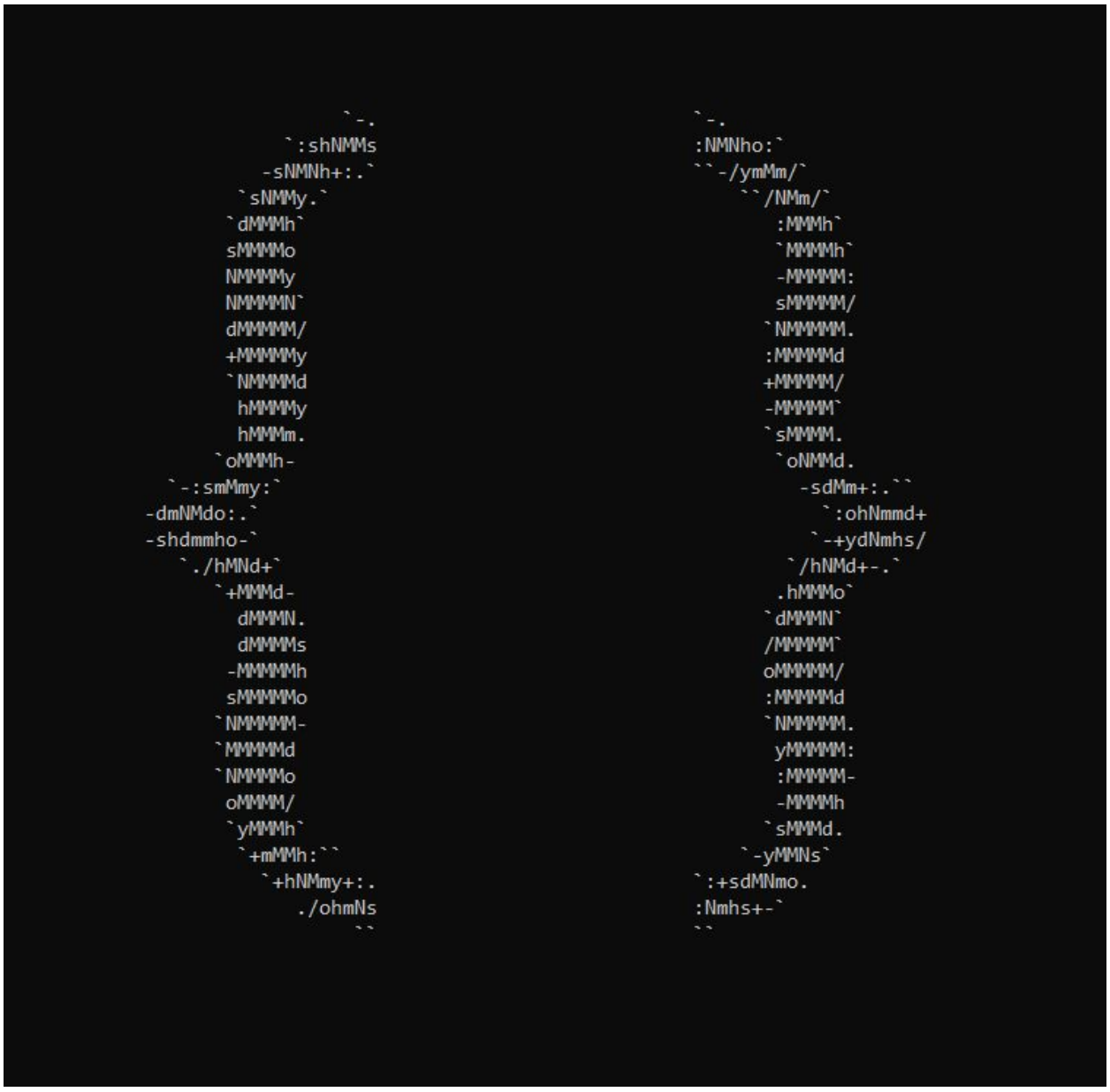
Code Standard for PegasOS

Version 1.1.0



C Files

Every .c file MUST have a commented header at the beginning of the file. This will serve as a description of the file's contents as well as the author, editors, and other metadata.

