# Memory

A bit is a binary digit, either `0` or `1`. A byte is eight bits, and can thus represent 256 unique values, such as `00000000` and `10010110`. Computer scientists often think in terms of hexadecimal, rather than binary, because each hex digit maps to four bits, and thus a byte is more compactly represented in only two hex digits. The mapping is:

```
0  0000
1  0001
2  0010
3  0011
4  0100
5  0101
6  0110
7  0111
8  1000
9  1001
a  1010
b  1011
c  1100
d  1101
e  1110
f  1111
```

Thus, the two example values above would be `0x00` and `0x96`.

Think of your computer as a huge array of bytes; say, $2^{32}$ or $2^{64}$ of them, depending on the system. Everything, absolutely everything about your program must ultimately be represented in bytes and stored somewhere in memory. That applies even to code; your C source code compiles down to machine code, which has a very precise mapping into bytes, and is stored somewhere in memory for the CPU to read, one instruction at a time, and execute.

Each of your variables also resides somewhere. The computer (more specifically, the C compiler and runtime) decides where everything goes, and unless you care to find out, this organization usually doesn't matter to you.

This massive array, your memory, naturally has indices. As good computer scientists, we number from zero, so the lowest index of a byte in memory is `0x00000000`, and the highest possible location in memory (on a 32-bit system) is `0xffffffff`. The index of a particular chunk of memory is called its *address*; the `&` operator lets you determine the address of any variable, and the `%p` format to `printf` is specially tailored for printing them out (they are also just numbers, so you can print them out however you like).

`addresses.c`

```c
#include <stdio.h>

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

        printf("The address of i is %p.\n", &i);
        printf("The address of main is %p.\n", &main);

        return 0;
}
```

This program prints out the address of a local function variable and also the address of a function itself (remember that your code is sitting in memory too, encoded as machine instructions). It produces something like this:

```
The address of i is 0xbf85416c.
The address of main is 0x80483e4.
```

As discussed, each primitive type has a particular fixed width. C also has arrays; an array has a particular size and a particular type, and is space set aside for that many of those values. For example, if each `int` takes up four bytes somewhere in memory, an array of five `int`s takes up twenty contiguous bytes somewhere in memory. The address of the array is synonymous with the address of the first element; the address of the second element is four greater (because it falls right after the first, which consumes those four indices of memory).

This program demonstrates both the concepts above, and the syntax for working with arrays.

arrays.c

```c
#include <stdio.h>

int main(int argc, char **argv)
{
        int arr[5];

        printf("The address of arr is %p.\n", &arr);
        printf("The address of the first element is %p.\n", &arr[0]);
        printf("The address of the second element is %p.\n", &arr[1]);
        printf("Each element occupies %zd bytes.\n", sizeof arr[0]);
        printf("The whole 5-element array occupies %zd bytes.\n", sizeof arr);

        return 0;
}
```

The output:

```
The address of arr is 0xbf95d9cc.
The address of the first element is 0xbf95d9cc.
The address of the second element is 0xbf95d9d0.
Each element occupies 4 bytes.
The whole 5-element array occupies 20 bytes.
```

Note that `&arr`, the address of `arr`, is the same as `&arr[0]`, the address of `arr[0]` (which is the first element, remember counting from zero!). Also note that the second element, `arr[1]`, falls immediately after `arr[0]`, and the size of the whole array is the number of elements times the size of each.

Arrays in C are fixed size and always homogeneous; you can not grow or shrink an array, and you can not have an array of values some of which are `int`s and some of which are `double`s.

This is a handy time to introduce pointers. A pointer type is an integer type big enough to store indexes into memory, that is, addresses. Thus, the type `int *` is for values that are addresses, specifically the addresses of `int`s.

