CODING STANDARD DOCUMENT

FOR THE CISOR COMPILER

Version 1, August 2022

Author: Philippe Caron

This document contains the coding standard for the CISOR compiler and its components.

Contents

Definitions

Line context — When programming, we often need to look in the immediate vacinity of the line being written for contextual cues on its impact. The code that is logically tied to the line being written will be referred to as "context". The size of the context is a measure of how many lines are in that code.

Stack context — The code that will be executed with the same stack pointer, generally indicated by accolades ({}) in C.

Fragmentation — The number of stack contexts (*not line context*) required to executed the code. For instance, a long list of sequential instructions would be completly whole, splitting that list into multiple functions would be fragmented. The ultimate fragmenation would be if each instruction was executed in a different stack context (i.e. a function for each instruction), that would obviously be very exagerated.

Banner — A banner is a comment structure to catch the eye and delimit subdivisions in the code. A few common ones are: //******//, //======//, //////////, etc.

1 Foreword

1.1 Aim

The aim of this coding standard is to maximize quick information absorption by the programmer as well as to minimize the strain on the programmer's mental state. There are a couple dimensions across which those objectives can be evaluated. For each dimension, the aim will be specified: all the way to one end (factor), all the way to the other (not a factor), or a balance of both.

Time to write (not a factor) Time to write will NOT be considered, if a few milliseconds are required to improve the code on relevant categories, that improvement is necessary. No advantage in information absorption would be gained by cutting corners, and it provides very minimal strain reduction.

Format (factor) Format should always be respected. Standard format allows the programmer to reflexivly look for variables without thinking. This improves both objectives; it accelerates information absorption, and reduces strain.

Density (balance) It is well known that a well aerated code is pleasant to read, as a general rule the more aerated the lesser the strain. At the same time, the denser the code, the more information is condensed, making it possible to scroll less, and increasing the likelyhood that the full context can be displayed on a single page. A reasonable balance between both should be achieved.

Fragmentation (factor) Fragmentation plays into density a bit, but has additional effects. More fragmenation reduces the size of the context and increases information absorption by reducing complexity of concepts, which in turn reduces the strain. Unlike density though, it has a concrete meaning in the compilation process. Too much fragmenation will decrease overall performance of the program (note that this can be solved with the `inline` keyword in C++), which is a special case because usually style has no effect on the output. Keeping in mind that the impact is very marginal though, and that it improves both objectives, code should always be fragmented to the limit of intuitive action. As a rule of thumb, a function's purpose should be possible to summarize into a single sentence, and should fit into a single screen.

Screen width (factor) Even if most programmers have more than one screen, the standard should reflect the use of a single screen split in two. Therefore the physical length of lines (and overall file) should be considered. This doesn't especially concern information speed, but strain will be significantly reduced by making file smallers and lines short enough to fit a half-screen.

Self similarity (factor) Mental orientation of the programmer within the code can sometimes be very difficult. This is in my experience the biggest source of strain, especially on files exceeding 1000 lines. Although those large files should be avoided, sometimes situations arise where they are necessary, and in that moment self similarity becomes the programmer's greatest enemy. As much as possible, and without impacting the other parameters, the programmer should make code sections visually unique (for instance by making custom banners before different subdivisions). This will make orienteering a lot quicker and easier.

Commentary (balance) A "reasonable" ammount of commentary is very hard to define because it will vary widely between a novice and an experienced programmer, and even more so between a newcomer, and a frequent contributer. With all that in mind, complicated functions should be thoroughly documented in a separate file. Comments should not be used as a way to explain the code if that explanation requires more than one line, rather they should point to a reference in the documentation. Similarly function purposes should be self-explanatory from their names and parameter names, and useless commentary (e.g.: "FILE*

open(char *name) // This function opens a file specified in the name, and returns a pointer to it") should be avoided. Commentary should mostly be used to reduce the context size by including cues to distant sections. This will reduce strain without becoming tautologic.

