

## Programming style

## Keep in mind

### A good program is not one that just works

Just working is not enough. A good software program is one that is

- Easy to read
- Easy to understand
- Well organized
- Well commented and documented

## Why paying attention to style?

Programming style is fundamental for many reasons:

1. It simplifies program reading & comprehension;
2. It facilitates program maintenance;
3. It reduces the possibility of making mistakes;
4. It allows quickly identifying syntactic and semantic errors;
5. It avoids irritating project reviewers.

Hence, adopt these rules **since the beginning!**

## Style rules

For all these reasons, programmers should follow some general guidelines aimed at improving source readability.

In particular, they concern the following aspects:

1. Horizontal spacing
2. Vertical spacing
3. Indentation
4. Comments
5. Code organization

## Horizontal spacing

It refers to a set of rules to follow to separate objects contained in the same line of code.

A **TAB** is not equivalent to a set of spaces!!!

```

... | ... | ... | ... | ... | ... | ... | ... |
int→  i;
char→  c;
float→x;

int...k;
char...a;
float...y;

```

Use an editor that does not replace **TABS** with spaces.

## Horizontal spacing

### Declarations

Insert a TAB after each type identifier.

```

int→   i;
float→ x;

```

### Multiple declarations

Insert a SPACE after each comma.

```

int→   i, .j, .k;
float→ x, .y, .z;

```

## Horizontal spacing

### Expressions

operators with 2 operands must have a space in both sides:

```
→ x = (a + 2) * (b - 1);
```

### Parentheses

Never put a space after a left parenthesis or before a right one.

```
→ a=( a+5)*( b- 2 ); // AVOID
→ a = (a + 5) * (b - 2); // CORRECT
```

## Horizontal spacing

### Semicolons (;)

- Never put a space before a semicolon;
- Always put a space after a semicolon if it is not the last character of the line.

```
→ for( i= 0;i <10 ;i++ ) // AVOID
→ for (i=0; i<10; i++) // CORRECT
```

## Horizontal spacing

### Conditional statements

- Always put a space between the instruction name and the parenthesis specifying the condition.

```
→ if( a < b )x = a; // AVOID
→ if (a < b) x = a; // CORRECT
→ while (v[i] < 0) i = i + 1;
→ for (i=0; i<10; i++) v[i] = 0;
```

## Vertical spacing

It refers to the use of **newline** to separate groups of statements.

```
int i, n;
int sum, v[10];

for (i=0; i<n; i++) v[i] = 0;

i = 0;
while (i < n) {
    v[i] = i;
    i++;
}
```

**good vertical spacing**

## Vertical spacing

It refers to the use of **newline** to separate groups of statements.

```
int i, n;
int sum, v[10];
for (i=0; i<n; i++) v[i] = 0;
i = 0;
while (i < n) {
    v[i] = i;
    i++;
}
```

**bad vertical spacing**

## Indentations

It refers to the space put at the beginning of a line. Each nested section must be right-shifted of a TAB.

```
int main()
{
    int i, k;

    for (k=0; k<dim; k++) {
        i = 0;
        while (v[i] < w[k]) {
            if (v[i] > max)
                max = v[i] + w[k];
            i = i + 1;
        }
    }
}
```

**good indentation**

## Indentations

It refers to the space put at the beginning of a line. Each nested section must be right-shifted of a TAB.

```
int main()
{
int i, k;

for (k=0; k<dim; k++) {
i = 0;
while (v[i] < w[k]) {
if (v[i] > max)
max = v[i] + w[k];
i = i + 1;
}
}
}
```

bad indentation

## Parentheses

There are **two religions**. As all religions, both are fine.

```
for (k=0; k<dim; k++)
{
i = 0;
while (v[i] < w[k])
{
if (v[i] > max)
{
max = v[i];
m = i;
}
}
i = i + 1;
}
}
```

```
for (k=0; k<dim; k++) {
i = 0;
while (v[i] < w[k]) {
if (v[i] > max) {
max = v[i];
m = i;
}
}
i = i + 1;
}
}
```

↑

But this allows you to save more vertical space.

