An Introduction to the C Programming Language

For students of Machine Oriented Programming

Erik Sintorn

DEPARTMENT OF SOME SUBJECT OR TECHNOLOGY

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2024
www.chalmers.se

Preface

This booklet is intended to be used for self-studies while taking the course *Machine Oriented Programming*. The text will give a quite brief introduction to the C programming language, with a focus on practical examples. The reader is assumed to have some experience with imperative programming and to have knowledge of basic computer technology.

Contents

1

Introduction

In this chapter, we will introduce the C programming language by examining a small first program. Before continuing, make sure you have installed the course's development tools and that you are comfortable compiling and running a C program on your host computer (not cross compiling a program for running on the simulator).

1.1 A short history of the C Programming Language

Translation of Roger's history.

1.2 A simple C program

C is a *procedural programming language* which means that the program's state can be changed by executing some *procedure*. A procedure, in this context, is simply a *subroutine* or, as they are called in C, a *function*. Specifically, every C program[1] must contain exactly *one* function called `main`, where the program starts.

To introduce the structure of a C program, we will start by looking at a very small example. Type the following program into your development environment, and make sure you can compile and run it.

```
1  #include <stdio.h>
2
3  // A function that calculates the square of the input
4  int square(int x)
5  {
6      return x * x;
7  }
8
9  int number = 5;
10
11 void main(int argc, char **argv)
12 {
13     int square_of_number = square(number);
```

[1] Actually, as you will learn later in the course, the very first code to run is usually some initializing startup code, but this is usually hidden from the programmer.

```
14      printf("The square of %i is %i.\n", number, square_of_number);
15  }
```

Starting from line one, we see that the program begins with an `#include` statement:

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

This line says that the contents of the file `stdio.h` will be included at the top of this c file before compilation. This file is part of the C Standard Library (see Section 3.5.1) and contains *declarations* of a number of useful functions that deal with user input and output. This is similar to they way the `import` statements work in Java or Python but, as will be seen in Section 3.1, the `#include` statement is much more rudimentary.

The next line:

```
// A function that calculates the square of the input
```

is just a comment. Comments can either begin with `//` and end on the end of the line, or be enclosed between a `/*` and `*/`. Comments are completely ignored by the compiler and are only used to make the code easier to understand.

```
int square(int x)
{
    return x * x;
}
```

Next, we define a *function*. The function is called `square` and has one integer parameter (named `x`). It calculates `x * x` and returns the result as an integer. This is a function *definition*, meaning that we provide the code that will be run when the function is called. In many cases (as in the `stdio.h` file included earlier), we only provide a function *declaration* which tells the compiler the name, return type and parameters of a function that is defined elsewhere (later on in the file, or in a different file). Functions are described in more detail in Section 2.1, and the way code is combined from different files is discussed in Chapter 3.

```
int number = 5;
```

Here, we declare a *global variable* of type `int` (a four-byte integer), and assign the value 5 to it. This variable will have its position in memory allocated during compilation and will exist throughout the whole program. The actual location of variables in memory will be discussed in Section 4.1.

```
void main(int argc, char **argv)
{
    int square_of_number = square(number);
    printf("The square of %i is %i.\n", number, square_of_number);
}
```

We now provide the function definition of the `main` function. This is where program execution will start. This function returns `void`, which is how we write in C that it does not return anything. There are two parameters: an integer `argc`, that says how many arguments where provided to the program on the command line, and a second parameter `argv` which is a *pointer* (a memory address) to an area in memory containing `argc` other pointers, each of which point to a string of characters containing an argument that was given to the program on the command line. The use of pointers to refer to variables in memory is an important part of C, and is usually the most difficult part for beginners to grasp. We will discuss pointers in more detail in Chapter 4.

Next, we define a new integer variable, `square_of_number`, also of type `int`. Because it is defined inside a function, this is a *temporary* variable which only exists (on the stack, or in registers) while we are executing this function. We call the function `square` with the global variable `number` as an argument, and store the returned value in `square_of_number`.

Finally, we print the result to the console. This is achieved by calling the function `printf` which has been declared in the `stdio.h` file. The `printf` function takes as its first argument a string[2]. Within this string we have inserted *tags*, on the form "`%i`". The first such tag means that `printf` should substitute the tag with the value of the second parameter and that that parameter is of type `int`. The next tag will be substituted for the third parameter, and so on[3]. The final two characters in the string, `"\n"`, denote a `newline` character and mean that the next text sent to the console will appear at the beginning of the next line.

As you can see, the general structure and syntax of a C program is very similar to other imperative languages, but there are a number of details that will need further clarification and that is what this compendium is for. It is *not* intended as a complete reference to the C language. There are several books and online sources that delve much deeper, and provide online learning examples. We urge you to use these resources if you find some topic in this compendium challenging. Two good starting points are:

`https://www.w3schools.com/c/index.php`
`https://www.programiz.com/c-programming`
`https://www.tutorialspoint.com/cprogramming/index.htm`

That said, this text is meant as a supplement to the Course book, and it will hopefully be sufficient for this course. As we are sure you already know, simply reading through the chapters will not get you very far, however. But make sure to code through the assignments in the course, and C might soon be your favourite language (until you discover C++).

[2]To be precise, it takes a pointer to a string of characters (bytes) in memory.

[3]A detailed description of `printf` is found on, e.g., `https://cplusplus.com/reference/cstdio/printf/`

2

Basic C Language

In this chapter, we will go through the basic structure of the C programming language.

2.1 Functions

2.1.1 Calling, Declaring and Defining a function

All code in a C program resides in a *function*. To call a function, it must have been *declared* earlier in the C file being compiled. A function declaration looks like:

```
<return type> function_name(<parameter 1 type> parameter_1_name,
                            <parameter 2 type> parameter_2_name,
                            ...);
```

So, if we for instance want to create a function called `max` that returns the maximum of two integers, we could declare it as:

```
int max(int a, int b);
```

This only tells the compiler that *there is* a function called `max` that takes two integers as parameters and returns an integer. We have not yet *implemented* or *defined* the function. Declaring a function, without giving the definition, is necessary if the function definition resides in a different C file, or if lies *after* the calling function[1].

```
1  int max(int a, int b); // Function declaration
2
3  void main(int argc, char **argv)
4  {
5      int a = max(2, 3); // Function call
6  }
7
8  // Function definition
9  int max(int a, int b)
10 {
11     if(a > b) return a;
12     else return b;
13 }
```

[1] It sometimes has to. Consider the case where function `a()` calls `b()`, which then calls `a()` again.

In the example above, the function `max` is first *declared* on line 1. This allows us to *call* the function on line 5. Finally, starting on line 9, we *define* the function by providing the code that shall be run.

A function can only be *defined* once, in one file, in a C program but needs to be *declared* before it can be called in all C files that need to call it. Note that a function definition also counts as a declaration, so the following code is perfectly valid:

```
1 int max(int a, int b) // Function declaration AND definition
2 {
3     if(a > b) return a;
4     else return b;
5 }
6 void main(int argc, char **argv)
7 {
8     int a = max(2, 3); // Function call
9 }
```

2.1.2 Function parameters

Unlike most modern languages (including C++), C *only* allows passing function parameters *by value*. That means that every time you call a function in C, the parameters are *copied* to the called function. Any changes that happen to the variables in the called function are local to that function:

```
1 void f(int a) {
2     a = 5;
3 }
4 void main()
5 {
6     int x = 0;
7     f(x);
8     printf("x = %i\n", x);
9 }
```

This program will print `x = 0` to the console. When calling function `f` on line 7, the value of `x` was *copied* into the parameter `a` and when that value is changed on line 2, the value of `x` in `main` is not affected.

