Project 0
Coding Style Guidelines

"Learn the rules like a pro, so you can break them like an artist."
Pablo Picasso

Penn OS Staff

Directions
These are important notes on style and convention for your reference. A portion of your Project 0 grade will be based on your code quality, and these same guidelines will be used for all projects in this class. Feel free to ask follow-up questions about style on Piazza.

Naming
Pick a convention and stick with it. Two popular conventions are `camelCase` and `under_score` Names should be descriptive of the variable's significance or the function's purpose. Name all your .c and .h files with the same convention as well.

Macros and `#define` constants are in `ALL_CAPS` by convention.

Headers
Header `(.h)` files are most effectively used to provide formal interfaces which are implemented in a corresponding .c file with the same name. Header files that may be included in multiple other headers or source files should use `#ifndef` guards to prevent accidental redeclarations. E.g.

```
/* myheader.h */
#ifndef MYHEADER_H
#define MYHEADER_H

void some_useful_function();

#endif
```

Constants
Define constants in C using `#define` macros, e.g.

```
#define BUFFER_SIZE 1024
```

This can greatly improve readability of your code and avoid painful bugs. Don't use variables to store constants. Also, don't hardcode numeric values, such as buffer size, the length of string literals, `STDIN_FILENO`, or `STDOUT_FILENO`.

Commenting

Comments should help the reader to understand the purpose of your code when it would otherwise be unclear. Block comments are useful in specifying the contract of a particular function.

This is a good use of block commenting:

```
/* Divides the dividend by the divisor, and places the quotient in the result
 * and returns TRUE on success.
 *
 * If dividing by zero, returns FALSE and result is unchanged.
 */
int int_divide_by(int dividend, int divisor, int *result);
```

This is an unnecessary use of commenting:

```
// iterate over chars until null
for (char* c = str; *c != '\0'; c++) { ... }
```

In addition, do not submitted commented-out code, even if it is extraneous.

Magic Numbers

The term magic number refers to using numbers directly in source code without explanation. In most cases this makes programs harder to read, understand, and maintain. Although most guides make an exception for the numbers zero and one, it is a good idea to define all your numbers in code as named constants. There are also several constants predefined in C that you can use. You can find more information on it here: https://en.wikipedia.org/wiki/Magic_number_(programming).

Debugging Information

If you want to leave in debugging information but don't want it to run, use a macro like this one:

```
#define DEBUG

#ifdef DEBUG
#define IF_DEBUG(x) x
#else
#define IF_DEBUG(x) ((void)0)
#endif

...
IF_DEBUG(printf("Process id: %d\n", pid));
```

Then, to turn off the debug statements (called compiling them out), just remove or comment out the `#define` DEBUG line.

Input Validation

Validate all input at the top of the function. You can avoid unnecessary code complexity by checking for invalid cases in an if conditional and returning immediately. E.g.

```
int parse_input(char* in) {
        if (!is_valid(in)) {
                return FALSE;
        }

        // parse in here, no else needed
        return TRUE;
}
```

Global Variables

Avoid the use of global variables when possible.

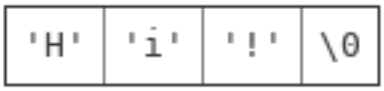
Compilers and Compiler Flags

You are free to use either `clang` or `gcc` to compile your projects, although in general we recommend clang because it's newer and provides more useful error messages. Always use the `-Wall` flag to report (almost) all warnings the compiler catches. Don't submit code with compiler warnings. Include the `-g` flag when building code for valgrind or gdb (see below).

Additional Tips:

Calling write(2) and using sizeof

Be careful of off-by-one errors when doing this. Strings in C are null-terminated. Therefore, when you write `char message[] = "Hi!";` what you get in memory is a pointer to the following array:



So when you compute `sizeof(message)`, you get 4 because it's an array of size 4. Then, when you call:

```
write(STDOUT_FILENO, message, sizeof(message)); // BAD
```

you print the three chars in the string plus the null character. But standard output streams are not null-terminated, so your terminal interprets the null character as a space. Therefore, you should instead call:

```
write(STDOUT_FILENO, message, sizeof(message) - 1); // BETTER
```

`strlen(3)`, by contrast, returns three (3) when called on message. It has the additional feature of working properly on dynamic memory and strings not defined as arrays in the current context.

```
#include <string.h>

write(STDOUT_FILENO, message, strlen(message)); // STILL BETTER
```

Note though that if your string is not null-terminated, this exposes you to a buffer overflow.

Using the return value and status of wait(2)

It's very important that your shell continues in its loop only once you are sure your child process has exited. Note that wait(2) has both a return value and a status code. For example:

```
int status = 0;
int returnValue = wait(&status);
```

Just because wait returns, this does not mean your child has exited. There are at least two important possibilities:

- `wait` encountered an error and returnValue = -1. E.g. your parent receives an unblocked signal (like Ctrl+C).
- `wait` returns because it has information to report about your child besides termination. E.g. the child was stopped.

If you continue in either of these cases, you risk leaking processes (i.e. forking new children before old ones terminate) which is very bad. Therefore, you need to check for a successful return value and for the information in status using the `W*` macros detailed on the man page.

Malloc and Free

You will be using malloc (or calloc) and free very often in this class. Being very careful about how and when you call them will help you avoid a lot of heartache due to weird behaviors, segfaults, and memory leaks.

When calling malloc, it is highly suggested to use sizeof to dynamically determine the number of bytes needed. For example:

```
char* foo = malloc(<number of characters> * sizeof(char))
```

This becomes even more important as you start writing and malloc-ing your own structs, as the size of your structs may change as you change your mind about what data to store where and in what form.

A best practice is to always free data structures in the same context as they were allocated. Hidden calls to mallocs and frees are a common cause of memory leaks and memory errors. As a general rule, unless you have a very good reason to do so (such as a function that was specifically written to cleanly destroy a structure), a function should avoid freeing memory that it didn't allocate or allocating memory and expecting the caller to free it.

A very useful trick to avoid mallocing when you know exactly how many bytes you need is to allocate memory on the stack. Since that memory is on the stack there's no need to free it. For example, if you happen to know you always need a buffer of 10 chars, you can do something like this instead of calling malloc:

```
function foo() {
  char[10] buffer;
  // do stuff with the buffer.
}
```

Note that this also means that the memory buffer points to will not be available after leaving the function foo, as that stack memory will be re-used by later function calls.

Recommended Tools

All of these programs are installed on the speclab machines and will help you throughout this class

- `vim/emacs` - Terminal-based text editors. Pick one and learn it. There are a lot of great tutorials for either online.
- `tmux` - Pane/window/session management. Tmux improves your workflow greatly (e.g. screen splitting saves you the effort of maintaining multiple ssh sessions and tmux can maintain persistent sessions on linux boxes).
- `valgrind` - Use this to check for memory errors and leaks (with `--leak-check=yes`). You make also want to look at the `--child-silent-after-fork` option.
- `gdb` - Terminal-based program debugger.
- `scan-build` - Static analysis tool. Reports bugs like uninitialized variables.