

C, string.h

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

Announcements

- Homework 9 due Monday on Gradescope
- Quiz 9 opens today, due Sunday on Gradescope
- Lab 11 next Tuesday (can check off for full credit by 12/5)

man string.h

DESCRIPTION

Some of the functionality described on this reference page extends the ISO C standard. Applications shall define the appropriate feature test macro (see the System Interfaces volume of POSIX.1-2017, [Section 2.2](#), [The Compilation Environment](#)) to enable the visibility of these symbols in this header.

The `<string.h>` header shall define `NULL` and `size_t` as described in `<stddef.h>`.

The `<string.h>` header shall define the `locale_t` type as described in `<locale.h>`.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided for use with ISO C standard compilers.

- ① ISO: International Standards Organization. They standardized the C language.
- ② There might be things that are not in the standard, but meet the standard, and possibly extend it.
- ③. `NULL`: A pointer to memory position zero.
- ④. `size_t`: the size of `size_t` depends on the system. It always the same width as a pointer.
- ⑤. `locale_t`: the user's locale, meaning regional settings.

◆ 1. `size_t` 是什么？为什么要用它？

“Alright, quickly—let’s talk about `size_t`.

`size_t` is the type that C uses to represent sizes and array indices. The important thing to remember is:

👉 `size_t` is always the same width as a pointer.

That means on a 64-bit machine, `size_t` is 64-bit; on a 32-bit machine, it’s 32-bit.

Why do we care?

Because if I allocate a huge chunk of memory—say several gigabytes—I need a type big enough to index all of it. A normal `int` might overflow, but `size_t` won’t. It’s guaranteed to be large enough to describe the size of any object in memory.

So whenever you see functions like `strlen`, `malloc`, `memcpy`, they all use `size_t` because it’s the safest, most portable way to talk about ‘how big something is’.”

◆ 2. `locale_t` 是什么？为什么存在？

“Next, very briefly—`locale_t`.

`locale_t` represents the user’s locale, meaning regional settings such as:

language

character classification rules

uppercase/lowercase rules

number formatting (commas vs periods)

time formats

For example, in the US, we write:

1,234.56

But in many European countries, the same number is written as:

1.234,56

Some string functions in C use the locale to decide how characters compare or convert—for example, upper/lowercase might behave differently depending on the language.

We’re not going to dive deep into `locale_t` in this course, but it’s good to know:

👉 It lets your program behave correctly for users in different regions.”

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided for use with ISO C standard compilers.

```
void *memcpy(void *restrict, const void *restrict, int, size_t);
void *memchr(const void *, int, size_t);
int memcmp(const void *, const void *, size_t);
void *memcpy(void *restrict, const void *restrict, size_t);
void *memmove(void *, const void *, size_t);
void *memset(void *, int, size_t);
char *strcpy(char *restrict, const char *restrict);
char *strncpy(char *restrict, const char *restrict, size_t);
char *strcat(char *restrict, const char *restrict);
char *strchr(const char *, int);
int strcmp(const char *, const char *);
int strcoll(const char *, const char *);
int strcoll_l(const char *, const char *, locale_t);
char *strcpy(char *restrict, const char *restrict);
size_t strcspn(const char *, const char *);
char *strdup(const char *);
char *strerror(int);
char *strerror_l(int, locale_t);
int strerror_r(int, char *, size_t);
```

1. What is a char in C? a 1-byte integer.

A string is just an array of chars ending in a '\0' byte. Memory itself is also an array of bytes.

So fundamentally: 🖱️ Strings and raw memory are the same thing: sequences of bytes.

This is why C can freely treat strings as pointers to bytes, and why some memory functions live inside string.h.”

2. Why does string.h contain both str and mem functions?

String functions (str*) operate on null-terminated byte arrays

→ They do NOT require a size parameter.

Memory functions (mem*) operate on arbitrary byte arrays

→ They MUST take a size parameter, because raw memory has no terminator.