## Separate declarations from code

Keep variable declarations **before** the code:

```
int main()
{
for (int k=0; k<dim; k++) v[k] = rand();
int max = 0;
int m = 0;
for (int i=0; i<dim; i++) {
if (v[i] > max) {
max = v[i];
m = i;
}
}
}
```

TO BE AVOIDED

## Separate declarations from code

Keep variable declarations **before** the code:

```
int main()
{
int k, i, m; // array indexes
int max; // maximum array element

for (k=0; k<dim; k++) v[k] = rand();
for (i=0; i<dim; i++) {
if (v[i] > max) {
max = v[i];
m = i;
}
}
}
```

Preferred way

## Comments

Comments must be used to explain the meaning of variables and functionality of parts of the program.

They must be:

1. short;
2. meaningful;
3. updated with code changes.


Do not exaggerate! Long and trivial comments can worsen readability!

## Comments

Example of useless comments


```
a = b + c; // computes a as b+c
count++; // increment the counter
z = f(x,y); // f requires 2 arguments
```

Such comments are not only **useless**, but even **dangerous**, because increase the amount of text to read and obscure the structure of the program.

 **Comments**

They should be inserted in the following situations:

1. at the beginning of a file, to explain its contain and functionality;
2. next to each declaration of variable, constant, or data structure, to explain its meaning;
3. before each function, to explain its functionality, the meaning of each argument and the return value (if any);
4. before non trivial operations;


 **Types of comments**

Depending on their position and length, different styles should be used to write a comment.

**Comments to variables**

Should be on the right of the declaration, separated by one or more TABs:


```
int  r;           // circle radius
int  x, y;       // center coordinates
```

 **Types of comments**

**One line comments**

They should be written just before the instruction (or group of instructions) to be explained:

```
// initialize array v
for (i=0; i<dim; i++) v[i] = 0;
```

 **Types of comments**


**Long comments**

They should be written before the code to be explained, highlighting them with some border.


```

-----
// Function even(n) returns 1 if n
// is an even number, 0 otherwise
-----

int  even(int n)
{
    if (n % 2 == 0) return 1;
    else return 0;
}
```



 **MISRA-C 2004**

C can be used to write well structured and expressive programs, but can also be used to write **perverse** and extremely **hard-to-understand** code.  
**The latter is not acceptable in a safety-related system.**



The **Motor Industry Software Reliability Association**, provided some **guidelines** for the use of C language in safety-critical systems.

URL: <http://caxapa.ru/thumbs/468328/misra-c-2004.pdf>

 **Example of MISRA rules** 

**56:** The *goto* statement shall not be used.

**57:** The *continue* statement shall not be used.

**58:** The *break* statement shall not be used, except to terminate the cases of a *switch* statement.

**61:** Every non-empty case clause in a *switch* statement shall be terminated with a *break* statement.

**62:** All *switch* statements shall contain a final *default* clause.

**63:** A *switch* expression should not represent a Boolean value.

## Example of MISRA rules

- 65: Floating point variables shall not be used as loop counters.
- 67: Numeric variables being used within a *for* loop for iteration counting should not be modified in the body of the loop functions.
- 69: Functions with a variable number of arguments shall not be used.
- 82: A function should have a single exit point.
- 101: Pointer arithmetic shall not be used.
- 118: Dynamic heap memory allocation shall not be used.

## Code organization

Organize code according to the following order:

1. Header files (standard first, yours later)
2. Global constants (separate them into groups)
3. Function prototypes
4. Global data structures
5. Functions definitions
6. Tasks definitions
7. Main function

## Code organization

### Very important programming rule

Any function (including main) should **NOT** be longer than one page.

- If a function is longer than one page, it means that you should define a new auxiliary function.
- A program should be self explanatory by the sequence of functions it contains.