1.2 IDE

The CISOR compiler project is developped mostly in the VS Code IDE (Linux) and Visual Studio 2019 (Windows). On Windows the use of Visual Studio is unavoidable, but on Linux it is not required to use the VS Code IDE (scripts are provided), or any IDE for that matter. This being said it is assumed that the programmer who wishes to edit this code will use an editor with colorizing features. For this reason, programmers should NOT use letter codes when defining variables. For instance, most corporate coding standards suggest using "m" for member variables, thus the class member "name" would be named "mName". In my opinion, this is an archaic standard that has been rendered redundant by a modern editor's standard features. Apart from the uselessness of the letter prefix, it also makes the code heavier, harder to read, and all-in-all hinders rapid identification by increasing self-similarities within the code.

1.3 Paradigm

It is also important to note that the first version of the CISOR compiler is developped with the intent of testing the backwards compatibility of the C* language, for this reason it is written in C, and should be able to compile itself. The subsequent version of CISOR should be written in C*. Although compatible, C is quite different to what C* aims to be. Most notably in that C is an imperative procedural language, and C* will be an object-oriented functionnal language. In order to keep a somewhat similar code structure across languages, the C written version of the compiler will contain some functions written in "standard C fashion" (meaning procedural), some others in a more functionnal style, and some more in an object-oriented approach. This coding standard specifies cues for the programmer to employ in order to make switching styles easier. By enforcing a slightly different coding standard for different styles, the hope is that the programmer will be able to almost subconsciously adapt their thinking based on the visual environment. This is important because switching paradigm frequently can lead to proper and effecient solutions that would normally be obvious to the programmer being overlooked in favor of subpar solutions from the previous paradigm. For instance, consider an objected-oriented style memory allocation function (constructor), and its procedural counterpart. In the OOP case, you expect the function to return a pointer, because the call to `malloc` should happen within the function (and the call to `free` should be in an eventual deletion function, the destructor). On the other hand, in the procedural case, the allocation function should return a boolean indicating if the initilization was successful, the pointer of preallocated memory should be passed as an in/out parameter. Similarly, `free` should be called outside of (after) the deletion function. Without an obvious way to distinguish between those two styles, a programmer moving from OOP to procedural could easily forget to free the memory after deletion, leading to a very hard to trace memory leak (no crash). Conversly, a programmer switching from procedural to OOP might free twice, leading to a double-free corruption. Either way, the standard will help in avoiding these mistakes.

2 Procedural C

Procedural C refers to C code written in the traditionnal imperative procedural fashion. The main concerns in defining the coding standards for procedural C are (1) pointer management and (2) internal consistency with the language's libraries.

2.1 Naming convention

The names of global variables (mainly functions) and files should NOT utilize capitalized letters. Even though the modern standard is the Java Object naming convention (ex: javaObjectNamingConvention). The purpose of this deviation from norm is to differentiate between procedural functions/objects and object-oriented functions/objects.

Function names should be easy to remember and follow the format set by existing library equivalents. For instance, considering the functions `fopen`, `popen`, a function that return a pointer to a stream should follow a similar pattern (ex: `sopen`). They should NOT contain any underscore. By contrast, objects should contain an underscore to seperate words.

2.2 Memory and structure management

Constructor functions should contain the prefix `cons`, they should take an allocated pointer of the right size as a parameter. **The constructor function is not responsible to check for** `NULL`. The allocation function should not modify the pointer in any way, merely populate the memory pointed by it. Calls to `malloc` should always contain a call to `sizeof` (in its function form, not operator form), even for `char`, although it's redundant: it will maintain the standard and convert to the proper type.

Destructor functions should contain the prefix `free`. Analogous to `free` itself, they will free the memory pointer, but also call any subdestructor required.

2.3 Pointers

In C, the pointer sign is a unary operator, not a type modifier. To highlight this relationship, like other unary operators, it should be preceding the variable and NOT following the type, which might be a hard habit to lose for a C++ programmer. This distinction will also help in creating a different mindspace when switching between C and C++ code.

