Strings

Readings: CP:AMA 13

The primary goal of this section is to be able to use strings.

Strings

There is no built-in C ***string*** *type*. The "**convention**" is that a C string is an **array of characters**, terminated by a ***null character***.

```
char my_string[4]  = {'c', 'a', 't', '\0'};
```

The *null character*, also known as a null ***terminator***, is a `char` with a value of zero. It is often written as `'\0'` instead of just `0` to improve communication and indicate that a null character is intended.

`'\0'` is equivalent to `0`. That is different from `'0'`, which is equivalent to 48 (the ASCII character for the symbol zero).

String initialization

The following definitions create equivalent 4-character arrays:

```
char a[4] = {'c', 'a', 't', '\0'};
char b[4] = {'c', 'a', 't', 0};
char c[4] = {'c', 'a', 't'};
char d[4] = { 99,  97, 116, 0};
char e[4] = "cat";
char f[4] = "cat\0";
```

Because they all have a null terminator, they are also strings.

C supports an *automatic* length declaration (`[]`), where the length is determined by the initialization.

```
int a[] = {4, 8, 15, 16, 23, 42};   // length is 6
```

If you combine the automatic length declaration with double quote(`"`), initialization, it adds the null terminator for you.

```
// these are equivalent
char a[4] = {'c', 'a', 't', '\0'};
char b[] =  "cat";
```

As we will explain later, the double quotes used in array **initialization** is **different** than the quotes used in expressions (*e.g.,* in `printf("string")`).

Null termination

With null terminated strings, we do not need to pass the *length* to functions. It is determined by the location of the `'\0'`.

```
// e_count(s) counts the # of e's and E's in string s

int e_count(const char s[]) {
  int count = 0;
  int i = 0;
  while (s[i]) {  // not the null terminator
    if ((s[i] == 'e')||(s[i] == 'E')) {
      ++count;
    }
    ++i;
  }
  return count;
}
```

It is good style to have `const` parameters to communicate that no changes (mutation) occurs to the string.

strlen

The `string` library (`#include <string.h>`) provides many useful functions for processing strings (more on this library later).

The `strlen` function returns the length of the *string*, **not** necessarily the length of the *array*. It does **not include** the null character.

```
// time: O(n)
int my_strlen(const char s[]) {
  int len = 0;
  while (s[len]) {
    ++len;
  }
  return len;
}
```

Here is an alternative implementation of `my_strlen` that uses pointer arithmetic.

```
int my_strlen(const char *s) {
  const char *p = s;
  while (*p) {
    ++p;
  }
  return (p - s);
}
```

Traditionally, string functions often used pointer notation. It is slightly faster than array index notation (`s[i]`), which requires an extra addition per iteration. In modern environments, the speedup is negligible.

Do **NOT** put the `strlen` function within a loop unnecessarily.

```
int char_count(char c, char *s) {
  int count = 0;
  for (int i = 0; i < strlen(s); ++i) {    // BAD !!!!
    if (s[i] == c) ++count;
  }
  return count;
}
```

By using an $O(n)$ function (`strlen`) inside of the loop, the function becomes $O(n^2)$ instead of $O(n)$.

Unfortunately, this mistake is common amongst beginners.

This will be harshly penalized on assignments & exams.

Lexicographical order

Characters can be easily compared (`c1 < c2`) as they are numbers, so the character **order** is determined by the ASCII table.

If we try to compare two strings (`s1 < s2`), C compares their *addresses* (pointers), which is not helpful.

To compare strings we are typically interested in using a **lexicographical order**.

> Strings require us to be more careful with our terminology, as "smaller than" and "greater than" are ambiguous: are we considering just the **length** of the string? To avoid this problem we use **precedes** ("before") and **follows** ("after").

To compare two strings using a **lexicographical order**, we first compare the first character of each string. If they are different, the string with the smaller first character *precedes* the other string. Otherwise (the first characters are the same), the second characters are compared, and so on.

If the end of one string is encountered, it *precedes* the other string. Two strings are equal (the same) if they are the same length and all of their characters are identical.