It is not uncommon that we *want* a function to change the value of a parameter. Consider a function `swap(x, y)` that should simply swap the values of x and y. The following code:

```
1 void swap(int x, int y) {
2     int tmp = y;
3     y = x;
4     x = tmp;
5 }
6 void main()
7 {
8     int a = 5, b = 2;
9     swap(a, b);
```

```
10      printf("a = %i, b = %i\n", a, b);
11  }
```

would not not accomplish anything, and would output `a = 5, b = 2`.

The solution, in C, is to use *pointers*. Pointers will be discussed in detail in Section 4.2, but we will take this opportunity to show one case where they are useful. A pointer is the *address* of a variable. If we send (copies of) the *addresses* of `a` and `b` to the function `sort`, we *can* modify their values and get the expected result.

```
1   void swap(int *x, int *y) {
2       int tmp = *y;
3       *y = *x;
4       *x = tmp;
5   }
6   void main()
7   {
8       int a = 5, b = 2;
9       swap(&a, &b);
10      printf("a = %i, b = %i\n", a, b);
11  }
```

This code would output `a = 2, b = 5`, but the syntax is probably quite confusing at the moment. We will return to this example later.

2.2 Basic Data Types

The only built-in data types in C are integers and floating point numbers. Anything more complex, such as a data type describing a player in a game, or the properties of some device, are built from these basic types using `structs`, `arrays`, and `unions`, as descibed in Chapter 5.

In addition to choosing whether a variable should be integer or floating point, we also have to specify how many bytes it should occupy (i.e., its range) and, in the case of integers, whether it should be signed or unsigned. For historical reasons, the C specification allows for a number of more or less confusing ways of describing integer types, but Table 2.1 describes the ones we will use in this course.

If you do not prefix your data type with `signed` or `unsigned`, it is assumed to be signed *except* if it is a `char` (a one byte integer), in which case its signedness depends on the architecture (!). Because the notation can be somewhat confusing, it is common to include the file `stdint.h` which allows us to use the short, descriptive names listed in the last column of Table 2.1.

The floating-point data types are simpler and consist only of `float` or `double` for the 4 and 8 byte data types respectively.

2.2.1 Type Casting

We often have to convert from one data type to another. This can be done either *implicitly* or *explicitly*:

Table 2.1: Integer data types

Integers			
type	range	bytes	short name
`unsigned char`	0 to 255	1	`uint8_t`
`unsigned short`	0 to 65535	2	`uint16_t`
`unsigned int`	0 to $2^{32}-1$	4	`uint32_t`
`unsigned long long`	0 to $2^{64}-1$	8	`uint64_t`
`signed char`	-127 to 127	1	`int8_t`
`signed short`	-32767 to 32767	2	`int16_t`
`signed int`	$(-2^{31}+1)$ to $(2^{31}-1)$	4	`int32_t`
`signed long long`	$(-2^{63}+1)$ to $(2^{63}-1)$	8	`int64_t`

```
1 void main()
2 {
3     float a = 200.501f;
4     int b = a;
5     unsigned char c = a * b;
6 }
```

In the example above, on line 4 the value 200.5 is cast to an integer and stored in variable `b`. Since an integer cannot store a floating point value, it will be truncated to 200. On the next line `b` is first promoted to a floating point number, then `a * b` is calculated as a floating point number (40100.2), then that value is cast to an `unsigned char` and stored in the variable `c`. Since an unsigned char can only store values up to 255, only the lowest byte of the result (`0xA4`) will remain in c.

If this seems a bit complicated, that is because it is. The C specification has a large number of very strict rules about what happens, and in what order, when casting between datatypes, but it can be hard to remember. Therefore, it is often better to *explicitly* describe what casts should be done:

```
1     float a = 200.501f;
2     int b = (int) a;
3     unsigned char c = (unsigned char)(a * (float) b);
```

2.3 Conditional execution

Choosing whether to execute code depending on some condition in C is similar to most high level languages. The `if` and `switch` statements are exemplified in Figure 2.1.

Note the `break` statement when using switch. It means that you should break out of the switch statement. If you forget that, the code will continue to execute the subsequent `case` statements, which *can* be used to your advantage in some cases, but is also a very common source of bugs.

One important thing to note, is that there is no `true` or `false` datatype in C.

if statement:

```c
if(a == 5) {
    printf("a is 5");
}
else if(a == 4) {
    printf("a is 4");
}
else if(a == 3) {
    printf("a is 3");
}
else {
    printf("a is something else");
}
```

switch statement:

```c
switch(a) {
    case 5: printf("a is 5"); break;
    case 4: printf("a is 4"); break;
    case 3: printf("a is 3"); break;
    default: {
        printf("a is something else");
        break;
    }
}
```

Figure 2.1: The `if` and `switch` statements. Both code snippets would produce the same result.

Instead, the result of a comparison is always an integer, with 0 meaning false and any other number meaning true. You will see examples of where this can be important in the following sections.

2.3.1 Conditional operator

In some simple cases, when a variable is to be assigned a value based on some condition, the *conditional operator* can be a cleaner way to express your intention:

```c
// Conditional operator:
// <variable> = <condition> ? <if condition is true> : <if condition is false>;
// Example:
int a = (b > 5) ? 20 : 30; // a is 20 if b is more than 5 and 30 otherwise
```

2.4 Iterations

Writing code that *iterates* or *loops* is also very similar to other imperative languages. The `for` and `while` loops are exemplified in Figure 2.2.

for statement:

```c
int pow(int v, int p) {
    int result = 1;
    for(int i=0; i < p; i++) {
        result = result * v;
    }
    return result;
}
```

while statement:

```c
int pow(int v, int p) {
    int result = 1;
    int i = 0;
    while(i < p) {
        result = result * v;
        i += 1;
    }
    return result;
}
```

Figure 2.2: The `for` and `while` statements. Both code snippets would produce the same result.

The `for` statement is generally used when we are iterating a known number of times, and the `while` statement is used when we only know that we should loop until some condition is met. In either case, we can use the `break` statement to immediately

break out of the loop, or the `continue` statement to jump back to the beginning of the loop:

```
int rand(); // Expecting this function to exist elsewhere
            // and that it returns a random number.
void main(void) {
    // Count the number of positive numbers we get
    // before we get a zero
    int result = 0;
    while(1) {
        int v = rand();
        if(v < 0) continue;
        result += 1;
        if(v == 0) break;
    };
}
```

2.4.1 The infamous `goto` statement

Unlike most other modern languages, C actually has a `goto` statement, which works exactly like the "jump" or "branch" instruction in assembler. This sort of low level statement has been removed from modern languages because it provides a very easy way to write completely undebuggable code, and we recommend that you forget that you ever saw this small section and never use it.

