1 Meaningful Names.

Names should carry some indication of the use of the content of code particles. Variables, enums, structs, functions and classes should all have meaningful names. What constitutes a meaningful name is highly contingent on the context. For example, naming a variable min in a clock program is probably acceptable. However, naming a variable min in a program to compute the minimum time for marathon runners is ambiguous.

1.1 Clearly Unacceptable

```c
int foobar(int a, int* b) {
    return b[1]*a;
}
```

1.2 Questionable

```c
int mult_2nd(int n, int* arr) {
    return arr[1]*n;
}
```

1.3 Clearly Acceptable

```c
int scale_y_coord(int scalar, int* coords_array) {
    return coords_array[1]*scalar;
}
```
2 Explanatory Comments.

Comments should explain the procedure to a competent programmer. Sections of the code whose intent may not be obvious should be commented in a way that lets other programmers know why the current design choice was chosen over others. Comments should list any assumptions the programmer is making about non-local code.

2.1 Clearly Unacceptable

//do some pointer voodoo and then call foobar
int* temp = malloc(sizeof(int));
temp* = x;
foobar(temp, 2);

2.2 Questionable

int* temp = malloc(sizeof(int));
temp* = x;
//Used foobar because it is the only function
//to provide both foo and bar services.
foobar(temp, 2);

2.3 Clearly Acceptable

//foobar doesn’t say it doesn’t keep the reference,
//so we have to make a copy in the heap.
//Hopefully, foobar has a way of deleting the copy...
/*
 * Fix this if anyone ever properly documents foobar
 * or a memory leak is detected!
 */
int* temp = malloc(sizeof(int));
temp* = x;
foobar(temp, 2);

3 No superfluous Comments.

Comments should explain the procedure to a competent programmer. Comments should not waste a reader’s time. Do not put comments that only
explain what every competent programmer would know from reading the code.

3.1 Clearly Unacceptable

d = d + 7; //add 7 to d
h = 0;    //set h to zero
m = 0;    //set m to zero

3.2 Questionable

d = d + 7; //Come on baby,
h = 0;     //Daddy needs a new pair of shoes!
m = 0;

3.3 Clearly Acceptable

d = d + 7; //Move to midnight of next week
h = 0;
m = 0;

4 Consistent Indenting Scheme.

There are several indenting schemes in use. A programmer should be aware of them and stick to a particular scheme in a project. A programmer should also be flexible enough to go with the group on which indenting scheme to use. Part of the indenting scheme is whether to use tabs or spaces, or some well-defined combination of both. The acceptable mixture of tabs and spaces is based upon the team members choice of IDE/editors. Setting the editor to expand tab stops to 6 spaces will generally show any inconsistencies in use of tabs and spaces.

4.1 Unacceptable

/* This function provides both foo and bar services.
 * The counter is the variable to be decremented
 * each time a foo event occurs.
 * The decrement is the amount to subtract
 * from the counter each time the system
 * gets fooed.*/
void foobar(int* counter, int decrement)
{
    if (fooCount > fooLimit)
    {bar(3);} else
    {if (fooCount == 0)
        activateBar();
    //if you reorder these, pay attention to the
    //autoincrement
    foos[fooCount] = counter;
        foo_decrementors[fooCount++] = decrement;
    }
}

4.2 Acceptable

/* This function provides both foo and bar services.
 * The counter is the variable to be decremented
 * each time a foo event occurs.
 * The decrement is the amount to subtract
 * from the counter each time the system
 * gets fooed.
 * When the counter reaches zero (or below)
 * the bar service will go off.
 * If there are already too many foo services,
 * will call bar with a status code of 3.
 */
void foobar(int* counter, int decrement) {
    if (fooCount > fooLimit)
        bar(3);
    else {
        if (fooCount == 0)
            activateBar();

        //if you reorder these,
        //pay attention to the autoincrement

        //autoincrement
        foos[fooCount] = counter;
            foo_decrementors[fooCount++] = decrement;
    }
}
5 Inclusion of Clarifying Whitespace.

Similar to paragraph separations in English text, vertical whitespace between objects, methods, structs and functions provides a visual clue to readers about the separation of logical code blocks. Similarly, inside long functions and methods, whitespace may indicate a logical break between main code blocks. Code without such whitespace clues make it difficult for a programmer to find a desired section of code.

5.1 Clearly Unacceptable

```c
int modulus(int dividend, int divisor) {
    if (decrement <= 0) {
        return -1;
    } // fi
    bool fix = FALSE;
    if (dividend < 0) {
        fix = TRUE;
        dividend = -dividend;
    } // fi
    while (dividend - divisor > 0) {
        dividend = dividend - divisor
    } // elihw
    if (fix) {
        dividend = dividend - divisor;
        dividend = -dividend;
    } // fi
    return dividend;
}
```

5.2 Questionable

```c
int modulus(int dividend, int divisor) {
    if (decrement <= 0) return -1;
    bool fix = FALSE;
```
if (dividend < 0)
{
    fix = TRUE;
    dividend = -dividend;
}
while (dividend - divisor > 0)
    dividend = dividend - divisor
if (fix)
{
    dividend = dividend - divisor;
    dividend = -dividend;
}
return dividend;

5.3 Clearly Acceptable

int modulus(int dividend, int divisor) {
    if (decrement <= 0)
        return -1;

    bool fix = FALSE;
    if (dividend < 0) {
        fix = TRUE;
        dividend = -dividend;
    }

    while (dividend - divisor > 0)
        dividend = dividend - divisor
    if (fix) {
        dividend = dividend - divisor;
        dividend = -dividend;
    }

    return dividend;
}