Pointers and dynamic memory management in C

Jakob Rieck

Proseminar C - 2014

Contents

1 Introduction 2

2 Pointers 4
   2.1 Declaration & Initialization . . . . . . . . . . . . . . . . 5
   2.2 Using pointers . . . . . . . . . . . . . . . . . . . . . . . . 6
   2.3 Special pointer (types) . . . . . . . . . . . . . . . . . . . 7
   2.4 Pointer arithmetic . . . . . . . . . . . . . . . . . . . . . . 7

3 Dynamic Memory Management 8
   3.1 Memory allocation . . . . . . . . . . . . . . . . . . . . . . 9
   3.2 Resizing memory . . . . . . . . . . . . . . . . . . . . . . . 10
   3.3 Using dynamic memory . . . . . . . . . . . . . . . . . . . 10
   3.4 Memory deallocation . . . . . . . . . . . . . . . . . . . . . 11

4 Problems involving pointers and dynamic memory manage-
  ment 12
There are four regions where data and variables are stored during the execution of a program: There is the data region, which is home to global variables that are preinitialized data. There is also another segment for global variables which are left uninitialized called .bss. Both of these segments are static in size and are therefore of no particular interest to me in this article. Additionally, there is the stack and the heap. The stack is a LIFO data structure that is being used to store local variables, function...

---

parameters and return addresses. When a function is called, a new so called
stack-frame is created which is big enough to encapsulate all local variables
in the called function’s scope. On return, the function’s stack-frame is de-
stroyed, rendering all local variables inaccessible.\(^2\) Even though the stack
can grow, there is a rather low size limit of a few megabytes. (Documenta-
tion is scarce, but in my tests, the default stack size limit seems to be just
8 megabytes on Linux and Mac OS X). This makes the stack unsuitable for
large data sets. Additionally, due to the way the last stack frame is destroyed
on return, one cannot, for example, return an array that had been declared
as a local variable, because in C, arrays are returned as pointers to their first
element. Since the memory should no longer be considered accessible after
the function returns, one cannot access said array. The so called heap offers
a way to get around these limitations: For the sake of this article, the heap
can be viewed as a blackbox that gets managed by the C standard library.
The standard library offers a few procedures to request, resize and deallocate
memory blocks of arbitrary size from the operating system (OS). In order to
use this memory effectively, a basic understanding of pointers is necessary.
This paper will explain what a pointer is, what pointers are needed for and
how to use them to effectively use dynamic memory management in C.

org/gustavo/blog/post/anatomy-of-a-program-in-memory/ (visited on
Apr. 29, 2014).
## 2 Pointers

A pointer in C is a variable that acts as a reference to a data object or a function.\(^3\) Most of the time, the actual value of the pointer is totally meaningless in itself, because the pointer’s value is the address of the data object of interest. In C, a pointer encapsulates both the address of an object and the type of that object.\(^4\)

In the example above, \(i\) is just a normal integer variable that gets assigned the value 42, which is reflected in the middle of the figure where the contents of memory are displayed. \(ptr\) is a pointer to an integer that gets assigned the address of \(i\). The pointer’s value is thus \(0x0000100C\) in this example - It points to \(i\).

Pointers are used for lots of things in C: They are required to implement dynamic data structures like linked lists and trees that grow and shrink depending on usage. In the case of a linked list, a pointer would point to the next element in the list. A tree would have pointers to its children. Furthermore, pointers are used to implement call-by-reference: Normally, a function in C works on copies of its parameters, which means that the function cannot change the original value. Using pointers as parameters lets us avoid this problem: We simply pass the address of a variable into a function, and said function can now modify the original, underlying memory. For an

---

\(^3\)Peter Prinz/Tony Crawford: C In a nutshell, 2006, p. 122.

\(^4\)Ibid., p. 122.
example, see 'Call-by-reference by example' below. Apart from the aforementioned use cases, dynamic memory management in C also relies heavily on pointers.\(^5\)

2.1 Declaration & Initialization

To declare a pointer, use the following syntax:

\[
\text{type } \ast [\text{cv-qualifiers}] \text{ pointerName } [= \text{ expression}];
\]

A pointer declaration consists of a type, that can itself be a pointer to a type, the asterisk (*) to denote that we are dealing with a pointer, a name and, optionally, an expression to initialize the variable.\(^6\) If you choose not to initialize a pointer explicitly, one of two things happen: Just like with normal variables, the value of a pointer with automatic storage duration is undefined. For all other pointers, the starting value is NULL.\(^7\) You can also choose to specify type qualifiers like const and volatile, just like you can for every other variable.