2.5 Operators

Most of the operators in C will be well known to you, if not from previous coding experience then from maths, and we will not go through all of them in detail as they can be easily found on the internet[2]. We will only quickly go through the different classes, and highlight some important details:

2.5.1 Arithmetic Operators, `+`, `-`, `*`, etc

We have already seen these used in the text and you probably know how they work. Worth noting are the increment and decrement operators, and making sure you understand the modulus operator.

```
int a = 10, b = 3;
int c = a + b; // Addition
int d = a - b; // Subtraction
int e = a * b; // Multiplication
int f = a / b; // Division
int g = a % b; // Modulus (remainder of an integer division)
int h = a++;   // Increment Operator (assign a to h, then increment a)
int i = ++a;   // Increment Operator (increment a, then assign a to i)
int j = a--;   // Decrement Operator (assign a to j, then decrement a)
int k = --a;   // Decrement Operator (decrement a, then assign a to k)
```

[2] e.g.: `https://www.tutorialspoint.com/cprogramming/c_operators.htm`

2.5.2 Relational Operators, ==, !=, >, <, >=, <=

These are the operators we use to compare variables and they are all probably known to you. Remember that, in C, the result of a relational operator is an integer (0 if false and 1 otherwise), so you *might* see expressions such as:

```
int a = 20 + (a > c); // 21 if a is more than c
```

2.5.3 Logical Operators, &&, ||, !

These are logical operators and are mostly used as in other languages:

```
if(a && b) // If a is non-zero AND b is non-zero
if(a || b) // If a OR b is non-zero
if(!a)     // If a is NOT non-zero (i.e., if a is zero)
```

2.5.4 Bitwise Operators, &, |, ^, ∼, «, »

The bitwise operators are easy to confuse with the logical operators, but they are *not* the same thing. The result of these operators are not a boolean value, but an integer where the operation has been performed per bit. While these operations look the same in most modern languages, you may not have come across them as often, and they will be very important in this course, so make sure you understand the following:

```
// Assume a = (binary) 10101010 and b is 00001111
c = a & b;       // Bitwise AND, c == 00001010
c = a | b;       // Bitwise OR, c == 10101111
c = a ^ b;       // Bitwise XOR, c == 10100101
c = ~a;          // Binary one's complement, c = 01010101
c = a << 2;      // Left shift operator, c = 10101000
c = a >> 6;      // Right shift operator,
                 //   c = 00000010 if a is unsigned
                 //   c = 11111110 if a is signed
```

2.5.5 Assignement Operators

There are several short-hand assignment operators that do an assignement and an operation in the same operator. A few examples:

```
c += b;   // Same as c = c + b
c <<= b;  // Same as c = c << b
c ^= b;   // Same as c = c ^ b
```

2.5.6 Precedence of operators

There are strict rules for which operators take precedence in C. For example, `a = b * c + d` means that we first multiply b and c, then add d. These rules are hard to remember for every single operator, however, and it is usually a good idea to make precedence clear using parentheses whenever precedence is not absolutely obvious.

3

Compilation

In this chapter, we will explain how the C code is turned into a binary file that can be executed on the computer. We will then go on to explain how libraries are created in C, and talk about the C standard library, that accompanies any C compiler.

To describe the process of compiling a whole C program, we will use a small example, consisting of a few files:

functions.h:

```c
float function(float a);
```

math.h:

```c
float pi_squared();
```

main.c:

```c
#include "functions.h"
void main()
{
    float a = function(2.0f);
}
```

functions.c:

```c
#include "math.h"
float function(float a)
{
    return a * pi_squared();
}
```

math.c:

```c
#define PI 3.14
float pi_squared()
{
    return PI * PI;
}
```

Figure 3.1: The source files of our program

The program consists of three C files: `main.c`, `functions.c`, and `math.c`. With each C file (except `main.c`) there is an accompanying `.h` file, that *declares* the functions that we want to be visible to other `.c` files.

Normally, when compiling a program from inside an *Integrated Development Environment* (IDE), like CodeLite or Visual Studio Code, we simply press the ”build” button and do not have to care much about what actually happens. In this chapter, however, we will go through the actual compilation steps, since it can be very useful to know how this works when things go wrong.

3.1 Preprocessor

The first step in compiling a program is to run the *preprocessor* on all `.c` files. We can invoke the preprocessor alone on the command-line like this:

```
gcc -E -P main.c -o main.i
gcc -E -P functions.c -o functions.i
gcc -E -P math.c -o math.i
```

Figure 3.2 shows the resulting three files.

main.i:

```
1  float function(float a);
2  void main()
3  {
4      float a = function(2.0f);
5  }
```

functions.i:

```
1  float pi_squared();
2  float function(float a)
3  {
4      return a * pi_squared();
5  }
```

math.i:

```
float pi_squared()
{
    return 3.14 * 3.14;
}
```

Figure 3.2: The result of running the preprocessor on the `.c` files

3.1.1 #include

As you can see, the job of the preprocessor is mostly quite simple. When it comes across an `#include "filename"` statement, it will simply cut and paste the contents of the provided file into the `.c` file being preprocessed. For example, in `main.i`, it has removed the include statement and inserted the function declatation from `functions.h`.

It is important to realize that the preprocessor is not smarter than this. Whatever text is in the included file will be inserted, in its entirety, into the preprocessed file.

3.1.2 #define

The file `math.c` begins with the statement `#define PI 3.14`. This tells the preprocessor that whenever it comes accross the word "PI", in the subsequent code, it will replace it with the text "3.14". Note that this is very different from declaring a *variable* called PI and assigning a value to it. The preprocessor does not understand the code at all, it simply replaces text, exactly as it has been told.

The define statement can also be used to construct slightly more complex *macros*, but we will not come across them in this course. A more in-depth explanation can be found at, e.g., `https://www.tutorialspoint.com/cprogramming/c_preprocessors.htm`.

3.2 Compiler

The next step is to take the preprocessed `.i` files and generate assembly code[1]. This is the job of the *Compiler*. We can run the compiler to produce assembly files with:

```
gcc -S main.i
gcc -S functions.i
gcc -S math.i
```

The resulting assembly files are listed in Figure 3.3[2].

You are not expected to understand this assembly code, but we will note a few important things about them. First, we can compile, for instance, the `main.i` file into assembly code *indepenently* of the other files. To create the assembly code for

[1] In practice, the compiler might skip creating the actual assembly code and merge the compilation step with the assembler step that produces machine code, but sometimes it is very handy to see the human readable assembly code.

[2] The output has been stripped of irrelevant lines and comments

main.s:

```
main:
    push {fp, lr}
    add  fp, sp, #4
    sub  sp, sp, #8
    mov  r0, #1073741824
    bl   function
    str  r0, [fp, #-8]
    nop
    sub  sp, fp, #4
    pop  {fp, lr}
    bx   lr
```

functions.s:

```
function:
    push {fp, lr}
    add  fp, sp, #4
    sub  sp, sp, #8
    str  r0, [fp, #-8]
    bl   pi_squared
    mov  r3, r0
    ldr  r1, [fp, #-8]
    mov  r0, r3
    bl   __aeabi_fmul
    mov  r3, r0
    mov  r0, r3
    sub  sp, fp, #4
    pop  {fp, lr}
    bx   lr
```

math.s:

```
pi_squared:
    str  fp, [sp, #-4]!
    add  fp, sp, #0
    ldr  r3, .L3
    mov  r0, r3
    add  sp, fp, #0
    ldr  fp, [sp], #4
    bx   lr
    .L4: .align 2
    .L3: .word  1092478984
```

Figure 3.3: The result of running the compiler on the `.i` files

`main.i`, the compiler needs to know the *there exists* a function called `function`, that it returns a float, and that it takes a float as parameter, but it does not need to know what that function *does*.

