The basics of programming in C

```
char* argv = malloc(4 * 1024 * 1024);

while ((bytes_read = read(argv_fd,
        argv + bytes_read_total,
        4 * 1024 * 1024 - bytes_read_total))) {
…
```

```
char* argv = malloc(4 * 1024 * 1024);

while ((bytes_read = read(argv_fd,
        argv + bytes_read_total,
        4 * 1024 * 1024 - bytes_read_total))) {
…
```

1. Use a wrapper that aborts when an allocation fails:
   char* argv = fs_xmalloc(4 * 1024 * 1024);

```
void* fs_xmalloc(size_t len)
{
   void *x = malloc(len);
   assert(x != NULL);
   return x;
}
```

Pros:

1. Fails visibly when overcommit is disabled, or when the limit on the number of VMAs is reached.
2. Can add instrumentation, e.g. accounting allocations, tracking allocation sites, etc.

```
char* argv = malloc(4 * 1024 * 1024);

while ((bytes_read = read(argv_fd,
        argv + bytes_read_total,
        4 * 1024 * 1024 - bytes_read_total))) {
…
```

1. Use a wrapper that aborts when an allocation fails:
   `char* argv = fs_xmalloc(4 * 1024 * 1024);`

2. Make use of properties of file systems that you work with:
   `char* argv = fs_xmalloc(4 * 1024 * 1024);`
   `read(argv_fs, argv, 4 * 1024 * 1024);`

We read from a procfs file.

1. Reads cannot be partial.
2. Reads do not block.
3. Reads from `cmdline` and `env` do not fail.

Quiz: why does nginx and similar software do file IO in a threadpool?

```
char* argv = malloc(4 * 1024 * 1024);

while ((bytes_read = read(argv_fd,
        argv + bytes_read_total,
        4 * 1024 * 1024 - bytes_read_total))) {
…
```

1. Use a wrapper that aborts when an allocation fails:

```
char* argv = fs_xmalloc(4 * 1024 * 1024);
```

2. Make use of properties of file systems that you work with:

```
char* argv = fs_xmalloc(4 * 1024 * 1024);
read(argv_fs, argv, 4 * 1024 * 1024);
```

3. Use constants to make the code more maintainable:

```
const size_t argv_max = 4 * 1024 * 1024;

char* argv = fs_xmalloc(argv_max);
read(argv_fs, argv, argv_max);
```

```
if (close(argv_fd) < 0) {
```

```
if (close(argv_fd) < 0) {
```

```
close(argv_fd);
```

A call to `close()` may fail if it needs to write something. We make no writes, hence there is no room for `close()` to fail.

Error handling is a must, but choose only errors that can be acted upon.

```
snprintf(path_buf, sizeof(path_buf),
    "/proc/%d/exe", pid);

report_error(path_buf, errno);
```

```
snprintf(path_buf, sizeof(path_buf),
    "/proc/%d/exe", pid);

report_error(path_buf, errno);
```

Library calls that fail are guaranteed to update `errno` to contain the reason for a failure.

Library calls that succeed are guaranteed neither to set `errno` to zero, nor to keep it unchanged.

```
snprintf(path_buf, sizeof(path_buf),
   "/proc/%d/exe", pid);

report_error(path_buf, errno);
```

Library calls that fail are guaranteed to update `errno` to contain the reason for a failure.

Library calls that succeed are guaranteed neither to set `errno` to zero, nor to keep it unchanged.

```
ssize_t getrandom(void *buf, size_t len, int flags)
{
    // try the fast syscall first
    ssize_t x = SYS_getrandom(buf, len, flags);
    if (x >= 0)
        return x;

    // errno contains ENOSYS
    // use /dev/urandom as a fallback
    int fd = open("/dev/urandom", O_RDONLY);
    …
}
```

A successful call to `getrandom()` may set `errno` to a non-zero value.

```
snprintf(path_buf, sizeof(path_buf),
   "/proc/%d/exe", pid);

report_error(path_buf, errno);
```

Library calls that fail are guaranteed to update `errno` to contain the reason for a failure.

Library calls that succeed are guaranteed neither to set `errno` to zero, nor to keep it unchanged.

```
ssize_t getrandom(void *buf, size_t len, int flags)
{
    // try the fast syscall first
    ssize_t x = SYS_getrandom(buf, len, flags);
    if (x >= 0)
        return x;

    // errno contains ENOSYS
    // use /dev/urandom as a fallback
    int fd = open("/dev/urandom", O_RDONLY);
    …
}
```

A successful call to `getrandom()` may set `errno` to a non-zero value.

