C, string.h

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

Announcements

- Homework 9 due tonight on Gradescope
- Lab 11 tomorrow(can check off for full credit by 12/5)
- No homework or quiz over break

strcat example

```c
#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;
}
```

Issues:

① May Segmentation fault if s1 or s2 are long. (buffer overflow).

② If I forgot '\0', strcat doesn't work.

" clang problem.c -fstack-protector "

it will show " stack smashing detected " (It gives more error information, some systems do this by default)

printf

```
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

a pointer to a format string (in register rdi)

telling the type checker: I don't care whats coming next.

We have calling conventions. The compilers will put the parameters in the right places, like registers, floating point registers or stack

(man 3 printf : check format string).

1. Format String Structure

The manual says:

A *format string* contains:

- **Ordinary characters:**
  These are copied *exactly as-is* into the output.
- **Conversion specifications:**
  These begin with `%` and tell `printf`:
  - that an argument must be consumed
  - how to interpret that argument
  - how to convert it into characters for output

The format string must contain *very specific information*, because without it:

`printf` would not know what arguments to read, in what order, or where in registers they live.

2. Components of a Conversion Specification

Each `%` item can include:

- optional **flags**
- optional **field width**
- optional **precision**
- optional **length modifier**
- a required **conversion specifier** (the final character)

“The simplest form is just `%` followed by a specifier character.”

```c
#include <stdio.h>

int main() {
    printf("23534624754765☺ç–½8");
    printf("x   x  \n\t");
    printf("\n------\n");
    int x = -2130;
    printf("A number: %d <=- like that\n", x);
    printf("A number: %o <=- like that\n", x);
    printf("A number: %u <=- like that\n", x);
    printf("A number: %x <=- like that\n", x);
    printf("A number: %X <=- like that\n", x);
    printf("%d + %d = %d (%s)\n", 2, 3, 2+3, "yay", 34);
    return 0;
}
```

printf prints exactly the bytes you give it
It does not automatically add a newline (unlike puts)

So we need to add '\n' to print a new line.

Conversion specifiers: a percentage sign, followed by what type of the thing is.

1 . Setting up a negative integer

```
int x = -2130;
```

We will print this same integer using different conversion specifiers.

2 . `%d`: signed decimal

```
printf("A number: %d <=-- like that\n", x);
```

Explanation:
`%d` prints the value **as a signed decimal integer**.
So the result is simply:

```
A number: -2130 <=-- like that
```

3 . `%o`: unsigned octal

```
printf("A number: %o <=-- like that\n", x);
```

Explanation:
`%o` interprets the bits of the int as an **unsigned integer**, then prints them in **base 8**.
Because `x` is negative, its bits are in **two's complement**, so interpreted as *unsigned*, it becomes a very large number.

Example concept:

```
-2130 (signed) → 0xFFFFF780 (unsigned
interpretation)
then printed in octal
```

This is why the result is a strange large octal number.

4 . `%u`: unsigned decimal

```
printf("A number: %u <=-- like that\n", x);
```

Explanation:
Again, interpret the bits as an **unsigned int**, then print in **decimal**.
This produces a huge number near `2^32`.

5 . `%x` and `%X`: hexadecimal

```
printf("A number: %x <=-- like that\n", x);
printf("A number: %X <=-- like that\n", x);
```

Explanation:

- `%x` prints the **hexadecimal representation** using lowercase `a-f`.
- `%X` prints the same value using uppercase `A-F`.

Since `x` is negative, C prints the two's-complement form:

Example (conceptually):

```
x = -2130
binary (32-bit) = FFFFF780
%x prints   fffff780
%X prints   FFFFF780
```

6 . Multiple arguments example

```
printf("%d + %d = %d (%s)\n", 2, 3, 2+3, "yay",
34);
```

! The format string only has 4 specifiers:

- `%d`
- `%d`
- `%d`
- `%s`

But the programmer passed **5 arguments**:

`2, 3, 2+3, "yay", 34`

Explanation:

- The first four arguments match the four format specifiers.
- The last argument (`34`) has **no corresponding % placeholder**.
- The compiler will **warn** but the program will still run.
- Extra arguments are simply **ignored** by `printf`.

Output:

```
2 + 3 = 5 (yay)
```

Passing Too Many Arguments:

```
useprintf-canned.c:13:53: warning: data argument not
used by format string [-Wformat-extra-args]
```

Passing Too Few Arguments:

```
useprintf-canned.c:13:28: warning: more '%' conversions
than data arguments [-Wformat-insufficient-args]
   13 |     printf("%d + %d = %d (%s)\n", 2, 3, 2+3);
```

Why the Compiler Warns for `printf`

The compiler *special-cases* `printf` because it's extremely common and extremely important.

Normally, C variadic functions **cannot** be type-checked. But for `printf`, the compiler actually **parses the format string** and checks argument count/mismatch.

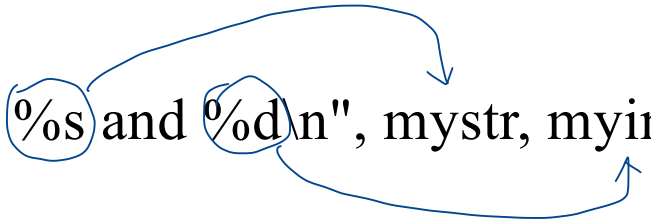
printf

Could you write printf?

printf

```
int printf(const char *format, ...);
```

```
printf("hi: %s and %d\n", mystr, myint);
```



You'd scan the format string one character at a time.

When you see %, you parse the specifier.

That tells you the type of the next argument.

You fetch the argument and convert it (e.g., convert int → its decimal characters).

You output everything to some destination.

"Where does the output actually go?"

`printf` ultimately sends characters to stdout**, which is a file-like object.**

To do that, `printf` must eventually call **the system call `write`**.

All the processes running on the machine : ps -A | less

How many things are running ? ps -A | wc -l (All of them have a view of memory as if they were the only program running in memory.

Processes

Process - approximately what we think of as a "running program"

- Operating System effectively has a giant array of processes started since computer turned on
- Try `ps -A`
- Has access to all memory (but only its own!)
- Operating System maintains data structure about each process
  - What program is running, who ran it, when it started, …
  - Array of "file like objects"

Processes

(a big array of processes. I can see information and stores information about all the processes are currently running).