Valid declarations include the following ones:

<table>
<thead>
<tr>
<th>Listing 1: Sample pointer declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 int \ast ptrToInt = NULL; // pointer to int</td>
</tr>
<tr>
<td>2 int ** ptrToPtrToInt = &amp;ptrToInt; // pointer to pointer to int</td>
</tr>
<tr>
<td>3 float \ast ptr = NULL; // pointer to float</td>
</tr>
<tr>
<td>4 // pointer to pointer to ... to int</td>
</tr>
<tr>
<td>5 int **************** a;</td>
</tr>
</tbody>
</table>

There is no limit on the number of indirection levels one can take: declarations like the one in line 4 in the example above are perfectly legal and help to illustrate this point, but are (hopefully) never found in real code. The &-operator is known as the *address-of* operator and yields the address of a variable. It can be used to initialize a pointer, as seen in the example above in line 2.

\(^5\)Prinz/Crawford: C In a nutshell (see n. 3), p. 122.
\(^6\)Ibid., p. 122.
Given a variable a of type type, &a yields the address of a, which is of type type *

2.2 Using pointers

As you already know, the actual value of a pointer, the address it is pointing to, is rarely of interest. Much more important is the value the pointer points to. Dereferencing a pointer allows the programmer to access that value. The asterisk (*) operator (sometimes also called indirection operator)\(^8\) is used for this. The following example demonstrates this concept.

```c
#include <stdio.h>

/* swap swaps the integers at a with the one at b */
void swap(int * a, int * b) {
    int c = *a;
    *a = *b;
    *b = c;
}

int main(int argc, char *argv[]) {
    int a = 42, b = 21;
    swap(&a, &b);
    printf("a = %d, b = %d\n", a, b); // prints a = 21, b = 42
    return 0;
}
```

"dereference" (*) and "address-of" (&) are complementary operators. Pointers can be compared for equality using == and !=, however the types of the two operands have to match or at least one one of them has to be either the NULL pointer or a pointer of type void *.\(^9\) Pointers to the same type can also be compared using >, <, <= and >=. The semantics of these operators are closely related to arrays and are discussed under pointer arithmetic.

\(^8\)Kernighan/Ritchie: The C Programming Language (see n. 7), p. 78.
\(^9\)Prinz/Crawford: C In a nutshell (see n. 3), p. 128.
2.3 Special pointer (types)

In an example above, I used `NULL` to initialize a pointer. This is a constant that can be used to initialize any pointer type. It signals a pointer does not point to a valid memory location. A pointer whose value is `NULL` may never be dereferenced.\(^{10}\)

There is a special pointer type, `void *`, that, contrary to intuition, does not refer to a pointer to nothing, but rather to a pointer to some unknown type. It is possible to assign any `void *` pointer to any other pointer type (and the other way around). As a `void *` pointer does not include the type information normal pointers carry around, it is not possible to dereference such a pointer.\(^{11}\) After all, what should be its value? An integer? A float? There is no right answer.

2.4 Pointer arithmetic

So far, we have learned how to use the "address-of" operator (&) to assign a value to a pointer, how to dereference (*) a pointer and how to compare two pointers for equality.

When you declare an array of length \(n\) and type `type`, all you really do is reserve a continuous block of memory, large enough to hold \(n\) objects of type `type`. The individual elements are placed in sequential order in memory, with no other data between them.\(^{12}\) If you assign the address of an element of an array to a pointer, you can decrement or increment that pointer to go back and forth in the underlying array (by one index). Additionally, it is possible to add an integer (both positive and negative ones) to a pointer. Consider the following declarations:

```c
1 int arr[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
2 int *ptr1 = &arr[0]; // assign the address of the first element of arr to ptr1
3 int *ptr2 = &arr[5];
```

We first declare an array `arr` of ten integers with `arr[i] = i` and then assign the address of the first element of that array to `ptr1` and the address of the sixth element to `ptr2`.

