Style and organization rules for C programs
how to enjoy freedom without stress

Gianni Di Caro
Dipartimento di Matematica, Università di Trento
via Sommarive 14, 38050 - Povo
dicaro@rtm.science.unitn.it

Abstract

The aim of this document is to present a coherent set of prescriptions about the style, the organization, the documentation and the precautions to be used while developing software in C language. These specifications are particularly important for projects that require the integration and the re-use of software modules developed in a scientific research context by different institutions and people, possibly at different times.

While C is a widely used language with a rich availability of compilers, the software designer and developer must be fully aware of the “low-level” nature of the language, that was born without extensive support for modularity, type checking, etc. It is therefore the responsibility of the designer to assure a high modularity and provisions for documentation, easy maintenance, and re-use.

The suggested rules are the result of an analysis of widely adopted rules for “literate” C programming.
# Contents

**Introduction**  
1

1 **The Code**  
   1.1 Human semantic aspects  
   1.1.1 Name usage  
   1.1.2 Pretty-printing  
   1.1.3 Structures and data types  
   1.1.4 Comments  
   1.2 Syntactic aspects

2 **The environment**  
   2.1 Source files  
   2.2 Directory organization and compilation  
   2.3 Documentation

**Acknowledgements**  
24

**Bibliography**  
25

A **Hungarian style**  
27
Introduction

Programming is the art of making a series of stepwise refinements of specifications.
K. Mani Chandy and J. Misra

The proposed rules are based on personal experience and on the analysis of various documents [Ame89, MAI93, ESA91, SH91, GFM+94, CEK+95, DLK95, Hor90, KR88, MAI93] dealing with the definition of a set of style and organization rules for C programming. These rules aim at improving the “software engineering” process and products [GFM+94, GJM91], the quality being measured by: reliability, correctness, robustness, safety, performance, user friendliness, testability, maintenance simplicity, reuse opportunity, understandability, productivity, etc. The definition of tailored rules for code writing, while not sufficient, is a crucial first step in the long and complex software optimization process.

Given the current intellectual ferment in the software engineering field, it is not surprising that there are not many standards or consolidated techniques. Nonetheless, the utility of selecting some reasonable set of rules is clear as soon as one considers the poor average quality of the software produced without common rules in a scientific research environment. In this context it is not unusual that code is developed with short-sighted objectives (for example, as a way to obtain computational results for publishing in the applied Computer Science area) and even a set of simple rules can greatly improve the situation, especially if software written by different researchers has to be integrated and re-used.

The rules proposed here are suitable in a context of non-professional but expert programmers, like researchers, with the need to develop sizable software modules that can be integrated in shared libraries. The purpose is to assure that these libraries, a sort of common knowledge resource, will be easy to use and to modify for anybody, with no need of knowing all internal details of the different modules (information hiding).
It is useful to separate the set of rules into two subsets:

**rules about the code** itself [KR88, Ame89], that determine:

- the **readability** (human semantic component), depending on name usage, on data structures, on indentation, on comments and on overall structure;

- **correctness, portability and efficiency** (syntactic component, computational complexity), that are strongly connected to the usage of operators and constructs;

**rules about the environment** i.e. the location and the content of source and headers files, the building and usage of libraries, the compilation tools, the enclosed documentation.
Chapter 1

The Code

The usage of specific constructs of the C language [KR88, Ame89] determine not only the correctness of a program with respect to the project specification, but also many other fundamental properties: the understanding of the algorithm, the possibility to modify or to improve the behaviour, the easiness to incorporate a part into a library of functional modules. These requirements motivate an appropriate usage of names, of text formatting style, of comments, of data structures and of functions. This first set of "literate programming" rules is indicated as the human semantic level because it involves many aspects connected to the understanding and reading of the code. In fact, these aspects are the fundamental ones to extract, at a local or global level, the semantic meaning. A second set of rules is related to the syntactic correctness of the code, its portability and efficiency. All these issues are grouped in the syntactic level, because they are closely related to the use of operators, of data types and of constructs.

1.1 Human semantic aspects

1.1.1 Name usage

An appropriate selection of the names used in a program gives a fundamental contribution to the comprehension and maintenance of the software. It is easier to follow the actions and the role and state of each single component if the semantics of the names is self-explanatory, and not ambiguous. As an example, the hungarian style, one of the most diffused styles among professional programmers, is briefly summarized in appendix A. We opted for less strict and more human-oriented rules.

