else if( i==0 )
  printf("The value is zero\n");
else
  printf("The value is positive\n");
```

Alternatively, we could follow the always-use-braces convention

```
if( i<0 )
{
  printf("The value is negative\n");
}
else
{
  if( i==0 )
  {
    printf("The value is zero\n");
  }
  else
  {
     printf("The value is positive\n");
  }
}
```

or the nearly-always-use-braces convention (the author's preference)

[22] Dutch: *geschakeld*

```
if( i<0 )
{
  printf("The value is negative\n");
}
else if( i==0 )
{
  printf("The value is zero\n");
}
else
{
   printf("The value is positive\n");
}
```

All four alternatives are equal in semantics.

The C language also features a *switch* statement that is a shorthand for cascading *if*'s. It is beyond the scope of this book.

Here is a more complicated Boolean expression:

```
if( (0<=x) && (x<10) )
  z=1;
else
  z=0;
```

This statement says, "If (the value in) variable *x* is greater or equal to 0, and *x* is less than 10, then set the variable *z* to 1, otherwise set it to 0". By the way, this if-statement could be reduced to

```
z= (0<=x) && (x<10);
```

since the value of *(0<=x) && (x<10)* happens to be 1 if the value in variable *x* is geater or equal to 0, and less than 10, and 0 otherwise!

We conclude with the templates of the if-statement:

```
if( expression )
  statement
```

or

```
if( expression )
  statement
else
  statement
```

4.3 While loop

Until now, there is a serious drawback with our repertoire of C constructs: the run-time of the computer is proportional to the amount of lines we write. If we want the computer to do more, we have to type more. That is now going to change; we introduce iterations, repetitions, loops or whatever we want to call them.

For example, suppose we want to print the integers 0, 1, 2, … upto but excluding 10. We could write:

```
#include <stdio.h>

int main( void )
{
  printf("0\n");
  printf("1\n");
  printf("2\n");
  printf("3\n");
  printf("4\n");
  printf("5\n");
  printf("6\n");
  printf("7\n");
  printf("8\n");
  printf("9\n");
  return 0;
}
```

and we typically see that for each number extra we want to be printed, we need to add one line: “the run-time of the computer is proportional to the amount of lines we write”. The alternative is a loop. The program below has the same output as the program we just saw.

```
#include <stdio.h>

int main( void )
{
  int i;
  i= 0;
  while( i<10 )
  {
    printf("%d\n",i);
    i= i+1;
  }
  return 0;
}
```

Not only is this shorter, it is also much easier to update. If we need the numbers till 20, we just have to change the 10 in 20.

The same remarks as for the if-statement apply to the *while* statement: the *while* keyword must be written lower case, the parenthesis are a mandatory part, and the so-called *body* of the while statement (the statement immediately after the while keyword) is a single statement. If more statements need to be repeated, use a block statement for the body.

```
while( expression )
  statement
```

Let's have a look at a more complex example. Say that we would like to create a program that prints a Fahrenheit-to-Celsius conversion table. Recall that the formula for that is

$$c = \frac{5}{9} \times (f - 32)$$

The table is easily accomplished with a while loop:

```
#include <stdio.h>

int main( void )
{
  int f;
  f= 0;
  while( f<=120 )
  {
    printf( "%4d degrees F = %4d degrees C\n", f, (f-32) * 5 / 9 );
    f= f + 10;
  }
  return 0;
}
```

If we run this program, it will produce a table of values starting at 0 degrees F and ending at 120 degrees F. The output will look like this:

```
   0 degrees F =  -17 degrees C
  10 degrees F =  -12 degrees C
  20 degrees F =   -6 degrees C
  30 degrees F =   -1 degrees C
  40 degrees F =    4 degrees C
  50 degrees F =   10 degrees C
  60 degrees F =   15 degrees C
  70 degrees F =   21 degrees C
  80 degrees F =   26 degrees C
  90 degrees F =   32 degrees C
 100 degrees F =   37 degrees C
 110 degrees F =   43 degrees C
 120 degrees F =   48 degrees C
```

The table's values are in increments of 10 degrees. We can see that we can easily change the starting, ending or increment values of the table that the program produces.

If we wanted our values to be more accurate, we could use floating point values instead:

```
#include <stdio.h>

int main( void )
{
  float f;
  f= 0.0;
  while( f<=120.0 )
  {
    printf( "%6.2f degrees F = %6.2f degrees C\n", f, (f-32.0) * 5.0 / 9.0 );
    f= f + 10.0;
  }
  return 0;
}
```

We see that the declaration for *f* has been changed to a *float*, and the *%f* symbol replaces the *%d* symbol in the *printf* statement. In addition, the *%f* symbol has some formatting applied to it: The value will be printed with six digits and two digits following the decimal point. We have also changed all literals from *int*'s to *float*'s (by appending a '.0').

Now let's say that we wanted to modify the program so that the temperature 98.6 is inserted in the table at the proper position. That is, we want the table to increment every 10 degrees, but we also want the table to include an extra line for 98.6 degrees F because that is the normal body temperature for a human being. The following program accomplishes the goal:

```
#include <stdio.h>

#define step  10.0
#define human 98.6

int main( void )
{
  float f;
  f= 0.0;
  while( f<=120.0 )
  {
    if( (f-step<human) && (human<f) )
    {
        printf( "%6.2f degrees F = %6.2f degrees C\n", human, (human -32.0)*5.0/9.0 );
    }
    printf( "%6.2f degrees F = %6.2f degrees C\n", f, (f-32.0)*5.0/9.0 );
    f= f + step;
  }
  return 0;
}
```

The extra output line with the human body temparature (98.6) needs to be inserted just before a regular output line is printed for a higher temperature (100). So, we're tempted to write

```
if( human<f )
```

However, this causes the "human body line" to be printed just before 100, but also before 110 and 120! It should only be printed once, namely in the slot 90..100. However, we wanted the program to still function when stepping with 20 degrees or 5 degrees, that's why we have introduced the constant *step*, and the two-sided check in the *if*.

4.4 For loop

When taking a closer loop at while statements, we recognize a pattern:

```
start ;
while( stay )
{
  statement
  step ;
}
```

Just before the while loop, there is an initialization (the *start* expression), there is an expression that determines whether to stop the looping or whether to stay, and the body of the loop typically steps (increments) a variable. This is such a common pattern, that the C language has an abbreviation for it: the *for* loop.

```
for( start ; stay ; step )
  statement
```

Again, if the body of the for loop consists of more than one statement a block must be used.

Let's examine the pattern of temperature table program from the previous section. We recognize the start, the stay, the stat(ement) and the step:

```
#include <stdio.h>

int main( void )
{
  float f;
  f= 0.0 ;
  while( f<=120.0 )
  {
    printf( "%6.2f degrees F = %6.2f degrees C\n", f, (f-32.0) * 5.0 / 9.0 );
    f= f + 10.0 ;
  }
  return 0;
}
```

So, this is rewritten using a *for* loop as follows:

```
#include <stdio.h>

int main( void )
{
  float f;
  for( f=0.0; f<=120.0; f=f+10.0 )
  {
    printf( "%6.2f degrees F = %6.2f degrees C\n", f, (f-32.0) * 5.0 / 9.0 );
  }
  return 0;
}
```

or even (since the body of the loop is a single statement, and using a shorter assignment operator):

```
#include <stdio.h>

int main( void )
{
  float f;
  for( f=0.0; f<=120.0; f+=10.0 )
    printf( "%6.2f degrees F = %6.2f degrees C\n", f, (f-32.0) * 5.0 / 9.0 );
  return 0;
}
```

As a general guideline, use a *for* loop when the number of steps (*iterations*) is known before hand, and use a while loop when the number of steps is not known before hand.

The most typical usage of a for loop is using an integer variable *i* that is incremented by one (*i++*) until it reaches an upper limit. As an example, consider the program below that computes the sum of the numbers 0 up to but excluding 10.

```
#include <stdio.h>

int main( void )
{
  int i;
  int s;
  s= 0;
  for( i=0; i<10; i++ )
    s= s+i;
  return 0;
}
```

4.5 The semicolon

In some languages, such as Pascal, the ; is a seperator (some call it a joiner) of statements. So, if *S1* is some statement and *S2* is a statement, then *S1;S2* is also a statement.

C on the other hand, does not have a (extra) symbol to separate (join) two statements. Statements are seperated (joined) simply by juxtaposition[23]. So, if *S1* is some statement and *S2* is a statement, then *S1 S2* is also a statement.

This sounds wrong at first, because there are a lot of semicolons in a C program. The truth is that the semicolons are actually *part of most statements*. For example, the function call statement has the form

```
func( arg ) ;
```

the assignment statement has the form

```
var = expression ;
```

the return statement has the form

```
return expression ;
```

but the block statement has the form

```
{ statement statement statement … }
```

without a semicolon.

4.6 Outside the scope of this book

There are several other forms of statements:

- empty statement (*;*) sometimes used in loops;
- multiple choice *switch* statement;
- "repeat" statement (*do*..*while*);
- *break* statement;
- *continue* statement;
- *goto* statement.

These are beyond the scope of this book.

[23] In plain English: "placing head to tail"; in Dutch "gewoon achter elkaar zetten".

5 Data types

Section 3.2 introduced types (integers, floating point, characters, strings and it even mentioned *void*). These types are known as standard types.

C also provides means to build bigger types from several smaller types, starting from these standard types. The first major construct is called *array*; it is a series of objects of the *same* type. For example, a string is actually an array of characters. The second major construct is a *struct* (i.e. a *structure*, also known as *record*); it is a set of objects of multiple types. For example, a date (day=25, month="December") is usually stored as a *struct*.

5.1 Arrays

An *array* lets us declare and work with a collection of values of the *same* type. For example, we might want to create a collection of five integers. One way to do it would be to declare five integers directly:

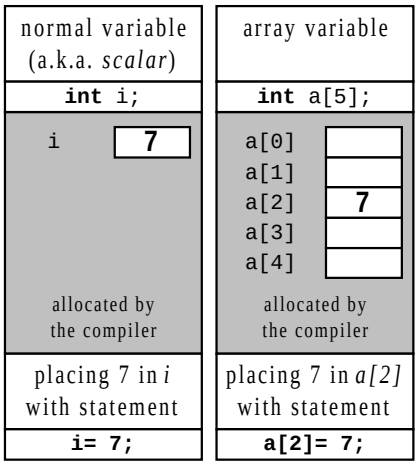
```
int i;
int j;
int k;
int l;
int m;
```

This is okay, but what if we needed a thousand integers? An easier way is to declare an *array of five integers*:

```
int a[5];
```

The five separate integers inside this array are accessed by *position*. This form of accessing is called *indexing*. All arrays (in C) start at index 0 and go to *n-1*. Thus, *int a[5];* contains five elements, indexed with 0, 1, 2, 3, and 4. For example:

```
int a[5];
a[0] = 12;
a[1] = 9;
a[2] = 7;   // see drawing
a[3] = 14;
a[4] = 1;
```



One of the nice things about array indexing is that we can use a loop to manipulate the index. For example, the following code initializes all of the values in the *a* array to 0:

```
int a[5];
int i;
for( i=0; i<5; i++ )
  a[i]= 0;
```

By the way, there are two occurrences of 5 in this code, both have to do with the *size* (5) of the array. If one changes, the other one should also change. Seasoned programmers make a constant for that.