The following strings are in lexicographical order:

```
"" "a" "az" "c" "cab" "cabin" "cat" "catastrophe"
```

The <`string.h`> library function `strcmp` uses lexicographical ordering.

`strcmp(s1, s2)` returns zero if the strings are identical. If `s1` precedes `s2`, it returns a negative integer. Otherwise (`s1` follows `s2`) it returns a positive integer.

```
// time: O(n), n is min of the lengths of s1, s2

int my_strcmp(const char s1[], const char s2[]) {
  int i = 0;
  while (s1[i] == s2[i] && s1[i]) {
    ++i;
  }
  return s1[i] - s2[i];
}
```

The <string.h> library function strcmp uses lexicographical ordering.

strcmp(s1, s2) returns zero if the strings are identical. If s1 precedes s2, it returns a negative integer. Otherwise (s1 follows s2) it returns a positive integer.

```
// time: O(n), n is min of the lengths of s1, s2

int my_strcmp(const char s1[], const char s2[]) {
  int i = 0;
  while (s1[i] == s2[i] && s1[i]) {
    ++i;
  }
  return s1[i] - s2[i];
}
```

To compare if two strings are *equal* (identical), use the `strcmp` function and check for **zero (false)**.

```
char a[] = "the same?";
char b[] = "the same?";
char c[] = "different";

trace_bool(strcmp(a, b) == 0);
trace_bool(!strcmp(a, b));
trace_bool(!strcmp(a, c));
```

```
strcmp(a, b) == 0 => true
!strcmp(a, b) => true
!strcmp(a, c) => false
```

Never use the equality operator (==) to compare strings. It compares the *addresses* of the strings, not their contents.

String I/O

The `printf` format specifier for strings is `%s`.

```
char a[] = "cat";
printf("the %s in the hat\n", a);
```

`printf` prints out characters until the null character is encountered.

`printf` does not print out the null character.

When using `%s` with `scanf`, it stops reading the string when a whitespace character is encountered (*e.g.,* a space or `\n`).

`scanf("%s", ...)` is useful for reading in one “word” at a time.

```
char name[81];
printf("What is your first name?\n");
scanf("%s", name);
```

Be very careful to reserve enough space for the string to be read in. **Do not forget the null character**.

`scanf("%s", ...)` automatically adds the null character.

The running time of `printf` and `scanf` with `"%s"` is $O(n)$.

example: understanding scanf

```
char name[10] = {0};
while (scanf("%s", name) == 1) {
  printf("Hello, %s!\n", name);
}
```

The input:

```
Samantha Bob [EOF]
```

Produces the following output:

```
Hello, Samantha!
Hello, Bob!
```

Afterward, what is stored in the `name` array?

B	o	b	\0	n	t	h	a	\0	\0

In the following example, the `name` array is `81` characters and can accommodate first names with a length of up to `80` characters.

```
char name[81];
printf("What is your first name?\n");
scanf("%s", name);
```

What if someone has a *really* long first name?

example 1: scanf and buffers

```
int main(void) {
  char name[8];
  char message[] = "Hello.";
  char prompt[] = "What is your name?";
  while (1) {
    printf("message: %s\n", message);
    printf("prompt:  %s\n", prompt);
    if (scanf("%s", name) != 1) break;
    printf("Welcome, %s!\n", name);
  }
}
```

In this example, entering a long name causes C to write characters beyond the length of the `name` array. Eventually, it overwrites the memory where `message` is stored, and if long enough, where `prompt` is stored.

This is known as a ***buffer overrun*** (or *buffer overflow*). The C language is especially susceptible to *buffer overruns*, which can cause serious stability and security problems.

In this introductory course, having an array with an appropriate length and using `scanf` is “good enough”.

