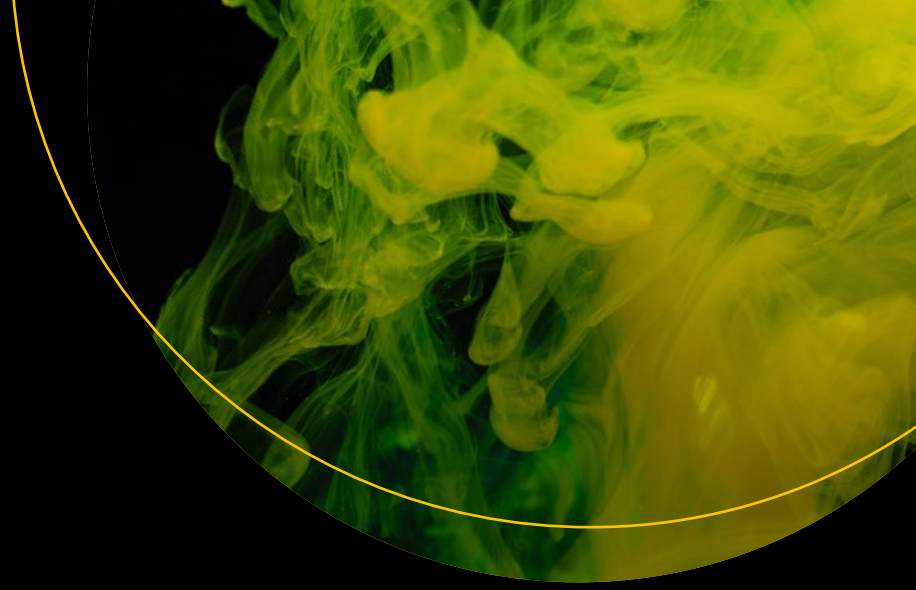

Pointers in C Programming

A Modern Approach to Memory Management, Recursive Data Structures, Strings, and Arrays

Thomas Mailund

Apress®

Pointers in C Programming

A Modern Approach to Memory Management, Recursive Data Structures, Strings, and Arrays

Thomas Mailund

Apress®

Pointers in C Programming: A Modern Approach to Memory Management, Recursive Data Structures, Strings, and Arrays

Thomas Mailund
Aarhus N, Denmark

ISBN-13 (pbk): 978-1-4842-6926-8 ISBN-13 (electronic): 978-1-4842-6927-5
https://doi.org/10.1007/978-1-4842-6927-5

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image by Engin Akyurt on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484269268. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Thomas Mailund is an associate professor in bioinformatics at Aarhus University, Denmark. He has a background in math and computer science. For the past decade, his main focus has been on genetics and evolutionary studies, particularly comparative genomics, speciation, and gene flow between emerging species. He has published *String Algorithms in C*, *R Data Science Quick Reference*, *The Joys of Hashing*, *Domain-Specific Languages in R*, *Beginning Data Science in R*, *Functional Programming in R*, and *Metaprogramming in R*, all from Apress, as well as other books.

About the Technical Reviewer

Juturi Narsimha Rao has 9 years of experience as a software developer, lead engineer, project engineer, and individual contributor. His current focus is on advanced supply chain planning between the manufacturing industries and vendors.

Acknowledgments

I am grateful to Helge Jensen, Anders E. Halager, Irfansha Shaik, and Kristian Ozol for discussions and comments on earlier drafts of this book.

CHAPTER 1

Introduction

Pointers and memory management are considered among the most challenging issues to deal with in low-level programming languages such as C. It is not that pointers are conceptually difficult to understand, nor is it difficult to comprehend how we can obtain memory from the operating system and how we return the memory again so it can be reused. The difficulty stems from the flexibility with which pointers let us manipulate the entire state of a running program. With pointers, every object anywhere in a program's memory is available to us—at least in principle. We can change any bit to our heart's desire. No data are safe from our pointers, not even the program that we run—a running program is nothing but data in the computer's memory, and in theory, we can modify our own code as we run it.

With such a power tool, it should hardly surprise that mistakes can be fatal for a program, and unfortunately, mistakes are easy to make when it comes to pointers. While pointers do have type information, type safety is minimal when you use them. If you point somewhere in memory and pronounce that you want "that integer over there," you get an integer, no matter what the object "over there" really is. Treat it like an integer, and it behaves like an integer. Assign a value to it, and may the gods have mercy on your soul if it was supposed to be something else and something you need later. You have just destroyed the real object you pointed at.