```
#define SIZE 5
int a[SIZE];
int i;
for( i=0; i<SIZE; i++ )
  a[i]= 0;
```

The following code reads values into an array and prints them out in reverse order.

```
#include <stdio.h>
#define SIZE 5

int main( void )
{
  int a[SIZE];
  int i;

  for( i=0; i<SIZE; i++ )
    scanf( "%d", &a[i] );

  for( i=SIZE-1; i>=0; i-- )
    printf( "a[%d]=%d\n", i, a[i] );

  return 0;
}
```

One of the classical problems for arrays is to sort them. The issue here is to do that fast. This is usually measured in the number of *comparissons* that are needed. The top algorithms sort an array of n objects using $n \cdot ^2\log(n)$ comparisons (so, an array of 1000 names takes 10 000 comparisons). But these algorithms require software techniques outside the scope of this book. We present a middle-of-the-road algorithm; it requires n^2 comparisons (so the 1000 names take 1 000 000 comparisons which is 100 times slower).

```
#include <stdio.h>
#define SIZE 10

int main( void )
{
  int a[SIZE]= {5,9,2,1,0,3,4,8,6,7};
  int i;

  // Print old order
  for( i=0; i<SIZE; i++ )
    printf("%d ",a[i] );
  printf("\n");

  // Sort
  for( i=0; i<SIZE; i++ )
  { // Find the smallest int in the section [i..SIZE) and put it at position i

    // Let p point to that smallest int found so far.
    int p= i;

    // Let j loop over the remaining [i+1..SIZE) int's.
    int j;
    for( j=i+1; j<SIZE; j++)
    {
      if( a[j]<a[p] )
        p= j; // j points to a smaller int than p so remember that
    }

    // At this moment p points to the smallest int in [i..SIZE)
    { // Swap cell p with i in array a (start new block for temp var h)
      int h=a[i]; // temporary storage for a[i]
      a[i]=a[p];
      a[p]=h;
    }
```

```
  }

  // Print new order
  for( i=0; i<SIZE; i++ )
    printf("%d ",a[i] );
  printf("\n");

  return 0;
}
```

There are several aspect worth noting

- The problem has been decomposed in smaller parts, seperated with a whiteline
- Each of the parts has a comment explaining its workings.
- This program features a loop in a loop. The so-called outer loop (loop with *i*) loops over all indexes, with the aim to put the next smallest value at position *i*.[24] The inner-loop (loop with *j*) loops over all remaining indexes, with the aim to find the smallest in the remaining part.[25]
- The array is sized using a constant *SIZE*.
- The array has an initilizer (*{5,9,2,1,0,3,4,8,6,7}*).
- There is a fragment that swaps two integers; it is coded in a block statement with a local variable *h*.

5.2 Structs

Structures in C allow us to group objects (variables) of different type into one package. Here's an example:

```
struct MyStruct { int a; int b; float c; char d; } s;
```

This is actually a variable definition (for variable *s*) together with a struct definition: if we want another variable of the same type, we only need to repeat the structure tag, not its actual definition:

```
struct MyStruct r;
```

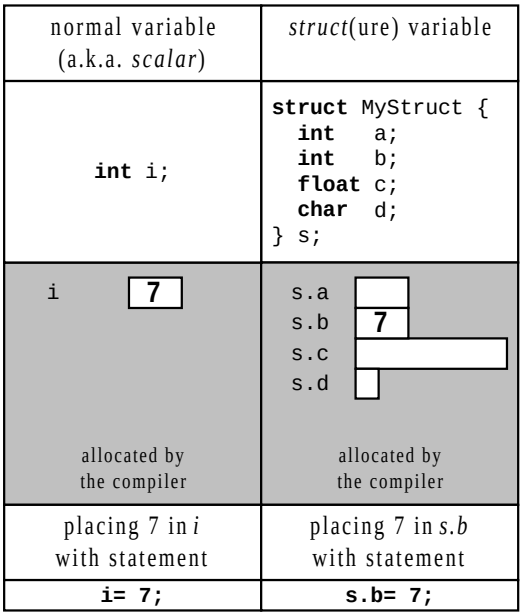
It is even allowed to have a struct definition without the variable definition, which only makes sense if the struct is used later on:

```
struct MyStruct { int a; int b; float c; char d; };  // No variable!
struct MyStruct s;
struct MyStruct r;
```

We access fields of structure using a dot (.), for example, *s.b= 7;* .

[24] Some developers actually write down an *invariant*: a proposition that stays true while the loop progresses. In this case, the invariant (of the outer loop) is: the array segment *a*[0..*i*) is correctly sorted and all values *a*[0..i) are less than any of the values in *a*[*i*..*SIZE*).

[25] The invariant of the inner loop is: *a*[*p*] is the smallest value in the segment *a*[*i*..*j*).



The program below uses a struct to store a date (a year, a month and a day). Observe that the fields of the struct can be read with *scanf* (e.g. *scanf("%d",&d.month);*), but they can also be used in the expression of the *if* (e.g. *d.month==12*) and can be printed (e.g. *printf("Christmas %d\n",d.year);*).

```
#include <stdio.h>

int main( void )
{
  struct date { int year; int month; int day; } d;
  printf("Enter year : "); scanf("%d",&d.year );
  printf("Enter month: "); scanf("%d",&d.month);
  printf("Enter day  : "); scanf("%d",&d.day  );
  printf("\nIt is ");
  if( d.month==12 && d.day==25 )
    printf("Christmas %d\n",d.year);
  else
    printf("%d %d, %d\n",d.year, d.month, d.day);
  return 0;
}
```

This has output (italic part entered by user)

```
Enter year : 2006
Enter month: 12
Enter day  : 25

It is Christmas 2006
```

5.3 Combinations

It is allowed in C to make arrays of structs, structs of structs, structs of arrays, or even an array of structs of an array, and so on. For example, to make an array for 10 date's we could write

```
struct date { int year; int month; int day; }  d[10];
```

With this definition, it takes the following code to set the last date to Christmas:

```
d[9].year= 2006;
d[9].month= 12;
d[9].day= 25;
```

Consider another example where we have a person record

```
struct person {
  struct date DayOfBirth;
  char        Name[20];
} Somebody;
```

and of course, we could have an array of thirty persons

```
struct person FirstYearStudents[30];
```

To check whether name of the last person starts with P, we would write

```
if( FirstYearStudents[29].Name[0] == 'P' ) …
```

5.4 Storing standard types

Values (and variables) of a different type use a different amount of storage space. Furthermore, the meaning of the bit patterns in that storage space also differs. This section explains storage size and bit patterns.

It is especially important when writing software that drives hardware, because then each bit needs consideration.

5.4.1 Storage size

Let us first have a look at the size of an expression. There is a special operator, *sizeof(E)*, that returns the size (in bytes) required to store expression *E*. So, when we run the following program (modern compiler on modern PC)

```
#include <stdio.h>

int main( void )
{
  signed         char sc;
  unsigned       char uc;
  signed   short int  ssi;
  unsigned short int  usi;
  signed         int  s_i;
  unsigned       int  u_i;
  signed   long  int  sli;
  unsigned long  int  uli;

  float               f;
  double              d;
  long double         ld;

  printf("signed         char: %d\n",sizeof(sc ) );
  printf("unsigned       char: %d\n",sizeof(uc ) );
  printf("signed   short int : %d\n",sizeof(ssi) );
  printf("unsigned short int : %d\n",sizeof(usi) );
  printf("signed         int : %d\n",sizeof(s_i) );
  printf("unsigned       int : %d\n",sizeof(u_i) );
  printf("signed   long  int : %d\n",sizeof(sli) );
  printf("unsigned long  int : %d\n",sizeof(uli) );

  printf("float              : %d\n",sizeof(f  ) );
  printf("double             : %d\n",sizeof(d  ) );
  printf("long double        : %d\n",sizeof(ld ) );

  return 0;
}
```

we get the following output

```
signed         char: 1
unsigned       char: 1
signed   short int : 2
unsigned short int : 2
signed         int : 4
unsigned       int : 4
signed   long  int : 4
unsigned long  int : 4
float              : 4
double             : 8
long double        : 8
```

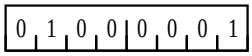
We see that *char's* only take one byte of memory to store, a *short* takes 2 bytes, an *int* and a *long* are the same, they both take 4 byte. The floating point type *float* takes 4 bytes and *double* and *long double* are the same: 8 bytes (remember these figures are not for C in general, they apply to modern compiler and a modern PC).

5.4.2 How is the value 65 stored?

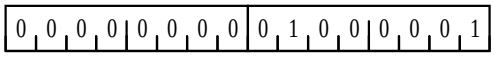
Let us next have a look at the bit patterns; at how a value is stored in memory. Suppose that we assign all variables the value 65 (with statements like *sc=65; f=65;*). How does the memory look like?