2.4 Initilization

The initilization of variales should ALWAYS take place first in the function. Variable names and equal signs should be aligned. If new variables are required in the function, a new stack context should be defined by the use of accolades.

2.5 Formatting

Logical sections should be separated by a newline. The opening accolade of a function should be after a newline, but other accolades can be inline (unless they are folling a multiline statement). Similar consecutive lines should have their logical components aligned so that they can be read like a table. Unless necessary, functions should return at a single point.

2.6 Includes

Includes should be in alphabetical order and there should be an empty line between native includes and project includes.

2.7 Example

```c
#include <string.h>
#include <stdio.h>

// Underscore to signify object
struct my_object {
  int   id;
  char *name; // Pointer on the variable, not the type
};

// No capital letters, "cons", function banner
////////////////////////////////////////////////////////////////////////////
int consmyobject(struct my_object *new_object, int id, char *name)
{
  // Aligned variables in initilization block
  char *tmp     = malloc(strlen(name) * sizeof(char)); // Use of sizeof
  int   success = 0;

  // Logic block separated by empty line
  if (tmp) { // Inline bracket
    new_object->id   = id;
    new_object->name = tmp;

    success = strcpy(name, new_object->name);
  }

  return success; // Single return point
}

// "free"
////////////////////////////////////////////////////////////////////////////
void freemyobject(struct my_object *new_object)
{
  free(new_object->name);
  free(new_object);
}

////////////////////////////////////////////////////////////////////////////
int main(int argc, char *argv[])
{
  struct my_object *the_object = malloc(sizeof(struct my_object));

  if (the_object &&
      argc == 2  &&
      consmyobject(the_object, 0, argv[1]))
  { // No inline bracket on multiline condition
    printf("%s\n", the_object->name);
  }

  {
    char *why = "This is an example of a stack context block";
    printf("%s\n", why);
  }

  return 0;
}
```

3 Object-oriented C

Although it is not an object-oriented language, some concepts of object-oriented programming can be integrated into C programming. The code produced is significantly different, hence why it has a separate coding standard.

3.1 Naming convention

The names of functions that utilize object-oriented programming should follow the Java object naming convention (ex: javaObjectNamingConvention). Functions should start with a lowercase letter, and object types should start with an uppercase letter. Object types should be defined using `typedef`.

3.2 Memory and structure management

Constructor functions should contain the prefix `new`, they should be entirely safe to call in any context. **They are responsible for all pointer allocations and checks**.

Destructor functions should contain the prefix `delete`. They should take the location of the pointer to be deleted as a parameter to ensure that the pointer can be marked null, and that no doulbe deletion is possible, they should be entirely safe to call in any context.

Contrary to the procedural standard, the checks are hidden with the intent of simplifying the use. This makes double deletes or allocation failures safe, and so careful attention should be exercised when using this standard. In practice though, their use is a lot easier.

3.3 Pointers

Pointer standards don't change.

3.4 Initilization

Initilization standards don't change.

3.5 Formatting

Function standards don't change.

3.6 Example

```
#include <string.h>
#include <stdio.h>

#include <procedural.h>

// Use of typedef
typedef MyObject struct my_object;

// "new"
////////////////////////////////////////////////////////////////////////////
MyObject *newMyObject(int id, char *name)
{
  MyObject *myObject = malloc(sizeof(MyObject));

  if (!consmyobject(id, name)) {
    free(myObject);
  }

  return myObject;
}

// "delete"
////////////////////////////////////////////////////////////////////////////
void deleteMyObject(MyObject **myObject)
{
  if (*MyObject) {
    freemyobject(*myObject);
    *myObject = NULL; // clear pointer
  }
}

////////////////////////////////////////////////////////////////////////////
int main(int argc, char *argv[])
{
  MyObject myObject = NULL;

  if (argc == 2  && (myObject = newMyObject(0, argv[1]))) {
    printf("%s\n", myObject->name);
  }

  return 0;
}
```