Names can be subdivided in two classes: simple names (like Var, gnat, foo, x) and composed names (like gnat_var, NewFoo, x_square).

- the first issue to consider deals with the meaning associated to the objects identified by a name. It is extremely important that names clarify the role of the variable, of the data type, of the constant as well the action carried out by the function or the macro. Even at the risk of some verbosity, the code must be easy to read, especially for external users.

Only the English language should be used in the documentation. Names length can be
contracted through commonly-used acronyms or abbreviations like: Get, Put, Tmp, Var, Ptr, Dim, Vec, New, Mem, Add, Del, Str etc., and i, j, k, l, .. for integer loops indices. Here are some examples:

<table>
<thead>
<tr>
<th>Correct Use</th>
<th>Incorrect Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>check_port_status()</td>
<td>ckpstat()</td>
</tr>
<tr>
<td>AddItemToQueue()</td>
<td>qadd()</td>
</tr>
<tr>
<td>InstallNewObject()</td>
<td>inst()</td>
</tr>
<tr>
<td>tmp, temp</td>
<td>t</td>
</tr>
<tr>
<td>VecDim</td>
<td>dim, d</td>
</tr>
<tr>
<td>status</td>
<td>s, stat</td>
</tr>
<tr>
<td>prohibition_period</td>
<td>pp, period, pperiod</td>
</tr>
<tr>
<td>period</td>
<td>v</td>
</tr>
</tbody>
</table>

The limited “waste” of time in the writing phase can be reduced by using advanced editing tools (e.g., emacs[Fre]), permitting the automatic completion of names, and the effort pays off dividends in the form of debugging advantages, reduced documentation, improved readability.

- Once an appropriate name is identified, the introduction of similar definitions needs to be avoided: terms like gnat, gnat1, gnat2, gant, Gnat generate a high probability of mistakes. The use of the underscore as prefix is prohibited, as in _gnat, _stack, because this convention is normally used for names defined internally by compilers.

- To allow a quick detection of the context of validity and definition of a variable or a function, the names that refer to global variables or functions exported outside of the definition file are written with the first letter capitalized, if the name is simple, and with the capitalization of the first letters of each composing name in the other case. Local variable and functions are written with lower-case letters, with underscores to separate components. An example with two files, export.c and import.c, that contain some groups of functions linked together is here presented:
If these rules are adopted it is immediate to see whether a variable (or the function) has to be searched in the current file or in a different one, if one or more external functions are able to modify its value, etc.

- Similar concepts are valid for the abstract data types. When `typedef` is used to define a new global and/or exported datatype, the name of the type has to follow the same rules described above. In the specific case of structs and unions the same rules should be applied both to the struct name and to every field. This prescription avoids ambiguities, see the following example:

```c
struct ExternalStruct { ... };  
struct OtherExternalStruct { ... };  
struct OtherExternalStruct GlobalVar;

void local_fun()
{
    struct ExternalStruct external;
    struct OtherExternalStruct other;
    ...  
    external.first_field = 1;  /* it is not clear if the struct, as type, is locally or globally defined, the definition of external has to be searched in the code */
    other.FirstField = 0;  /* it is immediately evident that the variable is local but that the structure type is globally defined */
    GlobalVar.Field = 2;  /* this expression too is without ambiguity */
    ...
}
```

- The constants, defined with `define`, `const` or `enum`, are expressed with uppercase letters and underscores as separators (`THIS_IS_A_CONSTANT`). The same rule holds for macros. Only if the macro is a “real” function (i.e. its action is an evaluation based on the argument values with no side-effects) the rule for function names can be used.
• If a class of functions operates on a same object, then the use of an homogeneous class of names like: `ActionObject()` is suggested. As an example, if the “object” is a configuration during an heuristic search (`Configuration`), all functions acting on it must have the form: `SetConfiguration()`, `ResetConfiguration()`, `StoreConfiguration()`, `GetConfiguration()` .... Moreover, if these functions act on specific data structures, these should have the form `ObjectType`. As an example, a data structure called `ConfigurationType` could be used by functions: `int SetConfiguration(ConfigurationType *config)` .... It can be useful to emphasize that this convention avoids conflicts with the convention `object_t` adopted by many modern compilers, in which, for example, the data type referring to a time measure is indicated with `time_t`.

