C Basics 6: Strings

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. **Strings**
   - **strings are char arrays**
   - **null char sentinel**
   - **comparing strings**
   - **the C String Library**
   - **strings vs. char's**
7. Library I/O

C Basics 6: Strings

8. Errors and Error Handling
9. Functions and Parameter Passing
10. Dynamic and Static Memory
11. Multidimensional and String Arrays
12. Structs (Structures)
13. Higher-Order Functions
14. Variadic Functions
15. Miscellaneous

Strings

C does *not* have a *built-in* string type (like Java does).

Instead, a "**C string**" is:

- an *array* of `char` (characters)
- terminated by the **null char**, `'\0'` (a **sentinel**)

Because they are `char` arrays, and array variables are effectively pointers, *strings are effectively represented as pointers*.

String types can be denoted as either arrays or pointers:
"`char str[]`" or "`char *str`"

It is more common to see the "`char*`" type specification for function parameters.

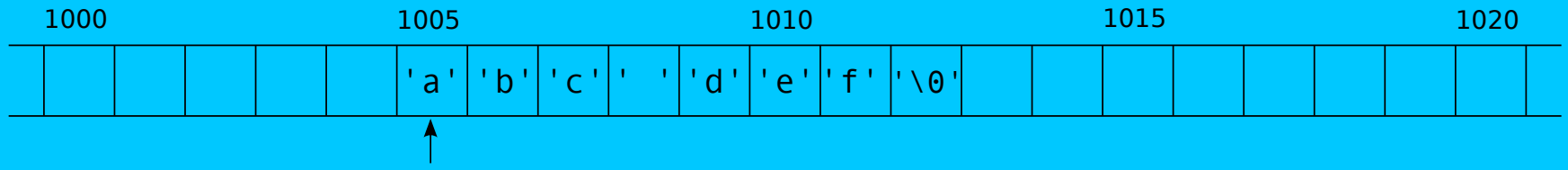
Strings (contd.)

To allocate memory for a string of a particular length, we use notation like:

`char str[4];` (can hold a three-char string + '\0')

String constants can be denoted with double-quote notation:
`"abc def"`
(same as in Java)

The constant `"abc def"` is effectively the *address* of a block of 8 bytes of memory, each byte holding a `char`:



String Initialization

C allows several different ways to initialize strings:
(all are 4-element arrays that hold strings of length 3)

```
char str[4];
str[0]='a'; str[1]='b'; str[2]='c'; str[3]='\0';

char str[4];
str = "abc";  //Error!  Not allowed by C standard
              //as would change an array pointer!

char str[] = {'a','b','c','\0'};

char str[] = "abc";

char *str = "abc";

char *str = malloc(4);
str[0]='a'; str[1]='b'; str[2]='c'; str[3]='\0';
```

The Null-Char Sentinels

Since C strings are arrays, and C doesn't maintain array sizes at runtime, the only way that C can detect the *end* of a string is by finding the *null-char sentinel*.

Failure to null-terminate strings is a common problem for new C programmers.

Remember that "reaching the array end" is meaningless in C since C doesn't know where the array end is.

The following code is likely to cause problems, since it fails to properly terminate the "string" `str`:

```
char str[4];
str[0] = 'a'; str[1] = 'b'; str[2] = 'c';
```

The Null-Char Sentinels (contd.)

(Remember that variables are not initialized in general, so the bytes in `str` will not have been "zeroed out"!)

Many string-related functions would continue tracing through `str[3]` *and the successive bytes in memory*, until a zero-byte is encountered (and interpreted as the string end).

This can cause strange results (e.g., "garbage" output) or even **segmentation faults** (illegal memory access).

If you are working with C strings and seeing "garbage" or segfaults, look carefully at your code to see if you are failing to null terminate strings.