Secondly, this assembler code can be created without any knowledge of where this code will reside in memory. Connecting the symbols between the assembly files and placing them at specific places in memory is the job of the *Linker* and will be described in Section 3.4.

In larger projects, it is very important that we can recompile a single `.c` file and that we do not have to recompile *all* files, whenever one of them changes.

3.3 Assembler

The next step is to assemble all of the `.s` files into *object* files. This can be done on the command line with:

```
as -c main.s
as -c functions.s
as -c math.s
```

An object file is a binary file[3], containing the machine code that will eventually run on the processor. Note, however, that this file is not an executable program. We still don't know the final addresses of variables and functions. The "`bl function`" command in *main.s*, for instance, has been assembled into the appropriate machine code for the `bl` instruction, but the *address* it should jump to is not yet available.

[3]The actual format of this file depends on the compiler. gcc will produce files on the ELF format.

3.4 Linker

The final step in producing our executable program is called *linking* the program. The linker takes as input all of the object files, and produces an executable:

```
ld main.o functions.o math.o -o program
```

The linker's main job is to arrange the code in memory, and turn all *symbols* (such as the call to a function called `function`) into actual memory addresses.

If you have run all of these commands on your host computer, with the appropriate gcc toolchain, the output will be an executable file that will run on your operating system (although it won't actually *do* anything visible). If you are cross-compiling for another machine (like the lab computer) and have used the appropriate gcc toolchain this last linking stage will not quite work. You will also have to supply some flags to inform the compiler that it should not expect to find a *runtime* library (see Section 3.5.2), and you would need to supply a *linker script* that tells the linker in what part of memory to place the functions and variables and where to start running the code. This is discussed in the Course book (Chapter 6).

3.5 Libraries

When writing larger programs, it is common to bundle some of your code into *libraries*, i.e., precompiled binaries that can be developed separately and linked with your program. This is also often the way C code is distributed over the internet. A library in C is just a collection of object files, merged into one single "library" file, along with a *header* file that contains the function declarations that are needed to use the library.

If we wanted to create a library so that someone else could use the functions we created for our toy program in Figure 3.1, we could do that in the command line as:

```
gcc -c functions.c math.c
ar cr libfunctions.a functions.o math.o
```

The first line compiles the `.c` files directly into object (`.o`) files. The second line bundles these object files into a single file called `libfunctions.a`. We can now copy our library into some other directory, along with the `functions.h` file. Note that we do not supply the `math.h` file, as we consider that internal to the library and that it should not be used from outside. Finally, we can *use* this library in any other program by calling, for example:

```
gcc main.c -L./libpath/ -I./libpath/ -lfunctions -o program.exe
```

There are a few things to note on this line. First, we are telling gcc to preprocess, assemble, compile, and link, all in one go, and (with the `-o` flag) to output a single

executable called `program.exe`. We also supply the `-L` and `-I` flags. They tell the compiler about an extra directory to look for libraries and include files, respectively. Finally, we use the `-l` flag to tell gcc to link with the library called `libfunctions.a` (it is just gcc convention that the file containing the library is called `lib<name>.a` and to only supply the name on the command line).

3.5.1 C standard library

The *C standard library* (often called `libc`) is an important library that follows any C compiler distribution. It is a collection of functions and macros that help with IO, maths, memory management, etc. For example, it supplies the include file `stdio.h` and the function `printf` that we used in Section 1.2. The C standard library is described in a bit more detail in the Course book (Chapter 6). When compiling a program with gcc, it links with `libc` by default. You can avoid this by adding the `-nolibc` flag when compiling.

3.5.2 Runtime library

The C standard library is meant to be easily portable between systems, but how can we have a portable version of `printf` if it is supposed to write to a console window on you Windows machine, and to a small ASCII display on your lab computer? This is achieved by having a separate library, the *runtime* library that takes care of all low level mechanisms. As an example, the `printf` implementation in `libc` expects there to be an *external* function called `_write` that takes care of putting characters onto some kind of output device. The `_write` function must be implemented by the runtime library, for the standard library to work. How to create a small runtime library for the lab computers is also discussed in the course book (Chapter 6).

3.5.3 Compiler Library

Another library that is usually linked with by default is the *compiler library*. This library is supplied by the compiler and contains precompiled code that may or may not be used depending on the target we compile for. For example, some small ARM processors do not have floating point hardware. If the programmer asks for, e.g., a floating point multiplication, the compiler then has to resort to doing this multiplication in software, and this is achieved by calling an existing function (you can find one such call in Figure 3.3)

3.6 Build Systems

A commercial computer program can easily consist of thousands of different C and header files, and manually typing all these commands for each file would be much too time consuming. Moreover, a large program can take quite some time to compile from scratch, and it can be very hard for a human to know exactly which files need to be recompiled when one file has been changed. This quickly led to the invention of *build systems* or *build automation tools*. One of the first, and for a long time the

most ubiquitous was `make`. This is a program that reads a single file (the `makefile`) where the user has written information about which include files each C file depends on, which C files make up each library, and so on. This was soon followed by software that could automatically parse the C files and *generate* makefiles. These days, the build system is often hidden by the IDE, but oftentimes a program has to compile on a number of different platforms, with different IDEs and build tools. In such cases it is common to use very large tools, such as `CMake`, which will read a high level description of the project structure and create the build files needed for any platform.

4

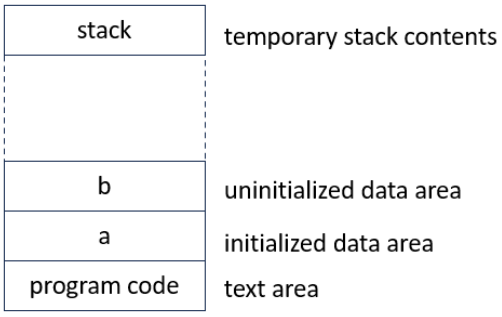
Memory, Arrays, and Pointers

In this chapter, we will describe how variable data is actually stored in memory, and how C uses pointers and arrays to access elements in lists and strings.

4.1 Variables location in memory

In C, and especially in machine oriented programming, it is often important to know where in memory variables reside. If a variable resides on the stack, for example, it can be very dangerous to store the address of that variable and try read or change it later in the program, when the stack's contents represent something different. Let's consider another toy C program:

```
1  int a = 5;
2  int b;
3
4  int count(int v) {
5      if(v == 0) return 0;
6      return 1 + count(v - 1);
7  }
8
9  int main()
10 {
11     b = 20;
12     int c = count(a + b);
13 }
```



The variables `a` and `b` are both *global* variables, and their location in memory is known when the program starts. These variables will reside in the same place in memory throughout the execution of the program and can be read and written from anywhere. The variable `a` is assigned a value when it is declared in the global scope, and will be copied from the executable file into the *initialized data area* which, in memory, will come right after the program code. The variable `b` is not initialized, and will be allocated a space in the *uninitialized data area*. The required size of this area is known from the executable file, and it will be initialized to 0 when the program starts.