Each line will be denoted by a single line comment, followed by a tab, then the following sections:

```
//  Sample C File
//  Author: Christopher Walen
//  Editor(s): Giancarlo Guillen, Jacob Thomas
//  Date Created: 5/4/2020
//  Summary: This file is for sending processes to the scheduler.
//  Last Edited By: Jacob Thomas (5/4/2020)
```

Following the commented header, will be the include linking to the header file that is associated with the .c file. Any other included libraries should be included in that header file.

```
//  Sample C File
//  Author: Christopher Walen
//  Editor(s): Giancarlo Guillen, Jacob Thomas
//  Date Created: 5/4/2020
//  Summary: This file is for sending processes to the scheduler.
//  Last Edited By: Jacob Thomas (5/4/2020)

#include "sample.h"
```

After the header #include, you can define your functions. Global variables and constants should be declared in the associated header file.

```
#include "sample.h"

int rectangleArea(int _width, int _height)
{
    return _width * _height;
}
```

Header Files

Every .c file MUST have a .h file associated with it. Global variables and constants should be declared in the header file, all necessary data structures, all necessary enumerators, and all functions should be prototyped here. The order of items should proceed as follows:

Header files must be initialized and formatted as so:

```c
#ifndef H_SAMPLE
#define H_SAMPLE

#include <stdio.h>
#include <stdlib.h>

#define EXAMPLE_CONST 10

typedef enum brands
{
    NONE, COCA_COLA, PEPSI, NESTLE
} brands;

typedef struct exampleStructA
{
    int val1;
    int val2;
    char ltrA;
} exampleStructA;

typedef struct exampleStructB
{
    int value1
    int value2;
    char letterA;
} exampleStructB;

int fooA();
int fooB();

#endif
```

This will allow you to include the header file whenever you need it, but the code will only appear once in compilation which will save us a lot of bloat.

You may indent your header file(s) if you wish.

```c
#ifndef H_SAMPLE
    #define H_SAMPLE

    #include <stdio.h>
    #include <stdlib.h>

    #define EXAMPLE_CONST 10

    typedef enum brands
    {
        NONE, COCA_COLA, PEPSI, NESTLE
    } brands;

    typedef struct exampleStructA
    {
        int val1;
        int val2;
        char ltrA;
    } exampleStructA;

    typedef struct exampleStructB
    {
        int value1
        int value2;
        char letterA;
    } exampleStructB;

    int fooA();
    int fooB();

#endif
```

Structures

Please *typedef* your structs, so that we can use them as normal types and avoid typing ‘struct’ all over the place. When you define your struct, use both the typedef convention and the normal convention, like so:

```
typedef struct rectangle
{
    int width;
    int height;
} rectangle;
```

You are NOT allowed to capitalize the first letter of structures. Try to limit the words in a structure’s name to one word. Use camelCase if it is more than one word long.

Indentation

Tabs, not spaces. This is non-negotiable.

Exception:
You are allowed to use spaces for long argument lists, such that they line up in other editors with different tab sizes. Here, the second line uses one tab followed by spaces to get the ‘6’ to line up with the ‘1’.

```
int secondFunc()
{
    myFunc(1, 2, 3, 4, 5,
           6, 7, 8, 9);
}
```

Blocks should be on new lines, as in the example above. There should not be any opening curly-braces on the same line as the function name. This is the same for loop-blocks, if-blocks, if-then-blocks, and switch-blocks.

A block of code should be delineated if the logic has moved onto a new level of scope. This will clearly separate scope between functions, conditionals, and loops such that we can understand what variables can be accessed by different pieces of logic.

Switch Statements

Switch blocks should be formatted as follows:

```
int foo(int _type)
{
    switch (_type)
    {
        case 1:
        case 2:
        case 9:
            printf("Statement A\n");
            break;
        case 3:
        case 5:
        case 7:
            printf("Statement B\n");
            break;
        case 4:
        case 6:
            printf("Statement C\n");
            break;
        default:
            printf("No Statement Found\n");
            break;
    }
}
```

Fall-through cases should be in either a numerical order, or a grouping order that makes sense for the given variable type (i.e. alphabetical for chars, or the order of an enum, etc.).

Defaults do not need to be included if they are not necessary. If a default is needed, please place it where it makes sense in the logic of the switch statement. In the above example, the default can go at the end since it represents not finding a statement, so we should see all of the valid statements first to understand why we might not have found a valid statement. Defaults may fall-through to other cases if another case is the ‘default case’.

Comments

Comments should be done on a single line as much as possible. If a comment is too long, it may be broken up into two separate lines. Comments should not be more verbose than is necessary; it should explain what that block of code or function is doing if it is not immediately apparent. Trivial functions/calculations do not need to be explained.

```
// This function performs several operations on a number of inputs
// The operations are arbitrary, and only serve as an example
int myFunc(int a, int b, int c, int d, int e, int f, int g, int h)
{
    int temp = a;
    temp *= (b - c);
    temp += (d * e * f);
    temp /= a;
    temp += (g * g) - (h * h);

    return temp;
}
```

You are allowed to have block comments, though they should be reserved for complex sections of code that require detailed explanations.