In practice you would **never** use this insecure method for reading in a string.

example 2: scanf and buffers

```
int main(void) {
  char command[8];
  int balance = 0;
  while (1) {
     printf("Command? ('balance', 'deposit', or 'q' to quit): ");
     scanf("%s", command);
     if (!strcmp(command, "balance")) {
       printf("Your balance is: %d\n", balance);
     } else if (!strcmp(command, "deposit")) {
       printf("Enter your deposit amount: ");
       int dep;
       scanf("%d", &dep);
       balance += dep;
     } else if (!strcmp(command, "q")) {
       printf("Bye!\n"); break;
     } else {
       printf("Invalid command. Please try again.\n");
     }
  }
}
```

In this banking example, entering a long command causes C to write characters beyond the length of the `command` array. Eventually, it overwrites the memory where `balance` is stored.

It writes four `char`s into the four bytes where `balance` is stored. The value of `balance` is a “re-interpretation” of those four bytes as an `int`, instead of four `char`s.

To read in a string that includes whitespace, the `gets` function reads until a newline (`\n`) is encountered (CP:AMA 13.3).

It is also very susceptible to overruns.

```
char name[81];
printf("What is your full name?\n");
char *result = gets(name);
if (result == NULL) {
  // handle the error
}
```

The return value is either the address of the string (success) or `NULL` (failure).

There are C library functions that are more secure than `scanf` and `gets`.

One popular strategy to avoid overruns is to only read in one character at a time (*e.g.,* with `scanf("%c")` or `getchar`). For an example of using `getchar` to avoid overruns, see CP:AMA 13.3.

While *writing to* a buffer can cause dangerous buffer overruns, *reading from* an improperly terminated string can also cause problems.

```
char c[3] = "cat";   // NOT properly terminated!
printf("%s\n", c);
printf("The length of c is: %d\n", strlen(c));
```

```
cat????????????????????
The length of c is: ??
```

The string library has "safer" versions of many of the functions that stop when a maximum number of characters is reached.

For example, `strnlen`, `strncmp`, `strncpy` and `strncat`.

strcpy

The `strcpy(dest, src)` function (part of <`string.h`>)

overwrites the contents of `dest` with the contents of `src`.

```
// time: O(n), n is length of src
char *my_strcpy(char *dest, const char *src) {
  char *d = dest;
  while (*src) {
    *d = *src;
    ++d;
    ++src;
  }
  *d = '\0';
  return dest;
}
```

For historical reasons, the return value of `strcpy` is the address of `dest`. This is not useful and typically ignored.

`strcpy` can be a source of buffer overrun: always ensure that the `dest` array is large enough (and don't forget the null terminator).

`strcpy` can also cause problems if the `dest` and `src` regions overlap.

Consider this dangerous call:

```
char s[9] = "spam";
my_strcpy(s + 4, s);
```

The null terminator of `src` is overwritten, so it will continue to fill up memory with `spamspamspam...` until a crash occurs.

strcat

`strcat(dest, src)` is similar to `strcpy`, except it copies (appends or con**cat**enates) `src` **to the end** of `dest`.

```
// time: O(n + m) n,m are lengths of src,dest

char *my_strcat(char *dest, const char *src) {
  strcpy(dest + strlen(dest), src);
  return dest;
}
```

Again, ensure that the `dest` array is large enough.

String literals

C strings in quotations (*e.g.*, `"string"`) that are in an **expression** (*i.e.*, *not* part of an *array initialization*) are known as ***string literals***.

```
printf("literal\n");

printf("literal %s\n", "another literal");

if (!strcmp(s, "literal")) ...

strcpy(dest, "literal");

int i = strlen("literal");

scanf("%d", &i);
```

String literal storage

Where are string literals stored?

For each *string literal*, a null-terminated `const char` array is created in the **read-only data** section.

In the code, the occurrence of the *string literal* is replaced with the address of the corresponding array.

The *"read-only"* section is also known as the *"literal pool"*.

example: string literals

```
void foo(int i, int j) {
  printf("i = %d\n", i);
  printf("the value of j is %d\n", j);
}
```

Although no name is actually given to each literal, it is helpful to imagine that one is:

```
const char string_literal_1[] = "i = %d\n";
const char string_literal_2[] = "the value of j is %d\n";

void foo(int i, int j) {
  printf(string_literal_1, i);
  printf(string_literal_2, j);
}
```

Do not try to modify a string literal. The behaviour is undefined, and it causes an error in `Seashell`.