All other variables in the program are declared inside functions. These are called *local* variables and will only "exist" while the function is executing. Usually, this means that they will exist on the *stack*, which grows every time a function is called,

and shrinks when the function returns. In some cases, local variables do not need to reside in memory at all, but only exist temporarily in processor registers.

4.1.1 Static and Constant variables

In some cases, we want a variable that ”exists” throughout the program, but that is only visible in a specific function or scope. This is achieved using the `static` qualifier:

```
1 int counted_function(int v) {
2     static int times_the_function_has_been_called = 0;
3     times_the_function_has_been_called += 1;
4     ...
5 }
```

A variable declared as `static` will reside in the initialized or uninitialized data area, just like a global variable, but is only *visible* in the scope that it was declared (i.e., the compiler will produce an error if we try to access it from outside the scope).

Both global and local variables can also be qualified by the keyword `const`:

```
1 const int a = 5;
2 int function(int v) {
3     const int b = 20;
4     ...
5 }
```

Using `const`, we are telling the compiler that this variable will not change, and trying to assign a new value to that variable will produce an error. `const` variables will reside either in the initialized data area or in the *text* area (with the program code). In either case, they may be placed in read-only memory.

4.2 Pointers

The use of *pointers* in C is usually the hardest part of the language for beginners to grasp. Understanding pointers is completely necessary for C to become a useful programming language, however. In this section, we will introduce the concept, and in the next sections we will go through some uses and examples.

A pointer is a special variable containing a *memory address* that points to another variable somewhere in memory. We will illustrate this in a small example in Figure 4.1.

At line 1, we create a global integer variable called `b` and initialize it with the value 20. It will reside at the first memory location in the initialized data area when we start the program. In this example, the address where `b` resides in memory is: `0x2100`.

Then, on line 2, we create another variable called `ptr_to_b`. The type of this variable is `int *`. The asterisk here means that `ptr_to_b` is of the type ”pointer

```
1 int b = 20;
2 int * ptr_to_b = &b;
3 int main()
4 {
5     // At this line, b == 20
6     *ptr_to_b = 40;
7     // At this line, b == 40
8 }
```

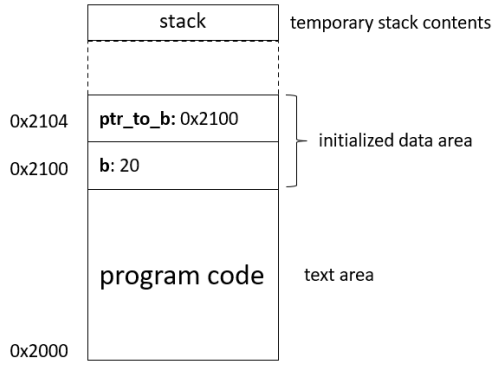


Figure 4.1: Example code and the memory contents before line 5.

to integer". This variable resides in the next available memory in the initialized data area, address `0x2104`. We immediately initialize this variable to be `&b`. The ampersand (`&`) here means "the address of `b`", so the value stored att memory address `0x2104` is `0x2100`.

Now, at line 6, we *dereference* the variable `ptr_to_b` and assign a new value to it. Dereferencing a pointer means "give me the variable that this pointer *points* to", and is done by putting an asterisk in front of the pointer [1]

So when the compiler sees "`*ptr_to_b = 40`" it will look read the value stored at `0x2104` and read that as a memory address (`0x2100`). It then assigns the value 40 to the integer at that memory address. Therefore, the next time we read the variable `b` (which resides at memory address `0x2100`) we will read the value 40.

If you feel lost at the end of this section, take a deep breath and try again. It is essential to understand that *a pointer is a variable containing the address of another variable, of a specific type*. Once that is clear, working with pointers will become second nature.

4.3 Accessing absolute memory addresses

One reason that pointers are important in machine oriented programming is that they allow us to express reading and writing from arbitrary memory addresses. When we do not have an operating system and drivers, the only way for the processor to communicate with peripheral hardware is through memory load and store operations.

Let us consider a new toy example, where we know that out program is loaded into a 32kb SRAM chip that is mapped to addresses `0x20000000 - 0x20007FFF`. We also know that there is a second 128kb SRAM chip mapped to addresses `0x30000000 - 0x3001FFFF`. Our program needs to read some data into memory for future processing. Since there is a lot of data to read, it will not fit together with our program on the smaller SRAM, so we will put it on the larger SRAM.

[1] So, in a *statement*, the asterisk means "dereference", but in a *declaration* it means "is a pointer".

```
1  char ReadValue(); // We expect this function to exist in some other file.
2  int main()
3  {
4      unsigned int BIG_SRAM_ADDRESS = 0x30000000;
5      char * output_pointer = (char *) BIG_SRAM_ADDRESS;
6      for(int i=0; i<44100; i++) {
7          *output_pointer = ReadValue();
8          output_pointer = output_pointer + 1;
9      }
10 }
```

On line 4, we store the known starting address of the larger SRAM into an unsigned integer. Then, on line 5, we define a pointer, `output_pointer`, to that address, and specify that it points to a `char` value (a byte). Next, we want to read one value at a time and place each value on the next available memory position.

In the loop that follows, we read one value and place it at the address that `output_pointer` points to, by *dereferencing* the pointer (Line 7). On Line 8, We then increase the pointer by one (so that it points to the next byte in memory), and repeat the process until all values have been read.

4.3.1 Volatile Pointers

Another situation where absolute memory accesses are necessary is when writing to, or reading from, hardware registers and ports. The only way the CPU has to communicate with outside peripherals is through load and store operations. So when, for instance, we want to read the current value from an external timer, we will do that by reading a specific memory address that is mapped to that timer's register (this will be discussed in detail elsewhere in the course). The fact that this value can change with no CPU interference introduces a problem for the compiler. Consider the following code:

```
1  void WriteValue(char value);  // We expect this function to exist in some
2                                // other file
3  int main()
4  {
5      char * timer_value_pointer = (char *) 0x12340000;
6      while(1) {
7          char current_value = *timer_value_pointer;
8          WriteValue(current_value);
9      }
10 }
```

The code is supposed to read some timer value, from a specific port at some address, and in every iteration of the loop it will write that value out somewhere (perhaps to a bargraph). The problem is that the compiler will generally assume that all loads and stores are memory accesses, *and* that the CPU it is running on is the only one controlling that memory. Therefore, it can look at this code and realize that no one is *writing* to the address we are reading all the time. Therefore, it might optimize the code by doing the read (line 7) *once* outside of the loop.

This is a perfectly good optimization in most programs, but since this particular address is *not* connected to memory, but to an external timer, it would be disastrous in this case. The simple solution to this is to use the `volatile` keyword:

```
7       volatile char * timer_value_pointer = (char *) 0x12340000;
```

This tells the compiler that this particular pointer points to an address that might be changed from outside, and so can not be considered for this kind of optimization.