If you are not careful, any small mistake can crash your program—or worse. If you accidentally modify the incorrect data in your program, all your output is tainted. If you are lucky, it is easily detectable, and you are in for a fun few days of debugging. If you are less fortunate, you can make business decisions based on incorrect output for years to come, never realizing that the code you wrote is fooling you every time it runs—or maybe not every time, just on infrequent occasions, so rare that you can never chase down the problem. When you have bugs caused by pointers (or uninitialized memory),

T. Mailund, *Pointers in C Programming*, https://doi.org/10.1007/978-1-4842-6927-5_1

they are not always reproducible. Your program's behavior might depend on which other programs are running concurrently on the computer. If you start debugging it, any code you add to the program to examine it will affect its behavior. Loading the program into a debugger will definitely change the behavior as well. I hope that you will never run into such bugs—known as Heisenbugs after Heisenberg's uncertainty principle—but if you mess around with pointers long enough, you likely will.

It sounds like pointers are something we should stay away from, and many high-level programming languages do try to avoid them. Instead, they provide alternative language constructions that are safer to use but provide much of the same functionality that we need pointers for in C. They are not as powerful but alleviate many of the dangers that raw memory pointers pose. In low-level languages such as C, we are programming much closer to the machine. The computer doesn't understand high-level constructions; it understands memory and chunks of bits, and in low-level languages, we can manipulate the computer at this fundamental level. We very rarely need to, nor do we want to, but when we choose to program in low-level languages, it is to get close to the machine, where we can write more efficient programs, measured in both speed and memory usage. And at this level, we get pointers—more efficient, more fundamental, and more dangerous. If, however, we approach using pointers in a structured manner, we can achieve the safety of high-level languages *and* the efficiency of low-level languages. The burden is on the programmer, rather than the language designer, but we can get the best of both worlds for anything that you can do in a high-level language—while maintaining the real power of pointers in the rare cases where you need more.

In this book, I will explain the basic memory model that C programs assume about the computer they run on and how pointers let us access data anywhere in memory. I will explain how you get safe access to memory, by allocating blocks of memory you need, so they are yours to manipulate, and how you can release memory when you no longer need it, so you do not run out of memory before your computations are done. I will explain how pointers are essential for building complex data structures and how you can approach this in a structured way, so they are safe to use. And I will show you how you can use pointers to functions to implement higher-order functions and polymorphic data structures.

I will not cover basic C programming. This is not an introduction to programming or the language. I will assume that you already know the basics and will jump directly into memory and pointers. I will not cover issues related to concurrency and interruptions and such either. That would lengthen the book substantially, and there are already excellent books where you can explore this further.

CHAPTER 2

Memory, Objects, and Addresses

Everything you manipulate when you run a computer program, and the program itself, has to reside somewhere in your computer's memory—on a disk, in its RAM circuits, in various levels of cache, or in a CPU's or GPU's registers. It is not something we necessarily think about when we write programs, but it is an obvious truth: if objects aren't found *somewhere*, we cannot work with them. The reason we can get away with not worrying about memory is that our programming language handles most of the bookkeeping.

Consider the classical "Hello, world!" program:

```
#include <stdio.h>

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

We don't need to think about the computer's memory when we write it (or execute it). Still, many objects must necessarily be represented in memory before we can run the program—the program itself, including the `main()` function we write ourselves and the `printf()` function we get from the runtime system. The two arguments we give to `main()`, `argc` and `argv`, are stored somewhere, as is the constant string `"Hello, world!\n"`.

T. Mailund, *Pointers in C Programming*, https://doi.org/10.1007/978-1-4842-6927-5_2

Or consider a simple function for computing the factorial of a number:

```
int factorial(int n)
{
  if (n <= 1) return 1;
  else return n * factorial(n - 1);
}
```

When we call the function, we must store the argument, `n`, somewhere. In the recursive case, we call the function again, and in the second call, we need another parameter `n`. We need another one because we need to remember the current `n` so we can multiply it to the result of the recursion. Each recursive call must have its own `n` stored somewhere in memory.

We don't have to worry about where the functions, variables, and constants live in memory when we write this code because the C compiler will generate the necessary machine code to handle it for us. It will allocate the space for constants and variables, and it handles writing function parameters and assignments to variables into the correct memory locations. When we read the value in a variable, it handles getting it from the right memory location for us as well.

However, when we choose to program in a low-level language, like C, the raw memory is never too far away. It is possible to hide memory entirely from the programmer, to pretend that objects are floating around somewhere and never wonder about where that is. However, it comes at a computational overhead, and it limits what we can do with a program in some ways. Low-level languages do not do this. They let us get the memory of objects and manipulate the memory directly. We do not do this willy-nilly because if we did, we would write unmaintainable software. Still, we have the power, and when we use this power carefully, and in a structured way, we can build the features that high-level languages provide using a single mental framework and with little computational overhead.

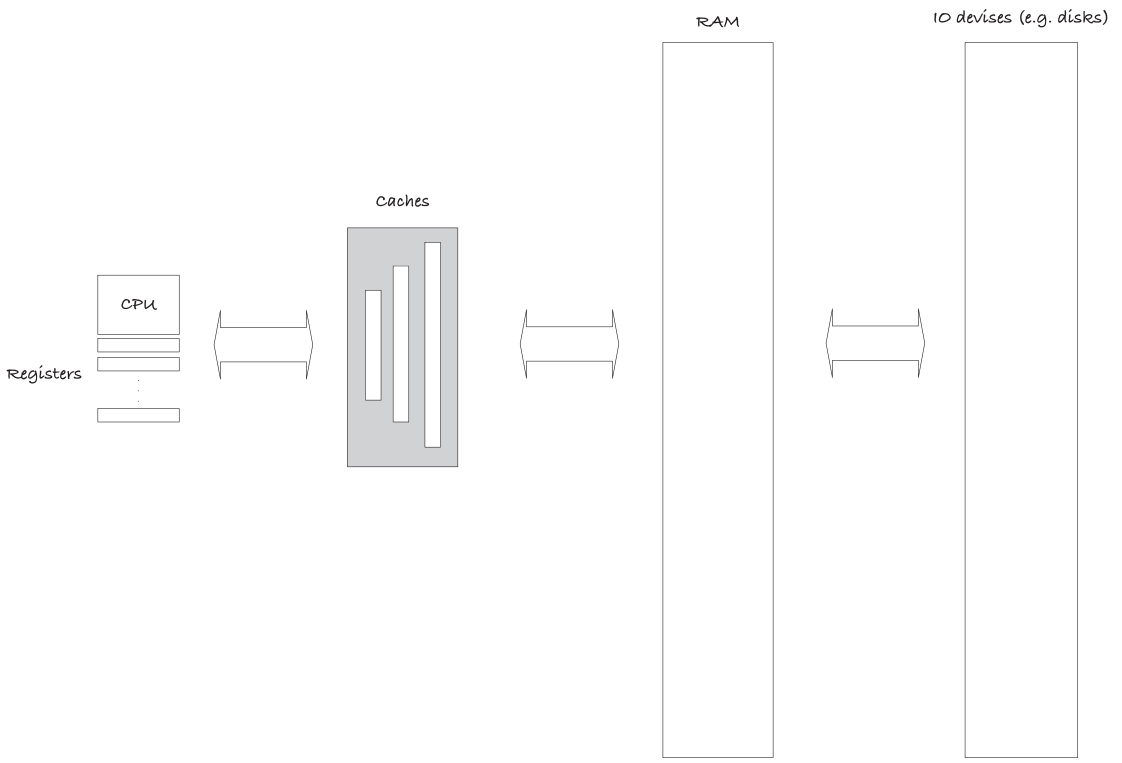

Figure 2-1. *Computer memory hierarchy*

Even though we work with low-level languages, we work with an abstraction of the computer's memory. A modern computer's memory is an immensely complex system, where data lives at different locations, and the time it takes to access it varies widely. A simplified model of a modern computer can look like that in Figure 2-1.

Objects that reside in a CPU's or GPU's registers are incredibly fast to access and manipulate. In comparison, accessing an object on a RAM chip takes geological ages. We cannot hold all the data we operate on in registers, there are too few of them, so we need to move data in and out of the CPU. To alleviate the long delay you get when the CPU has to access objects, the computer moves data you are currently working on into a cache, which the CPU can access faster than the main memory. When you switch to working on some other data, that goes into the cache, and the previous data goes back to main memory. When we need data from files, we usually write code that explicitly gets it from there, but if the computer runs out of main memory, it might also use the file system to swap data you are not using out of and data you are using into RAM.

Your hardware, operating system, and compiler work together to optimize the computational cost of memory access. Your compiler will analyze your programs and put objects in registers when possible. The computer's hardware will move objects from RAM into different levels of cache for faster access. If you are so unlucky that data needs to move to a disk, the operating system will handle that for you. We do not usually write programs that work on memory at this level of detail. It would be incredibly tedious to do, and we would write programs optimized for specific platforms. If you change the hardware, you have different levels of cache, with different performance trade-offs. Writing programs with an abstract memory model is hard enough; writing programs with the full complexity in mind would be close to impossible. We write programs with a simpler conceptual model of computer memory and let the compiler and hardware map from the simple model to the more complex.

In this book, we will pretend that there is only one level of memory, RAM. All data manipulation happens in the CPU, but the compiler will generate the necessary code to move data in and out of the CPU. We will not worry about this, but trust that it does this efficiently. An optimizing compiler is likely better at it than we are anyway, and it certainly is more efficient to write code if we do not worry about such low-level programming. So we will only worry about what our data is doing in that big block of RAM. This is close to how C's memory model work. If you write portable C, the language standard does not make many promises about what the memory looks like. Still, all objects sit in some memory, they have addresses that you can get, and if you have the address of an object, then you can manipulate that object. What you can actually do with the object depends on how you define it, but whatever you can do with an object, you can also do through its address.

The Memory of a Generic Process

The C standard doesn't specify how memory should be organized for running programs, but a typical process, that is, a running program, can look like Figure 2-2. At the lowest memory addresses, at the bottom, you have the code that the process runs. Code is data as well, it is the instructions that the CPU should follow, and it is part of the process' memory. Above that, you have the data that exists throughout the process' lifetime. When you declare global variables, they live as long as the program runs, and this is where they sit in memory. Some of this data will be read-only. There are constants defined in a program that you cannot change. String literals, those you define with `"..."`,

are usually immutable, they live in read-only memory, and your program might crash if you try to write to them. Global variables you define yourself, if not declared const, are mutable, and you can write to them. In the figure, I do not make a distinction between the two, but your data usually comes as both read-only and read-write.

