A C Primer for Java Programmers

Adam A. Smith

C was originally developed by Dennis Ritchie at Bell Labs between 1969 and 1973. It was used to a large degree to program the early Unix operating system. It was standardized in 1989 by the American National Standards Institute (ANSI), and later by the International Organization for Standardization (ISO). C is an extremely influential language. Its syntax has been widely copied by other languages, including C++, C#, Java, and JavaScript. For this reason C will look very familiar to Java programmers, though subtly wrong.

One of C's most notable features is that it translates very naturally into the underlying machine code, on which modern computers run. This is a strength in that the code you write can be very efficient. But it's also a weakness, in that less is done automatically for you by the compiler, which can significantly increase development time. Think of C as being like a manual-transmission or stick-shift car, in comparison to Java's automatic transmission. You have more control with C, but there aren't as many guard rails for you when you make a mistake. This can be very frustrating for programmers who are new to C (just like driving a stick might be very frustrating if you're used to an automatic).

This primer is not intended to turn you into an expert C programmer—that will only come after many hours of coding and solving difficult bugs. But it will help get you started, and answer some of your basic questions.

We will divide the differences between C and Java into three main categories:

1. **Syntax and practical differences:** These are the small differences between C and Java that can be stumbling blocks as you learn the new language.

2. **Memory and pointers:** C's memory management is much more hands-on, and it allows you to do a couple things that aren't even possible in a Java program. We will discuss how it works here.

3. **Data structures:** C does not have classes like Java does. However it does contain something called a `struct`, which is related. We will discuss what it is, and how it can be used to program in an object-oriented way.

Header File	Use	Example
`stdio.h`	standard input/output	`printf()`, `scanf()`
`stdlib.h`	common "standard library" functions	`malloc()`
`string.h`	string manipulation functions	`strlen()`, `strcmp()`
`math.h`	common math	`sqrt()`, `sin()`, `log()`
`time.h`	system time & random numbers	`time()`, `rand()`

Table 1: Common Header Files. These are some of the `.h` files commonly used in an `#include` directive at the beginning of a C file. There are many more, and advanced C programmers often make their own.

Syntax & Practical Differences

Here we detail the minor differences between C and Java. None of these are particularly difficult to master, but you will need to get used to them.

Comments

Comments work the same way they do in Java, using `//` for a single-line comment, or `/*` and `*/` for a multi-line comment. Since Javadoc is a Java tool, there is no need to make formal Javadoc comments. (However, it is always good practice to give your functions in-depth comments that explain their use.)

Including Libraries

C's equivalent to Java's `import` lines are **`#include`** lines, that tell the compiler where to find standard C functions that you haven't defined yourself. For example, the first line of many C programs is:

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

This indicates that the compiler should *include* C's standard input/output library, including useful functions such as `printf()` (which is described below). Notice that the line starts with a hashmark, and has no semicolon at the end. This is because it is technically a **preprocessor directive**, or a step that should be followed before we begin the actual compilation. We will encounter this again when we define constants.

Other common libraries that your program might include are shown in Table 1. These `.h` files are called "C header" files, and they are used to compile together source code from multiple files into one working executable program. Advanced C programmers often make their own `.h` files, allowing a program's source code to be spread among multiple `.c` files.

Functions & Prototypes

Like in Java, a **function** is a self-contained area of code to perform some task, and they are the basic building blocks of a program. Because C has no classes, there's absolutely no need to make a `class` block to contain the functions, like one does in Java. Further, it is incorrect to refer to C functions as "methods", because by definition a method is inside a class.

Instead, a C program consists mostly of the `#include` statements followed by prototypes (see below), and finally all of the individual functions. Functions look very similar to they way they do in Java:

```
// this function takes 2 int arguments and returns an int
int doSomeTask(int arg1, int arg2) {
    // internal code goes here
}
```

However C has an important rule: a function *must* be defined prior to its being called. That is, one function cannot call another function that is below it in the source code, since it hasn't been defined yet.* Of course Java is different: the compiler does not care about the order in which a class's methods are defined.

To get around this rule, every C function should have a corresponding **prototype** near the top of the source code. This is a definition of how to call the function, before it is fully defined. A function's prototype looks like its signature line, but it ends with a semicolon rather than an opening brace:

```
int doSomeTask(int arg1, int arg2); // prototype
```

This prototype lets the compiler know how to call the function, even when it's not defined until later in the source code. You *can* write a C program with no prototypes, but it's not a good idea—a function without a prototype can only be called by functions that occur later in the source code. So, after your `#include` statements, there should be a master list of all your functions, one per line, followed by the function definitions themselves.

Unlike in Java, a function that takes no arguments *must* have the keyword `void` between its parentheses. This is because historically a function with nothing between its parentheses meant that the programmer wasn't yet establishing whether or not the function took any arguments. Putting `void` in this position means that you are positively stating that the function has no arguments.

---

*This probably seems ridiculous and frustrating to modern programmers. The reason is that it makes the compiler quicker—it allows it to read through the source code fewer times while compiling the program. Given that compiling in the old days could take a minute or even longer, this is not unreasonable!

Variable Type	Guaranteed Min Size	Usual Size	Usual Min Value	Usual Max Value
`char`	1 byte	1 byte	-128	127
`unsigned char`	1 byte	1 byte	0	255
`short int`	2 bytes	2 bytes	-32,768	32,767
`unsigned short int`	2 bytes	2 bytes	0	65,535
`int`	2 bytes	4 bytes	-2,147,483,648	2,147,483,647
`unsigned int`	2 bytes	4 bytes	0	4,294,967,295
`long int`	4 bytes	8 bytes	$\sim -9.2 \times 10^{18}$	$\sim 9.2 \times 10^{18}$
`unsigned long int`	4 bytes	8 bytes	0	$\sim 1.8 \times 10^{19}$
`long long int`	8 bytes	8 bytes	$\sim -9.2 \times 10^{18}$	$\sim 9.2 \times 10^{18}$
`unsigned long long int`	8 bytes	8 bytes	0	$\sim 1.8 \times 10^{19}$
`float`	*none*	4 bytes	$\sim -3.4 \times 10^{38}$	$\sim 3.4 \times 10^{38}$
`double`	*none*	8 bytes	$\sim -1.8 \times 10^{308}$	$\sim 1.8 \times 10^{308}$
`long double`	*none*	16 bytes	$\sim -1.2 \times 10^{4932}$	$\sim 1.2 \times 10^{4932}$
`void`	*N/A*	*N/A*	*N/A*	*N/A*

Table 2: Basic variable types in C. Exact sizes may vary from system to system, but will be as least as big as indicated in the "Guaranteed Min Size" column.

The `main()` function should return an `int`. This `int` should be `0` if the program terminates normally, or nonzero (usually `1`) if there was an error. Therefore, a typical `main()` function might look like this:

```
// program starts here
int main(void) {
    // code here
    return 0;
}
```

Because there are no classes in C, there are no `public`, `private`, or `protected` keywords. There *is* a `static` keyword, but it means something different. (It allows a local variable to persist even when it is out of scope.)

C functions do not throw exceptions. The language predates their common use.

Basic Variables

C contains five basic variable types, that will be familiar to you. These are the **`int`** for integers, the **`double`** and **`float`** for real numbers, the **`char`** for single symbols (such as letters or numbers)*, and the **`void`** which represents a non-type. These all work more-or-less the same way they do in Java. (Though `void` has extra uses in C, as we've already seen.)

*Like in Java, a `char` is really a very small `int`. Every character has an equivalent number. (For example, `'A'` is 65.) If you wish to know them, search for the "ASCII table".

Unlike in Java, the sizes of these variables may vary from system to system. This is because C relies on the underlying hardware to define these types, and that can depend on many factors that are out of your control. The most common definitions are shown in Table 2, but you should understand that this might be different in your environment. If you believe that your system is different, you can determine the size (in bytes) of each using the C `sizeof` operator:

```
int intSize = sizeof(int); // 4 on most systems
```

There are also four keywords that can be used to modify variables, which are also shown in Table 2. The **`long`** modifier increases the size of an `int` or `double`, whereas the **`short`** modifier makes a smaller `int`. The **`unsigned`** modifier restricts an `int` or `char` to be nonnegative, doubling its range in positive territory. (By contrast **`signed`** is mostly redundant today, though on some systems it might be used to force an `int` or `char` to allow negative values.)

C does not have a boolean type.* Traditionally, one uses an `int` when one needs a true/false value: `0` means false, and nonzero (usually `1`) means true. This calls for some care, as the following code will compile without an error message:

```
if (myValue = 4) { // bug: should be ==
```

In all probability, the programmer who wrote this intended to use a `==` to test if the variable holds a `4`, and to execute the associated `if` block if it does. Instead, this code forces `myValue` to hold a `4`, and then this block will *always* trigger because `4` is nonzero. This would not compile in Java, because the value in the parentheses is an `int` rather than a `boolean`.

C does not have a string type. Instead, we use `char` arrays. We'll discuss this more in its own subsection below.

For the most part, the **operations** you are used to in Java work the same way in C. So when you see C code with `+`, `-`, `*`, `/`, `%`, `+=`, `-=`, `++`, `--`, `=`, `==`, `<`, `<=`, `>`, `>=`, `+`, `&`, `|`, `^`, and so on, you should assume that they are working the way you would expect. However `+` cannot be used to concatenate strings (because they are `char` arrays). Also, because there are no booleans, `&&`, `||`, and `!` operate on `int`s in C—returning an `int 0` for false, and nonzero for true. Finally, C has some extra operators that help direct memory management (`&`, `*`, and `->`), that we will discuss when we talk about memory and data structures.

Casting variables works the same way in C as it does in Java, using types in parentheses such as `(int)`.

*Though there are some recent additions to C that simulate them.

C also allows **global variables**, that do not exist in Java.* A global variable is simply one that is defined outside of a function, and can be accessed everywhere. However, global variables are usually considered to be very poor practice. You should avoid using them as much as possible.

C has a keyword **`const`** that is very similar to Java's `final`. It is used as a modifier of the variable type when it is declared:

```
const int CONSTANT_VALUE = 19
```

This line should be near the top of your code, close to the `#include` lines. Give the constant an `ALL_CAPITAL` name as you do in Java. It is okay to make your constants global.

There is also an older way to make a constant, using the **`#define`** preprocessor directive like this:

```
#define CONSTANT_VALUE 19
```

You may see this in some code, but its use is usually discouraged today. Using `#define` does not actually make a variable. Rather, it tells the compiler to do a search and replace just before compiling, in this case changing all instances of `CONSTANT_VALUE` to `19`. It is just as if you used the literal `19` throughout your code, without making a variable at all. It is potentially more efficient than using `const`, since it doesn't "waste" memory on a value that will never change.** However, modern compilers are smart enough to recognize `const` variables, and to do this substitution anyway. Therefore, you should use `const`, because it allows for better checking for bugs at compile time.

Control

The **`if`** and **`else`** keywords work the same way in C as they do in Java. Further, **`while`**, **`do`**, and traditional **`for`** loops all work identically. However, C has no "enhanced" for loops (e.g. `for (int a: myArray)`).

There are no try-catch blocks in C, because C functions do not throw exceptions.

Input & Output

To **print** to the terminal, one usually uses the **`printf()`** function. `printf()` is special, in that the number of arguments it takes varies. Its first argument

*Java does allow public static variables, that are much the same thing—but they are not true global variables because they are still part of a class.

**Using a few extra bytes is trivial on a desktop or laptop computer or a phone. But it might be important on small embedded systems!

Escape Sequence	Used to Print	Escape Sequence	Used to Print
`\n`	newline	`\"`	"
`\t`	tab	`\\`	\

Table 3: Common Escape Sequences to print special characters in `printf()`. This is not a complete list.

Format Specifier	Used to Print	Format Specifier	Used to Print
`%d`	`int`	`%c`	`char`
`%u`	`unsigned int`	`%s`	string (i.e. `char` array)
`%ld`	`long int`	`%p`	pointer
`%lu`	`unsigned long int`	`%%`	percent sign
`%f`	`float` or `double`		

Table 4: Common Format Specifiers for `printf()` and `scanf()`. There are many others in addition to these.

is always the string to print out. So the most basic example of `printf()` is the "hello world" example:

```
printf("Hello world!\n"); // basic output
```

Like in Java, you can print a newline by using the **escape sequence** `\n`. Other common escape sequences used to print special characters are shown in Table 3. `printf()` does not automatically append a newline to the end of its output like Java's `System.out.println()` does, so be sure to do it yourself.

Unlike in Java, we cannot easily concatenate strings with other variables to output complicated sentences. If you want to print out nonstring variables, you must use **format specifiers**, which are short codes that begin with a `%`. Each one necessitates a further argument in the `printf()`. The specifiers are locations in the string in which to insert the variables. For example, one of the most common specifiers is the `%d`, used to print an `int`:

```
int myValue = 6;
printf("The value is %d.\n", myValue); // "The value is 6."
```

This example has one extra argument corresponding to the `%d`: the variable to print. Here's another example, that uses three format specifiers to print two numbers and their sum. Its `printf()` needs four arguments:

```
int num1 = 3, num2 = 9;
printf("%d + %d = %d\n", num1, num2, num1 + num2);
        // prints "3 + 9 = 12"
```

There are many format specifiers—a subset is shown in Table 4. If you are curious, you may look online to find others.

If you use the wrong specifier, it can result in undefined behavior (that can differ from system to system). For example, this code doesn't work:

```
printf("%d\n", 4.0); // bug: prints weird number
```

The number to print is a `double`, but it is formatted as an `int`. Depending on your computer, it may print out the same wrong integer each time, or it may appear to print a random number. It's trying to print out an `int`, but it cannot find it. Some systems might try to interpret the raw bits of `4.0` as if it were an `int`. Other systems might print out data from a nearby area of memory that hasn't been initialized at all. If you get some weird outputs, it may pay to double-check that your format specifiers are correct.

`printf()` does not always print immediately. It often tries to make execution more efficient by delaying printing until there's a newline, and then printing everything at once. This can be frustrating when you're debugging: if your program terminates unexpectedly, some things that you thought you printed before the crash will never appear. However, you can make sure that everything is printed out by "flushing the print buffer". You can do this with `fflush()`, as shown here:

```
fflush(stdout); // makes sure everything's printed
```

The argument `stdout` is the standard-out stream, which usually links to the terminal. It is a global variable defined inside of `stdio.h`.

The most common way to **input a value** from the keyboard is with **`scanf()`**. To get an `int`, one might do the following:

```
int userInput;
printf("Pick an integer: ");
scanf("%d", &userInput); // input a number
```

The `&` sign is very important for `scanf()` to work. We'll discuss this in greater detail in the next section, when we talk about C and memory management. In brief, it tells the `scanf()` function where the argument is stored in memory, so that the variable can be modified—much like an argument in Java can be modified if it happens to be an object or an array.

Inputting a string from the user is more difficult. We will talk about how to do it when we discuss strings, below.

Arrays

There are multiple ways to create an **array** in C. The methods shown in this section are easier to use, but they result in arrays that are not permanent. This code creates a new array of 20 `int`s called `myArray`:

```
int myArray[20]; // array of 20 ints
```

Usually the elements of this array are *not* initialized. Do not assume that they all start out holding `0`.

Alternatively, we can also leave out the size but include the elements explcitly:

```
int fib[] = {0, 1, 1, 3, 5, 8, 13, 21, 34, 55}; // 10 ints
```

This results in an array of size `10`, holding the first numbers of the Fibonacci sequence.

The problem with these arrays is that they are treated like local variables. They will cease to exist after the function they're in has ended. Thus, these techniques cannot be used if the array needs to be permanent. Trying to return these arrays, or use them after the function has stopped, can result in strange errors in which your program terminates or unrelated data gets changed, seemingly at random. We will discuss this further when we talk about memory and pointers. We will also discuss a way to make a permanent array.

What's more, you must be *very* careful when indexing an array in C. In Java, you immediately get an `ArrayIndexOutOfBoundsException` when you try to access an illegal element in an array. For example, the array `fib` that we created has only 10 elements: `fib[0]` through `fib[9]`. If you tried to access `fib[10]` or `fib[-1]`, your Java program would immediately stop and let you know there was a problem. C has no such boundary checking: if you tried to access `fib[10]`, C would just look at the memory where `fib[10]` *would be if it existed*, and return that data as if it were an `int`. Even worse, if you mistakenly wrote to `fib[10]`, it would change the data where `fib[10]` would be—altering some unrelated data! Be careful!

Another issue is that C arrays *do not* have a `length` propery, unlike in Java. There is no way to calculate the length of the array from the array itself.* If the length of an array can vary, its length *must* be stored in a

*Some people might try to do a `sizeof(myArray)`. However, this will only work from inside the same function as the array was declared, and only if done using one of these "temporary" methods. It usually only tells you the size of the array's address in memory, which is not helpful.

separate `int`. It is very common when one writes a function that takes an array as an argument, that it has a second argument indicating that array's length:

```
// this function has an array argument & a length argument
int doThingWithArray(int array[], int arrayLength) {
```

Strings

Like Java, we use single quotes (') to designate a `char`, and double quotes (") to designate a **string**. However, C does not have a devoted string type in the same way that Java does. Instead, you must use `char` arrays. For example, the following creates a string that holds the word **"hello"**:

```
char myGreeting[] = "hello"; // char array used as string
```

The length of this array will be 6: every string *must* end with a `0` (the null character*), to designate the end of the string. Therefore `myGreeting[0]` will hold the `char 'h'` (equivalent to the `int 104`), `myGreeting[1]` will hold `'e'` (or `101`), and so on, until `myGreeting[5]` holds `0`.

Even though strings are really arrays, C has many built-in functions to make their use easier. Most of the functions detailed in this section are defined in the `string.h` header file, so be sure to `#include` it!

Inputting a string from the user can be difficult, because the programmer must make absolutely certain that enough memory has been pre-allocated to store it. Not doing so can result in very confusing bugs, since it might overwrite unrelated data. It might even result in a security hole in your program. The easiest way to get a string is with the **`fgets()`**** function. Here we assume that a constant `BUFFER_SIZE` has already been defined:

```
char string[BUFFER_SIZE]; // temporarily allocate space
printf("Enter a string: ");
fgets(string, BUFFER_SIZE, stdin); // input a string
```

The first argument to `fgets()` is the array of characters that has been pre-allocated for the string. There is no need for the `&` because string is already an array, and can be modified when it's an argument to a function. The second argument is the maximum size that can be input: the string will

*The null character `0` is *not* the same as the symbol `'0'`, which is `48`. The null character has no symbol, and so is sometimes represented as `''`.

**You may find online code that uses `gets()` instead of `fgets()`. DO NOT USE IT! `gets()` has serious security issues, that can allow a malicious hacker to gain control of a computer that is running your program. We will talk about this more in the next section.

automatically be truncated down to the right size to fit in the array, so that nothing gets accidentally overwritten. (Remember, the last element of the array must be `0`.) And the third argument is the standard-in stream, stating where to get the information. `stdin` is defined inside of `stdio.h`.

There are a number of standard functions made to work with strings. If you wish to use them, be sure to `#include <string.h>` at the beginning of your code. The simplest is **`strlen()`**, which finds the length of a string:

```
int length = strlen(myGreeting); // length of myGreeting
```

This will search the given `char` array for the first `0`, returning its index. So if `myGreeting` contains `"hello"`, the above code will return `5`. It takes linear time to run, since it must search for the `0`.

Use the **`strcmp()`** function to compare two strings. This will return `0` if the strings are equal:

```
if (!strcmp(string1, string2)) { // triggers if identical
```

If the strings are not equal, `strcmp()` returns an `int` based on the ordering of the two strings. It returns a negative number if the first string comes first, and a positive number if the second string comes first. It does this based on an order that is similar to alphabetical order, but using each `char`'s numerical value. The comparison will be alphabetical if the strings consist of only lowercase letters. But all numbers come before all letters, and all capital letters come before all lowercase letters.*

`strcpy()` is used to copy one string into another. Its two arguments are the location to copy the string to, and the original string. For example:

```
strcpy(copyString, originalString); // copies a string
```

Note that you are responsible for making sure that the destination string already has all the space it needs for the data being copied into it! All `strcpy()` does is take the characters from one string and copy them into the new string. It doesn't allocate any new space on its own. If you do not do this first, your program will likely overwrite unrelated data or terminate. For example:

```
char[strlen(originalString)+1] copyString; // allocation
```

*Look up the "ASCII table" if you wish to see the exact ordering.

Note that the `+1` is to make space for the `0` that must be at the end of every string.

The function **`strcat()`** is used to concatenate two strings. It looks much like `strcpy()`:

```
strcat(mainString, suffix); // concatenates suffix to end
```

Here, the string `suffix` is tacked to the end of `mainString`, permanently altering `mainString`. As with `strcpy()`, you are responsible for making sure that `mainString` has enough space already allocated, for this to be a legitimate operation. If there is not, you may write over unrelated data or cause your program to crash!

Finally, there is a function called `sprintf()` that can mimic much of the above functionality. `sprintf()` is identical to `printf()` detailed above, except that it takes an additional argument before the others indicating a *string* to write into. It does not print anything to the screen at all:

```
sprintf(string, "The value is %d.", value); // saves string
```

As always, you must make sure that the first string argument contains the space needed, or else risk writing over unrelated data.

Command-Line Arguments

If you use the command line to execute a program, you can pass information to the program via the **command-line arguments**. For example, using a Java program called `MyJavaProgram`, the user might type the following into the command line:

```
user@mycomputer> java MyJavaProgram 21 blue
```

This starts up the `main()` function of the `MyJavaProgram` class. A proper Java `main()` function takes a `String` array as its argument, usually called `args`. In this case, the `args[0]` will be `"21"`, and `args[1]` will be `"blue"`.

To do the same thing in a C program, your `main()` function should be defined as follows:

```
int main(int argc, char **argv) { // command-line args
```

The variable `argc` means "argument count". It will simply hold the number of command-line arguments (including the command to activate the program). The variable `argv` ("argument values") is a little more complicated,

and we will explain the significance of the ** below. For now, it suffices to think of it as an array of strings holding the command-line arguments. So if the user runs the program a.out by typing:

```
user@mycomputer> a.out 21 blue
```

then argv[0] will be "a.out", argv[1] will be "21", and argv[2] will be "blue". Using this technique, the C program can take needed information directly from the command line.

Memory & Pointers

Java does a lot of memory management for you, behind the scenes. C's use of memory will seem clunky and weird by contrast. But there are advantages to being able to do your own memory management. For one thing, not having to rely on Java's automated "garbage collection" can noticeably speed up your program. For another, there is less need to take the time to allocate throwaway data structures. We will discuss C pointers first, and then work into how memory management works, and its ramifications.

Pointers

In Java, a complex-type variable is really holding an address—a location in memory. When you pass an object to a method as an argument, you're really just passing the address. That's why changes to the object inside the method persist when it is done: the object inside the method is an *alias* of the one outside. For example, in Java you can do the following:

```
Arrays.sort(myArray); // Java's built-in sort function
```

When this method is done, the array will have been changed—even though it has no return value. It used myArray's address to effect all its changes.

In C, *any* variable can be handled this way, including basic variables like ints and doubles. The trick is to pass the variable's address in memory, rather than the variable itself. That's what a **pointer** is: it's a variable that holds the address of another variable. (It "points to" the other variable.)

To declare a pointer variable, we add a * before its name. (In this context, * has nothing to do with multiplication.) So to declare a pointer to an int, we use a line like the following:

```
int *myIntPointer; // can point to an int
```

Two new operators become useful when we're working with pointers: `&` and `*`. The `&` operator can be read as *"address of"*, and it returns the address of its variable. We can use it to set a pointer, as we do here:

```
myIntPointer = &value; // set pointer to be int's address
```

The `*` operator can be read as *"contents of"*, and it returns the variable to which the pointer points. We can use it to print out the `int` that `myIntPointer` points to like this:

```
printf("%d\n", *myIntPointer); // prints pointed-to int
```

If you actually wanted to know the value held in `myIntPointer` (not the value held by the `int` it points to), use the `%p` format specifier:

```
printf("%p\n", myIntPointer); // prints address (not int)
```

We don't use the `*` operator here, since we're interested in the value of the pointer itself.

For every variable type, there's a pointer type that can point to it. So there are `int` pointers (`int*`), `char` pointers (`char*`), etc. There are even pointers to pointers, such as the `int**` type that points to an `int*`. If you want to make a function that takes a pointer as an argument, use the `*` in its definition. This function takes a `double` pointer, and returns an `int` pointer:

```
int *doPointerThing(double *doublePointer) {
```

That's how functions like `scanf()` appear to modify their arguments. They aren't really doing so—the actual arguments are addresses, and those can't be changed by the function. But the values they point to are easily altered. This means that in C, some arguments are really functioning as extra return values in disguise.

Consider this function to swap the values of two `int` variables:

```
// exchange 2 ints
void swap(int *value1, int *value2) {
    int temp = *value1; // temp <- contents of 1st pointer
    *value1 = *value2; // contents of 1st <- contents of 2nd
    *value2 = temp; // contents of 2nd <- temp
}
```

This `void` function technically doesn't return anything. Instead, it just takes two `int` pointers, and exchanges their contents. To call the function, just do this:

```
swap(&x, &y); // x and y are just ints!
```

After it's done, `x` and `y` will have switched values.

This style of coding is extremely common in C, but it can't be done in Java. It lets a programmer return multiple values, or deal with primitive variables like they're complex types. It's efficient, because there's no need to mess with arrays or complicated temporary data structures. By taking in the *addresses* of the `int`s, everything becomes streamlined. However it does require care and understanding that are not needed in Java.

Finally, the `void` pointer is a "generic" pointer, that can point at anything:

```
void *anythingPointer; // not limited to one type
```

This kind of pointer can be especially useful as a return value, when a function needs to effectively return any kind of pointer.

You might be surprised that when we declare a pointer, we attach the `*` to the name of the variable, and not the type. The following code might look more intuitive to a Java programmer:

```
int* myIntPointer; // not standard C
```

The reason we don't do this is because every single pointer variable must have its own `*`. If you were to do this:

```
int* pointer1, pointer2; // bug: pointer2 is an int
```

The compiler would associate the `*` with `pointer1`, while `pointer2` would mistakenly be just an `int`. The proper way to code this is as follows:

```
int *pointer1, *pointer2; // both are int pointers
```

Pointers in C are incredibly flexible. Understanding them is vital to understanding the language.

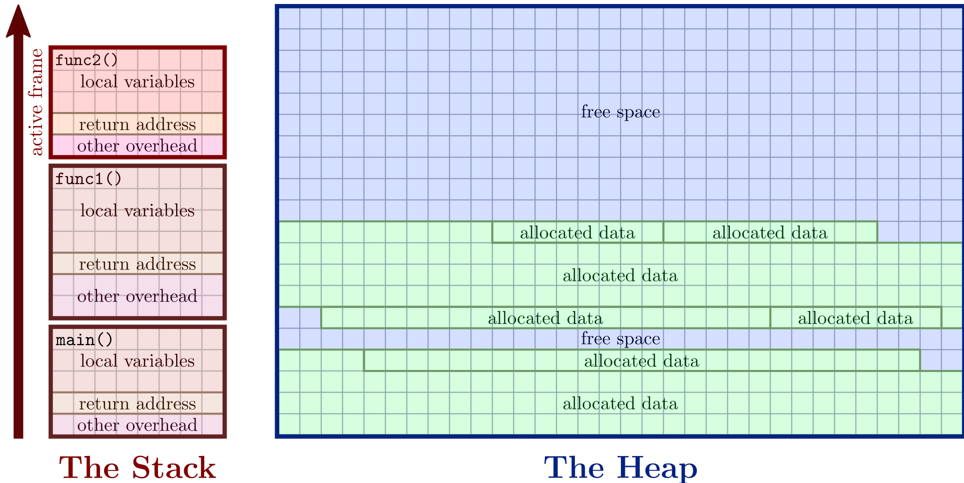

Figure 1: The stack and the heap. The stack is used to store separate data for each function. Here, `main()` has called `func1()`, which has called `func2()`, and each needs its own space for local variables and the like. Meanwhile the heap is a general-use space for allocation. Here, seven different blocks are currently checked out from the operating system.

The Heap & the Stack

Before we continue, we need to explain the heap and the stack. These are the areas of memory on a computer that we allocate to store variables and data structures.

First of all, in both C and Java we need memory every time we call a function. We need this memory for:

- all the local variables and arguments
- the address of the previous function, to return to upon completion
- other miscellaneous overhead

A function stores all this data in a structure called a **frame**, and each currently running function has its own frame. Each time a function is called, a new frame is created for it. When a function ends, its frame isn't needed anymore and it can be deleted. If a function is called recursively, the new incarnation of the function needs its own local variables and return address that are distinct from those of the old version. Therefore each call gets its own frame.

All of the frames are stored on a *stack* data structure, as shown in Figure 1. Properly, this stack is called the *execution stack*, but it is so important to a program that it is usually just called **the stack**. The top frame will always be the frame of the currently running function; it is called the **active frame**. When a program first starts up, the only frame is that of the `main()` function. But when a new function is called, its frame gets created and pushed onto the top of the stack, becoming the new active frame. When

a function ends, its frame is removed and the one beneath it is once more the active frame. Eventually the `main()` function's frame will be the only one left, and when it terminates the stack is empty, and the program is done.

By contrast, a program's **heap**[*] is organized more as a "lending library" of memory, as shown in Figure 1. Regions of contiguous memory can be "checked out" for the program's use. It is managed by the computer's operating system (OS), which has ultimate control over how much memory each active program gets. When a program needs memory from the heap, it uses a **system call** to flag the OS and request how much memory it needs. Since the OS is probably busy with other programs, this program must pause while it waits for an answer. Eventually the OS will be able to check if it has enough available memory. If so, it will likely approve the request, returning the address where the newly checked-out memory is located. But it may also deny the request and return a null pointer instead. If this happens, it will probably result in the program not being able to do its task, causing it to shut down. But assuming the request was approved, the program may use the memory for any needed data structure for as long as it wishes.

When a program is done with memory that was checked out, it is expected to **free** the memory—returning it to the heap. In C this must be done explicitly, or else it will never be returned to the heap for future use. This condition is called a **memory leak**, and it is a serious bug. If one program continues to request memory without giving any back, it will slowly eat up more and more of the computer's resources, until the computer slows to a crawl (because it is using virtual memory), and eventually other programs begin to fail due their memory requests being denied.

In general, data stored on the heap is more permanent. However, it takes more management, and allocating it can be slower because it requires the attention of the OS. By contrast, allocating data from the stack is more temporary. But it can be very fast because no system calls need to be made. Keeping this in mind, there are two major differences between memory management in Java and C:

1. In Java, local variables are always allocated from the stack, while arrays and objects are always allocated from the heap. But in C, you can often choose whether to allocate from the stack or the heap.

2. In Java, returning memory to the heap is done automatically by the Java *garbage collector*. This finds objects that aren't being used anymore, and automatically frees them. But in C, you must explicitly free memory that you previously allocated (or else have a memory leak).

[*]The heap we're talking about here is not at all related to the *heap data structure* that is used by heapsort and priority queues. Don't confuse them.

Allocating Arrays

The array declaration techniques you already know allocate memory from the stack:

```
int myArray[20]; // array of 20 ints on the stack
```

This has the problem mentioned above: the array is inherently linked to the function in which it is declared. When the function ends, so does the array. Soon, a new frame will be created, which overwrites the array.

There's another problem here that is less obvious: an array stored on the stack is stored very close to the function's return address. A malicious hacker *might* be able to trick your program into overwriting the return address, and executing hostile code![*] Thus, you should be very careful when indexing arrays with you allocated from the stack.

To make a permanent array from the heap, you must use the `malloc()` function, and store the result into a pointer variable. The name "malloc" means "memory allocation", and it is roughly equivalent to Java's `new` keyword. It takes one argument: the size of the array in bytes, and if successful it returns a pointer to the allocated memory. Here is an example:

```
int *myArray = (int*)malloc(20 * sizeof(int)); // from heap
```

This requests space for 20 `int`s: on most computers `sizeof(int)` will be `4`, and so this requests `80` bytes. Since `malloc()` returns a generic `void*`, you must cast this into an `int*`. You can permanently allocate a string in a similar way:

```
// allocate space for a length-30 string (plus 0 at end)
char *someString = (char*)malloc(31 * sizeof(char));
```

In C, arrays and pointers are almost interchangeable. After all, what is an array, but the address of a series of contiguous elements? Because the elements are contiguous, the address of the array is also the address of the first element, element `0`. But that's what a pointer is: the address of an element. So when we cast the results of a `malloc()` as, for example, an `int` pointer, we can treat it just like an `int` array. Look at this code:

[*] This is the big problem with the `gets()` function, which you should never, ever use. It takes an input string from the user but cannot control how big the input is, making it very easy for an intelligent adversary to overwrite data and hijack your program.

```
int *myArray = (int*)malloc(20 * sizeof(int)); // allocate
*(myArray + 9) = 13; // set element 9 to 13
myArray[9] = 13; // set element 9 to 13 (same thing)
```

The second and third lines do the exact same thing: they take `myArray[9]` (which is 36 bytes away from `myArray[0]` on a system with 4-byte `int`s), and set it to `13`. It is very common practice to use `malloc()` to allocate a pointer, but from that point on use it as an array.

This last detail illustrates an important detail about pointer arithmetic. You may add or subtract an `int` from a pointer. But when you do so, the `int` is first multiplied by the size of the type that the pointer points to. That is why in the above case adding `9` to an address resulted in an address `36` bytes away. This helps you to treat pointers like arrays. Further, if you subtract one pointer from another, the result is divided by the type size. So `myArray[9] - myArray[0]` is in fact `9`, even though the two addresses are `36` bytes apart.

Remember, when your program calls `malloc()`, it only *requests* memory from the heap. This request is never guaranteed: the operating system can always decide to deny it (usually when it cannot find enough space). If the request gets denied, `malloc()` will return a `NULL` pointer, which is address `0`. Thus, it is usually prudent to test that the program gets a valid address back. That way if there's a problem, the program can termiante in a controlled manner:

```
if (!myArray) {
    // allocation was denied; time to shut down gracefully
}
```

(Remember that `!` works on `int`s in C, since there are no booleans. So the code "`if (!myArray)`" is equivalent to the code "`if (myArray == 0)`".)

In Java, the garbage collector takes over your program periodically. It scans your objects and arrays, finding ones that are no longer being used and returning their memory to the heap, automatically. This makes the program less efficient, but it does make development much easier. However, C does not do this for you. Rather, you are expected to know when memory from the heap is no longer needed, and to return it explicitly. This is done via the `free()` function:

```
free(myArray); // give myArray back to the heap
```

If you do not free memory properly, you will have a memory leak bug—with all the bad effects detailed above. Every call to `malloc()` must have a corresponding call to `free()`.

To allocate a two-dimensional array, you need a pointer to a pointer. For example, the following code will allocate a 4×5 `int` array from the heap:

```
// start by mallocing the array of arrays
int **array = (int**)malloc(4 * sizeof(int*));
for (int i=0; i<4; i++) { // now for each subarray...
    array[i] = (int*)malloc(5 * sizeof(int)); // ...malloc
}
```

A three-dimensional pointer would require a `***int` type, and so on. Also remember that a string is already a `char` array. So if you want a string array, it must really be a 2D `char` array. This is why the `main()` argument `argv`, which holds an array of the command-line arguments, needs to be of type `char**`.

Segmentation Faults

One of the most common errors you will get in a C program is a **segmentation fault**. A **segment** is a region of memory on your computer, that your operating system allocates out to different programs. Both the heap and the stack are made of segments. However, the OS will not permit programs to access the wrong segment. This is for both protection and security: one rogue program shouldn't be able to interfere with the others or to bring the whole system down. Thus, whenever you try to access memory outside of your allotted region, the OS will forcibly halt your program. This is what a segmentation fault is.

These "seg faults" can be particularly frustrating, because your program often won't give you any clues as to what caused it. (Unlike Java, that will usually give you a helpful error message when it crashes.) Here are some of the most common causes behind seg faults:

- You tried to use a null pointer, or a pointer that hasn't been properly set yet. (This would usually be a `NullPointerException` in Java.)
- You tried to access an array beyond its bounds, reaching into a forbidden area of memory. (This would be an `ArrayIndexOutOfBoundsException` in Java.)
- You requested too much memory in a `malloc()`, and it returned a null pointer that you followed. (This would be an `OutOfMemoryError` in Java.
- Your recursive function could not stop and kept recursing, running out of stack space. (This would be a `StackOverflowError` in Java.)
- You accidentally overwrote the return address in a frame, and when the function ended you tried to follow it back to a forbidden address. (This cannot happen in Java.)

- You tried to use a pointer that has already been freed. (This cannot happen in Java.)
- You accidentally overwrote a pointer variable, and then tried to follow it to a forbidden address. (This cannot happen in Java.)
- You tried to use an array stored in the stack, after its function finished and its frame was overwritten. (This cannot happen in Java.)

Remember that your bug might not cause a crash immediately. It could corrupt your data in some way that only causes a segmentation fault later on.

Objects in C

As we have already stated, C does not contain classes.* This can limit the object-oriented programming that you are used to doing in Java. However, C *does* contain a way to make a sort of complex variable type, called a `struct` (short for "structure"). C's `struct` can be thought of as a `class` without any methods—it only contains fields. We will discuss how to use them here.

Defining a Basic `struct`

In its most basic form, a C **`struct`** looks much like you would expect. Here is a simple personnel record defined as a `struct`:

```
struct PersonnelRecord {
    char *familyName, *givenName; // strings
    unsigned long int idNumber; // big non-negative int
};
```

This code might go near the top of the `.c` file, near the prototypes. A `struct` has some important differences from a Java-style class:

- There are no visibility modifiers like `public` and `private`. All fields are inherently public.
- There are no methods.
- The definition must end with a semicolon.

However, `struct`s are seldom defined as they are above, because of one annoying detail. You must always use the keyword `struct` when referring to this type, such as when declaring a new variable:

*If you want to use true object-oriented programming with C-like memory management, look into the C++ language. C++ was derived from C in the early '80s. It contains real classes that are very similar to Java's classes.

```
struct PersonnelRecord myRecord; // declares new object
```

To get around this, we can use C's **typedef** ability. The typedef was originally created to make custom variable types, that were aliases for better-known types. For example, if you thought that using the keyword double was confusing, you could write code in which you define the word real to mean the same thing. Then, instead of declaring and using double variables, your code would instead use real variables, as if real were a standard C variable type.*

We can take advantage of the typedef in order to use structs more naturally. In this example, we are not technically making a struct called PersonnelRecord. Rather we are making an anonymous struct with no name, but using typedef to make PersonnelRecord an alias for this anonymous struct:

```
typedef struct {
    char *familyName, *givenName; // strings
    unsigned long int idNumber; // big non-negative int
} PersonnelRecord;
```

Now we can declare a new variable of this type without having to use the word struct:

```
PersonnelRecord myRecord; // declares new object
```

When you declare a struct like this, it is automatically allocated from the stack. Like in Java, you can assign its fields using the . ("dot") operator:

```
myRecord.familyName = "Lidell";
myRecord.givenName = "Alice";
myRecord.idNumber = 123456789;
```

Like everything allocated from the stack, it will be recycled once the function it was defined in has completed. Therefore the above code is not that useful.

Managing structs from the Heap

If you want to allocate a struct from the heap, you must declare a pointer and call malloc, like you did for arrays. This is shown here:

*Don't do this. Most people think that adding this ability to C was a bad idea, because using it makes code less standard and harder for other people to read. But typedef is still useful when making a struct.

```
PersonnelRecord *myRecord; // make an object pointer
myRecord = (PersonnelRecord*)malloc(sizeof(PersonnelRecord));
        // now allocate enough space from the heap
```

This object will persist, as long as you need it. Of course you will have to free it when you are done with it, to avoid a memory leak:

```
free(myRecord); // back to the heap!
```

Unfortunately, there is now another problem. The variable `record` is a pointer, not a `struct`, and therefore it's not valid to use the `.` operator on it to try to access a field. If you do so, you'll get a compiler error telling you that `record` is not a proper structure containing fields. You need to use the `*` operator to first access the *contents* pointed to by the pointer, which is the actual `struct`. However, under the standard C order of operations, `.` will occur before `*`. That means that you need to use parentheses, like here:

```
(*myRecord).familyName = "Lidell"; // 1st *, then .
```

To ease coding, there is another operator called the **arrow operator**, `->`. The `->` works just like a `.`, but it is used with pointers so that you don't need to use parentheses. So this line does the same thing as the above one:

```
myRecord->familyName = "Lidell"; // no ()s needed!
```

Practically speaking, you will see the `->` operator in C code much more often than you will see the `.` operator. A `struct` that was allocated from the heap is just more useful than one that was allocated from the stack, and ones allocated from the heap almost always use `->`.

Making "Methods"

A `struct` does not contain methods. Even so, C programmers often mimic them by making functions that are specially tuned to a particular `struct`.

A constructor can be made by creating a special function that takes values for all the fields and copying them into a newly allocated object:

```
// make a new PersonnelRecord, from given data
PersonnelRecord *makeNewRecord(char* familyName, char* givenName,
        unsigned long int idNumber) {
    // 1st allocate the new object
```

```
    PersonnelRecord *newRecord;
    newRecord = (PersonnelRecord*)malloc(sizeof(PersonnelRecord));
    // now fill it up
    newRecord->familyName = malloc((strlen(familyName)+1)
        * sizeof(char));
    strcpy(newRecord->familyName, familyName); // copy family name
    newRecord->givenName = malloc((strlen(givenName)+1) * sizeof(char));
    strcpy(newRecord->givenName, givenName); // copy given name
    newRecord->idNumber = idNumber;
    return newRecord;
}
```

This constructor has a very special feature: it allocates brand new strings and copies the old values into them, rather than using the strings directly. This is because strings in C are easily changeable. Making brand new strings helps to isolate this object's data, so that it won't be accidentally altered. The function starts by allocating enough space for the `struct` itself. Then it allocates enough space for the family name, and uses `strcpy()` to copy the family name argument into the family name field. Then it does the same thing with the given name. Finally it copies over the primitive ID number, and returns the brand new record.

We also need another function to handle deleting the `PersonnelRecord` when we're done with it.* This function needs to make sure that every part of the `PersonnelRecord` that was allocated from the heap is freed again.

```
// delete an old PersonnelRecord, returning it to the heap
void deleteRecord(PersonnelRecord *record) {
    free(record->familyName); // free family name
    free(record->givenName); // free given name
    free(record); // free structure itself
}
```

For other "methods", just be sure to pass the object pointers as arguments. For example, here's a function that creates a new string representing the `PersonnelRecord`, which functions kind of like a `toString()` function in Java:

```
// allocate & return string "familyName, givenName (ID#)"
char *makeStringFromRecord(PersonnelRecord *record) {
    // figure out length of ID# (log_10 + 1)
    int idLength = 1;
    if (record->idNumber > 0) idLength = (int)log10(record->idNumber) + 1;
```

*In C++, this function is called a *destructor*. A destructor is the opposite of a constructor, handling the tear-down when you're all done with an object. Java handles all of this automatically.

```
    // figure out total length & allocate string
    int length = strlen(record->familyName) + strlen(record->givenName)
        + idLength + 6;
    char *string = (char*)malloc(length * sizeof(char));
    // make string & return it
    sprintf(string, "%s, %s (%lu)", record->familyName, record->givenName,
        record->idNumber);
    return string;
}
```

Here's another function to compare two `PersonnelRecords`, which behaves similar to the `compareTo()` method inside Java's `Comparable` interface. It would be useful if you needed to sort an array of these objects. It returns `0` if two records are equivalent, a negative number if `record1` would sort before `record2`, and positive if it would sort after. It sorts the records by family name first, then given name if there was a tie, and finally by ID number if two people have the same names:

```
// compare 2 PersonnelRecords to see which is 1st
int compareRecords(PersonnelRecord *record1, PersonnelRecord *record2) {
    // 1st compare family names
    int diff = strcmp(record1->familyName, record2->familyName);
    if (diff) return diff; // return if !0
    // on tie, compare given names
    diff = strcmp(record1->givenName, record2->givenName);
    if (diff) return diff; // return if !0
    // finally compare ID #s
    if (record1->idNumber == record2->idNumber) return 0;
    else if (record1->idNumber < record2->idNumber) return -1;
    else return 1;
}
```

Conclusion

C has enjoyed a long and fruitful life, and it still influences language design today. Ultimately C++ was created by adding objects to C, and Java was created by cleaning up C++'s syntax and making it so that it could run on any computer after being compiled once.* Therefore, C tends to look primitive to modern students.

Nevertheless, there is still call for C programmers today. A lot of behind-the-scenes code for Python modules is still written in C, as is much of the code for modern operating systems. Understanding C is also a good first step toward understanding assembly language, since it translates more directly to

*The creators of C++ and Java would quite rightly argue that the process was at least a *little* more complicated than this.

assembly without as many intermediate steps. And many computer games are still written in C or C++, for the speed they can offer.

In short, you may use C every day in your coding career, or you may never use it at all. But there are undeniable benefits to understanding how the language works, especially as you explore the inner workings of your computer.