## General approach

The idea is to implement each library in a different file providing a set of functions for using it.

- Static global data can only be accessed through the provided library functions.

28

## Use of header files

C programs are normally organized into separately compiled **modules**.

**Module:** group of declarations and functions that are developed and maintained separately and possibly reused in different projects. Good examples are the **math** and **string** Standard Libraries.

Except for the main module, each module X consists of a

- source file (X.c):** contains global variable definitions, initializations and function definitions.
- header file (X.h):** contains **only**: structure type declarations, function prototypes, and **extern global variable** declarations.

- The X.c file must include the X.h file
- **Global variables** must be declared as **extern** in the X.h file
- Other modules can access the functionality in module X by inserting **#include "X.h"**
- X.c has to be compiled only if changed; the rest of the times the linker will link X's code into the final executable without needing to recompile it.

## Use of header files

- **Keep a module's internal declarations out of the header file**  
Global variables or functions that are used only in module X must be declared as **static** in X.c and **must not be mentioned in the X.h file**.
- **Always use include "guards" in a header file**  
When several source files include the same header files, the compiler generates an error if the same entities are defined multiple times. To avoid this, you can make sure that a given include file is only included in a particular source code once with the **#ifndef** directive. For example "geometry.h" would start with:  

```
#ifndef GEOMETRY_H
#define GEOMETRY_H
```

and end with:  

```
#endif
```

*Do not start the guard symbol with an underscore!* Leading underscore names are reserved for internal use by the C implementation.



## Use of header files

- **A.h should include all strictly needed header files, but no more**

If a structure type defined in module *X* is used as a member variable of a structure type *A*, then you must include *X.h* in *A.h*, so that the compiler knows how large the *X* member is.

However, do not include header files needed only by the *.c* file.

For instance `<math.h>` is usually needed only by the function definitions, therefore it should be included in the *.c* file, not in the *.h* file.

- **File *A.c* should first include its *A.h*, then the other required headers**
- **Never include a source *.c* file for any reason!**

Read more on:

<http://www.umich.edu/~eecs381/handouts/CHeaderFileGuidelines.pdf>



## Makefile

```

#-----
# Target file to be compiled by default
MAIN = balls
#-----
# CC is the compiler to be used
CC = gcc
#-----
# CFLAGS are the options passed to the compiler
CFLAGS = -Wall -lpthread -lrt -lm
#-----
# OBJs are the object files to be linked
#-----
OBJ1 = mylib1
OBJ2 = mylib2
OBJS = $(MAIN).o $(OBJ1).o $(OBJ2).o
#-----
# LIBS are the external libraries to be used
#-----
LIBS = `allegro-config --libs`

```

32



## Makefile

```

#-----
# Dependencies
#-----
$(MAIN): $(OBJS)
           $(CC) -o $(MAIN) $(OBJS) $(LIBS) $(CFLAGS)

$(MAIN).o: $(MAIN).c
           $(CC) -c $(MAIN).c

$(OBJ1).o: $(OBJ1).c
           $(CC) -c $(OBJ1).c

$(OBJ2).o: $(OBJ2).c
           $(CC) -c $(OBJ2).c
#-----
# Command that can be specified inline: make clean
#-----
clean:
    rm -rf *o $(MAIN)

```

33



## Important guidelines

Your project will be evaluated also based on the level of compliance with the following guidelines

1. Write code according to the given **style rules**
2. Do not use **dynamic memory** allocation
3. Define all **local variables** at the beginning of a function
4. Avoid using **nested critical sections** (use more local variables)
5. Avoid **numeric constants** in the code (exceptions: **0, 0.5, 1, 2, ...**)
6. Avoid functions **longer than one page** (use new functions)
7. Avoid lines **longer than 80 characters** (lines can be split)
8. Never use the **goto** and **continue** statements
9. Use **break** only in the **switch** statement
10. Organize your program in **different source files** (no more than 3)