Both sets of functions operate on bytes, so they are grouped together in string.h.

3. strcmp and strncmp — string comparison functions

“strcmp(s1, s2) compares two strings until one hits the null terminator: returns 0 if the strings are equal, negative if s1 < s2, positive if s1 > s2.

strncmp(s1, s2, n) compares only the first n characters. It will stop early if either string hits the null terminator ('\0').

These functions only work on C strings, meaning they must be null-terminated.”

4. memcmp — compare ANY memory block (including structs)

“memcmp(ptr1, ptr2, size) compares two memory regions byte by byte.

It does not care about: null terminators, structure layout, data types. It only looks at bytes.

You can use memcmp to compare two structs. However, you must be careful:

Structs may contain padding bytes. Padding bytes may contain uninitialized values, so two structs with identical fields may still compare as ‘not equal’ if the padding differs.

5. memset — initialize raw memory

memset(ptr, value, size) writes a byte value to every byte of a memory block.

The most common use is: memset(&s, 0, sizeof(s));

This clears all fields and also clears padding bytes.

It ensures that structs initialized this way can be safely compared using memcmp.

6. memcpy — copy raw memory

memcpy(dest, src, size) copies exactly size bytes from one memory region to another.

For simple structs (without internal dynamic memory), memcpy works perfectly:

```
memcpy(&s2, &s1, sizeof(s1));
```

This is equivalent to `s2 = s1;` but works for any data type. Because C treats everything as bytes, memcpy becomes a universal tool for duplicating data.

```

#include <string.h>
#include <stdio.h>

void pick_a_string(char *s, size_t length);

int main() {
    char a[20];
    char b[40];
    pick_a_string(a, 20);
    pick_a_string(b, 40);

    // Warm up! Is there an issue with
    // this code? Can you think of a
    // possible problem?

    int c = strcmp(a, b); strcmp(a, b, 20);

    if (c < 0) puts(a);
    else if (c > 0) puts(b);
    else puts("both the same");

    return 0;
}

```

if I have $a = \overbrace{aaa \dots a}^{20 \text{ a's}}$
 $b = \underbrace{aaa \dots a}_{20 \text{ a's}} z$

$c == 0$? a and b are the same.?

No! I cannot have 20 a's in array a . I need a $\backslash 0$

$\backslash 0 == 0$

so a is smaller

Do you see any potential issues in this code?

`strcmp` compares two strings by looking at each byte until it hits a $\backslash 0$.

If the string is not null-terminated, it keeps reading... possibly into garbage memory, return addresses, or stack frames.

If `pick_a_string` forgets to add a null terminator, `strcmp(a, b)` might scan off the edge of the buffer. That's undefined behavior, and it can crash your program—or worse, allow malicious code to hijack your program.

How to Be Safe — Use `strncmp`

This limits the number of characters `strncmp` will compare. Even if there's no null terminator, it won't read past the 20th byte.

It's a safety net — use it when you're dealing with manually built strings or unsure input.

```
#include <stdlib.h>
```

```
void *mcp(void *dest, const void *src, size_t n) {  
    for (size_t i = 0; i < n; i += 1)  
        ((char *)dest)[i] = ((char *)src)[i];  
    return dest;  
}
```

—

```
clang -O1 -S mcp.c
```

```
.LBB0_4:                                     # =>This Inner Loop Header: Depth=1  
    movzbl    (%rsi,%r10), %ecx  
    movb      %cl, (%rax,%r10)  
    addq      $1, %r10  
    addq      $-1, %rdi  
    jne       .LBB0_4
```

The implementation is correct, but the professor emphasizes that it is slow, for several reasons:

Reason 1 — Each loop iteration only transfers 1 byte.

But each iteration also requires multiple instructions: load 1 byte, store 1 byte, increment index, decrement counter, branch/jump

So the majority of the work is loop overhead, not data copying.
This is fundamentally inefficient for large memory blocks.