On top of that, you have the memory that the program allocates (and deallocates) while it runs. We call this memory area the *heap*, and in Chapter 9 we see how you can allocate memory from it in C. When the process needs more memory, the heap grows upward. When it gets rid of memory, the situation is more complicated. We do not remove a block in the middle and move all the data above it down, that would be time-consuming, and we cannot move objects we have the address of—then they would have moved away, and so accessing the data through an address would not work. Not to worry, though, it is something that C's runtime system will handle for you. At the top, we have the *stack*. The stack handles function calls, and it is where local variables and function arguments live. It typically grows downward. Between the stack and the heap, there is usually a barrier, a piece of memory that you are not allowed to access. It is there to prevent the stack and heap to grow into each other.

The memory that a process sees is rarely the physical memory the computer has. Between a running process and the physical memory, the CPU creates a "virtual" memory. That is the memory space that the program works with, and each time it needs to access memory, the hardware will map the virtual address to a physical one. In the old days, physical and virtual memory was the same, and any program could read and write data anywhere and execute any code from anywhere. This is, obviously, highly unsafe. The virtual memory protects processes from each other and provides a more straightforward address interface to programs.

Programs need to allocate memory for the stack and heap to use it, which typically involves asking the operating system to get a chunk of memory, which in turn will set up this virtual to physical mapping. That is the addresses that the program can freely access. Even though you could, in theory, address the full address space, in practice, the hardware will cause an interrupt if you access data outside of the memory the program got allocated by the operating system. This will typically result in the OS terminating the process. Thus, if you haven't gotten permission to read or write from somewhere, and you do it anyway, then it can be the death of your program.

Similarly, there is usually protection on which memory you can execute. You should not execute random data, so you are prevented from that. And since there are obvious security problems if you allow a program to write into its code, modifying it potentially based on user input, the executable memory is often read-only.

When you write a C program, you are not given any guarantees for how the data is positioned in memory. You have the `register` keyword to tell the compiler that you would like a given variable stored in a register, but this is an anachronism more than anything else. It is only a suggestion to the compiler, and it is allowed to ignore it. Your compiler is better at allocating registers than most programmers, and it *will* likely ignore the keyword altogether. The only practical consequence of using it is that you are then not allowed to take the address of the variable (that would be inconsistent with wanting to keep it in a register). I suggest you never use this keyword. If you do not take the address of a local variable, then the compiler will put it in a register if that makes the most efficient code. Don't interfere with its register allocation.

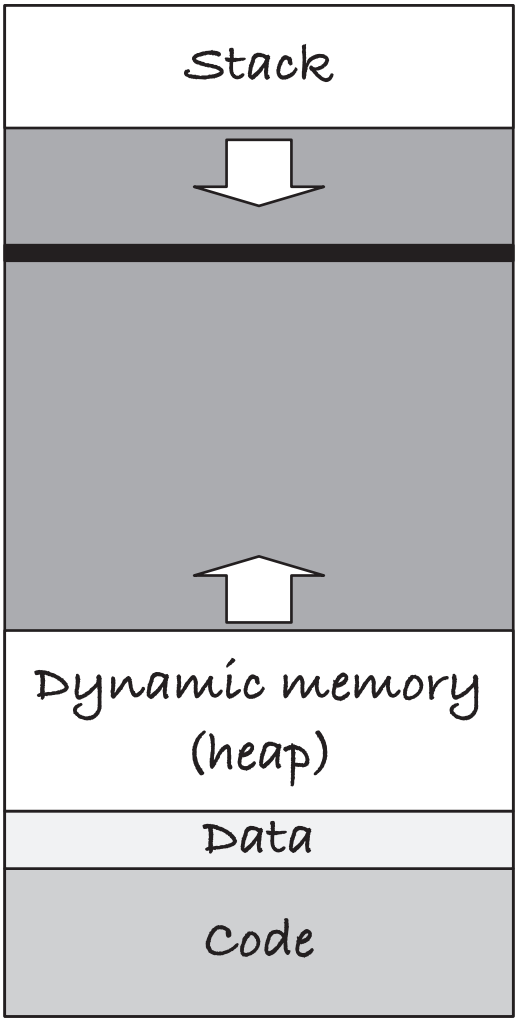

Figure 2-2. *A process' memory layout*

You likely have access to the system calls that lets you manipulate memory at the low levels described, but they are platform dependent, and code you write for one platform will not work on another. The interface to memory that C provides handles the interaction with the operating system, and if you want to write portable code, you should stick with that. Unless you have particular needs, that interface will do everything you need.

In portable C, you cannot assume that your program will run with a memory layout like that described earlier. C is designed to run on practically any hardware and any operating system, and the C standard thus makes few assumptions about the underlying platform. That being said, it is a useful mental model for thinking about your program's memory. You cannot assume that the stack lies at higher memory locations than the heap or that it grows downward instead of upward (and I honestly don't see when that would be relevant for you to worry about).

Even if you write your code in machine code, with full power to access memory as you please, you probably won't see exactly this layout. Addresses are usually scrambled by the architecture, as a defense against hacking attacks (it prevents an attacker from knowing where your code and data are, by randomizing it). If you write multithreaded programs, you need a stack per threat, and they can't all lie at the top of the process' address space. If you dynamically load libraries while executing your program, they need to go somewhere as well. That is code, but the code's location and size are already fixed in this model.

Still, there is a stack, and there is a heap—if not in reality, then conceptually—and I will present memory in this book as if we had processes like these. As long as you don't write your programs with this strong an assumption about the memory layout, it is a useful mental model of the memory you use and manipulate.

Objects, Sizes, and Addresses

While the C language doesn't describe how memory is organized, it does specify that each object has an address and a size. The address is where it sits, conceptually if not in fact, and its size is how many memory locations it takes up. By the C standard, each memory cell takes up one `char`, and larger objects take up more cells of memory. The C standard doesn't say what size a `char` actually is; it is just the minimum size of an object that we can put into one block.

You can get the size of an object using the `sizeof` operator. Try running this program:

```
#include <stdio.h>

int main(void)
{
  char c;
  printf("%zu %zu\n", sizeof(char), sizeof c);
  int i;
  printf("%zu %zu\n", sizeof(int), sizeof i);
  double d;
  printf("%zu %zu\n", sizeof(double), sizeof d);
  return 0;
}
```

I got

```
1 1
4 4
8 8
```

but the result will depend on your platform.

When we use `sizeof` on a type or a variable, we get the size of the type/object. Your result might vary from mine (I got size 1 for `char`, 4 for `int`, and 8 for `double`). The size of `char` is always one. That is guaranteed by the C standard. There are no other guarantees about the absolute size of other types, although there are some guarantees about the relative size of objects. For practically all modern hardware, a `char` is 8 bits, but the standard doesn't guarantee it. The constant `CHAR_BIT` will tell you how many bits a `char` contains in your own development environment, but I will be surprised if it isn't 8. If it isn't, then you are working on unusual hardware. If a `char` is 1 byte, that means that for my output, an integer is 32 bits (4 bytes) and a `double` is 64 bits (8 bytes).

All sizes are relative to the minimal size that C works with, and that is the size of a `char`. For the variables, you do not need the parentheses. You can write `sizeof c` instead of `sizeof(c)`. For the types, you do need the parentheses. If you want the size of an object or type related to a variable, that is, the variable itself or something it refers to in cases of structures or arrays, you should prefer to get the size through the variable.

You have specified the type when you declared the variable, and if you use the type once more with `sizeof`, you have two references to it. If you change one and not the other, you can get in trouble. It is better to specify the type once and get it automatically from the variable after that.

If you want to know the address at which a variable sits, you can put an ampersand, &, before the variable:

```
#include <stdio.h>

int main(void)
{
  char c = 1;
  printf("%d %p\n", c, (void *)&c);
  int i = 2;
  printf("%d %p\n", i, (void *)&i);
  double d = 3.0;
  printf("%f %p\n", d, (void *)&d);
  return 0;
}
```

The program prints the (integer) value of a character, the value of an integer, and the value of a `double`, together with the memory addresses where the variables sit. The formatting code `%p` gives us the text representation of the address when we call `printf()`. It will print the memory addresses. The `(void *)` cast is there because the `%p` wants a void pointer. We see more to those in the next chapter.

There are no hard rules for where C should put variables, nor is there any rule that says that you can meaningfully compare the address of objects you haven't allocated together. That being said, if you see that the printed addresses are numbers close together, then the addresses probably are. If your memory addresses are laid out in the process' memory locations as described in the previous section, the preceding program gives you where they sit. I got the result:

```
1 0x7ffee0d888ff
2 0x7ffee0d888f8
3.000000 0x7ffee0d888f0
```

which tells us that the double was put first in memory, then the integer, and then the character; see Figure 2-3. The memory locations are ordered from the bottom and up, so the integer, for example, sits at address 0x7ffee0d888f8 (bottom) to 0x7ffee0d888b (top).

The 8 bytes from 0x7ffee0d88f0 contain the `double`. Immediately after the `double`, we have the `int`. From the `sizeof(int)` call in the previous program, we know that an `int` takes up four memory cells on my machine, but there is a gap, the gray area, up to the `char`, found at address 0x7ffee0d888f0. C can put the variables where it wants, and you have no guarantee that they are consecutive for two separate variables. This layout is what I got on my computer when I translated the program with the compiler and options that I used. If I change any of the options, for example, change the optimization settings, things could look very different. Do not make assumptions about where individual variables are put in memory; the C standard does not make any promises. It only promises that your objects have an address and a size that is determined by its type.

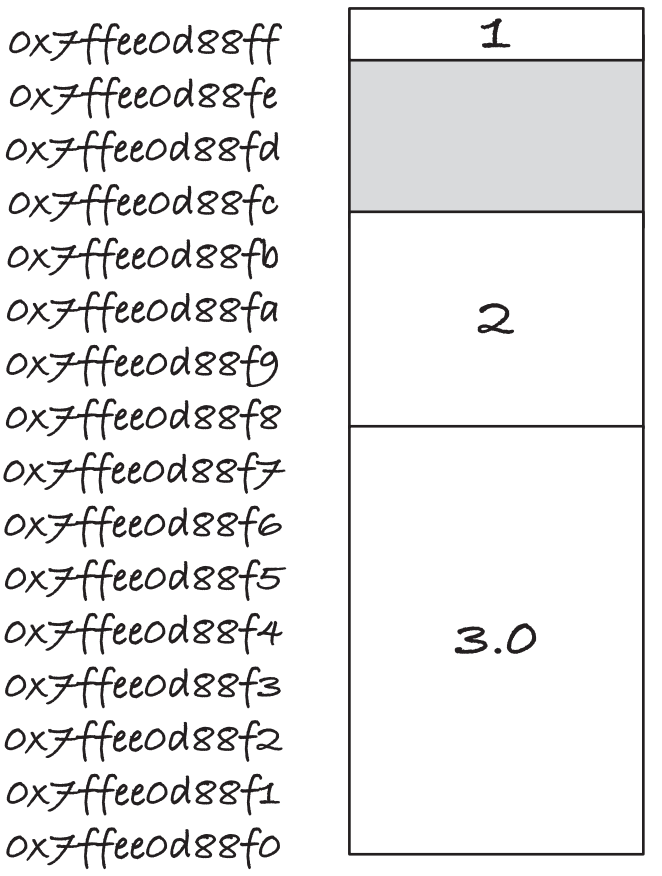

Figure 2-3. *Memory locations for a char, int, and double*

More technically, a block of memory you have allocated in a single operation has an address and a size. From the beginning of the allocated memory and up to its size, you have consecutive addresses, and you can meaningfully compare these addresses and reason about the memory layout. Memory that you have allocated independently, you should not make any assumptions about. Maybe you can use their addresses to work out where the memory sits relative to each other, or maybe you cannot. If you want to compare addresses, stick to looking at addresses within one allocated block.

Memory Allocation

What does it mean to allocate memory? How do we get the memory that our variables sit in? And how do we get more when we need it? Most memory management is automatic in C. When you declare a variable, the compiler generates code for allocating the memory to hold it. For global and static variables, it sets aside memory that will last as long as the program runs. For local variables and function arguments, which you can think of as the same thing, the compiler generates code to get memory for them when you call a function. This memory is allocated on the stack, and it only lives as long as the function call that allocated it. We return to stack-allocated memory later in the chapter.

Although it is a good bet that local variables sit near each other on the stack, you cannot make assumptions if you want your code to run everywhere. Individual variables are independently allocated, and then the language makes no promises about how they relate. But you can allocate more than one value at the same time, and then we get a few more promises.

There are different ways that we can allocate multiple objects at the same time. The simplest is through *arrays* that we will cover in detail in Chapters 5 and 6. An array allocates several objects of the same type and put them, one after another, in consecutive memory locations. In the following program, we allocate an array of five integers and get the addresses of the individual integers:

```
#include <stdio.h>

int main(void)
{
  int array[5];
  printf(" array    == %p\n", (void *)array);
  for (int i = 0; i < 5; i++) {
```