### 1.1.2 Pretty-printing

The rules about indentations, parentheses and blanks follow:

**indentation:**

• (main rule!) instructions start at the same column if and only if they are at the same logical level. The same holds for comments;

• the global variables, the functions and the preprocessor instructions must be indented at the first column;

• the declaration lists of variables or functions arguments, should be structured on more lines if they are too long to fit in a single one:

```c
char *a, b, c,
    d, foo,
    gnu;

int fun(int first_arg,
        float second_arg,
        char third_arg)
```

Uncorrelated variables in the variables declaration part should be written on different lines;

• a long assignment instruction can be broken on more lines, with indentation at the right of the separation sign between lvalue and rvalue:

```c
left_value = (first_function(argument) * second_function(argument)) / denominator;
```

Analogously, in the case of statements with multiple long conditions every condition can placed in a different row, with indentation corresponding to the conditional statement parenthesis:

```c
if (first_condition() * useful_variable &&
    second_condition() * another_useful_variable)
{
    ...
```
The same considerations have to be applied to complex for loops or to long expressions with the ternary operator :?, which are better visualized if placed on multiple lines:

```c
var_to_assign = (var_1 == var_2)
    ? assign_1
    : assign_2;
```

**brackets:**
- the control structures should be written as follows:

```c
keyword
{
    body
}
```

as an example:
```c
if( cond )
{
    if_body
}
else
{
    else_body
}
```

and analogously for while, for, do-while and switch:
```c
switch( expr )
{
    case value_1:
        body_1;
    case value_2:
        body_2;
    ...
}
```

If the body consists of a single instruction, the parentheses can be omitted, but not the indentation and a blank line after the instruction:
```c
for(i = 0; i < iterations; i++)
    h += i + 2;
```

```c
... /* new instructions */
```

- the structs and the unions follow a rule similar to that of statements:

```c
struct i_am_a_struct
{
    int first;
    int second;
    char third;
};
```

The same is valid for the typedef;

- round brackets must be always used to group together homogeneous operations and/or to explicitly impose precedence rules:

```c
x = (var_correlated_to_b + b) + (c * d) + e;
```

**blank spaces:**
• *variables and operators* have to be written with blank spaces as separators, with the exception of prefixed/postfixed increments or decrements, of the operators ++ and --; and of items close to parentheses ( ):

```c
for(i = 0; i < number_of_iterations; i++)
```

• blank separation lines have to be inserted between the end of a *function* and the start of the next;

• adjacent but logically separated *instructions blocks* have to be separated by a blank line.

### 1.1.3 Structures and data types

Let us now consider the use of data structures and the definition of new data types and symbolic constants.

• all information associated with an object must be grouped in a *struct* to create an entity that is easy to understand, to localize, and to modify.

• *typedef* should be used for each defined structure, to create a new datatype with a well defined semantics.

• numerical values are prohibited, apart from cases where the meaning of a value is self-evident. The *numerical constants* have to be defined with symbolic expressions through `#define`, `const`, `enum`. `enum` must be preferred over `#define` for groups of constants that refer to the same object. The use of symbolic names allows both to modify with a single action the constant value and to give a clear meaning (no strange “magic” numbers). In particular, the array dimensions must be declared through symbolic expressions.

• some datatypes and constants of common use have to be defined in a convenient file header. As an example, a boolean type does not exist in C, and therefore it must be introduced:

```c
typedef short boolean;
```

Connected to it there are the constants *TRUE* and *FALSE*:

```c
#define TRUE 1
#define FALSE 0
```

A symbolic constant *ERROR* could be useful for the generic error value returned from a user-defined function.

### 1.1.4 Comments

Comments are crucial to describe in a correct and concise way the *functions carried out* and the *implementation details*. It is possible to distinguish two classes of comments: *heading comments* and *inlined comments*.
heading comments: they are comment lines that must be introduced at the beginning of:

- each file containing a set of functions,
- each function definition,
- each group of logically connected functions,
- each header file.

The next example shows the situation:

```c
file:     group_of_mixed_functions.c

/* initial comments about the program and/or the local functions */
/* comments about the first group of functions */
... /* function definitions */

/* description for the function fun */
int fun()
{
    ...
}
...

/* comments about the second group of functions */
...
/* function definitions */
```

These general comments are necessary to immediately grasp the objectives of a group of functions, the program organization and the actions of the different functions. To provide readable and complete information the comments lines should be structured according to specific description fields whose contents vary for the different cases cited above.

