The Stack

In which our hero peeks under the covers and discovers one of the most basic mechanisms needed during the execution of a program, the famous “stack”.

15.1 A Stack

Envision a stack of identical plates. When you need a plate, you just grab one off the top of the stack; call this operation, of taking something off of a stack, a pop. When you’ve cleaned a plate and want to put it back, you place it on top of the stack; call this operation, of putting something back on the stack, a push. And now you understand one of the most basic data structures in programming, called a stack, and the operations it provides, pop and push. A stack follows what is called a LIFO discipline, or last-in, first out, i.e., when you push a plate onto the stack it’s the last one that has been added, and thus when you pop a plate off the stack, this last one in is the first one out\(^1\).

However, we are not yet at the point where we need to introduce data structures, or how they are used, in C. So why this discussion about stacks? Well, since you asked, the reason is this: the C language itself manages a stack internally, (somewhat) invisibly to the programmer. This is not a stack, but rather, the stack, sometimes called the C run-time stack or the call stack. Inevitably, we will call this the stack, and you will know what we mean.

In this chapter, we’ll introduce the C stack, and motivate its usage. The C stack is a convenient and fast way for C to allocate space in memory for what we call local variables (sometimes also called automatic variables, because the stack manages them automatically for the programmer), as well as to implement the machinery of a function call and subsequent return. Understanding the stack, and how it fits into how C uses memory, is yet another requirement for mastery of C. So time to pay attention!

---

\(^1\)Make sure to occasionally take your plates out and shuffle them; otherwise, you will repeatedly use the top few. Look for other household tips in our monthly newsletter.
15.2 A Simple Example

Let’s begin with a simple example. Here, inside a function `foo()`, we decide we need a variable, \( x \), and thus define it and assign it to the value 100:

```c
int foo() {
    int x = 100;
    ...
}
```

You know, from previous chapters, that C lets you allocate variables in this manner, and places them in memory somewhere. The thing you don’t know is this: where in memory does C place such a variable? How does C manage memory for these variables?

The answer to these questions are simple. C allocates variables like \( x \) on its stack, a data structure managed by the C run-time for this very purpose (and a few others, as we’ll see below). The stack is usually located at the very far end of the address space (i.e., at high addresses); when C needs more space for variables, it grows the stack (this is like the push operation described above); when those variables are not needed anymore, C shrinks the stack (this is like pop). C does all of this automatically for you, the oblivious programmer –
and that is why we sometimes call variables managed in this manner **automatic variables**.

Figure 15.1 (page 2) shows how memory is laid out, with our new model of how the stack fits into the picture. As you can see in the diagram, the stack starts at the very highest address of the address space, and grows towards lower addresses.

The top of the stack is marked by what we’ll call the **stack pointer**, or the **C Stack Pointer (CSP)** for our model of the C Abstract Machine. The important thing about this: all of the data between the CSP and the end of the address space is **live**, or **valid**, and can be accessed legally by the program. When C needs more space on the stack, it simply can change the value (by decreasing it, as the stack grows towards lower addresses) of the CSP to allocate more space; when it is finished using that space, it can change the value again (by increasing it), thus freeing up said memory.

# 15.3 Space For Parameters And Local Variables

One major usage of the C stack is for local variables (as we’ve discussed) as well as parameter passing. In fact, as we’ll see, allocating space for local variables within a function and for its parameters is a very similar operation; the main difference is that parameter values are automatically initialized by the caller of the function, whereas local variables are (in a properly written function) initialized within the function itself.

Let’s use a simple example here of a function that computes a factorial, and a call of this function from **main()**:

```c
int factorial(int n) {
    int i, result = 1;
    for (i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}

int main() {
    int fact = factorial(7);
    return fact;
}
```

This function takes a single parameter (**int n**) and then uses two local variables (**result** and **i**) to perform its computation. So how would C allocate space for these variables when **factorial()** is called from **main()**?
Figure 15.2: Before, During, After Function Call

Figure 15.2 shows the (simplified) state of the stack before, during, and after the call to \texttt{factorial()}. As you can see from the picture, before the call takes place, the stack only contains room for the local variable \texttt{fact}, which will (eventually) hold the result of the call. During the call to \texttt{factorial()}, the stack is grown to include room for parameter \texttt{n} and local variables \texttt{i} and \texttt{result}. Finally, when the function call has returned, the stack is decreased again and only \texttt{fact} remains live, and holds the final value of the computation.

15.4 Call And Return

There is one more important conceptual piece of information that is placed on the stack, and is essential for the function call to work. That piece of information is known as the return address – basically, when a function is finished executing, it needs to know where in the code to jump back to in order to continue the program’s execution in the expected manner.

Let’s look at an example to make this clear. Figure 15.3 shows the same sequence of function call and return as with our previous example (\texttt{factorial()}), but this time including the extra detail of the return address. As you can see from the diagram, first the argument values are pushed onto the stack. Then, the return address is pushed onto the stack, and control is transferred to the entry point of the function; as we will see later in the book, a single machine-level instruction (such as \texttt{call} on x86) usually accomplishes this task. Finally, space is made for local variables (\texttt{i} and \texttt{result}), and the function can begin to run.

On the return path, the expected corollary actions take place. First, the stack size is decreased such that the return address is at the top of the stack. Then, some kind of machine-level return instruction is executed, which pops the top of the stack and jumps to that point in the code. Finally, the stack space for the input parameters to the function is reclaimed.

The figure also shows how the C Instruction Pointer (CIP) changes throughout the sequence of operations. As we set of for the func-
Figure 15.3: Function Call/Return With Return Address

At this point, you should have a good idea as to how call and return from a function works, including how space is made available for parameters, local variables, and even the return address to jump to when the function completes. When a series of function calls takes place, the stack thus consists of a series of structures that enable you to figure out exactly which calls are currently being processed.

For example, imagine a piece of code in which function main() began and then called function f() somewhere in the middle of its
computation. Then, in the middle of \( f() \), \( g() \) is called. Finally, in
the middle of \( g() \), function \( h() \) is called. We can denote this chain
of function calls as follows: \( \text{main()} \rightarrow f() \rightarrow g() \rightarrow h() \).

If we examine the state of the stack at this point (when \( h() \) is ex-
cuting), we see a trail of what exact calls have been made up to this
point in the execution of the program. Figure 15.4 shows how this
state has been constructed, one function call at a time. As each func-
tion is called, a new record of the activity is pushed onto the stack,
including arguments to the function, return address, and space for
local variables. We call each record a stack frame (or, sometimes,
an activation record). The stack is thus simply, at any given point
during the execution, a series of such frames, with the currently ex-
cuting function atop the stack, and all the calls waiting to complete
below it.

15.6 Finally, Why \text{increment()} Doesn’t Work

Back in the chapter on functions [AD17], we started with a simple
example that didn’t quite work as expected. A function \text{increment()}
incremented a value that was passed to it but had no effect on the
caller:

```c
1  void increment(int value) {
2       value = value + 1;
3  }
4
5  int main() {
6      int x = 0;
7      increment(x);
8      return 0;
9  }
```
Now you should have a good idea as to why. When `increment()` is called, room for its parameter, `value`, is created on the stack. Then, the value of `value` is initialized by the caller to the current value of `x`. This method, of passing a copy of the value of a variable to the function is referred to as **pass-by-value** in the world of programming languages; C only passes arguments by value.

Inside the function, the code meaninglessly increments `value`, having no effect upon the variable `x` in `main()` that we truly wish to increment. Operating on a parameter updates the *copy* that has been created during the function’s lifetime. To update a variable that in the caller’s stack frame, we need to do something more, and that something more (as you may have guessed) is using a pointer, a task we will take on shortly.

Figure 15.5 shows the variables involved during this example before, during, and after the call. As you can see, `increment()` only operates on its own copy of the value of `x`, which it refers to as `value`. The function dutifully increments `value` by one, but the computation is pointless, leaving the value of `x` in the caller unmodified.

### 15.7 Summary

We have learned about one of the most fundamental aspects of a running C program: the **stack**, an idea as old as computing itself [D00]. The stack is involved in space allocation for local variables and parameters, as well as in proper control of flow during function call and return. We’ve also seen why parameters can be modified without worry during a function call, as a parameter is just a copy of whatever value the caller passed to the function. Finally, we’ve seen why a naive implementation of a function to change a value in the caller does not work; something more is needed to enable this essential behavior.

One final note: our model here of the C runtime stack is somewhat simplified; as we’ll see (much later, when looking at assembly code), there are a few more items that get pushed onto and popped off of the stack during program execution. However, from the C pro-
Aside: C Only Passes By Value

C only passes arguments to functions by value. This behavior has many benefits, including the fact that the function can freely change its arguments without worry of harming values elsewhere in the program. But pass-by-value semantics have their limitations; how do we change a value of a variable that is passed to a function? C overcomes this limit through the use of pointers, as we will soon see; however, other languages use other means. For example, in C++, you can declare a parameter to be a reference to a variable, which is another way of saying you are passing a pointer to that variable instead of just its value. In such pass-by-reference languages, changing the value of a reference parameter has the intended effect of changing it in the caller.

grammer’s standpoint, thinking of it as a place where local variables, parameters, and return addresses reside is likely enough to provide an understanding of the critical machinery behind the all-important function call.
References

Chapter on **Functions**
*A reference within this book to itself: how millenial. Sorry, millenials, you get a bum rap.*

[D00] “Recollections on operating system design”
Edsgar Dijkstra
/www.cs.utexas.edu/~EWD/transcriptions/EWD13xx/EWD1303.html
*In the summer of 1959, Dijkstra invented the stack to help with function calls, specifically to enable the recursive calling of a function `foo()` from `foo()` itself! Dijkstra notes many others invented such a structure at the same time, and thus no one really gets credit for it. We’ll learn more about recursion later when we need it for more advanced data structures.*