4.4 Pointer Arithmetic

In the example in Section 4.3, we used a pointer to a `char` (one byte) in memory. There is a small but important point to be made about how arithmetic operations work on pointers. Let's say we have exactly the same toy program as before, but the values we need to store are 32 bit floating point values, instead of bytes:

```
1  float ReadValue(); // We expect this function to exist in some other file.
2  int main()
3  {
4      unsigned int BIG_SRAM_ADDRESS = 0x30000000;
5      float * output_pointer = (float *) BIG_SRAM_ADDRESS;
6      for(int i=0; i<44100; i++) {
7          *output_pointer = ReadValue();
8          output_pointer = output_pointer + 1;
9      }
10 }
```

This code would work just fine, which might look weird considering line 8. Since we are now reading and storing floating point values (four bytes), the address that `output_pointer` points to must be increased by 4, in every iteration of the loop. When adding x to a pointer, we are telling the compiler to add *the size of the type of element that the pointer points to* to the address. So, for example:

```
1  int main() {
2      char * char_pointer = (char *) 0x20000000;
3      short * short_pointer = (short *) 0x2000000;
4      int * int_pointer = (int *) 0x20000000;
5      char_pointer += 1;  // Now points to address 0x20000001
6      short_pointer += 1; // Now points to address 0x20000002
7      int_pointer += 1;   // Now points to address 0x20000004
8  }
```

This might seem strange, but actually leads to much cleaner code in many cases.

Since pointers are often used to point out the *starting address* of a list of elements in memory, and we add an offset from that address to access a specific element, there is an alternative way of writing this is illustrated in the next example:

```
1 float ReadValue(); // We expect this function to exist in some other file.
2 int main()
3 {
4     float * big_sram_pointer = (float *) 0x30000000;
5     for(int i=0; i<44100; i++) {
6         big_sram_pointer[i] = ReadValue();
7     }
8 }
```

The syntax `ptr[i]` is *equivalent* to `*(ptr + i)`; It means "Take the address that `ptr` points, increase it by the size of `i` *elements*, and dereference that pointer".

4.5 Arrays

In the previous sections, we have seen examples of iterating through a list of variables (an array) when we know the absolute starting address of the list. Obviously, C supports allocating arrays of elements without hard-coded addresses as well:

```
1 short value_array[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
2 int main()
3 {
4     int square_array[10];
5     for(int i=0; i<10; i++) {
6         square_array[i] = value_array[i] * value_array[i];
7     }
8 }
```

Hopefully, the syntax is fairly intuitive from your knowledge of other imperative languages (e.g., Java). On the first line, we declare that we want an array of `short` (2 byte) elements. The square brackets can be empty, because we immediately define the values of the array on the same line (comma separated within the curly brackets). Since this array is declared outside of any function, the memory for the array will be allocated in the *initialized data area*. The compiler will count the number of elements and knows the size of each element and allocates memory for the array before the program starts. The variable `value_array` is a `const` pointer to the first element of the allocated array.

On line 4, another array is declared. This time, no initial values are given and so the size of the array must be given within the square brackets. Since this array is local, it will be allocated on the stack and will only "exist" while the function is running. Note that, unlike higher level languages, an array in C is always of constant size, since the compiler needs to know the size when producing the code[2].

The program then enters a loop and, for each element, the i:th element in the first array is squared and the result is stored in the second array. This is done, just as in the previous sections, using the pointers to the start of the arrays in memory (`value_array` and `square_array`).

[2]Allocating memory dynamically is possible, and common practice, when writing programs for machines with operating systems. How this works is discussed in detail in the Course Book, Chapter 6.

4.5.1 Pointers as Function Parameters

Let's say we want to isolate the code that squares the values of an array in a function. This is done by passing the pointers to the start of the arrays as arguments to the function:

```
1  void SquareArray(short * src_array, int * dst_array, int num_elements) {
2      for(int i=0; i<num_elements; i++) {
3          dst_array[i] = src_array[i] * src_array[i];
4      }
5  }
6
7  int main()
8  {
9      short value_array[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
10     int square_array[10];
11     SquareArray(value_array, square_array, 10);
12 }
```

After the call to `SquareArray` on line 11, the array `square_array` will contain the squares, just as before. It is worth remembering here that we claimed in Section 2.1.2 that *all* function parameters in C are sent *by value*. Yet, the contents of `square_array` are changed after the function call. This is because the array itself is *not* a function parameter. The parameter is a single *pointer* to the start of the array, and that pointer is copied as a function parameter.

In the same way, we can use pointers to variables when we want to change the value of "a parameter" (where we would have used pass by reference, in other languages):

```
1  void swap(int * x, int * y) {
2      int temp = *y;
3      *y = *x;
4      *x = temp;
5  }
6
7  int main()
8  {
9      int a = 1, b = 2;
10     swap(&a, &b);
11     // a now equals 2 and b equals 1
12 }
```

This example was discussed in the introduction, but bears repeating now that you have a better grasp of what pointers are. The function `swap` takes as input *pointers* to two integers, which allows it to modify the values that they point at.

4.5.2 Strings

In C, there is no built-in or otherwise special datatype to represent strings. Instead, a string is simply a certain number of characters stored contiguously in memory, i.e., an array of `char`. Each character is represented by a single byte (which can

have a value between 0 and 255) and each character is assigned a certain value. The mapping between byte values and letters is defined in the ASCII standard[3]. This standard also contains some non-characters. For instance, the byte value 10 means "end of line" and the byte value 0 means "end of string". When we want to do operations on strings, for example compare two strings and see if they are equal, we do this by passing around the *pointer* to the beginning of the strings:

```
int compare_string(char * str0, char * str1)
{
    while(1) {
        char character0 = *str0;
        char character1 = *str1;
        if(character0 != character1) return 0;
        if(character0 == 0) return 1;
        str0 += 1;
        str1 += 1;
    }
}
```

This function will take the pointer to the starting characters of two strings, then look at the subsequent chars until they either differ (so the strings are not the same and we return 0), or one of them is zero (then we have reached the end of the strings and we return 1).

A string can be declared in several ways:

```
char * string1 = "Hello";
char string2[] = "Hello";
char string3[] = {'H', 'e', 'l', 'l', 'o', 0};
char string4[] = {72, 101, 108, 108, 111, 0};
```

All of these declarations mean exactly the same thing.

4.6 2D Arrays

4.7 Pointers to Pointers

We have seen how pointers can be used to point to basic datatypes such as `int` or `char`. In this section we will see that we often need pointers that point to other pointers.

Figure 4.2 shows an example of declaring a pointer to another pointer. As you can see, it works exactly the same as creating a pointer for any other type:

```
    int ** ptr_to_ptr_to_a = &ptr_to_a;
```

The type of a "pointer to a pointer to an integer" is `int **`, and to get the address of a pointer, we use the & character just as before.

[3]More info on the ASCII standard at `https://en.wikipedia.org/wiki/ASCII`

```
1  int main()
2  {
3      int a = 0x1234;
4      int * ptr_to_a = &a;
5      int ** ptr_to_ptr_to_a =
6                   &ptr_to_a;
7  }
```

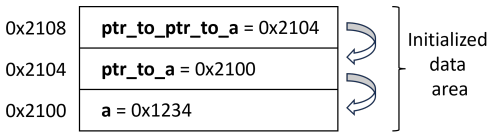


Figure 4.2: Example code and the memory contents after line 5.