Additional reading: vDSO and how Linux implements syscalls like `gettimeofday()`.

```
void print_symlink(const char* path)
{
   static char buf[PATH_MAX + 1];

   ssize_t sz = readlink(path, buf, sizeof(buf));
   if (sz < 0) {
      report_error(path, errno);
      return;
   }
   buf[sz] = '\0';

   report_file(buf);
}
```

```
void print_symlink(const char* path)
{
   static char buf[PATH_MAX + 1];

   ssize_t sz = readlink(path, buf, sizeof(buf));
   if (sz < 0) {
      report_error(path, errno);
      return;
   }
   buf[sz] = '\0';

   report_file(buf);
}
```

Make functions static unless they are part of your public API:

```
   static void print_symlink(const char* path)
```

Pros:

1. The function will not be added to the table of exports of the executable file.
2. More room for optimisation because the compiler knows all callers of the function.

```
static char *create_hello_string()
{
    static char result[50];
    pid_t pid = fuse_get_context()->pid;
    sprintf(result, "hello, %d\n", pid);
    return result;
}
```

```
static char *create_hello_string()
{
    static char result[50];
    pid_t pid = fuse_get_context()->pid;
    sprintf(result, "hello, %d\n", pid);
    return result;
}
```

1. This is not MT-safe.

```
static char *create_hello_string()
{
    static char result[50];
    pid_t pid = fuse_get_context()->pid;
    sprintf(result, "hello, %d\n", pid);
    return result;
}
```

1. This is not MT-safe.

2. `snprintf(res sizeof(res), “hello, %d\n”, pid)`

Do not use functions like `sprintf()` because they do not limit the number of bytes that they read or write. That is a recipe for a memory corruption.

```
buf = realloc(buf, cmd_buffer_size);
if (buf == NULL)
    goto cleanup;
```

```
buf = realloc(buf, cmd_buffer_size);
if (buf == NULL)
    goto cleanup;
```

1. This is a memory leak. A call to `realloc()` does not free the previous buffer.

Remark: FreeBSD has `reallocf()` which frees the previous buffer.

2. All remarks on `fs_xmalloc()` apply to `fs_xrealloc()` as well.

```
if (fd_path != NULL)
    free(fd_path);
```

```
if (fd_path != NULL)
    free(fd_path);
```

```
free(fd_path);
```

When designing your APIs, make them friendly to "`goto out`"-style cleanup.

```
fd_path = (char *) malloc(INITIAL_BUFFER_SIZE);
```

```
fd_path = (char *) malloc(INITIAL_BUFFER_SIZE);
```

```
fd_path = fs_xmalloc(INITIAL_BUFFER_SIZE);
```

```
while ((entry = readdir(proc_dir)) != NULL) {
    if (entry->d_type == DT_DIR) {
        int pid = atoi(entry->d_name);
        if (pid > 0) {
            …
```

```
while ((entry = readdir(proc_dir)) != NULL) {
    if (entry->d_type == DT_DIR) {
        int pid = atoi(entry->d_name);
        if (pid > 0) {
            …
```

This style leads to too much indentation.

Prefer this way:

```
while ((entry = readdir(proc_dir)) != NULL) {
    if (entry->d_type != DT_DIR)
        continue;

    int pid = atoi(entry->d_name);
    if (pid <= 0)
        continue;

    …
```

```
#include <stddef.h>

int main()
{
   const size_t N = 2*1024*1024;
   char *envp[N];

   envp[0] = "oops";

   return 0;
}
```

```
#include <stddef.h>

int main()
{
   const size_t N = 2*1024*1024;
   char *envp[N];

   envp[0] = "oops";

   return 0;
}
```

Do not allocate big variables and arrays on the stack.

Linux does not grow the stack by more than 256 pages at a time, and `pthread_create()` starts new threads with 2M stacks by default.

```
#include <stddef.h>

int main()
{
   const size_t N = 2*1024*1024;
   char *envp[N];

   envp[0] = "oops";

   return 0;
}
```

Do not allocate big variables and arrays on the stack.

Linux does not grow the stack by more than 256 pages at a time, and `pthread_create()` starts new threads with 2M stacks by default.

Quiz: what is the difference between code generated by

```
gcc -O2 test.c
```

and

```
gcc test.c
```

?

Why does the debug version segfault, and the optimised version run without an error?

Hint: use `` `objdump -d` `` to disassemble the executable file in both cases.