You may section parts of your code using the following style:

```
//==========   Geometric Shapes   ==========//

typedef struct rectangle
{
    int width;
    int height;
} rectangle;

//==========   Vector Operations   ==========//

typedef struct vector2
{
    int x;
    int y;
} vector2;

vector2 multiplyScalar(vector2 _vector, int _scalar)
{
    vector2 _temp;

    _temp.x = _vector.x * _scalar;
    _temp.y = _vector.y * _scalar;

    return _temp;
}
```

Loops

Loops of all kinds should have the block beginning on a new line. Variables used for iteration do not have to follow the guidelines for regular variables: they may be single letters and are not required to have an underscore granted they are of limited or local scope. They may also be somewhat descriptive of their meaning in the iteration.

```
for (int i = 0; i < 500; i++)
{
    printf("%i\n", i);
}
```

```
for (int yy = 0; yy < 10; yy++)
{
    for (int xx = 0; xx < 10; xx++)
    {
        printf("%i", _array[xx][yy]);
    }
    printf("\n");
}
```

```
int count = 0;
do
{
    printf("%i\n", count++);
}
while (count < 500);
```

Conditionals

If statements and if-else statements should have the block beginning on a new line, and must be clearly encapsulated with opening and closing brackets even if the chosen branch has only one line of code.

```
if (inputFile != NULL)
{
    logicLoop();
}
else
{
    return -1;
}
```

You may use single-line conditionals to save space, however it must either be commented or be readily apparent what the logic represents.

```
char _member = (person.hasMembershipCard) ? 'y' : 'n';
```

```
int _returnCode = (!feof(inputFile)) ? calculateChecksum(inputFile) : encryptFile(inputFile);
```

If a piece of logic requires a large number of if-statements that are enumerable, consider using a switch statement instead.

```
if (optionA)
{
    printf("A\n");
}
else if (optionB)
{
    printf("B\n");
}
else if (optionC)
{
    printf("C\n");
}
```

VS

```
switch (_option)
{
    default:
    case optionA:
        printf("A\n");
        break;
    case optionB:
        printf("B\n");
        break;
    case optionC:
        printf("C\n");
        break;
}
```

Constants

Do not use the const prefix, instead use #define.

All #define variables should be in CAPITAL LETTERS ONLY. Use underscores where a space would make sense.

```
#define MAX_WIDTH 50
#define MAX_HEIGHT 50
```

Globals

Global variables may use the *static* keyword, and should not have a preceding underscore. They should be declared before any functions are defined, after the associated .h file is included. Like regular variable names, they should be descriptive without being overly verbose or too short.

```
#include "sample.h"

static FILE* in;
static FILE* out;
```

#Define

You may use #define for more than just constants or ‘magical numbers’. Please be sure to leave comments about what a particular non-constant #define does if it is not immediately apparent.

```
// Access the position in the memory array,
// aligned to the 4th bit
#define MEM(addr) (Mem[addr >> 2])
```

Variables

Please spell out names for variables. I would much rather see this:

```
typedef struct exampleStructB
{
    int value1
    int value2;
    char letterA;
} exampleStructB;
```

Than this:

```
typedef struct exampleStructA
{
    int val1;
    int val2;
    char ltrA;
} exampleStructA;
```

However, if you're using a well-known equation and wish to use the variables in that equation, that is fine. Otherwise, do not use single-letter names.

```
int a = 20;
int b = 10;
int h = sqrt(a * a + b * b);
```

The primary reason for this is it avoids confusion about what a variable is meant to do, so that if someone else is looking at the code they can better follow what is going on.

For local variables, prefix an underscore to the variable. This will avoid confusion if a function contains the same variables that a structure has, for example.

```
typedef struct rectangle
{
    int width;
    int height;
} rectangle;

int rectangleArea(int _width, int _height)
{
    return _width * _height;
}
```

Multiple Variables of the Same Name

You are allowed to prepend additional letters before the underscore of a variable name if you must have the same name as that of a pre-existing variable in the function or in a structure. This should not be necessary however, and is more of a ‘catch-all’ than a requirement.

Functions

camelCase for function names. If the same function name appears in another section of code, be more specific with your function name. If you absolutely must have the same name, prepend the first letter of the file name or component name, followed by an underscore.

```
int t_calculateArea(int _base, int _height)
{
    return (_base / 2) * _height;
}

int r_calcualteArea(int _width, int _height)
{
    return _width * _height;
}
```

Just like with the variables, don’t be afraid to spell out the function names.

```
int exampleFunc()
{
    printf("Example Function\n");
}
```

VS

```
int explFunc()
{
    printf("Example Function\n");
}
```

Line Length

Keep the length of your lines of code as short as possible while maintaining readability. For a more general guide-line, do not have lines longer than 128 characters including indentation. Depending on the level of scope in a piece of logic this may not be possible, in which case keep the length of the text itself without indentation to a maximum length of 128 characters.