The preferred aesthetic appearance of the comment block is as follows:
The suggested description fields are in the following list:

**type:**  ● in the “general comment” at the beginning of a file it states the type of functions contained:
- if the content is a pool of functions contributing to a specific program, and therefore depending on other functions and headers developed for the application with the `main()` function placed in a different file, then the type field will sound as: “pool of functions for the program *application* (name of the file containing `main()` )”;
- if the module contains functions written for a definite application, without a `main()` but with a self-consistent nature then the type field could be: “pool of functions developed for the program *application* (archived) that can be archived in a library (name of the library)”;
- if the `main()` function is present the type field will be: “(principal or unique) pool of functions developed for the program *application*”.

● when the type field is used for the description of an individual function it should specify if the C function is a *function*, a *simple procedure* or a *complex procedure* (see discussion at pag. 14);

● when the field is used in a header file, the content will be: “header file (private or public) linked to the file (file name)”, where the distinction between private and public will be clear in the Section 2.1;

**purpose:** intended for all comments, except those in header files. It gives a description of the application or of the pool of functions or of the function. The amount of details should be appropriate to the position of the comment.

**copyright(c):** the institution and/or person holding the copyright, with instructions (or reference) about terms and conditions for using, modifying and distributing the code.

**author(s):** the author and his affiliation/internet/WWW coordinates should be given at the beginning of each file;

**version:** the current version number with date and possibly a short description of changes with respect to previous versions;

**update:** this optional field can be used at the beginning of the file. It can be useful to keep track of changes and of their motivations, authors and dates;
**libraries**: placed at the beginning of a file, with a list of the libraries to be linked when using the pool of functions contained in the file;

**headers**: similar to the previous, but listing the header files;

**group**: comment field for each sub-group of functions in a given file, its scope is to describe the functionalities contained in the sub-group;

**inputs**: used only before individual functions to describe the input parameters that are passed to the function. It could be useful to describe also the global variables that are used inside the function. Parameters that do not are modified should be declared as **const** or passed by value.

**outputs**: describing the meaning of the returned value of the function or the error code in the case of procedures. It could be useful to list the global vars or the input parameters that are modified by the function.

**prerequisites**: describe the condition that have to be set or the functions that have to be called before using the function that this comment is applied to.

**inlined comments**: introduced in the code to explain single actions or declarations. Normally they are indented at the same level of the code. If the space is sufficient they can be placed on the same instruction line. Redundant comments should be omitted (this is the case of self-evident operations, especially if proper meaningful names are used). These comments should be used to explain “what” is done and “how” it is done, to clarify the role of a constant or the meaning of a variable or of a struct field (again, when it is not evident).

A complete example follows:

```c
/*
 * type: main and unique functions group for the check_style program
 * purpose: take a .c file in input and check the programming style
 * author: P. Spiderman, spider@marvel.comics.com
 * update: February 31, 1999: author
 *       June 31, 1999: Mr. Fantastic
 *       insert check of use of pointers
 * libraries: -lm -lgulp
 * headers:
 */

#define MAX_LEN 100000 /* max file length */
FILE *Fp; /* data file pointer */

/*
 * group: functions that parse the file
 */
```

11
/*
 * type: procedure
 * purpose: make a first level parsing of the file
 * inputs: contents of data file
 * outputs: error condition if ....
 */

int first_parsing(char *data_array)
{
  char *last_position;

  /* scan the array searching the # characters */
  while(read_from_array(data_array) != ERROR)
  {
    /* do some action */
  }

  data = last_position; /* the last # position */

  if( last_position == NULL)
    return(ERROR)

  ...  
}

1.2 Syntactic aspects

Let us now consider the rules aiming at improving the correctness, portability and efficiency of a C program [KR88, Ame89]. Some potentially dangerous situations are analyzed and the use of data structures and functions is discussed.