Let's revisit the example of a `swap` function from Section 4.5.1, but this time we want to swap two *pointers*:

```
1  void swap(char ** x, char ** y) {
2      char * temp = *y;
3      *y = *x;
4      *x = temp;
5  }
6
7  int main()
8  {
9      char * name0 = "Chimpanzee";
10     char * name1 = "Bonobo";
11     swap(name0, name1);
12     // name0 now points to "Bonobo" and name1 to "Chimpanzee"
13 }
```

The variables `name0` and `name1` are both *pointers* to the places in memory where two text strings start. We want to swap these two pointers, so that `name0` points to the start of the string "Bonobo" and `name1` points to the start of the string "Chimpanzee" (without actually changing any of the strings).

Just as in the previous `swap` example, we cannot send `name0` and `name1` as parameters to the swap function, since they will be copied to the stack and any changes will be local to the swap function. Instead, we send pointers to `name0` and `name1`, which are dereferenced in the swap function to change their respective contents.

Thus, the the parameters to the swap function are:

```
void swap(char ** x, char ** y)
```

That is, `x` is a "pointer to a pointer to a char". When dereferenced we have that `*x` is a "pointer to a char".

4.8 Function Pointers

We will conclude this chapter by talking about *function pointers*. Just as a pointer can point to, e.g., the beginning of a string of characters in memory, it can point to the beginning of a piece of code in memory (a function). This can be extremely useful as it, for instance, allows us to send a function as a parameter to another function. Let's consider a very simple toy example:

```
1  int function(int a) {
2      return a + 1;
3  }
4  int main() {
5      int (*function_ptr)(int) = &function;
6      int a = (*function_ptr)(1);
7  }
```

On line 5, we declare a new function pointer and assign it the address of an existing function. On line 6, we then dereference that function pointer (which gives us a function) and call that function with the parameter 1. There is really nothing new about this, a function is just another thing that we can point to, but the *syntax* is often quite confusing to beginners[4].

A function pointer is declared as:

```
<return type of function> (*<name of function pointer>)(<parameters of function>)
```

The pointer (address) to a specific function is obtained with the & operator just as for any other data type, and that pointer can be dereferenced to call the actual function with the * operator, as we saw in the previous example. However, somewhat surprisingly, it is *also* allowed to use the function name itself to mean "the address of this function" and to call the function pointer without dereferencing it. That is, the example above *can* look like:

```
1  int main() {
2      int (*function_ptr)(int) = function;
3      int a = function_ptr(1);
4  }
```

This code is equivalent and arguably "prettier", but it is less consistent with how pointers to ordinary data types work.

Let us look at a slightly more useful example of using function pointers:

```
1  int Double(int value) {
2      return 2 * value;
3  }
4  int Square(int value) {
5      return value * value;
6  }
7  void map(int (*an_operation)(int), int * items, int num_items)
8  {
9      for(int i=0; i<num_items; i++) {
10         items[i] = (*an_operation)(items[i]);
11     }
12 }
13 int main()
14 {
15     int values[] = {1, 2, 3, 4};
```

[4] Okay, the syntax can be confusing to experienced programmers too.

```
16      map(Double, values, 4); // Double every item in values
17      map(Square, values, 4); // Square every item in values
18      // values is now {4, 16, 36, 64}
19  }
```

Here, we create a general function called `map` that takes as input a list of items, and an operation that shall be performed on each item. There is nothing new in this example, so go through the code and make sure you understand how it works.

5

Advanced data types

In real programs, we naturally need data types describing more complex objects than a single integer. In this chapter, we will learn about how more complex data types can be created in C.

5.1 typedef

One of the simplest ways in C where we create a new name for a data type is the `typedef` keyword:

```
typedef <existing name> <name of aliased data type>
```

This can be useful in many cases. For instance, we have already talked about using the file `stdint.h` to get better names for integers of different sizes. This is achieved using `typedef`:

```
...
typedef signed long long int64_t;
typedef unsigned long long uint64_t;
typedef signed long int32_t;
typedef unsigned long uint32_t;
typedef signed short int16_t;
typedef unsigned short uint16_t;
...
```

5.2 Structs

For more complex data types we will use *structures*. These will look familiar to people who are used to *classes* in other languages, but a `struct` is a much simpler concept, as it can not have *methods*, *constructors*, *inheritance*, or any other fancy functionality. It is just a compound data structure, consisting of a compilation of other data types. A simple example of the usage of a `struct` follows:

```
struct Player {
    int score;
    int health;
};
int main() {
```

```
6       struct Player player_one;
7       player_one.score = 0;
8       player_one.health = 100;
9   }
```

So `player_one` in this example is an instance of the `struct` "Player", and has two *fields* called "score" and "health", which can be read and written to using the notation `player_one.score` or `player_one.health`.

Note that (for mostly historical reasons, and unlike most modern languages), the name of the struct, "Player" is not a type in C, so we can *not* write:

```
1   Player player_one; // Incorrect!
```

to declare an instance of the `struct`. It must be preceded by the `struct` keyword. To remedy this, we can make use of `typedef`:

```
1   typedef
2       struct {
3           int score;
4           int health;
5       }
6       Player;
7   Player player_one; // Correct
```

Here, the datatype we want to create an alias for is an *unnamed* `struct` containing the fields "score" and "health" and we give that datatype the alias "Player". We can then use "Player" as our data type in the way we are used to from other languages.

5.2.1 Initialization of a `struct`

When declaring a struct, we can simulatenously initialize it like this:

```
1   Player player_one = {0, 100};
```

The values within the curly brackets will be assigned to the fields of the `struct` in the order that they appear in the declaration. C also allows for a convenient initialization of named fields:

```
1   Player player_one = {.score = 0, .health = 100};
```

In this case, the values can come in any order, and any field that is not named will be initialized to zero.

5.2.2 Nested structures

A struct can also have other structs as fields. In the following example we first declare a struct that represents a 2D coordinate, and then a line that has one starting coordinate and one end coordinate:

```
typedef struct { int x; int y; } Coord2D;
typedef struct {
        Coord2D start, end;
} Line;
```

We can also declare a struct nested in another struct:

```
typedef struct {
    struct { int x; int y; } start, end;
} Line;
```

In this case, we declare two instances (”start” and ”end”) of a *nameless* struct, to be fields of our ”Line” struct. In either case, the child structs can be defined together with the parent struct during declaration, or accessed like any other field after declaration:

```
int main()
{
    Line line = {{1, 2}, {3, 4}};
    line.start.x += 1;
}
```

5.2.3 Pointers to `structs`

When a variable of `struct` type is declared, it will of course have its place in memory just like any other variable. The fields of the struct will lie contiguously (in the order that they are declared) in memory along with the fields of any nested structure. See Figure 5.1 for an example.

Note that the exact positioning of each field will have to adhere to the rules for correct *alignment*, which means that there will sometimes be padding (unused space) in memory. This is discussed in greater detail in the Course book, Chapter 1.5.

```
typedef struct {
    int x;
    int y;
} Coord2D;
typedef struct {
    Coord2D start, end;
} Line;
Line line;
```

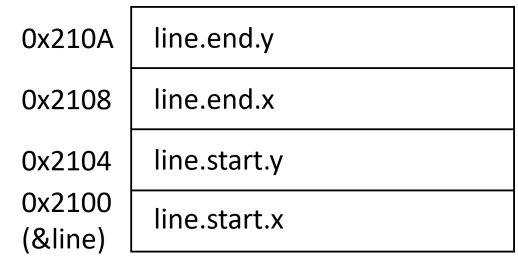


Figure 5.1: Declaration of a struct and how it ends up in memory.