Note the subtle difference between the following two definitions:

```
int main(void) {
  char a[] = "mutable char array";
  char *p  = "constant string literal";
  //...
}
```

Once again, it is helpful to think of the string literal as a separately defined `const char` array.

```
const char string_literal_1[] = "constant string literal";

int main(void) {
  char a[] = "mutable char array";
  char *p  = string_literal_1;
  //...
}
```

Arrays vs. pointers

Earlier, we said arrays and pointers are *similar* but **different**.

Consider again two similar string definitions:

```
void f(void) {
  char a[] = "pointers are not arrays";
  char *p  = "pointers are not arrays";
  ...
}
```

- The first reserves space for an initialized 24 character array (`a`) in the stack frame (24 bytes).
- The second reserves space for a `char` pointer (`p`) in the stack frame (8 bytes), *initialized* to point at a string literal (`const char` array) created in the read-only data section.

example: more arrays vs. pointers

```
char a[] = "pointers are not arrays";
char *p  = "pointers are not arrays";
char d[] = "different string";
```

a is a char array. The *identifier* a has a constant value (the address of the array), but the elements of a can be changed.

```
a = d;              // INVALID
a[0] = 'P';         // VALID
```

p is a char pointer. p is initialized to point at a string literal, but p can be changed to point at any char.

```
p[0] = 'P';         // INVALID (p points at a const literal)
p = d;              // VALID
p[0] = 'D';         // NOW VALID (p points at d)
```

An array is more similar to a **constant** pointer (that cannot change what it "points at").

```
int a[6] = {4, 8, 15, 16, 23, 42};
int * const p = a;
```

In most practical expressions `a` and `p` would be equivalent. The only significant differences between them are:

- `a` has the same value as `&a`, while `p` and `&p` have different values
- The size of `a` is 24 bytes, while `sizeof(p)` is 8

Arrays of Strings

An array of strings can be defined as a 2D array of `char`s, but this approach is awkward and rarely used.

Instead, an **array of pointers** is more common.

```
char *aos[] = {"my awesome array", "of string", "literals"};
```

In the above example, `aos` is an array of pointers, with each pointer pointing to a string literal.

Even though it is not a "proper" 2D array, any `char` can be accessed as if it was in a 2D array of `chars`.

For example, `aos[0][1]` is `(aos[0])[1]`, which is `'y'`.

```c
// equivalent definition

const char str_lit_0[] = "my awesome array";
const char str_lit_1[] = "of string";
const char str_lit_2[] = "literals";

char *aos[] = {str_lit_0, str_lit_1, str_lit_2};
```

This array of pointers can be passed to a function, but as with all arrays, also pass the array length:

```c
void aos_function(char *aos[], int num_strings) { ... }
// OR
void aos_function(char **aos,  int num_strings) { ... }
```

For complicated technical reasons, do not worry about adding `const` to parameters/definitions that are arrays of pointers.

Until we learn how to use dynamic memory, defining an array of *mutable* strings is a little more awkward.

Define each mutable string separately.

```
char s0[] = "my mutable array";
char s1[] = "of strings";
char *aos[] = {s0, s1};
```

A 2D array of `chars` requires that each string is allocated the same fixed number of `chars` (regardless of the actual string length).

```
char aos2d[3][21] = {"my", "two dimensional", "char array"};
```

This is awkward because a function would need to know the fixed length in advance.

```
void aos_function(char aos2d[][21], int num_strings) { ... }
```

If necessary, the array could be “re-interpreted” (cast) as a 1D array, and the fixed lengths could be passed as parameters.

Goals of this Section

At the end of this section, you should be able to:

- define and initialize strings
- explain and demonstrate the use of the null termination convention for strings
- explain string literals and the difference between defining a string array and a string pointer
- sort a string or sequence lexicographically

- use I/O with strings and explain the consequences of buffer overruns
- use <`string.h`> library functions (when provided with a well documented interface)