

Strings: From Java to C

Introduction: C != Java

Many students learn Java as a first language. Java makes strings easy. You can create strings easily, you can concatenate strings and other objects to create new strings, you do not need to worry about the size of a string, and when you drop all references to a string, the memory is reclaimed automatically. Most of all, a string in Java is an object; you are not supposed to know about or care about the internal structure of a string.

In C, strings are just the opposite. Creating strings is pretty easy, concatenating strings is not so easy, you must keep track of how much space a string uses, and you have to clean up your own garbage. Finally, unless you know about and care about the internal structure of a string, you will surely be plagued by segmentation violations.

This document explains to Java programmers how to understand and use strings in C.

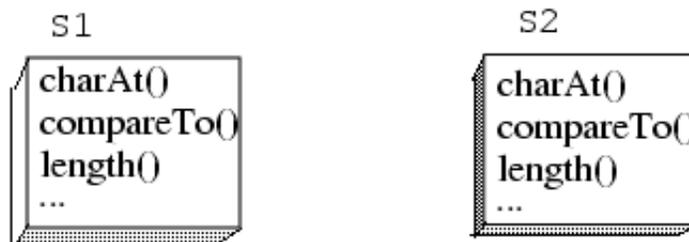
The Basic Difference

A string in C is a nul-terminated sequence of chars. That sequence occupies a contiguous set of memory locations.

A String in C is a nul-terminated sequence of chars
The standard C library contains several string functions



A String in Java is an object with several methods



A string in Java is an object, an instance of the class called String. As such, a string has a set of methods. The internal structure, storage, and location of the object is, frankly, none of your business.

Opening Remarks: C Programs are Supposed to Be Efficient

C programs are known for their speed. C compiles to machine language. The machine language runs on the *bare metal* of the CPU. There is no bounds checker for arrays, there is

no runtime system to interpret byte codes into the native environment. There is no garbage collector thread picking up and recycling unused objects.

A core principle of the C programming heritage and culture is to write code that makes sensible, effective use of memory and time. For example, when you pass a string to a function, only its address is passed to the function, not the entire string. Copying the entire string would use more memory (to hold the copy) and would take more time (to copy the bytes from the original to the copy.) Of course, if you *need* a copy of a string, you can make a copy. A fluent, skilled C programmer only copies strings when there is a clear need. Gratuitous copying and duplication of strings is inefficient, hence bad style.

Output and String Concatenation: `printf` vs +

How can you print out a combination of strings and numbers ? In Java, the natural solution looks like:

```
int i = 13;
System.out.println("Your lucky number is " + i );
```

Java automatically converts the integer value *i* into a string, combines that string with "*Your lucky number is* " and then prints that string to the system output object. Notice how Java makes it easy, in fact, almost strongarms you into creating new objects just to print out a simple message.

The C version of the same idea is:

```
int i = 13;
printf("Your lucky number is %d\n", i);
```

The `printf()` function eliminates the need to create a new string. Rather than appending the string version of *i* to the text message, `printf` simply sends the characters in the format to the output, and when it sees the `%d` notation, `printf()` converts the value to characters and prints out those characters. No new memory is allocated, no characters are copied, no garbage is created. C is much more ecologically friendly language.

A slight variation on the same solution is:

```
int i = 13;
printf("%s %d\n", "Your lucky number is", i);
```

Again, no new strings are created. The format "`%s %d\n`" tells `printf` how to interpret the two arguments.

Combining Strings: `sprintf()` vs +

Sometimes in C you need to create a new string from other strings or from different values. Here is a Java example:

```
int a = 5; int b = 7;
String msg = "The sum of " + a + " and " + b + " is " + (a+b);
```

Here is a C equivalent:

```
int a = 5; int b = 7;
char msg[ENOUGH_SPACE];

sprintf(msg, "The sum of %d and %d is %d", a, b, (a+b));
```

The C function *sprintf()* works just like *printf()* except it puts the output in an array of characters rather than sending them to standard output. In the example above, the array called *msg[]* will contain the result of inserting the values *a*, *b*, and *a+b* into the format. In this case, we do create a new string, and the size of the array, *msg*, must be large enough to hold the result.

Compare the C solution to the Java solution. In the Java code, the new string *msg* will automatically be assigned enough storage to hold the string created by the combination of the various text strings and string versions of the integer values.

There is no way of knowing exactly how the Java system will manage memory as it combines the strings and integers, so there is no easy way to know how to make the code more efficient. On the other hand, *sprintf* is simpler. It traverses the format string, copying characters to the destination string. Each time it sees a *%d*, *sprintf()* converts the next numerical value to a sequence of characters and copies those characters to the destination string.