As with any data type, we can refer to an instance of a struct through a pointer. Since a structure can be a fairly large amount of data, this is usually how we pass structures in function calls:

```c
typedef struct {
    char title[100];
    char author[100];
    int num_pages;
    short ISBN;
} Book;

// We expect this function to exist and do something important
void ProcessBook( Book * book );

int main() {
    Book book = { "Middlemarch", "George Elliot", 370, 1234 };
    ProcessBook(&book);
}
```

In the above example, a struct representing a book is sent to the function `RegisterBook`, but the parameter to the function is a *pointer* to a ”Book”, rather than a ”Book”. If we had not used a pointer, all of the data that makes up the structure would have to be copied to the stack which could become very costly both in terms of performance and memory. By just sending the address to where the book is already stored in memory, we avoid this.

Since it is so common to use pointers to structures, there is a special syntax that allows us to directly access the fields from a pointer to a struct:

```c
void ProcessBook( Book * book )
{
    // We can dereference the book to get at the fields:
    (*book).title = "Something New";

    // Or we can directly access the fields from the pointer:
    book->title = "Seomething New";
}
```

5.2.4 Incomplete declarations

There are cases where we want two different structures to refer to each other, or we might want one structure to refer to another instance of itself. Consider the following example:

```c
typedef struct {
    char * name;
    int birth_date;
    Book * best_book;     // Incorrect
} Author;

typedef struct {
    char * title;
    Author * author;
} Book;
```

When declaring the `Author` structure, we are trying to use a `Book` as one of the fields, but that has not yet been declared. If we changed the order, we would have the opposite problem (the `Author` would not yet have been declared when trying to declare the Book). In such cases, we will need to create an *incomplete* declaration of the structures:

```
// Incomplete declarations
struct Book;
struct Author;

typedef struct Author {
    char * name;
    int birth_date;
    struct Book * best_book;     // Incorrect
} Author;

typedef struct Book {
    char * title;
    Author * author;
} Book;

int main()
{
        Author author = { "George Elliot", 1819 };
        Book book = { "Middlemarch", &author };
        author.best_book = &book;
}
```

On the first two lines, we tell the compiler that *there is* a struct called ”Book” and a struct called ”Author”, but they are not yet defined. We can then, when declaring the struct ”Author” say that it shall have a field that is a pointer to a ”Book”, without error. The compiler does not yet know what a Book is, but it knows that any *pointer* is just a 32 bit address, which is all it needs. We do need to declare it as `struct Book *`, since we have not yet declared the `typedef`.

5.2.5 Bit fields

When programming in C, and especially in machine oriented programming, it is often important to use as little memory as possible for a structure. Consider the following unusually silly example:

```
typedef struct {
    char is_alive;   // 1 if alive, 0 if dead   (1 bit)
    char age;        // Monkeys do not live past 100 years (7 bits)
    char strength;   // On a scale from 0 to 10 (9 bits)
    short num_bananas; // Max bananas is 1000 (10 bits)
} Monkey;
```

Although we use the smallest possible datatypes for each field, the size of one `Monkey` would be 6 bytes (one extra byte due to alignement). Since the total information needed is 27 bits, we could pack all of this into a single `unsigned int`, and use

shifting and masking operations to get the data out, but that is not very easy to read. Alternatively, C allows for so called *bit fields* that can hide these operations for us:

```
typedef struct {
    unsigned int is_alive : 1;      // 1 if alive, 0 if dead
    unsigned int age : 7;           // Monkeys do not live past 100 years
    unsigned int strength : 9;      // On a scale from 0 to 10
    unsigned int num_bananas : 10;  // Max bananas is 1000
} Monkey;

int main()
{
    Monkey monkey;
    monkey.strength = 35;
}

```

We now inform the compiler that, e.g., `is_alive` only requires 1 bit. It will place all of these variables in the same unsigned int (starting at the lowest bit), and when we access individual elements, it will do the shifting and masking for us.

If we should add another variable, that does not fit in the remaining 5 bits:

```
    ...
    unsigned int num_bananas : 10;  // Max bananas is 1000
    unsigned int thrown_bananas : 8;
} Monkey;
```

the size of the struct will be 8 bytes, with `thrown_bananas` taking up the lowest 8 bits, and the rest being unused. As you will see during the course, bit fields are especially useful for declaring structs that map to *ports* that communicate with hardware units.

5.3 Unions

Another space-saving mechanism in C is *unions*. Consider the the following struct that describes a vehicle in a video game:

```
typedef struct {
    int speed;
    int weight;
    short type;       // 0 = CAR, 1 = BOAT, 2 = AIRPLANE
    char num_wings;   // Only applies to airplanes
    char num_sails;   // Only applies to boats
    short num_wheels; // Only applies to cars
} Vehicle;
```

The last three fields in this struct take up 12 bytes, but only one of them will carry any important data (depending on the `type` of the vehicle). Here, we can use a `union` to create a more compact representation:

```
typedef struct {
    int speed;
    int weight;
    short type;       // 0 = CAR, 1 = BOAT, 2 = AIRPLANE
    union {
        char num_wings;   // Only applies to airplanes
        char num_sails;   // Only applies to boats
        short num_wheels; // Only applies to cars
    };
} Vehicle;
```

By grouping the fields together in a `union`, we tell the compiler that these three variable will occupy the *same* memory. So if we, for instance, change the value of `num_sails`, we also change the value of `num_wings` and `num_wheels`, but that is okay if the values are mutually exclusive.

Unions can also be very useful if we want to access the same memory in different ways, depending on the situation:

```
union TwoBytes {
    unsigned short data;
    struct {
        unsigned char lower_byte;
        unsigned char upper_byte;
    };
    struct {
        unsigned short first_bit : 1;
        unsigned short second_bit : 1;
    };
};

int main()
{
    union TwoBytes word;
    word.data = 0xFFFF;
    word.lower_byte = 0x01;
    word.second_bit = 1;
 }
```

Here, we use both unions and bit fields to create a type where we have immediate access to either the whole two-byte word, the upper and lower bytes, or individual bits.

5.4 Enums

Finally, we will look at final bit of syntactic sugar in C, called *enumerations* or `enum`. In our ”Vehicle” struct in the previous section, we had an integer field called ”type”, and in the comment we explain what different values of that field would mean. C allows us to use an enumeration type instead, so that we can use readable names, rather than numbers to describe the type of vehicle:

```
enum VEHICLE_TYPE { CAR, BOAT, AIRPLANE };
typedef struct {
    int speed;
    int weight;
    VEHICLE_TYPE type;
    union { ... };
} Vehicle;

int main()
{
    Vehicle vehicle;
    vehicle.type = BOAT;
}
```

It should be noted that the underlying type of an `enum` is still just an integer. In the example above, CAR is 0, BOAT is 1, and AIRPLANE is 2. We can, if we want, assign specific values to the different names. Enums do not provide much in terms of type safety, but can be used to make the programmers intention clear, and make the code more readable.

DEPARTMENT OF SOME SUBJECT OR TECHNOLOGY
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY