A Guide to Mastering the Nuances of C

Erik Anderson, Kaylin Finke, Nick Nestor, Robin Cosbey,
Chris Riley, Brandon Tarquinio, and Ted Weber

May 2, 2019

Contents

Chapter 1

Memory and C

C's model of memory is a large, one-dimensional array of data. An address in memory can be thought of as an index into this array. Unlike many other higher level languages, C has much more direct control over memory and memory allocation. The programmer, not the environment, is expected to understand which sections of memory are to be used and to not mistakenly wander into other sections, as well as to explicitly say when they are finished using it.

1.1 Memory Layout

The main advantage of C is the ability to have extreme, granular control over the memory of the program. Before we begin, it is important to explain how the memory is organized in a C program. When you run a C program, the computer provides an address space that it can work in[1]. This includes variables, function calls, and the literal code that will execute to run the program. The two major components in this are the **stack** and the **heap**.

1.1.1 The Stack and Heap

The main working area of the program and the section of the address space where the program will run is called the stack. Every function call made gets its own block of the stack, as shown in Figure 1.1. Once the function call is completed, the area of the address space that holds the function is erased to make room for more calls. This includes removing any local variables or data that the function has initialized.

While the stack holds temporary memory, the heap is the permanent memory segment of the program. The heap allows you to designate places in address space that are permanent for the entirety of the program. For example, a program contains several functions that all need access to an array. The best way to ensure that the array is not erased after each function call and remains in the same location in memory is to designate a permanent space in memory to hold the array. This does not allow for the loss of any data after the function call is completed.

[1]See: https://en.wikipedia.org/wiki/Virtual_address_space

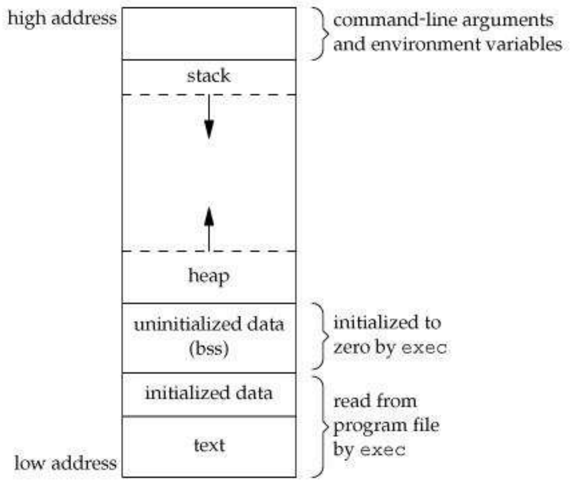
high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

Figure 1.1: Layout of memory allocation in C

1.1.2 Garbage Collection

Having explicit control over memory leads to the issue of **garbage collection**. In some higher level languages, the compiler is able to figure out what data or structures are no longer in use and remove them. In C it isn't that easy. Old data and structures will remain on the stack and heap until they are explicitly removed.

1.1.3 Pass By Value

C is a pass by value language. This means that when a function foo(x) is called, foo(x) gets a copy of x, rather than x itself. If foo were to add 1 to x, the value of x outside of the function would remain unchanged.

1.2 Variables

When a function is invoked, it gets placed on the stack, performed, and then is forgotten. This becomes problematic if the user wants to recall variables from that function after it has finished.

1.2.1 Initialization

The way to fix this problem is to store the variables within the heap as global or static variables so they can still be referenced after the function call ends. An initialized variable is stored within the data segment while uninitialized are stored in the BSS segment.

```
static int k = 12; //static variable stored in the data segment
static int l; //static variable stored in the BSS segment
```

When k is declared, the OS is declaring a spot in memory, 4 bytes on x64 linux, and putting the value 12 within it. However, when l is declared, no value is assigned so the OS chooses a 4 byte spot with whatever was previously in that location unchanged. In general, it is important to initialize variables when they are created.

1.3 Addresses and Pointers

An address is the location of the actual bytes used to store variables in memory and are represented as a hex value. This allows for direct access to a variable and the ability to change that variable, instead of changing a copy of the value. To get the address of a variable, the "&" is used. The value of a pointer is the address of another variable. It is denoted with a "∗".

```
int a = 4; //declares an integer in main's stack frame with the value 4
int *b = &a; //assigns b as a pointer to the location of a
```

```
printf("%p\n", b); //prints the address b points to
printf("%p\n", a); //prints the value of a
printf("%p\n", *b); //prints the value of what b points to
```

Since b is a pointer, printing b will produce the address, in hex, of a. The value of a, which is 4, will be printed when either a or $*b$ are given to printf because b is pointing at a's location. Calling $*b$ will produce the value that the location b is pointing at.

```
int* square(int p){
    int q = p*2;
    int* qq = &q;
    return qq;
}
```

```
int *c = square(3);
printf("%d\n", *c); //prints the value c points at, which is 6
int *d = square(8);
printf("%d\n", *c); //after calling square() again, this will print 16
printf("%d\n\n", *d); //prints the value d points at, which is 16
```

The example above shows the impermanence of the stack; the value that $*c$ pointed to is overwritten by the second call. Printing $*c$ a second time should produce 6 but instead 16 is printed.

```
int* improved_square(int r){
    int s = r*2;
    int* ss = (int *) malloc(sizeof(int));
    *ss= s;
    return ss;
}
```

```
int *e = improved_square(3);
printf("%d\n", *e); //prints the value e points at , which is 6
int *f = improved_square(8);
```

```
    printf("%d\n", *e); //prints the same value of e, which is 6
    printf("%d\n\n", *f); //prints the value f points at, which is 16
```

In this case, malloc fixes the issues with stack instabilities from the previous example. Using malloc points f to a different location than e. This will be explained further in Chapter 2.

Chapter 2

Dynamic Memory Allocation

C is a language that forces the decision of dynamic memory allocation on you more than other programming languages. Although this does require extra time to ensure error-free code, there can be benefits as well. Dynamic memory allocation is the idea that the programmer is performing the memory management manually. This is done in C by utilizing the stdlib library functions **malloc**, **calloc**, **realloc**, **sizeof**, and **free**.

2.1 malloc and free

We will begin by discussing the most common function for dynamic memory allocation, **malloc**.

```
void *malloc(size_t size)

Parameters:
    size - size, in bytes, of the memory block you need.

Returns on success:
    A void * to the start of the memory you requested.
Returns on failure:
    A Null pointer.
```

This will allow us to get memory from the heap to use. In Java all memory allocated from the heap using the keyword **new** is *Garbage Collected* when it is no longer needed. In C you have to manually clean up your own garbage using the following function:

```
void free(void *ptr)

Parameters:
    ptr - Pointer to previously allocated memory (from a call to malloc,
    calloc, or realloc) that the programmer wishes to free.

Purpose:
    This function returns the memory allocated to the pool of available
    memory so it can be used for other things.
```

```
Notes:
   Do not call multiple times on same pointer or on a pointer to memory
   that was not allocated using malloc, calloc, or realloc.
```

Here is an example using **malloc** and **free**:

```
   #include <stdio.h> // Needed for printf() and constants used.
   #include <stdlib.h> // Needed for malloc() and free().
   #include <string.h> // Needed strncpy() and strlen().

   int main() {
      const int str_len = strlen("RandomString");
      char *string = (char *) malloc(sizeof(char) * (str_len + 1));
      if(string != NULL) {
         strncpy(string, "RandomString", str_len);
         printf("String: %s \n", string);
         free(string);
         return EXIT_SUCCESS;
      }

      return EXIT_FAILURE;
   }
```

The result of running the previous program is:

```
String: RandomString
```

This is a slightly contrived example. A more common one is to return a new string or array from a function. Note that the memory returned from **malloc** can contain garbage in it, this is fine for most purposes but when it is not you can use the function **memset** from string.h to set every byte of memory to whatever value you need. You can also use **calloc**, which we describe next, instead of **malloc**.

2.2 calloc

This function is very similar to **malloc** except that **calloc** will set every byte of the memory allocated to 0 for you.

```
void *calloc(size_t n, size_t size)

Parameters:
n - number of elements allocated.
size - size, in bytes, of each element.

Returns on success:
   A void * to the start of the memory you requested.
Returns on failure:
   A Null pointer.
```

Here is an example using **calloc**:

```
#include <stdio.h> // Needed for printf() and constants used.
#include <stdlib.h> // Needed for calloc() and free().
#include <string.h> // Needed strncpy() and strlen().

int main() {
    const int str_len = strlen("RandomString");
    char *string = (char *) calloc(str_len + 1, sizeof(char));
    // All (str_len + 1)(sizeof(char)) bytes in memory, starting at
    // where string is pointed to, are set to 0.
    if(string != NULL) {
        strncpy(string, "RandomString", str_len);
        printf("String: %s \n", string);
        free(string); // Returns memory back to heap
        return EXIT_SUCCESS;
    }

    return EXIT_FAILURE;

}
```

The result of running the previous program is:

```
String: RandomString
```

2.3 realloc

This function will re-size the amount of memory allocated to a specific memory block. However, it may or may not move your data. Useful for when you realize you need more or less space than anticipated.

```
void *realloc(void *pointer, size_t size)

Parameters:
pointer - Pointer to previously allocated memory. If NULL a new block is
    allocated.

size - New size of the memory block.
```

Here is an example using **realloc**:

```
#include <stdio.h> // Needed for printf() and constants used.
#include <stdlib.h> // Needed for realloc() and free().
#include <string.h> // Needed strncpy() and strlen().

int main() {
    int str_len = strlen("RandomString"); // note we did not use const
    char *string = (char *) malloc((str_len + 1) * sizeof(char));
    if(string != NULL) {
        strncpy(string, "RandomString", str_len);
    }
    str_len += strlen("AndMore");
    string = (char *) realloc(string, str_len + 1);
    // Needed more space to fit the rest of message.
    if(string != NULL) {
        strcat(string, "AndMore");
        printf("String: %s \n", string);
        free(string);
        return EXIT_SUCCESS;
    }

        return EXIT_FAILURE;
}
```

The result of running the previous program is:

```
String: RandomStringAndMore
```

Advantages

Waste: When it comes to static memory allocation there are issues of waste to consider. If you allocate memory statically you may allocate more than you need. This cannot be changed during the execution of the program if this is realized. With dynamic memory allocation you have the ability to reduce the memory given so it may be used elsewhere during run time. This also means if you allocated too little memory you can increase the amount of memory available.

Disadvantages

There are multiple problems that can stem from dynamic memory allocation and they must be avoided carefully.

Memory Leaks: When allocating memory dynamically there are times when the memory has served its purpose and is no longer needed. To free the memory the programmer must manually free the memory so it can be used by the system. If this is not done the memory will never be freed to be used by other resources and if done enough times this can be a large waste of memory and in the worst case can cause you to run out of memory during program execution. Note for small programs this may not be an issue. Memory is automatically freed when the program exits. For instance in the above examples it was not necessary to call free before returning since the operating system handles it for us and we weren't going to use any more memory so we wouldn't have hit that boundary. On a larger program however this can be a major concern.

Freed too soon: If the memory allocated is freed too soon then any access to that location *may* cause errors. This is even worse than always causing errors as you might not catch this during debugging. It is very important to free up memory only when the programmer is certain it is no longer going to be used.

Chapter 3

Pointers and Arrays

Knowing how pointers work in C is a valuable tool because C allows unrestricted access to the address space given to a program.

3.1 Arrays

There are two ways to declare an array in C.

```
int a[] = {1, 2, 3, 4, 5, 6, 7}; //an array containing 7 elements
int* b = a; //points to the array a
```

Since *b* is a pointer, it can be used to access the elements in *a* as if *b* itself were an array. The valid indices of *b* in this case are 0 through 6.

```
printf("%d ", a[0]); //prints the first element in a
printf("%d\n", *b); //prints the first element in a as well
printf("%d ", a[1]); //prints the second element in a
printf("%d\n", *(b+1)); //prints the second element in a as well
```

As shown, portions of an array can be accessed using box notation or with pointer arithmetic.

3.1.1 Index checking

In many other programming languages, the program ensures that the programmer is not exceeding the bounds of an array. C does not have index checking and the compiler will allow an array to go out of bounds. Index checking is very time consuming and since C is used for many low-level operations, speed is incredibly important. This brings more complications to the programmer because they must be aware of their bounds and make sure they are not exceeding them. Failing to do so will result in buffer overflows and off-by-one errors, among others. As such, bound checking is very important to the correctness of a program.

3.2 Strings

In C, strings are just arrays of characters and the end of a string is denoted with a null character symbol. Without this character, many "string" functions will cause errors, as they will only know when to stop by seeing the null character. There are two ways to declare a string.

```
char* test = "Erik Andersson"; //declaring a string by character pointer

char test2[] = {'E', 'r', 'i', 'k', ' ', 'A', 'n', 'd', 'e', 'r', 's',
               's', 'o', 'n', '\0'}; //declaring a string with an array
```

Using a character pointer to declare a string, such as with test, implicitly places the null character at the end of the string. However, when declaring test2 with an array, the null character must be included at the end, as it will not be added for you.

```
  printf("%s\n", test);
  printf("%s\n\n", test2);
```

The above example shows how strings are traversed. The print starts at the beginning of test and test2 and prints characters until the null character is reached.

```
  printf("\"%c\"\n", test2[14]);
  printf("\"%c\"\n\n", *(test+14));
```

This block of prints shows how char* declaration "gives" the null character. If the last character in either is printed, the null character is returned.

```
  int i = 14;
  test2[i++] = '4';
  test2[i++] = '4';
  test2[i++] = '4';

  printf("%s\n", test2);
```

This final string example shows what can go wrong if a programmer forgets to null terminate the string or access the out of bounds of the array. Although test2 is an array of 15 characters, it can still access locations past 14. When test2 is printed, it will continue to print characters until it finds the next null character, which could be anywhere.

Chapter 4

Assorted Topics

4.1 Structures

In C there are complex data types called structs which allow for multiple variables to be put together under one name but within separate memory locations.

```
struct Foo{
    char* name;
    int size;
    double bigSize;
    char box[4];
};

void test (struct Foo *a){
    a->name = "Bobby";
    (*a).size = 55;
    a->bigSize -= 20000.00;
    a->box[0] = 'b';
    a->box[1] = 'a';
    a->box[2] = 't';
}
int main(){
    struct Foo foo;
    foo.name = "Erik";
    foo.size = 42;
    foo.bigSize = 40000.00;
    foo.box[0] = 'a';
    foo.box[1] = 't';
    foo.box[2] = 'e';
    foo.box[3] = '\0';
    printf("%s %d %f %s\n", foo.name, foo.size, foo.bigSize, foo.box);
    test(&foo);
    printf("%s %d %f %s\n", foo.name, foo.size, foo.bigSize, foo.box);
}
```

In this example, the outline for declaring a struct is done in a global scope. This way, a "Foo" struct can be declared anywhere in the program. In main, the struct is initialized at the top and then values are assigned to each variable in the struct. Printing shows the values of the variables in the struct before and after alteration by test().

4.2 Declarations and definitions

4.2.1 Function Declarations

In C, variables and functions can be declared or defined. In a function declaration, only the function's return type and arguments need to be known, not the implementation of the function. Similarly, variables can be declared, but not given a value, causing them to be undefined.

```
#include <stdio.h>
// this is a function declaration, and does not contain actual code
int add10(int num);

int main() {
    // this is a variable declaration
    int num;

    // this is a variable definition
    num = 0;

    printf("%d \n", add10(num));
}

// this is a function definition, notice how this contains actual code,
// whereas the declaration does not
int add10(int num) {
    return (num + 10);
}
```

4.2.2 Defining Variables in Loops

When defining variables, there are a few things to keep in mind. Depending on the C standard you adhere to, you may need to declare variables for while loops outside of the loop itself, as seen below. Variables only exist within the scope they are declared, so if you declare a variable within a loop, it will no longer exist once a loop terminates.

```
int i;
for(i = 0; i<5; i++){
    char* test = "apple";
    //prints apple
    prtinf("%s", test);
}
printf("%s", test); //will not work
```

If you were to run this code on your computer, your program would crash, as the second printf would attempt to use the *test* variable, but it no longer exists once you leave the loop.

4.3 Break and continue statements

Break and continue are two commands that are used to handle the flow of control statements. Break will take you out of whatever control statement you are currently in.

```
for(int i = 0; i < 8 ; i++){
    if(i == 5)
        break;
    printf("%d",i);
}
```

The loop will continue until i reaches 5 and then it will break out of the loop. The output will never print past 4.

```
0
1
2
3
4
```

Continue works in a similar way to break in that it allows you to alter the execution of a control statement. However, instead of leaving the structure, continue will put you back at the "top" of the structure.

```
for(int i = 0; i < 5; i++){
    if(i * 2 == 4)
        continue;
    printf("%d", i);
}
```

When i = 2 it will trigger the continue, skipping the print statement. Therefore, the output will not include 2.

```
0
1
3
4
```

4.4 Postfix and prefix operators

In C, the order of operator execution is important. variable++ and ++variable do not have the same effect in certain circumstances. When the operator comes before the variable, the operator works on the variable before the variable is returned. When the operator comes after the variable, the variable is returned and then the operation is conducted

```
#include <stdio.h>

int main()
{
    int i = 1;

    printf("%d \n", i);
    printf("%d \n", i++);
    printf("%d \n", ++i);
}
```

Output:

```
1
1
3
```

As this example shows, the first print works as expected, but the second and third have interesting outcomes. The second print statement begins operating when the value in i is 1, and this value is returned to the print statement before it is incremented. The third print statement begins operating when the value in i has been incremented to 2, and before returning the value in i to the print statement, it is incremented, returning and printing 3.

Chapter 5

Further Reading

- The C Programming Language by Brian W. Kernighan and Dennis M. Ritchie
- Programming in C by Stephen G. Kochan

Bibliography

[1] Figure 1.1 from http://cs-fundamentals.com/assets/images/code-data-segments.png