Programming for Speed: functions vs properties

I have seen many instances of code like:

```
string name = "john q public";
for( i=0 ; i< name.length() ; i++ ){
    ...
}
```

and many instances of a C version of the same code:

```
char name[] = "john q public";
for( i=0; i< strlen(name) ; i++ ){
    ...
}
```

Both fragments work. Both make clear that the loop repeats for the number of characters in the string. What could be wrong with that code? The answer is that *strlen()* is a function. Each time through the loop, the condition *i < strlen(name)* is tested, and that test calls the *strlen()* function. But *strlen()* steps through the entire string counting characters until it sees a nul character. A more efficient use of CPU time is:

```

char name[] = "john q public";
int len = strlen(name);
for( i=0; i<len ; i++){
    ...
}

```

Here, the length of the string is computed once. That value is used as a limit for the loop. Unless you expect the string to change during the course of the loop, there is no need to check it every time through the loop.

Combining Strings: `strcpy()` and `strcat()` vs +

What if you need to combine two strings? In Java you can write:

```
String combo = "Java makes " + "string handling easy.";
```

In C, you need to write:

```

char combo[LARGE_ENOUGH];
strcpy(combo, "Java makes ");
strcat(combo, "string handling easy");

```

or you could write:

```

char combo[LARGE_ENOUGH];
sprintf(combo,"%s%s", "Java makes ", "string handling easy.");

```

Again, when you program with strings in Java, you do not need to worry about the amount of space available; Java does that for you. When you program in C, though, you have to worry about the amount of space. You need to make sure the array that will contain the combination of the strings is large enough for the result. If the new string overflows that array, your program may fail in various, unpredictable ways.

Warning: Make Sure You Have Enough Space

A popular error for new C programmers is:

```

char msg[] = "My favorite food is";

strcat(msg, " pizza");

```

This is an error because the array, *msg* created for the string is just large enough to hold the characters in the original string. The *strcat()* function will add the additional data to the end of the the string. In the example shown here, there is no additional space for the additional data. The word "pizza" will be recorded in memory that probably belongs to another variable.

One solution is to write this:

```
char msg[100] = "My favorite food is";  
  
strcat(msg, " pizza");
```

If you define an array with a specific size, the compiler will make the array that large, even if the initial string you store does not use all the space. In the example shown, the 100 chars of storage is enough to hold the initial string and the added data.

Summary: C Strings Sequences of chars in Memory

When you want to output messages from a C program, use `printf()` rather than building up a single string just so you can print it out. That's what `printf()` is for.

When you want to combine several string and/or numerical values into one string, use `sprintf()` rather than a sequence of `strcpy()` and `strcat()` calls.

`strlen()` is a function that takes cpu time to run. It is not a property of a string. Each time you call `strlen()`, you use cpu time, so call it once and store the result if you need to check the length several times.

Avoid copying and combining strings unless your need to. Copying a string requires memory and time.

Finally, and most important of all, memory for a new string does not get created automatically. You have to define it as an array of chars, or you have to use `malloc()` to create it when the program is running. Make sure the amount of memory you define or allocate is large enough for the data you store there. C does not prevent you from exceeding the space in an array.

Strings and Functions

Consider the function `fgets()`. A typical use of this function is:

```
char buff[LEN];  
  
while( fgets(buff, LEN, fp) != NULL ){
```

Notice that `fgets()` does not return a string. Instead, it stores a string in an array you provide. This is the C way to write functions that get things for you. The caller of the function provides the storage and the function fills in the storage.

It's like taking your own shopping bag to the store. You give the bag to the person at the fruit counter, and that person puts some apples in the bag. You take the bag home. If you want two bags of apples, you get two bags and send each one to the store.

The alternative is for the store to provide bags. Soon you end up with house full of bags. You need to keep calling `free()` to dispose of the bags. Of course, Java does garbage collection. With Java, you do not have to keep your house tidy; the maid cleans up after you. In C, you do your own housecleaning or you end up with a lot of wasted storage.

Therefore, when you write functions that get things, do not return a dynamically allocated chunk of memory. Demand, instead, that the caller pass a block of memory as an argument.

Learning More

Read the manual pages about the main string functions, `strlen()`, `strcpy()`, `strcat()`, `strcmp()`, and think about how they work. The basic rule is that a string in C is a sequence of characters in memory with a trailing nul character. A string is not an abstract data type with a secret internal structure. It is just an array of chars.