`*ptr1` can now be used to access `arr[0]`. Because arrays are continuous chunks of memory, you can add an integer \(n\) to `ptr1` and dereferencing

\(^{10}\)Prinz/Crawford: C In a nutshell (see n. 3), p. 124.
\(^{11}\)Ibid., p. 124.
\(^{12}\)Ibid., p. 111.
the resulting pointer allows us to access \texttt{arr[n]}. Subtraction of integers works just the same: \(^*(\texttt{ptr1} - n)\) is the same as \texttt{arr[-n]}. Suppose \texttt{ptr1} would not point to the start of the array, then accessing negative indices yields values at lower indices. Because addition is commutative, the following holds true: \texttt{ptr1 + n = n + ptr1} and thus \(^*(\texttt{ptr1} + n) = \texttt{arr[n]}\). Finally, \(\texttt{*(ptr1 + n) = arr[n] = arr[p] = *(n + ptr1)}\). This last statement may look confusing, but array subscripting is actually defined as pointer arithmetic.\(^{13}\) The comparison operators introduced above can be used to check if one pointer points to an element at a higher- or smaller index than another pointer. In the example above, \((\texttt{ptr1} < \texttt{ptr2})\) would be true because \texttt{ptr1} points to the first element, whereas \texttt{ptr2} points to the sixth.

Another legal operation is to subtract two pointers of the same type that point to elements in the same array. The result is an integer that reflects the index difference between the two pointers. Suppose \texttt{ptr1} references \texttt{a[n]} and \texttt{ptr2} references \texttt{a[n+i]}. Then \texttt{ptr2 - ptr1 = i}.\(^{14}\) The following example is a snippet you can actually try out for yourself demonstrating the concept.

```c
int *a, *b;
int arr[10] = {0,1,2,3,4,5,6,7,8,9};
a = &arr[0];
b = &arr[9];
printf("%d\n", b - a); // prints 9
```

### 3 Dynamic Memory Management

Dynamic memory management in C consists of three important aspects: Firstly, one has to request a pointer to a block of memory large enough to work with. The memory can then be accessed through the pointer, either by relying on pointer arithmetic or, if you prefer, using the familiar array syntax, until it is manually free’d by the programmer. Dynamic memory management in C is a manual process, so you are also responsible for cleaning up after you. In the following sections, I am going to explain the basic procedures required to allocate- (request) and deallocate (free) memory.

\(^{13}\)Programming Languages - C ISO/IEC 9899, second edition, 1999, p. 70.
\(^{14}\)Prinz/Crawford: C In a nutshell (see n. 3), pp. 127-128.
3.1 Memory allocation

Memory allocation in C is handled by `malloc()` and `calloc()`. `Malloc` (and every other memory related procedure discussed here) is declared in `stdlib.h`, so be sure `#include` that file. `malloc`’s declaration is as follows:

```c
void * malloc(size_t size);
```

`malloc` has one parameter named size of type `size_t`, an unsigned integer type that is commonly used to hold information about the size of data objects in memory. In this case, `malloc` wants to know how many bytes you want to allocate. Because the size of various data types like int, long and short varies from system to system, you should make use of the `sizeof` operator, which returns the size of an object or type in bytes. The following snippet illustrates this concept:

```
Listing 3: malloc() usage and error handling

// Allocate memory large enough to hold 40 int
int * ptr = malloc(40 * sizeof(int)); // Preferred way
if (ptr == NULL) {
  // error handling
} else {
  // continue normal execution.
}

/* The following line might work on 32-bit (= 4 byte) machines, but is not portable */
int * ptr = malloc(40 * 4);
```

On failure, `malloc()` returns NULL, otherwise it returns a typeless pointer that points to the beginning of the allocated block in memory. The return value is implicitly converted into the needed pointer type (in the example above we assigned the return value of malloc to a pointer to int!). The contents of the block of memory are undefined, so do not rely on any particular value.\footnote{Prinz/Crawford: C In a nutshell (see n. 3), p. 168.}

`calloc()` has a slightly different declaration:

```c
void * calloc(size_t count, size_t size);
```

In contrast to `malloc()`, `calloc()` takes two parameters: count is the number of elements you want to allocate and size is the size of the
type of the elements you want to allocate. Furthermore, calloc() actually initializes the memory it allocates to zero, before returning the pointer. Just like malloc(), calloc() returns NULL on failure.