In addition to simply storing the results of the address-of `&` operator, you can also go the other way, and turn the address back into a value, with the dereference `*` operator. If you have an `int` pointer `p`, the expression `*p` evaluates to the value of the memory at the address stored in `p`, as an `int`. You can also assign to `*p` to change that part of memory.

pointers.c

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5          int i = 0;
6          int *p = &i;
7
8          printf("i = %d\n", i);
9          printf("*p = %d\n", *p);
10
11         i = 1;
12
13         printf("i = %d\n", i);
14         printf("*p = %d\n", *p);
15
16         *p = 2;
17
18         printf("i = %d\n", i);
19         printf("*p = %d\n", *p);
20
21         return 0;
22 }
```

As you might imagine, this produces:

```
i = 0
*p = 0
i = 1
*p = 1
i = 2
*p = 2
```

In addition to assigning a pointer from the address of another variable, you can also allocate anonymous portions of memory of arbitrary size with `malloc`, a standard function found in `stdlib.h`. Thus:

pointers2.c

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

int main(int argc, char **argv)
{
        int *p;

        p = malloc(sizeof *p);
        if (!p) {
                fprintf(stderr, "Could not allocate memory!\n");
                return 1;
        }

        *p = 2;

        printf("*p = %d\n", *p);

        free(p);
        return 0;
}
```

It is possible for you to run out of memory, in which case `malloc` returns a so-called null pointer; null pointers are guaranteed to be false, so we can check for failure. Note that at the end, we `free` the memory we allocate. Every single time you call `malloc`, it returns a new chunk of memory of the size you request, and every single one of them must be `free`d. This is crucial, and failure to do so results in memory leaks.

Now for a little secret: Pointers and arrays are the same thing.

Some simple arithmetic is allowed on pointers. For example, with an `int` pointer `p`, `p + 1` evaluates to the address of the next `int` in memory after `p`'s target (thus, the actual address increases by 4, not 1; it is 1 value, not one address). Naturally, `p + 2` evaluates to the one after that, and, obviously, `p + 0` is `p` itself.

The syntax `a[i]`, used for array indices, is in fact syntactic sugar for `*(a + i)`. Thus, here is the array-of-five-`int`s example, with `malloc`.

arrays2.c

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

int main(int argc, char **argv)
{
        int *arr;

        arr = malloc(5 * sizeof arr[0]);
        if (!arr) {
                fprintf(stderr, "Could not allocate memory!\n");
                return 1;
        }

        printf("The address of arr is %p.\n", arr);
        printf("The address of the first element is %p.\n", &arr[0]);
        printf("The address of the second element is %p.\n", arr + 1);
        printf("Each element occupies %zd bytes.\n", sizeof *arr);

        free(arr);
        return 0;
}
```

Finally, a second little secret: Strings are not a fundamental type in C, but rather are simply arrays of characters. Each character takes up one byte (the size of a `char`), and there is always an implicit null character `'\0'` at the end, so that you know how long the string is.

Here, for example, is a simple re-implementation of the standard `strlen` function (the real one is found in `string.h`). It simply counts the number of actual characters in the string, not including the terminal null.

strlen.c

```c
#include <stdio.h>

unsigned int strlen(char *str)
{
        unsigned int i = 0;
        while (str[i] != '\0')
                ++i;
        return i;
}

int main(int argc, char **argv)
{
        char greeting[] = "Hello, world!\n";

        printf("The greeting occupies %zd bytes.\n", sizeof greeting);
        printf("It has %u characters.\n", strlen(greeting));

        return 0;
}
```

Another real-life example of arrays and strings is one we have been ignoring so far: the arguments to `main`. The `argv` argument is a `char` pointer pointer; in fact, it is an array of strings (an array of arrays of characters). How big is the array? Naturally, `sizeof` would only tell us the size of a pointer. Thus, we also get `argc`, the count of the arguments.

Here is a very simple program similar to the standard `echo` utility, which just prints its arguments.

echo.c

```c
#include <stdio.h>

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

        for (i = 0; i < argc; ++i)
                printf("%s\n", argv[i]);

        return 0;
}
```

The first argument, `argv[0]`, is the name of the program as you typed it; the real `echo` utility doesn't print it, but this one does, for demonstration purposes.

```
$ ./echo
./echo
$ ./echo blah
./echo
```

```
blah
$ ./echo foo bar
./echo
foo
bar
```