```
signed         char  41
unsigned       char  41
signed   short int   00 41
unsigned short int   00 41
signed         int   00 00 00 41
unsigned       int   00 00 00 41
signed   long  int   00 00 00 41
unsigned long  int   00 00 00 41
float                42 82 00 00
double               40 50 40 00 00 00 00 00
long double          40 50 40 00 00 00 00 00
```

The output above (memory is dumped in hex) shows that a (signed or unsigned) *char* with value 65 is filled with the expected bit pattern:



If we take a look at the longer integers, we see that they get extended with zeros. For example a *short* with value 65 is stored as follows.



The reader may wonder about the *float* (or (*long*) *double*) representation. Usually (as for the compiler used in this example) a *float* is stored according to the IEE 754 standard:

```
65(dec)= 41(hex)= 1000001(bin)= 1.000001 E 110 (bin) (normalized)

sign: +        coded as 0
exp:  110(bin) coded as 10000101 (the +127 notation)
mant: 0000010...0000 (drop the leading 1)

0 10000101 00000100000000000000000 (sign, exp, mant concatenated)

0100 0010 1000 0010 0000 0000 0000 0000 (bin)

42 82 00 00 (hex)
```

5.4.3 What does the pattern A0…A0 mean?

We can also reverse the game. Let us assume that we fill all the bytes that make up a value (so the two bytes of a *short*, or the 4 bytes of an *int*) with the bit pattern 10100000 (bin), that is A0 (hex) or 160 (dec).

If this is the memory reserved for a unsigned char, it represents the decimal value 160. If this memory is reserved for a signed char, it represents the decimal value –96. Why? Because by interpreting it as signed, it is looked at with two's-complement glasses. It has the MSB (most significand bit) 1, so it is negative.

The table below illustrates all cases.

```
signed         char: -96
unsigned       char: 160
signed   short int : -24416
unsigned short int : 41120
signed         int : -1600085856
unsigned       int : 2694881440
signed   long  int : -1600085856
unsigned long  int : 2694881440
float              : -2.721135e-019
double             : -1.587369e-151
long double        : -1.587369e-151
```

5.4.4 Which type to use?

When we want to store a number, which type to use? The rule of thumb is to always use *int*. It is the fastest and easiest. If storage matters (we have little memory and we have a large amount of numbers to store) only then think about using *short* or *char*. If *int* is too small (we want to store the value 40000 and the code needs to run on e.g. an 8051) use *long*.

Be very careful when mixing (sub)expressions of a different type in one expression. The following program checks whether signed integer *i* with value –1 is smaller than unsigned integer *n* with value +1.

```
int main( void )
{
  signed   int i= -1;
  unsigned int n= +1;
  if( i<n ) printf("yes"); else printf("no");
  return 0;
}
```

To our big surprise, this program prints

```
no
```

Why is that? Because we mix two types (signed *i* and unsigned *n*) in one expression (*i*<*n*) and then the dark rules of C apply. In this case ANSI C dictates that the unsigned wins, i.e. the signed value is interpreted as an unsigned. The signed *i* equals –1, so it has bit pattern 1111111111111111. With an unsigned interpretation, this has the value 65535. So the dark rules of C translate the expression to

```
if( 65535<1 ) printf("yes"); else printf("no");
```

which clearly should print no!

Do we have to know this? If we make a living out of C programming yes! Otherwise, just remember not to mix types in one expression; then we're relatively safe.

Another warning: never compare floats for equality. In the following program, we assign float *f* the value of 1 fifth. We print its value, which is indeed 0.2, and compare it with 0.2.

```
int main( void )
{
  float f= 1.0/5.0;
  printf( "f=%f\n", f );
  if( f==0.2 ) printf("yes"); else printf("no");
  printf("\n");}
```

This prints

```
f=0.200000
no
```

Why "no"? because the binary representation of 0.2 is not finite:

```
0.00110011001100110011001100110011
```

and then one 0.2 might be stored differently then another

```
0.00110011001100110011001101 // rounded (up)
0.00110011001100110011001100 // chopped (rounded down)
```

in the IEEE 754 notation.

So, remember to never compare floats for equality, rather use "less than":

```
if( abs(f-0.2)<1E-10 )
```

5.5 Storing a struct

What is the memory layout of a struct?

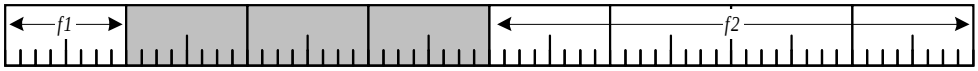
Suppose we have a struct like the following:

```
struct Two { char f1; int f2; } s;
```

On a PC we know that *sizeof(s.f1)* equals 1 and *sizeof(s.f2)* equals 4.

However on that same PC, *sizeof(s)* is not 5 but 8 bytes!

The reason for this is that (by default) compilers optimize for speed (and not for size). And for speed, it helps when integers start at an address that is a multiple of 4 (this is due to the PC hardware architecture). As a result, the struct has the following memory layout:



The gray bytes are not used. This is called *padding*. By padding structs the compiler achieves *alignment* of variables (fields), leading to fastest code.

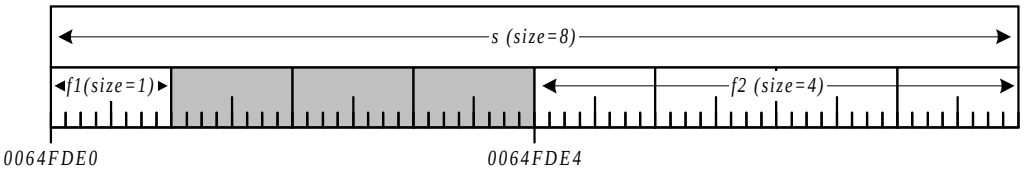
It is very easy to check this for ourself. Simply run the following code (there is a "new" operator here: & which returns the memory address of a variable[26], and a new printf template %p which is used to print addresses)

```
printf( "start of s.f1 at %p, size of s.f1 is %d\n", &s.f1, sizeof(s.f1) );
printf( "start of s.f2 at %p, size of s.f2 is %d\n", &s.f2, sizeof(s.f2) );
printf( "start of s    at %p, size of s    is %d\n", &s,    sizeof(s)    );
```

which prints

```
start of s.f1 at 0064FDE0, size of s.f1 is 1
start of s.f2 at 0064FDE4, size of s.f2 is 4
start of s    at 0064FDE0, size of s    is 8
```

confirming (detailing) the previous picture.



5.6 Outside the scope of this course

There are several types and type related aspects not covered in this book:

- enumeration types (*enum*)
- union types (*union*)
- bitfields
- multi dimensional arrays
- pointers (*) – this is probably the biggest omission in this book
- strings (since they are a form of pointers)
- *typedef* 's for giving the constructed type a name
- the *const* modifier

[26] Actually "memory address" means "pointer", but we promissed not to do pointers in this book ☺.

6 Functions

Most languages allow one to create *functions* (subroutines, procedures) of some sort. Functions let us chop up a long program into named sections so that the sections can be reused throughout the program. It is even possible to group popular functions into a separate C file(s) and convert them into a reusable library. This is actually what has been done for e.g. *printf* and *scanf* !

6.1 Divide and conquer

One of the most important aspects of a function is the fact that it chops-off a part of the main program. This is an instance of the "divide and conquer" paradigm of solving big problems.

Let's start with an example of a (overly) simple program.

```
#include <stdio.h>

int main( void )
{
  // Intro
  printf("Welcome to my fabulous program HELLOWORLD\n");
  printf("Version 1.0 by Maarten Pennings\n”);

  // Here starts the real content of my fabulous program
  printf("Hello, world!\n");

  // Done with the real content of my fabulous program
  printf("HELLOWORLD says bye now...\n");

  return 0;
}
```

Next, let's "chop-off the introduction into a named section".

```
#include <stdio.h>

void PrintIntro( void )
{
  printf("Welcome to my fabulous program HELLOWORLD \n");
  printf("Version 1.0 by Maarten Pennings\n”);
}

int main( void )
{
  PrintIntro();

  // Here starts the real content of my fabulous program
  printf("Hello, world!\n");

  // Done with the real content of my fabulous program
  printf("HELLOWORLD says bye now...\n");

  return 0;
}
```

We have now created a *function* with the name *PrintIntro*. It is called from *main* via the function call statement *PrintIntro( );*. Note that the parenthesis in a function call are mandatory. A function header (the function name, its parameters and the types) together with its body are called a *function definition*.

In this example, the function *PrintIntro* accepts no parameters (that's what the "*(void)*" after *PrintIntro* is specifying), nor does it return any result (that's what the *void* before *PrintIntro* is specifying). In general, C functions can accept an unlimited number of parameters, and they can return a value of practically any kind. The

(types) of its parameters, the type of its result and its name, are together called the *signature* of a function (and the signature is defined by the function header).

Note that there is no ; after the header in the first line. If we accidentally put one in, we will get a huge cascade of error messages from the compiler that make no sense. Also note that there is no semicolon at the end of the body.

By now, the mystery of *main* should have been resolved. It's just a function, like any other. The only special thing about it, is that the operating system (the loader) will call it, so its parameters, result and name (in short, its signature) are fixed – otherwise the loader wouldn't know what to call, what to pass nor what to do with the answer.

For illustrative purposes, let's create another function for the exit message.

```
#include <stdio.h>

void PrintIntro( void )
{
  printf("Welcome to my fabulous program HELLOWORLD \n");
  printf("Version 1.0 by Maarten Pennings\n”);
}

void PrintExit( void )
{
  printf("HELLOWORLD says bye now...\n");
}

int main( void )
{
  PrintIntro();
  printf("Hello, world!\n");
  PrintExit();
  return 0;
}
```

Many people believe that the above order of the functions is logical. C doesn't care in what order we put our functions in the program, as long as the function signature is known to the compiler before it is called. So

```
#include <stdio.h>

 void PrintExit( void )
{ … }

void PrintIntro( void )
{ … }

int main( void )
{
  PrintIntro();
  printf("Hello, world!\n");
  PrintExit();
  return 0;
}
```

is also ok, but

```
#include <stdio.h>

void PrintExit( void )
{ … }

int main( void )
{
  PrintIntro();     // C compiler will give an error that it doesn’t know PrintIntro
  printf("Hello, world!\n");
  PrintExit();
  return 0;
}

void PrintIntro( void )
{ … }
```

is *not* ok! The *main* function is calling *PrintIntro*, before the definition of *PrintIntro* is given (this problem can be fixed though, read on!).

6.2 Returning results

Let's make a function that's a little bit more complex, one that returns a value. It's hard to come up with a meaningful function with that signature. But there is one. The standard function *rand* returns a pseudo random number.[27]

```
int rand_seed= 10;