ANSI instructions: only statements from the standard ANSI[Ame89] committee are allowed;

jump instructions: goto, break and continue should be completely avoided; The motivation for this prohibition is that, if it is not followed, the program readability and the action of automated correctness-checking systems could be seriously compromised. Of course, The break is admitted in the switch statement, whereas in the other cases it should be substituted by boolean variables, except for cases where the jump off point is easily locatable (analog discussion can be done for the continue statement);

variable initialization: it is better to avoid variable initialization in the declaration point, (except for the static variables inside functions). Comments are obligatory if this rule is not followed. The best place to initialize variables is immediately before their use, to control the starting values before the action.

Default values assigned by the compiler should not be trusted because portability problems could arise.

constants: if the meaning a value is not obvious, symbolic constants should be used, for example:
for(i = 0; i < MAX_ITER; i++). The type of the constants must be consistent with its
usage, to avoid implicit casts. The use of `enum` for a homogeneous group of constants can improve the code readability.

Pointers have to be compared with the `NULL` constant and no assumptions can be done about its specific value;

**preprocessor usage**: the preprocessor is a very powerful tool but it is also a source of errors that are difficult to detect, and therefore it should be used only in a limited way. In addition, let us warn that `#define` has a global validity and therefore it is not possible to delimit the range of its action.

The preprocessor is useful in case of conditional code compilation depending on machine and/or compiler type, to obtain portable code:

```c
#include <gnat.h>
#define MAX_LEN 1000

#include <gnatgnat.h>
#define MAX_LEN 2000

#define abs(x) x>0?x:-x
  // abs(a-b) is expanded as: a-b>0?a-b:-a-b ...that is wrong !!! */
#define correct_abs(x) (((x)>0?(x):-(x))
```

The same technique can be used to discriminate with respect to some flags, but the same effect can be obtained through run-time conditional, using `if` instead of `#if`'s. At the cost of a limited loss of efficiency the flexibility and readability increase.

The preprocessor must not to be used to define data types, to avoid side effects and to allow type-checking by the compiler (types have to be defined using `typedef`):

```c
#define CharPtr char *
CharPtr a, b; /* but now b is a char not a char */
```

Functions can be defined as macros only if they are pure functions according to the discussion of par. 1.2. Parentheses must be used to protect arguments from unwanted side effects:

```c
#define correct_abs(x) ((x)>0?(x):-x))
```
global, static, and extern variables: global data which can be modified by more than one software module are to be avoided, because there is no protection from unwanted changes. The basic concept is that each datum should belong to a module managing it. A good way to obtain this result is by using access functions, i.e., an interface that hides details and guarantees protection against unwanted changes (if you want, this is a way to simulate an object oriented language through simple C).

extern ColorType ColorsTable[][color]; /* global variable */
circle_draw(x, y, ray, ColorsTable[color]);

extern ColorType ColorsTable(int color); /* global function */
circle_draw(x, y, ray, ColorsTable(color));

Let us assume that the way to extract the color from the color table changes. In the first case all instances of ColorsTable[color] must be changed in all files, whereas in the second case the interface does not change and changes will occur only inside the function ColorsTable(). This information hiding approach should be applied also to static global variables.

It is necessary to declare as static all global but not exported data and the local functions of a file, whereas the exported items have to be declared as extern. In this way it is possible to avoid unwanted inclusions and the context of the various components is immediately clear;

data structures access: functions and/or macros with a clear semantics should be used to access the fields of exported data structures of fundamental interest. In this way, the name of the fields is hidden and a more intuitive meaning is linked to them. This technique can be used also to create and to manage data structures in an implicit way (see the initialization routines provided by many object-oriented languages for object creation):

struct RectangleType rectangle;
int GetRectangleWidth(struct RectangleType *rectangle);
int SetRectangleWidth(struct RectangleType *rectangle, int width);

/* manage a rectangle internally through a descriptor */
typedef int RectangleDescriptor;
RectangleDescriptor create_rectangle(int width, int height);
int get_rectangle_width(RectangleDescriptor desc);

functions: the definition prototype of each function must be inserted into a file header following the specification of Section 2.1. The prototype allows type-checking by the compiler (automated prototype-extraction packages are available as free software).

Functions can belong to one of the following classes:
true functions that do not modify their arguments but return a value depending on an evaluation of their inputs (for example, mathematical functions);

simple procedures that carry out a sequence of actions identifiable as a logically atomic action (for example, read data from a file and store them into an array);

complex procedures where different “heterogeneous” actions are executed (for example, read a file, get inputs from a keyboard and send data to a printer).

