The basics of programming in C

```
int bytesRead = read(fd, cmdline, ARGV_MAX – 1);
close(fd);

if (bytesRead == –1) {
    report_error(path, errno);
    …
```

```
int bytesRead = read(fd, cmdline, ARGV_MAX – 1);
close(fd);

if (bytesRead == –1) {
    report_error(path, 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.

```
int bytesRead = read(fd, cmdline, ARGV_MAX – 1);
close(fd);

if (bytesRead == –1) {
    report_error(path, 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.

```
int bytesRead = read(fd, cmdline, ARGV_MAX – 1);
close(fd);

if (bytesRead == –1) {
    report_error(path, 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()`.

```
char ** splitToStrings(char *buffer, const size_t size)
{
    char **res = malloc((size/2 + 1) * sizeof(char *));
    …
```

```
char ** splitToStrings(char *buffer, const size_t size)
{
    char **res = malloc((size/2 + 1) * sizeof(char *));
    …
```

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

```
static char ** splitToStrings(…)
```

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.

```
char *buffer = malloc(size);
if (!buffer) {
    …
```

```
char *buffer = malloc(size);
if (!buffer) {
    …
```

```
char *buffer = fs_malloc(size);
```

Do not add handling of errors that cannot reasonably occur.