// from K&R: produces a random number between 0 and 32767.
int rand( void )
{
    rand_seed = rand_seed * 1103515245 +12345;
    return (unsigned int)(rand_seed / 65536) % 32768;
}
```

The *int rand(void)* line declares the function *rand* to the rest of the program. It specifies that *rand* will accept no parameters and returns an integer result. Note that even though there are no parameters, we must use the *(void)*. They tell the compiler that we are declaring a function rather than simply declaring an *int*.

The *return* statement is important to any function that returns a result.[28] It specifies the *value* that the function will return. By the way, the return statement also causes the function to exit immediately. This means that we can place multiple *return* statements in the function to give it multiple exit points.[29] If we do not place a return statement in a function, the function returns when it reaches } and returns some spooky value (many compilers will warn us if we fail to return a specific value). In C, a function can return values of nearly any type: *int*, *float*, *char*, *struct*, etc.

There are several correct ways to call the *rand* function. For example: *x= rand( );*. The variable *x* is assigned the value returned by *rand* in this statement. Note that we must use *( )* in the function call, even though no parameter is passed.[30]

We might also call *rand* this way:

```
if( rand()>100 )
```

Or this way:

```
rand();
```

In the latter case, the function is called but the value returned by *rand* is discarded. We probably never want to do this with *rand*, but many functions return some kind of error code, and if we are not concerned with the error code (for example, because we know that an error is impossible) we can discard it in this way. This is not theoretical: the *printf* statement returns the number of characters printed or a negative value if an output error occurs.

As we saw above, functions can use a *void* return type if we intend to return nothing.

---

[27] The *rand()* function uses a (global) variable *rand_seed*. It is an integer, and it is initialized with 10. Global variables are usable in *any* function body that occurs later in the file. Global variables are condemned in a lot of coding conventions; local variables (i.e. variables defined in the body of a function and only accessible by that function) are by far preferred. However, local variables "die" when the function ends, and that is not what we want in this case (otherwise *rand()* would return the same "random number" each time).

[28] In this example, the return expression has an operator that is outside the scope of this book, namely a typecast operator, written as *(unsigned int)*.

[29] A function with multiple exit points is also condemned by some developers as bad style.

[30] Otherwise, *x* is given the *memory address* of the *rand* function, which is generally not what you intended. Memory addresses is a pointers topic, and hence not part of this book.

6.3 Passing parameters

C functions can accept parameters of any type. For example, the function *fac* defined as

```
// Returns factorial of n (n>=0)
int fac( int n )
{
  int f;
  int i;

  f= 1;
  for( i=2; i<=n; i++ )
    f= f*i;
  return f;
}
```

returns the factorial of *n*, which is passed in as a single integer parameter.[31]

To pass multiple parameters, separate them with commas:

```
int add( int a, int b )
{
  return a+b;
}
```

Given these two functions, we could now type

```
int main( void )
{
  int i;
  int j;
  i= fac(5);
  j= add(i,3);
  j= j * fac( add(1+2,3)+4 );
  return j;
}
```

not that this is a useful program…

6.4 Passing arrays

Actually, the type of the parameters maybe any type construction. In this section we restrict ourselves to *array* parameters. A useful feature is that we may leave out the size of the array – in the declaration of the parameter.

```
int sum( int row[] )
{
  return …
}
```

The above function *sum* may now be called with an integer array with 5 items, but also with an integer array of 5000 items. However, the function has one big flaw: how is it supposed to know the size of the array? There is no magic in C, so we have to fix that ourselves:

```
int sum( int size, int row[] )
{
  int s;
  int i;
  s=0;
  for( i=0; i<size; i++ )
    s= s + row[i];
  return s;
}
```

Given this functions, we could now type

[31] Recall that the factorial of 5 (mathematically written as 5!) is 5×4×3×2×1 and, for example, 3!= 3×2×1.

```
int main( void )
{
  int row1[5];
  int row2[5000];
  … // filling row1 and row2
  printf( "sum[row1]=%d\n", sum(5,row1) );
  printf( "sum[row2]=%d\n", sum(5000,row2) );
  printf( "sum[row2[0..99]]=%d\n", sum(100,row2) );
  return j;
}
```

Note the 3rd *printf*.

6.5 Scope

The *scope* of a variable describes where in a program's text a variable may be used. Scope is a syntactical aspect of a variable. The scope of a variable is the portion of the program code for which the variable's name has meaning and for which the variable is said to be visible. Entrance into that scope typically begins a variable's lifetime and exit from that scope typically ends its lifetime. C has two notions of scope: global variables, and variables in a block (which includes functions, since their body is a block). A global variable may be referred to anywhere in the program, a block variable may only be referred to in that block. It is erroneous to refer to a variable where it is out of scope.

```
int v1;                 // v1 is global, can be used anywhere in f and in main
int f( int v2 )         // v2 is local to f, can be used anywhere (and only) in f
{
  int v3= v2*v2;        // v3 is local to f, can be used anywhere (and only) in f
  if( v1<v3 )
  {
    int v4=v3-v1;       // v4 is local to this then-part, can be only in then-part
    return v4;
  }
  else
  {
    return v1;
  }
}

int main( void )
{
  int v5= 5;            // v5 is local to main, can be used anywhere (and only) in main
  v1= 1;
  return f( v5+v1 )
}
```

Variables in an inner scope, hide variables from the outer scope if they have the same name. In the fragment below, there is a global integer *x*. Since it is global, *main* can use it (assign 5 to it, pass it to *f* etc). Similarly, since *x* is global, *f* could also use it. However *f* has a local *x* (its parameter) which hides the global *x* (makes the global x inaccesible).

```
int x;              // global x

int f( int x )      // introduces a local x at the scope of function f
{
  x= x+3;           // local x (of f)
  return x*2;       // local x (of f)
}

int main( void )
{
  int y;
  x= 5;             // sets global x to 5
  y= f( 3 );
  printf("%d\n",x); // global x
  y= f( x );        // global x
  printf("%d\n",x); // global x
}
```

The above program prints two times a 5, since variable *x* of *main* is not changed. Even in the latter call to *f*, the *x* is not changed. This mechanism of passing a parameter (that can not be / is not changed) is known as *call by value*.

There are mechanisms in C to have the value of variable *x* changed by function *f*. To achieve this, *x* should not be passed by value, rather x should be *passed by reference*. However, in C passing by reference requires in-depth knowledge of pointers, which is outside the scope of this book.[32]

6.6 Function prototypes

Many coding conventions consider it good form to use function prototypes for all functions in a program. A *prototype* declares the function signature, i.e. its name, its parameters, and its return type to the rest of the program prior to the function's actual definition. A prototype is a header followed by a semicolon (;) instead of a body.

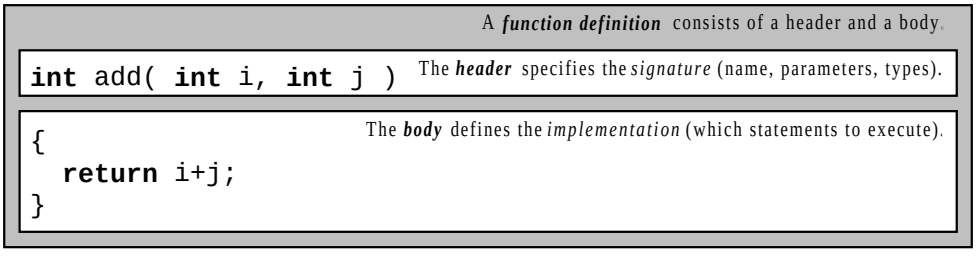
As we saw above, the following program will give compiler warnings (unfortunately, it will not give us an error that the call to *add* does not have enough parameters!).

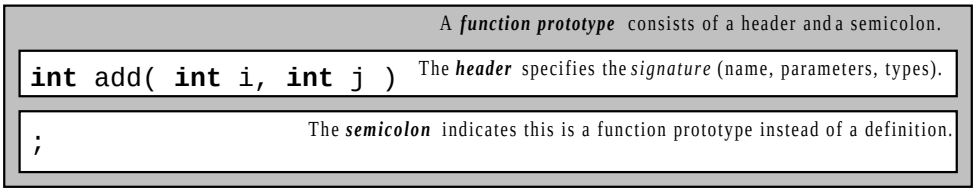
```
#include <stdio.h>

int main( void )
{
  printf("%d\n",add(3)); // missing second parameter is not trapped
  return 0;
}

int add( int i, int j )
{
  return i+j;
}
```

One way to solve this is to exchange the place of *main* and *add*. Another way to solve this problem is prototypes. C lets us place function prototypes at the beginning of (actually, anywhere in) a program. If we do so, C checks the types and counts of all parameter lists. Try compiling the following:

```
#include <stdio.h>

int add( int i,int j );  // the prototype (note presence of semicolon and absence of {})

int main( void )
{
  printf("%d\n",add(3)); // due to earlier prototype, we now get a warning
  return 0;
}

int add( int i, int j )
{
  return i+j;
}
```

The prototype causes the com piler to flag an error on the erroneous call to *add* in the *printf* statement.





[32] But, the second parameter of *scanf("%d",&i)* is passed by reference!

7

7 Old examiniations

Since these are copies of real examinations, they're in Dutch.

7.1 Trial for 2005

Hogeschool Eindhoven
Studierichting Hogere Informatica

Vak	: SPR3/PRCE
Docent	: Maarten Pennings, Agnes Veugen
Tijdstijp	: Dit is een proeftentamen voor PRCE samengesteld uit bestaande PRC tentamens

Opgave 1. Expressies (1 punten per vraag; totaal 5) *her2002 (verkort)*

Geef voor elk van de volgende statements wat de output zal zijn.

1a) `printf( "%d", 0x30 &  0x11 );`
1b) `printf( "%d", 0x30 && 0x11 );`
1c) `printf( "%d", 0x03 << 0x02 );`
1d) `int i=10; printf( "%d", i=i+1 );`
1e) `printf( "%d", !23 );`

Opgave 3. Structs en functies (20 punten 2, 4, 4, 5, 5) *her2002 (aangepast)*

We gaan in deze opgave werken met datums. Een datum representeren we met een struct met drie integers, voor jaar, maand en dag.

3a) Geef de **`struct`** voor `Datum`.
3b) Schrijf een functie

```
int DatumKerstmis( struct Datum d )
```

waarbij de functie 1 oplevert als datum d op 25 december valt, en 0 als d niet op 25 december valt.

3c) Schrijf een functie

```
void DatumPrint( struct Datum d )
```

die een datum print door eerst de dag te printen (1..31), dan een minnetje, dan de maand (1..12) ook weer gevolgd door een minnetje en tenslotte het jaar (bijvoorbeeld 31-12-1999).

3e) Schrijf een functie

```
int DatumVergelijk( struct Datum d1, struct Datum  d2 )
```

die –1 retourneert als `d1` kleiner is (vroeger is in de tijd) dan `d2`, 0 retourneert als `d1` gelijk is aan `d2` en +1 retourneert als `d1` groter is (later is in de tijd) dan `d2`. Voorbeelden: 30-12-1999 is groter dan 29-12-1999 en groter dan 31-10-1999, maar kleiner dan 1-1-2000.

Tip: kijk eerst of het jaar van `d1` kleiner is dan het jaar van `d2` (zoja, return –1) of dat het jaar van `d1` groter is dan het jaar van `d2` (zoja, return +1); ga dan pas de maanden (net zo) en de dagen vergelijken.

Opgave 4. TelZe her2003 (aangepast)

Schrijf een functie, *TelZe* genaamd, dat met precies twee argumenten aangeroepen moet worden:

```
int TelZe( char s1[], char s2[] )
```

De functie bepaalt het aantal keer dat in beide argumenten op overeenkomstige plaatsen hetzelfde karakter staat.

Voorbeeld:
Aanroep: `TelZe( "peter", "patse" )`
Output: 2 (op de 1e plaats staat in beide argumenten een p en op de 3e plaats in beide een t.)

Nota bene: de C compiler zorgt ervoor dat er altijd een karakter extra in het karakter array staat (bij gebruik van de "…" notatie), namelijk een ascii waarde 0 aan het einde!

Opgave 4. Breuken (8 punten per vraag; totaal 32) *ten2002 (aangepast)*

We gaan in deze opgave werken met breuken. Breuken kunnen we representeren met een (integer) teller en een (integer) noemer. We definieren daarom het volgende struct om in C met breuken te werken.

```
typedef struct Breuk
{
    int teller;
    int noemer;
};
```

4a) Schrijf een functie

```
struct Breuk BreukLees( void )
```

die twee integers inleest en deze opslaat in een Breuk die geretourneerd wordt.

4b) Schrijf een functie

```
void BreukPrint( struct Breuk a )
```

die een breuk print door eerst de teller te printen, dan een slash, en dan de noemer (bijvoorbeeld 25/200).

4c) We brengen in herinnering dat het produkt van de breuken t_a / n_a en t_b / n_b gelijk is aan $(t_a \times t_b) / (n_a \times n_b)$.
Voorbeeld: 2/3 maal 5/7 is 10/21.

Schrijf een functie

```
struct Breuk BreukMaal( struct Breuk b1, struct Breuk b2 )
```

die het produkt van *b1* en *b2* berekent en dit retorneert.

4d) Schrijf de *main( )* functie die eerst twee breuken inleest (met behulp van *BreukLees( )*), ze daarna vermenigvuldigt (met behulp van *BreukMaal( )*) en die tenslotte het hele sommetje print (met behulp van *BreukPrint( )* en wat *printf's*).
Voorbeeld: het vermenigvuldigingssommetje van 2/3 en 5/7 wordt alsvolgt uitgevoerd.

```
2/3 x 5/7 = 10/21
```

(einde van het tentamen)

7.2 Real examination 2005

Hogeschool Eindhoven
Studierichting Hogere Informatica
deeltijd

Vak	: SPR3/PRCE tentamen
Docent	: M.Pennings, A. veugen
Datum	: 10 april 2006
tijd	: 18.00-19.40 uur
Hulpmiddelen	: dictaat, boeken, aantekeningen
Normering:	: 20(opg1) + 5(opg2) + 10(opg3) + 15(opg4) + 25(opg5) + 15(opg6) + 10(bonus)

Waarschuwing: dit tentamen is alleen voor *electro* studenten.
Werk netjes: = schrijven in plaats van = = , puntkomma's of accolades vergeten etc. kost punten.

Opgave 1 Expressies (20 punten, 2 per deel-opgave)

Geef voor elk van de fragmenten aan wat de output zal zijn (op een PC).

1a. `printf("%d", 0x5A | 0x3C );`
1b. `printf("%d", 0x5A || 0x3C );`
1c. `printf("%d", 6 << 2 );`
1d. `printf("%d", 6 <= 2 );`
1e. `printf("%d", 6 == 2 );`
1f. `printf("%d", !6 );`
1g. `printf("%d", 27 / 4 );`
1h. `printf("%d", 27 % 4 );`
1i. `printf("%d", 'A' – 'C' );`
1j. `printf("%d", sizeof(char) );`

Opgave 2 Pre- and post increment (5 punten)

2. Wat is de output van het volgende fragment.

```
int i=4; printf("%d", ++i ); printf("%d",i); printf("%d",i++);
```

Opgave 3 Printf (10 punten, 2 per deel-opgave)

Geef voor elk van de fragmenten aan wat de output zal zijn (hint: de ASCII waarde van 'j' is 106_{10} of $6A_{16}$).

3a. `printf("%d", 106 );`
3b. `printf("%5d", 106 );`
3c. `printf("%x", 106);`
3d. `printf("%X", 106);`
3e. `printf("%c", 106);`

Opgave 4 Theorie (15 punten, 3 per deel-opgave)

4a. Wat is *linken*?
4b. Wat is *padding*?
4c. Wat is *indentatie*?
4d. Wat is een *escape character*?
4e. Wat is een *prototype*?

Opgave 5. Priemgetallen (25 punten, 10+5+10)

We brengen in herinnering dat de C-expressie *a % b* de rest-bij-deling van *a* door *b* oplevert. Dat betekent dus dat als *a* % *b* gelijk is aan 0, dat er geen rest is, ofwel dat *b* een deler van *a* is.

5a. Schrijf een functie

```
int AantalDelers( int a )
```

die het aantal delers van *a* retourneert. De preconditie is $a \geq 1$ (dwz dat de functie alleen hoeft te werken voor $a \geq 1$). Hint: gebruik een *for*-lus in deze functie.

Voorbeeld: `AantalDelers(6)` retourneert 4, omdat 1, 2, 3, en 6 de vier delers van 6 zijn.

5b. Als een getal precies twee delers heeft, dan noemen we dat getal een *priemgetal*.
Schrijf een functie (wederom met preconditie $a \geq 1$)

```
int IsPriem( int a )
```

die *1* retourneert als *a* een priemgetal is, en die 0 retourneert als *a* geen priemgetal is.

Het is de bedoeling dat *IsPriem( )* gebruik maakt van *AantalDelers( )*, maar geen *if*, *while*, of *for* gebruikt.

5c. Schrijf de *main( )* functie die een integer *a* inleest, *IsPriem()* gebruikt, en een van de volgende teksten afdrukt als uitvoer:

- `Invoer niet correct (moet positief zijn)` als $a \leq 0$,
- `Een priemgetal` als *a* groter dan 0 en priem is,
- `Geen priemgetal` als *a* groter dan 0 en niet priem is.

Opgave 6 Structures (15 punten)

Er zijn veel toepassingen waarbij een grote serie getallen opgeslagen moet worden (temperatuur meetwaardes van een weersstation, pixels in een plaatje, etc.). Vanzelfsprekend komt daar de vraag dat zuinig te doen. Een van de technieken is run-length-encoding (RLE). RLE is gebaseerd op de hoop dat opeenvolgende getallen gelijk zijn; niet alleen de waardes worden opgeslagen maar ook het aantal keer dat die waarde achter elkaar staat. De reeks

13, 13, 13, 13, 13, 14, 14, 15, 15, 15, 13, 12, 12, 12, 12, 10, 10, 10, 13, 13, 13

wordt alsvolgt RLE gecodeerd

13 (5x), 14 (2x), 15 (3x), 13 (1x), 12 (4x), 10 (3x), 13 (3x)

De 21 oorspronkelijke getallen zijn dan in 7 paren (14 getallen) gecodeerd.

Om een element uit deze RLE rij op te slaan maken we gebruik van het volgende structure

```
struct Paar { int waarde; int aantal; }
```

Een ***struct*** *Paar e;* met *e.waarde=13* en *e.aantal=5* codeert het eerste element “13 (5x)” uit onze voorbeeld rij. De hele rij is natuurlijk een array van die elementen.We gebruiken bovendien de truc dat een speciaal element het einde van het array aangeeft; dat is een element dat zijn *aantal* op 0 heeft staan.

Kortom, onze voorbeeld rij zou alsvolgt RLE gecodeerd zijn.

```
struct Paar rij[] = {{13,5}, {14,2}, {15,3}, {13,1}, {12,4}, {10,3}, {13,3},
{9999,0} };
```

6. Schrijf een functie *PrintVoluit()* die een RLE gecodeerde rij voluit afdrukt. In het volgende programma

```
#include <stdio.h>

struct Paar { int waarde; int aantal; };