True functions and simple procedures are the easiest to write, to understand, to reuse, and can help in writing modular code. They contain data structures, concepts and actions in an “atomic” entity. On the other hand, complex functions are typically used only for the specific task they are developed for.

To pass a variable number of arguments, the ANSI committee [Ame89] introduced the stdarg interface that has a set of macro definitions va_list, va_start, va_end located in the file stdarg.h. The similar and widely-used macro package varargs has been outdated by the new macros standard of stdarg.h.

utility functions: the code often is cluttered with checks on function return values. For some standard actions, like file opening or dynamic memory allocation, it could be convenient to insert the check into ad hoc routines (that stop the execution with meaningful error messages in case of errors): as for example:

```c
void *safe_malloc(int bytes, char *msg)
{
    void *ptr;
    if ((ptr = malloc(bytes)) == NULL )
    {
        fprintf(stderr, "%s", msg);
        exit(1);
    }
    return(ptr);
}
void any(void)
{
    float *f;
    f = safe_malloc(VEC_SIZE * sizeof(float), "Not enough space for array f\n"");
    ...
}
```

use of standard data types: the standard data types need to be used in the right way. For example, a variable can be declared as unsigned only if one is sure that only non-negative values will be used, char have to be avoided for integers with a limited range of values, etc. For portability, the sizeof() function has to be used to compute the byte length of an item.

memory usage: assumptions about data alignment or about structs or array contiguity must be avoided, because the computer internal organization and the memory management policy change for different computers.
Chapter 2

The environment

The word environment in this context means the organization and the location in the file system of the set of source, header, documentation, and compilation management files that belong to an application or a library project (a synonym is programming in-the-large).

2.1 Source files

According to well-known modern programming practices, the module must be the base unit of a software project. In detail, the module consists of a source file (.c) and a public header files (.h), the public “interface” part. In addition, a private header file contains information used only in the source file. A project is composed of a collection of linked modules and of a global header file. The description of the contents and the organization of each component (designed with modularity and information localization in mind) are as follows:

public header: the name is the same as that of the source file with the usual “header” suffix .h. It contains the declarations of all exported objects (the header will be included in all modules that want access the services provided by the given module). In detail, this header contains the prototypes of exported functions, the abstract data types and the constants necessary to use them (the interface to the functions), and all the constants, global variables, macros and types utilized by the source file functions but also by other modules.

The interface has to be perfectly consistent, i.e. the inclusion of this file header has to be sufficient to completely use the services of the module.

As emphasized in Section 1.2 it would be better to avoid exporting global variables out of a module. When this is necessary, an extern declaration must be used. The adopted convention is as follows: the extern declaration is used before the global exported variables in the public header, while the definition of the variable is in the private header (without the extern keyword). Because exported variables should be rare, the complication of taking care of the variable in two headers is acceptable.

The organization of the public header follows (note: comments lines should be used in the file to separate different sections):
• preprocessor flags to avoid multiple inclusions. The widely used convention is to put at the beginning of the header file:

```c
#ifndef HeaderFileName_H
#define HeaderFileName_H
```

and at:

```c
#endif /* #ifdef HeaderFileName_H */
```

• list of all `#include` files necessary for using the exported objects (note: nested inclusions must be considered in the `makefile`'s list of dependencies);

• collection of all exported symbolic constants defined through `#define`, `enum`, `const`;

• list of all exported `macros`;

• list of all public `abstract data types`;

• list of `global variables` that can be accessed by other modules (with the keyword `extern` in front of each variable);

• list of all prototypes of exported `functions` (keyword `extern` must be used).

**Private header**: all objects that are used *only* by the local functions of the module are placed in this header. The name is the same than that of the source file with a `P` appended and the usual `.h` (for example `appl.c`, gives `applP.h`).

The file organization is:

• preprocessor flag to avoid multiple inclusions;

• list of all `#include` necessary for the local functions, variables and types;

• list of the prototypes of all external function imported from other modules (if they are not already present in the included headers, as they should be);

• list of all privates constants, defined through `#define`, `enum`, `const`;

• list of all private `macros`;

• list of all private `data types`;

• list of `private variables` that are declared explicitly as `static`;

• list of all `exported global vars`. This section duplicates the corresponding section in the public header with the `extern` keyword omitted (the variables are defined to be used in the module and not declared as `extern`);

• prototypes of all the `functions` called only inside the module and declared as `static`.