```
    printf("&array[%d] == %p\n", i, (void *)&array[i]);
  }
  printf("sizeof array == %zu\n", sizeof array);
  printf("5 * sizeof(int) == %zu\n", 5 * sizeof(int));

  return 0;
}
```

An integer takes up `sizeof(int)` memory addresses, so five of them takes up `5 * sizeof(int)`, and that is the size of the array. The integers lie in contiguous memory, with `array[i + 1]` `sizeof(int)` after `array[i]`; see Figure 2-4. The value of an array, the preceding `array`, is the address of the first of the integers.

The integers in the array are part of the same memory allocation, and you are guaranteed that they are structured this way in memory.

With dynamic memory allocation, the topic for Chapter 9, you explicitly allocate memory blocks of the desired size. There, as well, you have a block of memory where the addresses are contiguous. You can use them more freely than you can with arrays, but in practice, you use them either to store array-like data or to store `structs` and `unions`.

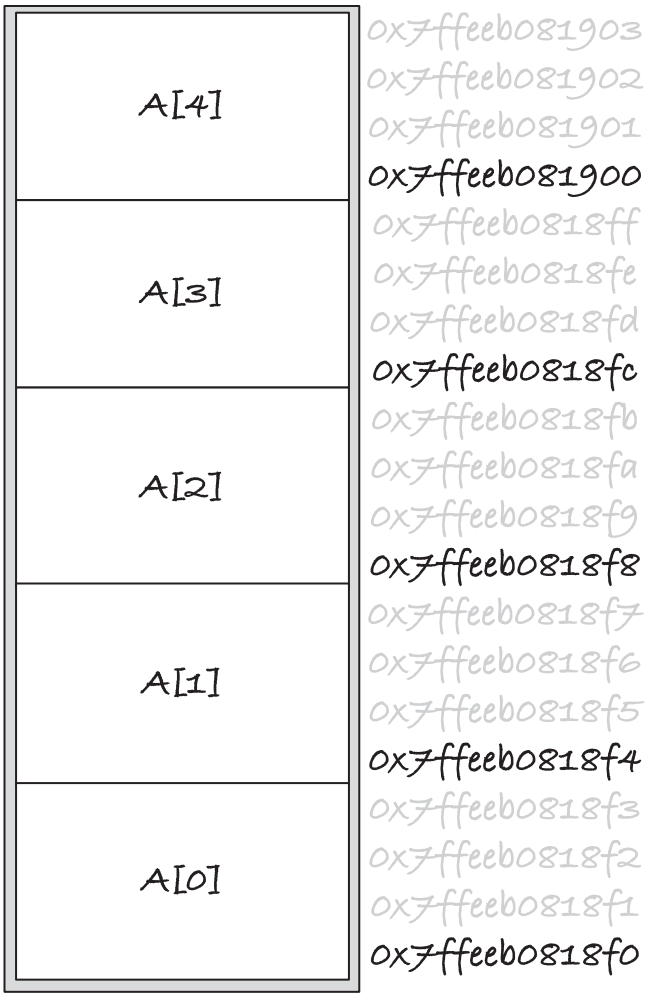

Figure 2-4. *Memory layout of an array*

With both `struct` and `union`, you have a single memory allocation when you declare a variable, but a `struct` usually contains more than one data type, and so does a `union` although its purpose is to store different types in the same memory location. When you define a variable of a `struct` or `union` type, you are guaranteed to get a chunk of memory of the relevant type's size that you can index as consecutive memory addresses. For unions, you get a block of memory that is large enough to hold the largest element, and all the elements sit at the first address in the union.

If you run this program

```
#include <stdio.h>

union data {
  char c;
  int i;
  double d;
};

#define MAX(a,b) (((a)>(b))?(a): (b))
#define MAX3(a,b,c) MAX((a),MAX((b), (c)))

int main(void)
{
  union data data;
  printf("sizeof data == %zu\n", sizeof data);
  printf("max size of components == %zu\n",
         MAX3(sizeof data.c, sizeof data.i, sizeof data.d));

  printf("data at   %p\n", (void *)&data);
  printf("data.c at %p\n", (void *)&data.c);
  printf("data.i at %p\n", (void *)&data.i);
  printf("data.d at %p\n", (void *)&data.d);

  return 0;
}
```

you might get something like

```
sizeof data == 8
max size of components == 8
data at   0x7ffeebd2c900
data.c at 0x7ffeebd2c900
data.i at 0x7ffeebd2c900
data.d at 0x7ffeebd2c900
```

A `double` is the largest of the three types (on my machine), and the union gets that size—but see the next section for more details about `union` sizes. All the elements in the union sit at the same address, the address of the union itself, but of course you cannot use them all at the same time. That is not the purpose of unions. You can treat the memory block that the union holds as all three of the types, but a union only holds one of the types at any given time. Therefore, they can store their data in the same memory block and at the same address.

For structures, you get the memory to hold all of the components at the same time, so their size is at least enough to hold all of them. The elements come, one after another, in the order you define them, and the first element is at the first address of the structure. However, between the elements in the struct, there might be unused memory.

When I run this program

