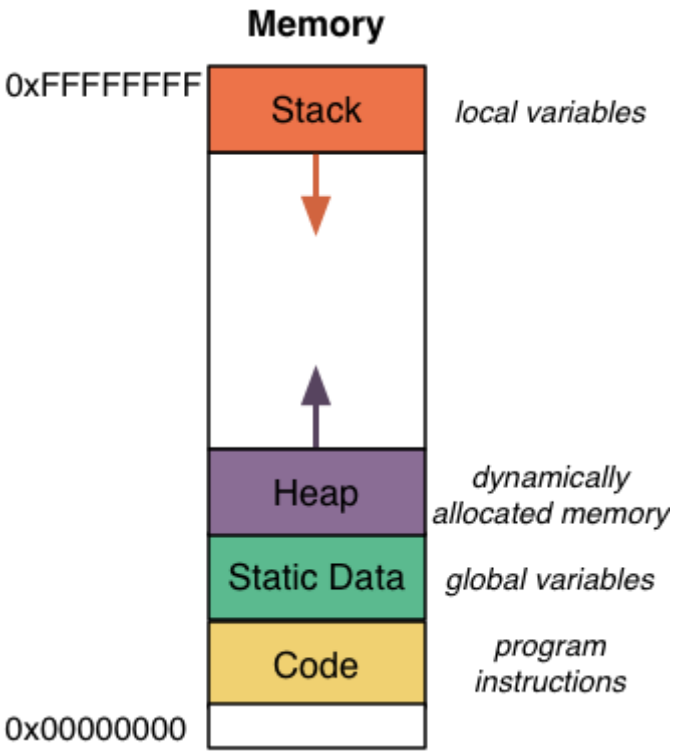
A Program's View of Memory

coursera.org/learn/pointers-arrays-recursion/supplement/sVQ8O/a-programs-view-of-memory

On a 32-bit machine, where addresses are 32 bits in size, the entire memory space begins at 0x00000000 (in hex, each 0 represents 4 bits of 0) and ends at 0xFFFFFFFF (recall that 0x indicates hexadecimal, and that each F represents four binary 1's). Every program has this entire address space at its disposal and there is a convention for how a program uses this range of addresses. the figure below depicts where in memory a program's various components are stored.



Code

Given the repeated claims that *everything is a number* it should come as no surprise that a program itself can be represented by a series of numbers. Producing these numbers is the job of a compiler, which takes a program in a language like C and converts it into a bunch of numbers (called object code) which is readable by the computer. An instruction in C which adds two numbers together, for example, might be encoded as a 32-bit instruction in object code. Some of the 32 bits will tell the machine to perform addition, some of the bits will encode which two numbers to add, and some of the bits will tell the processor where it should store the computed sum. The compiler converts the C code into object code and also assigns each encoded instruction a location in memory. These encoded program instructions live in the Code portion of memory, shown in yellow in the figure above.

Static Data

The static data area contains variables that are accessible for the entire run of the program (*e.g*. global variables). Unlike a variable that is declared inside a function and is no longer accessible when the function returns, static variables are accessible until the entire program terminates (hence, the term *static*). Conceptually, a static variable's box remains "in the picture" for whole program, whereas other are usable for only a subset of the program's lifetime. These variables are placed in their own location in memory, just past the code portion of the program, shown in the figure above.

The final two sections of memory are for two different types of program data that are available at specific times during a program's execution. The **Heap** (in purple) stores dynamically allocated data. The **Stack** (in orange) stores the local variables declared by each function. The stack is divided into *stack frames* that are available from starting when the function is called, and last until it returns. The conceptual drawings throughout this book have primarily shown pictures of stack frames as boxes (one for each function) with local variables. In actuality, the stack is one contiguous piece of memory; each stack frame sits below the stack frame of the function that called it.

With these details about how pointers work and how code is placed in memory, we can show a more detailed view of what happens during the execution of our swap function. The next video shows the same swap function's execution, but with a contiguous view of the stack and memory addresses as the values of pointers. It also shows the code of the program stored in memory in 4-byte boxes. The return address field of the stack—previously depicted in our conceptual drawings as a blue call site location–is also made explicit in this video. The return address is the address of the instruction that should be executed next after the function being called completes and returns.

The *calling convention* —the specific details of how arguments are passed to and values returned from functions resembles an x86 machine; the arguments to a function reside in the stack frame of the function that called it. This is slightly different from the conceptual drawings we have shown previously, in which the arguments were placed in the stack frame of the function being called. For a conceptual drawing, this is both sufficient and easy to understand. Hardware details will differ slightly for every target architecture.