**Global header (optional)**: it contains all resources defined at the application level and useful or necessary for more modules. Organized in a way analogous to that of the previous headers:

• preprocessor flag to avoid multiple inclusions;
• list of all global `include files;
• list of all global `symbolic constants`;
• list of all global `macros`;
• list of all global `data types`;
• list of all global `variables`: this section should be empty;

**source file (`.c`):** contains the functions corresponding to services offered by the module. It is organized in the following way:

• a section including the optional `global header`;
• the `public header`;
• the `private header`;
• one or more groups of `exported functions`;
• one or more groups of `local functions declared as static`;

As a complete example: let us consider a software project for an application named *new application* where the functions set is divided into more modules, containing homogeneous groups of functions. The `source file` of the i-th module is named `module\text{i}.c`, and therefore its private and public headers will be called respectively `module\text{i}.P.h` and `module\text{i}.H.h`.

For generality the case of exported global variables will be considered. The source file of the first module will look like:
file: module_1.c

/* comment fields here... */

#include "application_global_header.h"

#include "module_1.h"
#include "module_1P.h"

/* groups of functions exported to other modules */

void ExportedFunction(int arg)
{
    ...
}

...

/* groups of functions local to this module */

static void local_function(int arg)
{
    ...
}

...
After the previous emphasis on modularity in project design, it is appropriate to remark that programmers have to be very careful against possible errors because the C compiler does not enforce appropriate modularity rules. For example, programmers must pay attention to the use of `static` and `extern`, or to the inclusion of global variables or the re-definition of a function prototype. Most suggested rules deal with modularity and information hiding, but the compiler does not provide good support to check for possible mistakes (see [MoY] for a detailed discussion about the drawbacks of C programming). C programming requires a high level of effort to obtain modular code, effort that is useless as soon as a single programmer in a project does not follow the rules. We are aware of strong criticisms against the use of C (see [MoY] for a detailed discussion about the drawbacks of C programming). We nonetheless decided that C can still be used in a scientific environment, provided that a suitable compromise is reached between modern software engineering practices and the almost complete freedom allowed by C (including the freedom to write unmanageable “spaghetti” code, with no re-use potential!).
2.2 Directory organization and compilation

The basic prescription is that the files are placed in the file system in such a way that finding of sources, libraries, headers and documentation is straightforward. In addition the arrangement in standard and highly structured places simplifies inclusions, linking and compilation actions.

Some of the concepts used in Unix[CoF90] systems will be followed, since they are widely diffused and allow a rational and efficient organization. The organization is that of a root directory with the following subdirectories:

- **bin** - for the executable files
- **lib** - for the libraries built with the developed functions
- **src** - for the source files .c
- **inc** - for the header files .h
- **obj** - for the object files .o produced by compilation
- **tmp** - for temporary files
- **doc** - for documentation files.

Each directory will be organized in subdirectories, one for each application (let us note that the creation of distinct subdirectories will be carried out with a standard `makefile` starting from the `src` subdirectories). This criterion allows to speed-up the search for applications, sources, documentation (one knows in advance where to look for them).

If more machines are available in a shared file system different binary files can be produced for the various machines and the directories `bin`, `lib`, `obj` will be structured in subdirectories, one for each architecture.

For example, if the set of functions contained in the module named `TabuSearch` has been compiled for SunOS and HP-UX operating systems, then the directory organization will be:
src:
  TabuSearch:
    TabuSearch.c
    README
    Makefile

inc:
  TabuSearch:
    TabuSearchP.h
    TabuSearch.h

lib:
  SUNOS:
    TabuSearch:
      TabuSearch.a
  HPPA:
    TabuSearch:
      TabuSearch.a

obj:
  SUNOS:
    TabuSearch:
      TabuSearch.o
  HPPA:
    TabuSearch:
      TabuSearch.o

doc:
  TabuSearch:
    TabuSearch.ps
    TabuSearch.1

To better exploit this file organization a standard Makefile model should be used. A simple Makefile that can be useful as a starting reference is given. Let us assume that the .c and .h files are in the subdirectory TabuSearch of src, with the directories lib, obj, inc, doc already created. By typing a single command `make all` (the shell’s commands are those of a Unix environment) the appropriate directories will be created (`make dir`), the functions will be compiled and archived in a library,
file: Makefile