Reason 2 — Using cl means you are doing 8-bit work

cl is the lowest 8 bits of the 64-bit register rcx

Because you cast to char*, the compiler assumes it's doing byte-by-byte operations, so it uses:

movzbl → load a single byte
movb → store a single byte

Why is that bad?

Because:

The CPU can copy 8 bytes in one 64-bit move

Or 16 bytes using SSE

Or 32 bytes using AVX

Or 64 bytes using AVX-512

Using cl means you're only using 8 bits of your register bandwidth, wasting the CPU's natural 64-bit or 128-bit capabilities.

Reason 3 — The loop overhead is too large

```
addq $1, %r10  
addq $-1, %rdi  
jne .LBB0_4
```

These are not doing useful copying—they exist only to keep the loop going.
When the loop runs millions of times, this overhead dominates the runtime.

The standard library versions use advanced optimizations such as:

1. loop unrolling
2. copying 4, 8, 16, 32 bytes per iteration
3. SIMD vectorized instructions
4. alignment optimizations
5. architecture-specific strategies
6. cache-friendly techniques

if you use -O3 instead of -O1, you can see loop unrolling and vectorization.

```
.LBB0_6:                                     # =>This Inner Loop Header: Depth=1
    movzbl    (%rsi,%r10), %ecx
    movb      %cl, (%rax,%r10)
    movzbl    1(%rsi,%r10), %ecx
    movb      %cl, 1(%rax,%r10)
    movzbl    2(%rsi,%r10), %ecx
    movb      %cl, 2(%rax,%r10)
    movzbl    3(%rsi,%r10), %ecx
    movb      %cl, 3(%rax,%r10)
    addq      $4, %r10
    cmpq      %r10, %rdx
    jne       .LBB0_6
    jmp       .LBB0_17
```

```
.LBB0_12:                                     # =>This Inner Loop Header: Depth=1
    movups    (%rsi,%rdi), %xmm0
    movups    16(%rsi,%rdi), %xmm1
    movups    %xmm0, (%rax,%rdi)
    movups    %xmm1, 16(%rax,%rdi)
    movups    32(%rsi,%rdi), %xmm0
    movups    48(%rsi,%rdi), %xmm1
    movups    %xmm0, 32(%rax,%rdi)
    movups    %xmm1, 48(%rax,%rdi)
    movups    64(%rsi,%rdi), %xmm0
    movups    80(%rsi,%rdi), %xmm1
    movups    %xmm0, 64(%rax,%rdi)
    movups    %xmm1, 80(%rax,%rdi)
    movups    96(%rsi,%rdi), %xmm0
    movups    112(%rsi,%rdi), %xmm1
    movups    %xmm0, 96(%rax,%rdi)
    movups    %xmm1, 112(%rax,%rdi)
    subq      $-128, %rdi
    addq      $-4, %r9
    jne       .LBB0_12
```

there are special registers that do vector math so they can load multiple values in at a time.

```

#include <string.h>
#include <stdio.h>

int main() {
    char buffer[100];
    const char *s1 = "This is a test";
    const char *s2 = " of string.h";
    buffer[0] = '\0';
    strcat(buffer, s1);
    strcat(buffer, s2);
    puts(buffer);
    return 0;
}

```

1. strcat does NOT check the size of the destination buffer

```

char buffer[100];
buffer[0] = '\0';
strcat(buffer, s1);
strcat(buffer, s2);
strcat assumes:

```

- buffer is a valid, null-terminated C string
- There is enough space to append the new data

But it never verifies this.

If the combined length of s1 + s2 is > 99 bytes, strcat will overflow the buffer.

This is one of the most common causes of buffer overflow bugs in real C code.

2. If we forget buffer[0] = '\0', the program becomes dangerous

```

char buffer[100];
strcat(buffer, s1);

```

buffer contains uninitialized garbage.

strcat will search through that garbage looking for a '\0', which can result in:

- writing past the end of the array
- reading uninitialized memory
- a crash
- completely undefined behavior

So although this code works, it relies on the programmer manually doing the right thing.