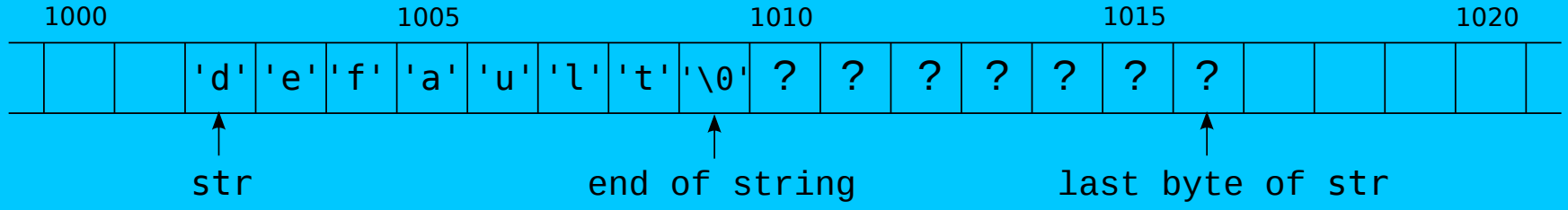
Strings Have Fixed Maximum Size

Strings are arrays, and standard arrays have fixed size.

It is common to allocate arrays large enough for the maximum size string expected and normally use only a portion of it:

```
char str[15] = "default";
```

The null-char sentinel marks the end of the used space:



(Remember that `str` can hold a string of a maximum length of 14 characters because the null-char sentinel consumes one byte!)

Comparing Strings

Because strings are not a built-in type in C, you *cannot* use the **equality operator**, ==, to test whether two strings are the same (contain the *same seqence of characters*).

Instead, you must use a string library function like `strcmp()`.

"`if (str1 == str2)...`" is *syntactically valid*, so it will not cause compiler errors, but it is *almost never what is wanted*.

What == will do is test whether the two strings are at exactly the *same memory address*—i.e., the *same string object*.

This is exactly what happens in Java with *reference types*, which is why you use `.equals()` or `.compareTo()` when comparing `String`'s.

The C String Library

Strings typically must be manipulated using library functions from the *C String Library*:

- `strlen` – length of a string (not counting final `'\0'`)
- `strcmp`, `strncmp` – *compare* two string to determine if *equal* or their *lexicographic order*
- `strchr` – determine if a `char` occurs in a string
- `strcat`, `strncat` – *concatenate* two strings (does *not allocate* dynamic memory for result)
- `strstr` – determine if a string is a *substring* of another string
- `strcpy`, `strncpy` – *copy* a string (does *not allocate* memory)

The C String Library (contd.)

Note that in order to use the C String Library functions, a program must contain the following header include:

```
#include <string.h>
```

One thing to note about the string library functions is that there is not a *substring* function.

Substrings are created using *pointer arithmetic* and `strcpy()`:

```
char full_name[] = "John Q. Smith";
char *last_name = full_name + 8;  //uses full_name array

char new_last[6];  //create space to hold separate last name
strcpy(new_last, full_name + 8);  //copy last name over

char middle_init[3];
strncpy(middle_init, full_name + 5, 2);
middle_init[2] = '\0';  //Must make middle_init valid C string
```

String Examples

Example code using strings:

```
//Make a copy of a string:
char str[20];
strncpy(str,"sample string",20);

//Read in a line from terminal:
char line[100];
fgets(line,100,stdin);

//Print out str and line:
printf("str: %s\nline: %s\n\n",str,line);

//Another way to make a copy of a string:
char *linecpy = malloc(strlen(line)+1);
strcpy(linecpy,line);

/Checking if str is "test":
if (strcmp(str,"test")==0)...
```

Strings vs. Characters

Beginning C programmers often have trouble understanding the difference between *strings* and *char's*.

Since a C string is a null-char-terminated *array* of `char`, `"A"` and `'A'` are not equivalent:

- `"A"` is of type `char*`, and is the two-element array: `'A'`, `'\0'`
- `'A'` is of type `char`, and is the single byte/`char 'A'`

Another source of confusion is the **empty string**, `""`:

- since it is a string, it must be a `char` array;
- it is the *one-element array* containing of just `'\0'`
- `""` is not the same as `'\0'`, however
- `""` and `'\0'` are of different types so not even comparable

Escape Sequences

We have seen that there are a number of **escape sequences** that can be used to represent characters.

These escape sequences can be included in strings to represent the corresponding characters.

They are heavily used to include non-printing characters such as newlines:

```
printf("The value of x is: %d\n", x);
```