# paths for directories
ROOT_DIR= /usr/spiderman/MyWeb
SRCDIR= $(ROOT_DIR)/src/$(SRC)
INCDIR= $(ROOT_DIR)/inc/$(SRC)
LIBDIR= $(ROOT_DIR)/lib/$(ARCHITECTURE)/$(SRC)
OBJDIR= $(ROOT_DIR)/obj/$(ARCHITECTURE)/$(SRC)
DOCDIR= $(ROOT_DIR)/doc/$(SRC)

# source and target files
SRC= TabuSearch
OBJ= $(OBJDIR)/$(SRC).o
LIB= $(LIBDIR)/$(SRC).a

# compilation options
CC= gcc -I$(INCDIR)
CFLAGS= -Wall -O2
LIBRARIES= -limage -ldsp
LDIR= /usr/local/lib
LINK= $(CC) -L$(LDIR) $(LIBRARIES)
AR= ar rc

# main target
all: dir $(OBJ) $(LIB)
  @echo
  @echo 'Main target finished.'
  @echo
  @echo
  @echo
  @echo

# create the sub-directories, move the header files in the $(INCDIR) and
# create symbolic links for them in $(SRCDIR)

dir:
  mkdir $(LIBDIR)
  mkdir $(OBJDIR)
  mkdir $(INCDIR)
  mkdir $(DOCDIR)
  mv $(SRC)-h $(SRC)-h $(SRC)-h $(SRC)-h $(INCDIR)
  ln -s $(INCDIR)/$(SRC)-h .
  ln -s $(INCDIR)/$(SRC)-h .

# make the obj
$(OBJ): $(SRC).c $(SRC).h $(SRC)-h
  @echo
  @echo 'Making $(SRC).o ...'
  $(CC) -o $(OBJDIR)/$(SRC).o -c $(CFLAGS) $(SRC).c
  @echo '... done !'
  @echo

# make the lib
$(LIB): $(OBJ)
  @echo
  @echo 'Making $(LIB) ...'
  $(AR) $(LIB) $(OBJ)
  @echo 'Installing $(LIB) ...'
  ranlib $(LIB)
  @echo '... done !'
  @echo
2.3 Documentation

A fundamental component of any software project is the documentation, that must describe the objectives and any useful information to correctly use and to exploit all the potentiality of an application or a library. Therefore a README file must always be included in the src directory; where the author(s) explain the goals and the features of the product, summarizing, if possible, the function and/or functionality list with detailed descriptions, pointers to accompanying documents, etc. For important applications it is useful to make formatted documents (the use of latex [Lam86] is encouraged) in .tex, .ps, or .man format to insert in the doc directory.

As an example of documentation we refer to [?] structured as it follows:

- an introduction of the problem (with bibliographical references) to whom the routines refer, the files of the package, the main data structures and some other interesting information to better understand the building philosophy.

- an exhaustive list of all the used function, where for each function are reported the purpose, inputs, outputs and prerequisites fields. The functions are all logically grouped (for example the exported functions, the tracing functions, etc.);

- a complete example that make clear how to use the functions;

- the description all parameters that the user have to set;

- requisites (functions, variables) that have to be supplied directly from the user;

- a more complex example that show the meaning of all the principal components.

While it is impossible to give a rigid specification, a well-done README file should at least contain:

- author(s)

- data

- version

- affiliation

- architectures in which the code has been tested

- known problems

- goals

- functionalities

- a list to describe the single functionalities
Acknowledgements

This document has been produced with financial support from Progetto Speciale 1995: Algoritmi e software per ottimizzazione, e modelli di sistemi complessi of the Dipartimento di Matematica dell’Università degli Studi di Trento.
Bibliography


Appendix A

Hungarian style

The set of rules of the hungarian style [SH91], assigns variables names as a composition of a type and a qualifier, and function names as a composition (optional) of type, action(s) and parameters. Therefore, for example cpFirst stands for a char pointer to the first element of such values interval. Functions, complex types or variables are generated by means of composition of elementary types. This approach can be formalized and it is possible to generate a types algebra. The main problem stands in its poor readability, its crypticity: types need being defined before and a “training” to recognize quickly the meanings is required as the conventions to understand the code need knowing.