3.2 Resizing memory

realloc() can be used to resize a memory block. It’s declaration is as follows:

```c
void * realloc(void * ptr, size_t size);
```

realloc() takes two parameters, ptr, a pointer previously returned by either malloc(), calloc() or realloc() itself, and a size parameter that specifies the new size of the memory block. realloc() first tries to extend the available memory at the address you passed in via ptr and if there is not enough memory available directly following the pointer you passed in, realloc allocates a new block of memory, copies over the data from the old buffer, free()'s the old buffer and returns a pointer to the new buffer. Just like malloc() and calloc(), realloc() returns NULL to indicate failure.

3.3 Using dynamic memory

Memory allocated using malloc() or calloc() can be accessed only through the returned pointer. You can dereference the pointer to read and write the first element, or add an integer n to the pointer and dereference the resulting pointer to access the n-th value. If you prefer array syntax over pointer arithmetic, you can also use that. (See example below)

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

/*
 * This program uses its first argument from the command line, converts it to an integer n, sums up 1..n and
 * prints the sum of those numbers.
 * The algorithm is deliberately ‘dumbed-down’ to illustrate
 * the relationship between arrays and pointer.
 */

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

```

16Prinz/Crawford: C In a nutshell (see n. 3), p. 170.
if (argc < 2)
    return EXIT_FAILURE;

// strtol converts its first argument, a string into an
// integer by trying to interpret the value as base 10
// in this case.
int num_elements = strtol(argv[1], NULL, 10);

int * arr = calloc(num_elements, sizeof(int));
int sum1 = 0, sum2 = 0;

if (arr == NULL)
    return EXIT_FAILURE;

for(int i = 0; i < num_elements; i++) {
    // Access using array syntax
    arr[i] = (i + 1);
    sum1 += arr[i];

    // Access using pointer arithmetic
    *(arr + i) = (i + 1);
    sum2 += *(arr + i);
}

free(arr);

if (sum1 != sum2) {
    printf("I can’t tell you what the sum of the first n natural number is...
    ");
} else {
    printf("The sum of the first n natural numbers is %d
", sum1);
}

return EXIT_SUCCESS;

3.4 Memory deallocation

On program termination, all dynamic memory allocated using malloc() is returned to the OS. However, it is good practice to explicitly deallocate the
memory using free() once you are done using it. Otherwise, in case your program runs longer than you think or is stuck in a loop you are wasting memory that could be used much more efficiently by the OS.

Here is free()'s declaration:

```c
void free(void *ptr);
```

free() takes a pointer previously obtained by a call to malloc() and deallocates the memory. free() returns no value.\(^{17}\)

### 4 Problems involving pointers and dynamic memory management

- There is no implicit bounds checking in C, buffer overruns (or underruns) are illegal and result in undefined behaviour.\(^ {18}\)
- The default value of pointers in automatic storage duration is undefined, not NULL, so checking for NULL is not enough to make sure you are allowed to reference the pointer in question!
- Your program will crash if you dereference the NULL pointer so make sure this cannot happen. Check malloc()'s return value to make sure it did not fail.
- Be aware of operator precedence: \((\ast ptr)++ \neq \ast (ptr++)\). When in doubt, just use parenthesis to convey meaning.\(^ {19}\)
- Always free() pointers returned by malloc() explicitly, otherwise you could end up consuming way more memory than you really need.
- Make sure not to lose track of any pointers returned by malloc() that have not been free()d. If you loose such a pointer, the memory will not be returned to the OS until your program exits.
- Make sure to never dereference a previously free()’d pointer. The memory there does not belong to you anymore.
- Pointer declarations and types can be quite complex and difficult to read. If you have trouble understanding a declaration, use cdecl on Linux machines or visit http://www.cdecl.org online.

\(^{17}\)Prinz/Crawford: C In a nutshell (see n. 3), p. 170.
\(^{18}\)Kernighan/Ritchie: The C Programming Language (see n. 7), p. 84.
\(^{19}\)Ibid., p. 79.
References

Prinz, Peter and Tony Crawford: C In a nutshell, 2006.