```
#include <stdio.h>

struct data {
  char c;
  int i;
  double d;
};

int main(void)
{
  struct data data;
  printf("sizeof data == %zu\n", sizeof data);
  printf("size of components == %zu\n",
         sizeof data.c + sizeof data.i + sizeof data.d);

  printf("data at   %p\n", (void *)&data);
  printf("data.c at %p\n", (void *)&data.c);
  printf("data.i at %p\n", (void *)&data.i);
  printf("data.d at %p\n", (void *)&data.d);

  return 0;
}
```

I get the output

```
sizeof data == 16
size of components == 13
data at   0x7ffeec6988f8
data.c at 0x7ffeec6988f8
data.i at 0x7ffeec6988fc
data.d at 0x7ffeec698900
```

So the `struct` variable `data` takes up 16 memory addresses, even though the data in it only take up 13 bytes (or technically 13 `sizeof(char)`). The components come in order; first we have `c`, then `i`, and then `d` with `c` at the same address as the `struct`, but there is some padding between `c` and `i`; see Figure 2-5. If you rearrange the order of the elements, you get them in a different order in memory, but there is likely always some padding.

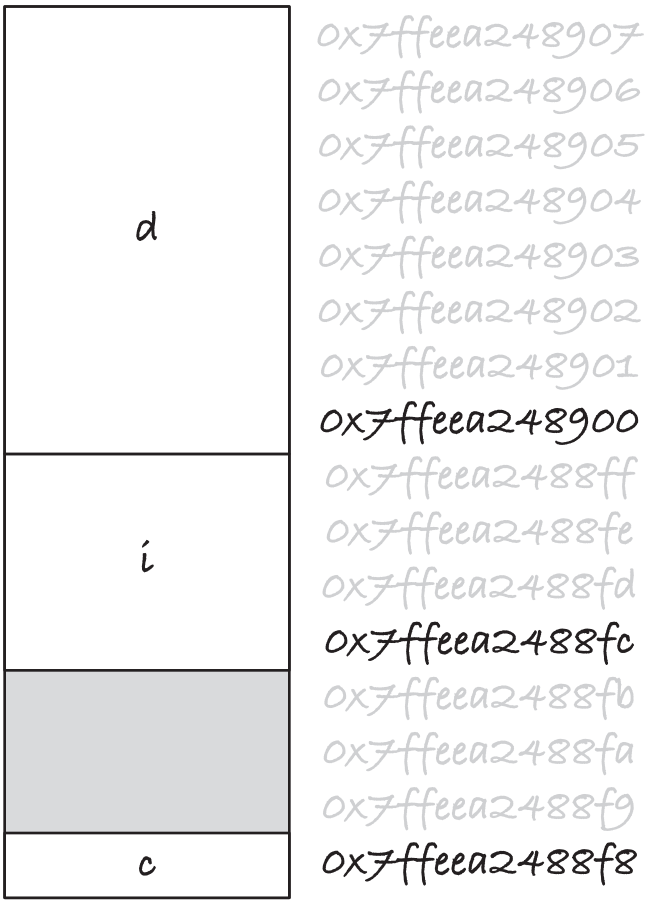

Figure 2-5. *Memory layout of a struct*

The padding might not only be between the components of the struct. You are guaranteed that the first address is where the first component sits, but there can be padding after the last components. If I move c to the bottom of the `struct`

```
struct data {
  int i;
  double d;
  char c;
};
```

I get the output

```
sizeof data == 24
size of components == 13
data at   0x7ffeef73a8f0
data.c at 0x7ffeef73a900
data.i at 0x7ffeef73a8f0
data.d at 0x7ffeef73a8f8
```

shown in Figure 2-6. The structure is now 24 long, with a gap between `i` and `d` and a segment of unused memory after `c`.

C does not give you many promises about how `struct` memory should look. The first element at the first address, the elements in order, and that is it. Why does it add this padding? It is not to be malicious. It has to do with memory *alignment*.

Alignment

In the abstract memory model, an address is just an address, and we can put any object there. An object takes up a certain amount of memory, say 4 bytes for a 32-bit integer, so if we put an integer at address *a*, then that address and the following three bytes is where the integer lives. However, on actual hardware, there is more structure to a computer's memory. The memory is not a sequence of bytes, but rather computer words of some given size, for example, 64 bits. The bus that carries data from memory to the CPU works with words of certain sizes. If you ask to get an integer from memory, and it sits in a single word, the computer needs to fetch that single word. If you put an integer at a location that spans more than one word, the computer has to fetch both words and then do some bit manipulation to put it into a register. And even if you ask for a 32-bit integer that sits inside a 64-bit integer, there might be more work for the computer to represent it as an integer in the CPU, if it doesn't sit at a certain offset in its word.