CS 241: Systems Programming
Lecture 33. Variadic Functions

Spring 2020

Prof. Stephen Checkoway

Student evals are online

Primary learning goals from course website

- the UNIX command line (in particular the BASH shell)
- a command line editor like Neovim, Emacs, or Nano
- Various command line utilities
- the Git version control system
- C compilers like Clang and GCC
- debuggers like GDB
- linting tools like shellcheck.

More learning goals

More learning goals

- how to write safe shell scripts (specifically BASH-flavored shell scripts);
- how and especially when to program in C;
- what undefined behavior is;
- what memory safety is;
- how to use Github;
- how to set up continuous integration with Travis-CI; and
- how to work with regular expressions.

Parameters vs. arguments

Parameters: variables in a function declaration/definition

Arguments: the data you pass to functions

```
void foo(int x, float y) { /* … */ } // Parameters: x and y

foo(37, 8.2f); // Arguments: 37 and 8.2f
int direction = 8;
float scale = -15e-6;
foo(direction, scale); // Arguments: direction and scale
                       // or 8 and -15e-6
```

Variable number of arguments

Need a way to handle variable length argument lists

- Format strings
  - `printf(char const *fmt, ...);`
  - `scanf(char const *fmt, ...);`
- Sentinel value (special value that marks the end, often **`NULL`**
  - `execl(char const *path, char const *arg0, ...);`
- Additional parameter when given specific fixed arguments
  - `open(char const *path, int flags, ...);`
  - `fcntl(int fd, int cmd, ...);`

Variable arguments in C

Two mechanisms (used to be) available:

```
#include <varargs.h>
```

- Old style, not supported — do not use!

```
#include <stdarg.h>
```

- New style — do use!

Types

Somewhere in `stdarg.h` there is

```
typedef /* stuff */ va_list;
```

Need one of these as an argument pointer

```
va_list ap;
```

Function prototypes

Use "`...`" in function prototype

```
void varfoo(char const *fmt, ... );
```

Variable argument marker `...` must be

- At the end of the parameter list
- Following at least one fixed parameter

Using variable arguments

Three macros used

- `va_start(va_list ap, last)`
- `va_arg(va_list ap, type)`
- `va_end(va_list ap)`

There's a fourth one that's rarely used

- `va_copy(va_list dest, va_list src)`

va_start

Macro used to initialize argument pointer

`va_start(ap, last);`

- `ap` — argument pointer
  - initialized to the first argument
- `last` — last fixed parameter in the parameter list

```
void foo(int x, int y, int z, ...) {
  va_list ap;
  va_start(ap, z);
  // ...
}
```

va_arg

Macro used to access arguments

Returns next argument in list; advances to the next position

Needs to know type of the next argument

```
double dbl = va_arg(ap, double);
char const *str = va_arg(ap, char *);
```

va_end

Macro to clean environment up when done

```
va_end(ap);
```

Each `va_start()` and `va_copy()` must be paired with a va_end() in the same function

```c
void strange_print(int next, ...) {
  va_list ap;

  va_start(ap, next);
  while (1) {
    switch (next) {
    case 'i': printf("%d", va_arg(ap, int)); break;
    case 'f': printf("%f", va_arg(ap, double)); break;
    case 's': printf("%s", va_arg(ap, char *)); break;
    default: va_end(ap); return;
    }
    next = va_arg(ap, int);
  }
}

strange_print('i', 37, 's', "text", 'f', .25, 0);
```

Open (from musl libc)

Open takes a third parameter (the file system permissions) when creating a file

```
int open(const char *filename, int flags, ...) {
 mode_t mode = 0;
 if ((flags & O_CREAT) || (flags & O_TMPFILE) == O_TMPFILE) {
  va_list ap;
  va_start(ap, flags);
  mode = va_arg(ap, mode_t); // file creation permissions
  va_end(ap);
 }
 // ...
}
```

When **implementing** a function with a variable number of arguments, how does the programmer know how many arguments there are?

A. Use the `va_number(ap)` macro

B. Format string specifies the number of arguments

C. An explicit "sentinel" value is used at the end of the argument to mark the end

D. The number of additional arguments is passed as a parameter

E. *Some* mechanism must be used to indicate how many there; it varies by function

What do you think happens if the program accesses more arguments than were passed to the function or an argument of the wrong type?

A. This is prevented by the type system (i.e., a compiler error)

B. The default value of 0 is returned

C. A garbage value is returned

D. The program segfaults

E. It's undefined behavior

Implementing printf via vfprintf

```
int printf(char const *fmt, ...) {
  va_list ap;
  va_start(ap, fmt);
  int ret = vfprintf(stdout, fmt, ap);
  va_end(ap);
  return ret;
}
```

Implementing `vfprintf` involves reading the format string character by character and deciding what argument to read next based on the character after a `%`

In-class exercise

https://checkoway.net/teaching/cs241/2020-spring/exercises/Lecture-33.html

Grab a laptop and a partner and try to get as much of that done as you can!