void PrintVoluit( struct Paar r[] )
{
  …
}

int main( void )
{
  struct Paar rij[] = {{13,5},{14,2},{15,3},{13,1},{12,4},{10,3},{13,3},{9999,0} };
  PrintVoluit(rij);
  return 0;
}
```

zou de uitvoer zijn

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

(einde van het tentamen)

8

8 Exercises

This chapter contains a section with exercises per week. It starts with an introduction on using developer studio.

As a mindset, implement the assignment in each execise as if it where a spec from another company that is paying us. Don't do too little, don't do too much.

> If the exercise says "the output should be a table with the numbers nicely aligned", alignment is part of the assignment. It is not allowed to skip that part.

> If the exercise asks to make a program printing a table with the powers for 3, don't write a "better" program that asks for a number *n* and then prints a table with the powers of *n*.

8.0 Microsoft Developer Studio

In the exercises, we use Microsoft Developer Studio.

Using developer studio is quite complex the first time. It has a notion of a *project*. In essence, a project is a list of source files (in our projects that will be a list containing just one source file) that all need to be compiled and linked together in order to create the executable. A project is what gets built and without a project, developer studio will build nothing.. In addition to projects, developer studio has the notion of a *workspace*. A workspace is a set of projects. This is to accommodate large development activities where a single application consists of several executables working together. The workspace (the set of projects) is the unit of work in developer studio. For example, there is a window showing all files of all projects so that any file can be changed easily.

8.0.1 Advise on setting up files, directories, workspaces and projects

We will now give an advise how to set up files, directories, workspaces and projects for solving the exercises using Microsoft developer studio. These are the steps:

- Step 1: Create on a network drive (not on the harddisk of the PC we're working on) a directory for PRC. We would make that a subdirectory of the directory for SPR. We believe it is wise not to use capitals in files when working cross file systems and operating systems (the network drive might be a Unix file system). Fire up Explorer and create a directory structure like the following[33]

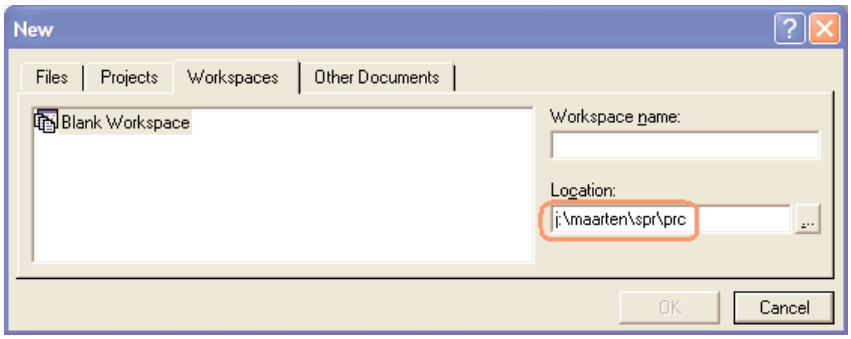
  ```
  J:\maarten\spr\prc
  ```

  Step 1 is a one-time action.

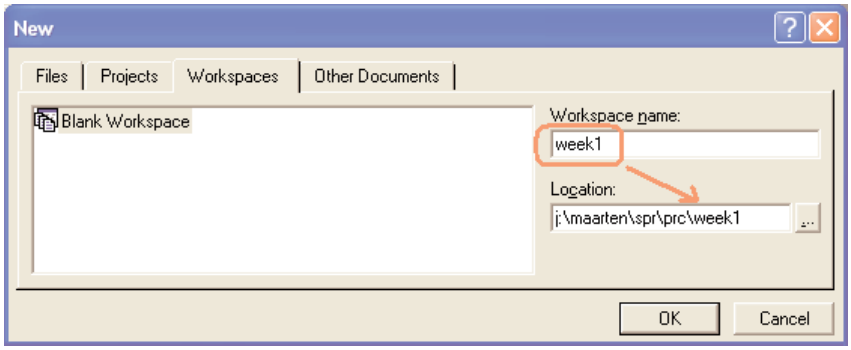
- Step 2: We would make a workspace per week[34]. Developer studio creates a directory per workspace. Fire up developer studio, chose *File | New* and then tab *Workspaces*. In the *Location* editbox enter the just created directory (or use the "…" button to browse).

[33] This is an example. Maybe the drive letter is different for you, maybe there are additional top-level directories (*h:\home\studs\maarten\spr\prc*).

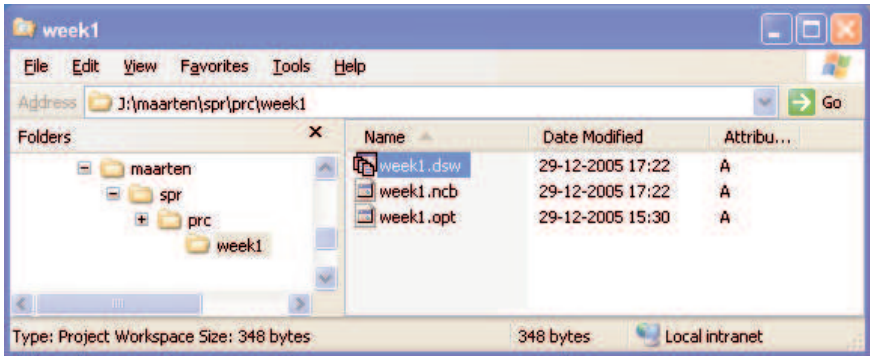
[34] This is a matter of taste. We could make one workspace for all exercises (but then the list of exercises would be long), or we could make one workspace per exercises (but that would mean many extra (workspace) files). A workspace per week is also easy for the teacher when a week is ready for review.



Next, type a name in the *Workspace name* editbox. We suggest *week1*. Observe that developer studio automagically extents the *location*!



Press *Ok* to create the blank workspace for *week1*.
Let's check the directory stucture and files created by developer studio in Explorer.



We see that indeed, developer studio has created a directory *week1* in *J:\maarten\spr\prc*. In that directory we see "administrative files" that developer studio has created to administer our *week1* workspace. Next time we want to work on the *week1* workspace, it suffices to *File* | *Open* file *week1.dsw* (dsw = developer studio workspace) or even double click *week1.dsw* in Explorer.

Every week, there is a new set of exercises that we will group in a workspace. Hence, step 2 needs to be executed every week.

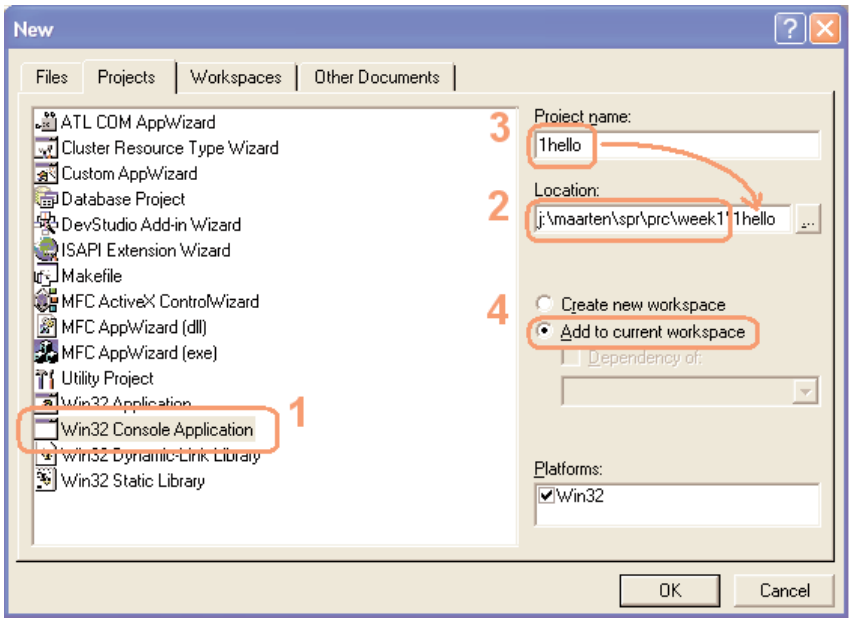
- Step 3: We make a project per exercise. A project is added to the workspace of the week it belongs to. So, make sure we have the right workspace open (as created in step 2) and chose *File* | *New* and then tab *Projects*.[35]

It is important to select *Win32 Console Application*.[36]

[35] As an alternative, right click *week1* in the workspace *window*, and select *Add new Project to Workspace*.

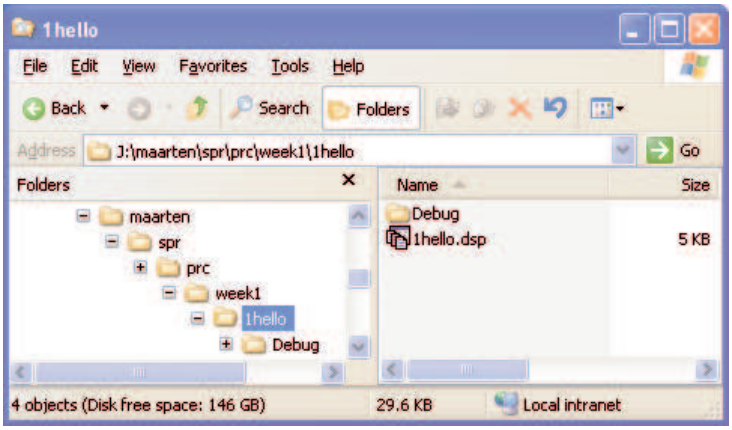
[36] This determines the kind of executable the project builds. We are *not* going to write applications that pop-up windows, rather we make an old fashioned text ("command-line") oriented application. Failing to select the right project kind will result in errors during the build.

In the *Location* editbox enter the directory of the workspace (or use the "…" button to browse). Next, type a name in the *Project name* editbox. We suggest to start with the exercise number (so that exercises are sorted in a convenient order) followed by a descriptive name (so that we as humans recall what it does). So, for example, *1hello*. Observe that developer studio once again extents the *location*!



Make sure the *Add to current workspace* is selected before creating the project by hitting *Ok*.
In the Wizard that pops up, just select *An empty project* and click *Finish*.

Let's check the directory structure and files created by developer studio in Explorer.



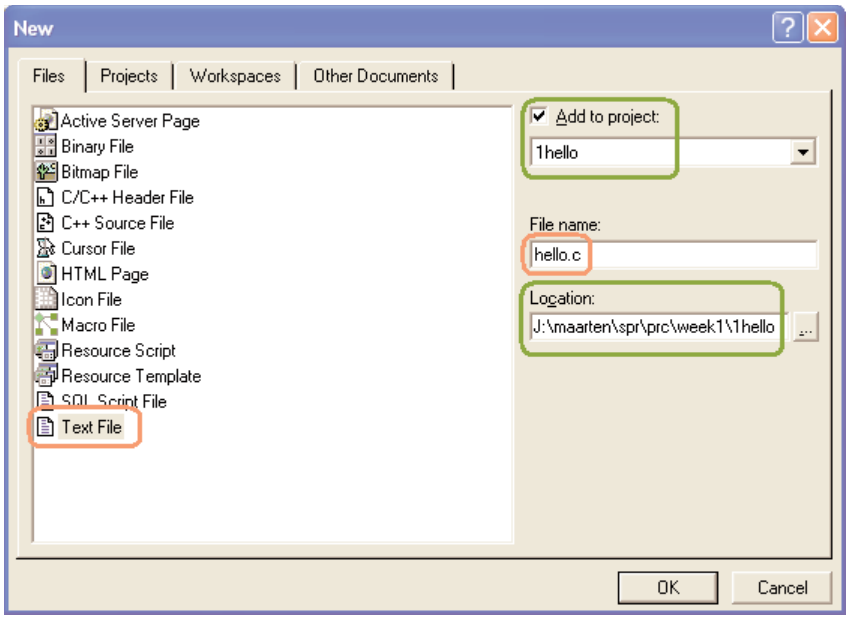
We see that developer studio has created a directory *1hello* in *J:\maarten\spr\prc\week1*. In that directory we see an "administrative file" that developer studio has created to administer our *1hello* project. It has also created a *Debug* directory that will be filled with temporary files as soon as we build this project.

Every exercise must have its own project (otherwise it can't be built), so step 3 must be repeated for every exercise.

- Step 4: We add a c source file to a project. So, make sure we have the right project open (as created in step 3) and chose *File | New* and then tab *Files*.

  Check the *Location* box (it should be ok), and check that the new file will be added to the right project (*Add to project*). As file type select Text File, and in the *File name* box enter the name of the c source file.

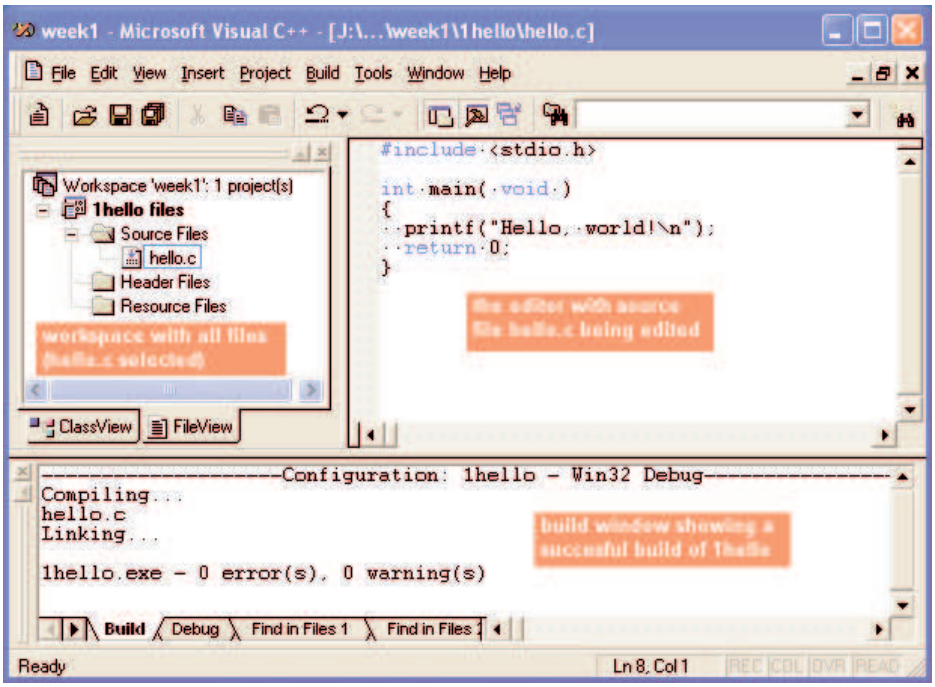
It is important to enter *.c* as extension. [37]



Finally, hit Ok.

8.0.2 The edit-compile-link-execute cycle in developer studio

Usually the main screen of developer studio is split in three parts: the workspace (top left), the editor (top-right) and the (build) log (bottom), as shown in the figure below. If one of those parts has disappeared, use the *View* menu to get them back.



The figure shows we have a workspace (*week1*), with a project (*1hello*), with a file (*hello.c*).

[37] Developer studio supports more programming languages then just C, most notably C++. Failing to use the .c extension might result in developer studio using the C++ compiler instead of the C compiler, which might lead to unexpected behavior.

- We can now edit any file. Select it in the workspace (top-left) and type in the top-right window.
- We can then start the built (via *Build* | *Build* or F7).
- If there are no build errors, we can execute *1hello.exe* via *Build* | *Execute* (or ctrl-F5).

  When simply running a program, use ctrl-F5 and not F5. The latter command pops up a console window, runs the program, and when the program terminates, the console window is closed immediately. The former command also pops up a console and runs the program, but when the program terminates it asks "Press any key to continue". Only after pressing any key, the console window is closed, given us time to inspect the output generated by our program in the console window.

8.1 Exercises for week 1 – Program

8.1.1 *1hello*: Hello, world!

Read the section on setting up files, directories, workspaces and projects in 8.0. Create a *week1* workspace, a *1hello* project and a file *hello.c* with the famous "Hello, world!" program (see Section 1.5). Type the c code in, save all (files, project, workspace), build the program an execute it.

8.1.2 *2errors*: Bugs in the program

Add a second project *2errors* to *week1*. Add one c file to that project (*errors.c*) and copy the following program:

```
#include <stdio.h>

int main( void )
{
  printf("A line 1/n")
  printf("Another line\n");
  printf('Last line');
  return;
}
```

This program has three syntax errors. Build the program and see how the compiler reports them. Double click them to open the offending line in the c file. Fix them.

The program also has a semantic error. Which one?

8.1.3 *3random*: Random errors

Add a third project *3random* to *week1*. Add one c file to that project (*random.c*) and copy and paste the contents of the first program (*hello.c*). Make each of the following errors by itself and then run the program through the compiler to see what happens.

- Delete the first line (#include) of the above program and see what the compiler does when we forget to include *stdio*.
- Delete a semicolon ';' and see what happens.
- Leave out one of the braces '{' or '}'.
- Remove one of the parenthesis '(' or ')' next to the *main* function.
- Change *main* in *Main*.
- Change *printf* in *Printf*.
- Delete random characters or words.

- Add random characters or words.

By simulating errors like these, we can learn about different compiler errors, and that will make our typos easier to find when we make them for real.

Also try pressing F1, either with the cursor on some item in the editor (else, prinf), or with the cursor on the number of a compiler editor in the build log window.

8.2 Exercises for week 2 – Intermezzo on input and output

Start with creating a *week2* workspace (in addition to the *week1* workspace).

8.2.1 *1pow3*: Powers of three

Add a project *1pow3* to *week2*. Add one c file (*pow3.c*). It should look something like this.

```
#include <stdio.h>

int main( void )
{
  …
  printf("%d^%d=%d", 3, 0, 1 );
  printf("%d^%d=%d", 3, 1, 3 );
  printf("%d^%d=%d", 3, 2, 3*3 );
  printf("%d^%d=%d", 3, 3, 3*3*3 );
  …
}
```

but the output should be as follows (note that the answers are *aligned*)

```
The powers of 3
3^0=    1
3^1=    3
3^2=    9
3^3=   27
3^4=   81
3^5=  243
3^6=  729
3^7= 2187
3^8= 6561
3^9=19683
```

We must achieve that by modifying the format strings of *printf* (adding padding instructions to the placeholders and adding escape sequences).

8.2.2 *2calc*: Simple calculator

Add a project (*2calc*) to *week2*, and create a c file (*calc.c*) that implements a simple calculator. The user should be able to enter two (floating point) numbers, and the program should print the product (*) and the quotient (/).

8.2.3 *3powb*: Powers of b

We are going to improve the “powers of three” program: we make it more flexible by using a variable. Add a project *3powb* to *week2*. Add one c file (*powb.c*). It should look something like this.

```
#include <stdio.h>

int main( void )
{
  …
  printf("%d^%d=%d", b, 0, 1 );
  printf("%d^%d=%d", b, 1, b );
  printf("%d^%d=%d", b, 2, b*b );
  …
}
```

and it should have a declaration of *b* and a *scanf* for *b*. When the user runs the program and enters 3 for *b*, the output of this program should be equal to the output of the original “Powers of three” program (including alignment).

8.2.4 *4printf*: Printf errors

Add one more project (*4printf*) to *week2*; we are going to investigate what happens if *printf* gets the wrong values.

Let the *main* function contain these lines:

```
// too many or too few numbers
printf("1=%d, 2=%d\n", 20, 30, 40 );
printf("1=%d, 2=%d\n", 20 );

// too many or too few strings
printf("1=%s, 2=%s\n", "ape", "nut", "mary" );
printf("1=%s, 2=%s\n", "ape" );

// mixing numbers and strings
printf("1=%d, 2=%d\n", 3, "ape" );
printf("1=%s, 2=%s\n", "ape", 3 );
```

How many errors do we get during compilation? Why?

How many errors do we get when running? Why?

8.3 Exercises for week 3 – Expressions

8.3.1 *1check*: Expressions

First, fill out the middle column of the following table

Expression	Human answer	Computer answer
17 & 22		
17 | 22		
13 & 15		
13 | 15		
13 && 15		
0xA ^ 0xC		
20 % 2		
21 % 2		
127/10		
127.0/10.0		
9 << 2		
3<=5		
3!=5		
!(3<5)		
5++		

Next, create a *week3* workspace, and a *1check* project with a file *check.c*. The code should check the expressions.

```
#include <stdio.h>

int main( void )
{
  printf( "%d & %d = %d\n", 17, 22, 17 & 22 );
  … all others too …
  return 0;
}
```

Enter, compile/link and execute this. Fill out the last column of the previous exercise and explain any differences.

8.3.2 *2powbp*: powers of b improved

Add a project *2powbp*. Add one c file (*powbp.c*). Improve the "powers of b" program by adding a variable *p* and ten assignments to *p* interspersed with the *printf* 's. The goal is to only execute a total of 9 multiplications.

Hint

```
int main( void )
{
  …
  printf("%d^%d=%d", b, 0, p );
  p= p*b;
  printf("%d^%d=%d", b, 1, p );
  …
}
```

8.3.3 *3abc*: abc formula

Let's do some mathematics. Recall that a quadratic equation

$ax^2+bx+c = 0$

has two solutions

$$x_1 = \frac{-b-\sqrt{b^2-4ac}}{2a}$$

$$x_2 = \frac{-b+\sqrt{b^2-4ac}}{2a}$$

That is, if we have as equation

$2x^2-6x+4 = 0$

then

$a=2,\ b=-6,\ c=4$

so that we have as solutions

$$x_1 = \frac{--6-\sqrt{(-6)^2-4\cdot 2\cdot 4}}{2\cdot 2} = \frac{+6-\sqrt{36-32}}{4} = \frac{6-\sqrt{4}}{4} = \frac{6-2}{4} = \frac{4}{4} = 1$$

$$x_2 = \frac{--6+\sqrt{(-6)^2-4\cdot 2\cdot 4}}{2\cdot 2} = \frac{+6+\sqrt{36-32}}{4} = \frac{6+\sqrt{4}}{4} = \frac{6+2}{4} = \frac{8}{4} = 2$$

Conclusion: our equation has two solutions for *x*, namely 1 and 2.

Let's check that

$2\cdot 1^2-6\cdot 1+4 = 2-6+4 = 0$

$2\cdot 2^2-6\cdot 2+4 = 8-12+4 = 0$

Write a program (project *3abc*) that solves the quadratic equation. Below we'll find a skeleton. Note the second line that includes the math library; we'll need it for the function *sqrt* that computes the square root of a *float*.

```
#include <stdio.h>
#include <math.h> // We need this for sqrt()

int main( void )
{
  float a;
  float b;
  float c;
  float x1;
  float x2;

  scanf( … a … );
  scanf( … b … );
  scanf( … b … );

  x1= … ;
  x2= … ;

  printf("The solutions of %fx^2+%fx+%f=0 are",a,b,c);
  printf("x1=%f", x1 );
  printf("x2=%f", x2 );
}
```

Try to solve

$2x^2-6x+4 = 0$

Next, try the following two

$x^2-9 = 0$

$x^2+9 = 0$

Why does the latter not work?

8.4 Exercises for week 4 – Statements

Create a *week4* workspace, and make projects for each of the exercises in this section.

8.4.1 *1abcd*: Improving abc formula

Create a project *1abcd*; it will be an improved version of the one of last week. As we may recall, the equation

$x^2+9 = 0$

caused a run-time error. The problem is "under" the square root.

$$x_{1,2} = \frac{-b \pm \sqrt{\overbrace{b^2 - 4ac}^{D}}}{2a}$$

The formula contains the square root of b^2-4ac. This is called the discriminant and usually abbreviated to D. The discriminant has the interesting property of being an indicator for how many solutions the quadratic equation has.

- When D is negative, we can not take the square root out of it, so there are no solutions.
- When D is positive, there are two solutions.
- When D is zero, the square root of D is zero, so the formula reduces to $x_{1,2}= -b/2a$, so both solution are equal.

Copy the abc program from last week, add a variable D that computes the discriminant and add *if* 's testing D so that the program prints something along the lines of the following three variants

```
The equation …x^2+…x+…=0 has 2 solution(s)
x1= …
x2= …
```

```
The equation …x^2+…x+…=0 has 1 solution(s)
x= …

The equation …x^2+…x+…=0 has 0 solution(s)
```

8.4.2 *2powfor*: Power of "for"

We have already written three versions of the program that prints the power-of table. We're now going to write the final one (*2powfor*):

- Keep the variables *b* and *p*, and let *b* be input (*scanf*).
- Use a *for* loop (so there should only be one * in the whole program).
- Using a *pow* function is forbidden.
- Add an extra variable *n* (input by the user via *scanf*) that denotes the number of rows to print.

8.4.3 *3updown*: Updown game

There is a simple "game" for which the *while* loop makes more sense than a *for* loop.

The rules of the game are as follows:

- pick a (positive integral) number
  - when it is even, half it
  - when it is odd, multiply it by three and add 1
- keep on doing this, unless it is 1, then the game stops.

So, when we start with 9, we get the following up/down sequence.

```
[9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

Write a program (*3updown*) that let's the user input one integer, and then prints out the up/down sequence as in the example above (including the comma's, spaces, brackets and a single linefeed).

By the way, can you find a number for which this program does *not* stop at 1?

8.4.4 *4updowntab*: updown table

Write a program (*4updowntab*) that prints a table of up/down sequences, started from 1 and ending at a number that the user may input (*scanf*). For example, when the users inputs 20, the following table should be printed.

```
[1]
[2, 1]
[3, 10, 5, 16, 8, 4, 2, 1]
[4, 2, 1]
[5, 16, 8, 4, 2, 1]
[6, 3, 10, 5, 16, 8, 4, 2, 1]
[7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
[8, 4, 2, 1]
[9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
[10, 5, 16, 8, 4, 2, 1]
[11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
[12, 6, 3, 10, 5, 16, 8, 4, 2, 1]
[13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
[14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
[15, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1]
[16, 8, 4, 2, 1]
[17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
[18, 9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
[19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
[20, 10, 5, 16, 8, 4, 2, 1]
```

8.5 Exercises for week 5 – Data types

Create a *week5* workspace.

8.5.1 *1toosmall*: Too small

Write a program (*1toosmall*) that reads (*scanf*) 10 integers and then prints those that are smaller than the last one (so we need an array to store them all).

For example, when the users inputs the following 10 numbers:

```
55  3  12  7  31  2  27  2  21  12
```

the last one is 12, so the output is

```
3  7  2  2
```

8.5.2 *2dice1*: Dice 1

Write a program (*2dice1*) that repeatedly reads (*scanf*) the number of dice[38] eyes until the stop command is given (the number 0). Next, the program prints a table of how often each of the numbers occurred.

So, on input 6, 1, 2, 3, 3, 1, 6, 4, 3, 1, 3, 6, 4, 0 the program prints

```
1 3
2 1
3 4
4 2
5 0
6 3
```

Note: only use one array

```
int dice[6];
```

8.5.3 *3dice2*: Dice 2

Write a program (*3dice2*) that repeatedly reads (*scanf*) two numbers (the number of eyes on two dices) until the stop command is given (either of the two numbers smaller than 1 or greater than 6). Next, the program prints a "graph" of how often each sum (2..12) occurred.

The output should look like below (where each # stands for 1 occurrence).

```
 2 ###
 3 #####
 4 #######
 5 #####
 6 ###########
 7 ###############
 8 #############
 9 ##########
10 #########
11 #####
12 ##
```

8.5.4 *4sortdate*: Sort dates

Write a program (*4sortdate*) that has an array of 5 dates (a month, and a day *struct*). The dates must be read (*scanf*), then sorted (oldest date first) and finally printed.

Hint: the sort algorithm from the theory can be used with one modfication: the comparisson for numbers

```
if( a[j]<a[p] )
```

should be changed in a comparisson for dates. When is one date smaller than the other? When the month of the one is smaller than the month of the other, or, in case the months are equal, when the day of the one is smaller than the day of the other.

[38] The Dutch word for "dice" is "dobbelsteen".

8.6 Exercises for week 6 – Functions

Create a *week6* workspace.

> It is forbidden to use global functions or global arrays for the sole purpose of passing parameters.

8.6.1 *1max*: Max

Write a function *max*, that has two integer parameters and returns the biggest of them. Write a program (*1max*) that reads three integers (in *main*) and then prints the biggest one using the just developed *max* function. Of course, the *main* function does not have any *if* itself either.

8.6.2 *2digits*: Number of digits

Write a function *NumDigits*, that takes a (positive) integer and returns the number of digits (in its decimal notation). So passing

```
8530
```

should result in 4 (hint keep on dividing by 10 until it is 0). Write a program (*2digits*) that keeps on reading integers and printing their number of digits until a non-positive integer is entered.

8.6.3 *3pos*: Pos

Write a function *int Pos(int v, int a[ ], int size)*, that searches an array *int a* (0..*size*-1) for an occurrence of *v*. If *v* occurs in *a*, its index should be returned, if *v* does not occur in *a*, *-1* should be returned.

Write a program (*3pos*) that has global array *nums* with an initializer, function *Pos* and *main*. Function *main* should let the user input a number and then search for that number in *nums* using *Pos*. The result should be printed.

As an example, assume the array is 0, 1, 4, 9, 16, 25, 36, 49, 64. When the user enters 25 the program prints 5, when the user enters 40, the program prints –1.

8.6.4 *4days*: Days

The following function returns the number of days since the "birth of Jezus", that is since January 1st 0001.

```
int NumDaysSince0001_01_01( int y, int m, int d )
{
  int n;
  if( m<3 ) { m+=12; y--; }
  n= d - 1 + (153*m+3)/5 - 92 - 306 +365*y + y/4 - y/100 + y/400;
  return n;
}
```

You must copy this function (*NumDaysSince0001_01_01*) unmodified to your program.

Write an extra function *NumDays* that does the same as *NumDaysSince0001_01_01*, but takes a *struct date* (as in 5.2) instead of three *int*s. Let *NumDays* call *NumDaysSince0001_01_01*.

Thirdly, write a function *DaysPassed* that takes two *struct date*'s, and returns the number of days between the second date and the first (this returns a negative number if the first date is later than the second).

In *main* have two date variables be entered by the user, and print the number of days between them. Call the program *4days*.

9 Mistakes

9.1 Mistakes in C

This chapter lists some common mistakes in C programs.

- Using the wrong character case – case matters in C, so we cannot type *Printf* or *PRINTF*. It must be *printf*.

```
Printf( "Ape" );                    printf( "Ape" );
```

- Forgetting to use the & in *scanf*.

```
scanf( "&d", i );                   scanf( "&d", &i );
```

- Type mismatch in actual parameter and the format in *printf* or *scanf*.

```
printf( "%s", 12 );                 printf( "%i", 12 )
```

- Too many or too few parameters following the format in *printf* or *scanf*.

```
printf( "%d and %d", a );           printf( "%d and %d", a,b )
```

- Forgetting to declare a variable before using it.

```
int Sum( int n )                    int Sum( int n )
{                                   {
                                      int i;
  int s= 0;                           int s= 0;
  for( i=0; i<n; i++)                 for( i=0; i<n; i++ )
    s+= i;                              s+= i;
}                                   }
```

- Putting = when we mean = = in an *if* or *while* statement.

```
if( x=0 )                           if( x==0 )
```

- Using *a* = = *b* on floats.

```
float a= …;                         float a=…;
float b- …;                         float b- …;
                                    float epsilon= 1E-9;
if( a==b )                          if( abs(a-b)<epsilon )
```

- Forgetting to increment the counter inside the while loop; this results in an *infinite loop* (the loop never ends, so the program never ends).

```
int i= 0;                           int i= 0;
int s= 0;                           int s= 0;
while( i<n )                        while( i<n )
{                                   {
  s+= i;                              s+= i;
}                                     i++
                                    }
```

- Accidentally putting a ; at the end of a *for* loop or if-statement so that the for statement has no effect.

```
for( x=0; x<10; x++ ) ;             for( x=0; x<10; x++ )
  printf("%d\n",x);                   printf("%d\n",x);
```

- Forgetting braces after *if*, *while* or *for*.

```
int i= 0;                          int i= 0;
int s= 0;                          int s= 0;
while( i<n )                       while( i<n )
  s+= i;                           {
  i++;                               s+= i;
                                     i++
                                   }
```

- C has no range checking, so if we index past the end of the array, it will not tell us about it. It will eventually crash or give us garbage data. The most common instance of this error is accessing an array *at* its upper bound.

```
int a[3];
int i= a[3];                       // only a[0], a[1] and a[2] exist
```

- A function call must include () even if no parameters are passed. For example, C will accept *x=sin;* , but the call will not work as expected. The memory address of the *sin* function will be placed into *x* instead. We must say *x=sin(3);*.
- Using the / operator with two integers will often produce an unexpected result (no remainder), so think about it whenever we use it.

9.2 Mistakes in Developer Studio

- Forgetting to select *Win32 Console Application*. This results in an error saying *_winamin_* does not exist.
- Forgetting to specify a .c extension. This results in studio chosing the .cpp extension (for c plus plus) which has an extended syntax with respect to c.

(end of book)