Coding and documentation standard

Cons T Åhs, Alexander Bottema

Edited and translated by Lars-Henrik Eriksson

February 16, 2008

1 Introduction

Writing good code (easy to read, easy to maintain, efficient) is a difficult art. It demands experience and planning. Good code uses abstraction to enhance readability and maintainability. The task is easier if the programming language used is designed to support the writing of good code and supports abstraction. Unfortunately, many languages still in use today do not have that property. It is still possible to write good programs, but the task becomes more difficult.

This document gives you a number of rules for writing good code in C. The C language has rather primitive support for abstraction. This problem must be approached in a sensible way. The collection of rules is intended to highlight the difficulties and suggest ways of overcoming them. The rules are not to be taken as absolute requirements but as (strong) recommendations. You can attack the difficulties in different way, as long as you follow the spirit of these rules and the problems pointed out here are handled/avoided. In some cases it is difficult to follow the spirit of a rule without actually following the rule itself. The prohibition against the use of `goto` is a good example.

The rules can be summarised by the following general principles:

Rule 1.0.1: Be clear
By clearly expressing what it to be done, the code will be easier to understand and modify.

Rule 1.0.2: Be consistent
Being consistent - in constructing, formatting as well as documenting code - increases readability.

Rule 1.0.3: Be abstract
Abstract code focuses on the task to be done rather than how it is done. It is substantially easier to maintain abstract code as changes in general are kept within limited parts of the code.

2 Rules

2.1 Documentation

Rule 2.1.1: All code should be documented.
The code in itself expresses the concrete way things are done. Someone

reading the code is more interested in an abstract view.

Rule 2.1.2: Code should be implicitly documented
Implicit documentation in the form of meaningful function and variable names enhance readability.

Rule 2.1.3: Code should be explicitly documented
Explicit documentation in the form of function headers and comments enhance readibility and understanding. Note that there are different prospective readers of explicit documentation. Someone who wants to change something needs more concrete descriptions while someone who wants to use the code (as a library) wants something more abstract.

You should also have a comment header for each entire code file, with the names of the authors, a short overview of the file, any dependencies etc.

Example (correct)

```
/*-------------------------------------------------------------------*
 * Function     : List_Insert(List l, ElementRef e) -> void          *
 * Input        : l     Any list.                                    *
 *                e     Element to insert into list.                 *
 * Precondition : none
 * Return       : void                                               *
 * Effects      : Destructively inserts element e somewhere into     *
 *                list l.                                            *
 * Ownership    : l declares ownership of e.                         *
 *-------------------------------------------------------------------*/
 void List_Insert(List l, ElementRef e)
 {
     ...
 }
```

(Ownership (*ägandeskap*) means assuming responsibility for release of allocated memory.)

Rule 2.1.4: Code should be accompanied by external documentation

Dependencies between different files/modules are best expressed using documents separate from the code. The same goes for installation instructions and user guides. As for other kinds of explicit documentation, there can be several different audiences, with different demands on how the documentation is written.

2.2 Modules

C doesn't support modules in the ordinary sense of the word, but it can be simulated by keeping the interface in a .h file and the implementation in a corresponding .c file. A .h file may only include declarations.[1] A .c file can have both declarations and definitions.[2]

[1] A declaration is something which does not generate code.
[2] A definition is something which generates code.

Rule 2.2.1: Export as little information as possible

Every name which is visible outside of a "module" is a potential for name collision. Visible information might lure another user into writing representation-dependent code.

Use `static`[3] to keep a name local to a module.

Rule 2.2.2: Let the name of the module be a part of exported names

The chance of name collisions is lower and clarity is improved. Additionally, the name itself will be part of the implicit documentation.

If you get a name collision between two different modules, there is no alternative to resolve the problem except by rewriting one of the modules. If you don't have access to the source, this can prove somewhat difficult.

One good and clear way of following the rule is using the name of the module as a prefix of all operations.

Rule 2.2.3: Use consistent naming principles within every module

Same reason as above.

Rule 2.2.4: Do not exploit implicit dependencies between modules

If the implicit dependency is changed, the code will stop working. Also, readibility will suffer.

An implicit dependency means that module **A** makes use of module **B**. If you need to use both, it might be "sufficient" to use module **A**, but then you have created an implicit dependency.

Rule 2.2.5: Prevent the contents of an .h file to be read several times

Reading the same .h file several times can lead to redefinitions of e.g. data types. The code will not compile.

A file `interface.h` should be written like this to prevent its contents to be read several times:

```
#ifndef _interface_h
#define _interface_h
...
#endif
```

Rule 2.2.6: A module should not define more than one ADT

A user who only wants to use a particular ADT shouldn't unexpectedly be given others as well.

If there is an ADT T_0 which makes use of another ADT T_1 both of them will (unfortunately) be visible. This is a consequence of the lack of a proper module system in C.

[3] The word `static` has different semantics in different contexts.

2.3 Rules for representation-independent code

Rule 2.3.1: Code should be representation-independent
Code which is independent of a particular representation code is easier to write, read and maintain.

Rule 2.3.2: Concrete data should be named
It is easier to understand a meaningful symbol than a constant value

Rule 2.3.3: Use the same naming convention for computed properties

The name should not reveal whether a property is stored or computed. If this is changed, the name will imply incorrect information.

Example The least amount of information needed to describe a vector is its coordinates. Often, its length is also of interest. The length can be computed from the coordinates - but if obtaining the length is a common operation, the length can be stored to save time. The drawback is higher memory consumption and (slightly) increased time to create a vector.

Rule 2.3.4: Avoid operations which are too specific.
An(other) user is only interested in the abstract and relevant properties of an ADT.

Example (incorrect)

```
typedef struct { double dx, dy, dz; } Vector;
...

typedef struct
{
    Vector    position;
    Vector    direction;
} Ray;

/* Constructor for rays. */
Vector Ray_create(Vector pos, Vector dir);

/* Does the ray r hit an object in the world w. */
Bool   Ray_hit(Ray r, World w);
```

Comment: That a ray hits an object in a world is not an abstract property of a ray. Possibly, it is an abstract property of a world.

Rule 2.3.5: Abstract away the representation in type definitions
A type can be simple or composite; a property which should be expressed when it is defined and not when it is used.

If it is visible that a `struct` has been used to represent a type, it must be written everywhere the type is being used. This leads to lots of work if we want to change the representation.

Example (incorrect)

```
void foo(struct Foo foo, int bar) {
    ... }
```

Example (correct)

```
typedef struct Foo Foo;
...

void foo(Foo foo, int bar) {
   ... }
```

Rule 2.3.6: Truth values must be typed.
Boolean expressions has different semantics compared to expressions which evaluate integers.

This is an ideomatic abomination in C. Unfortunately everything vaguely reminiscent of integers - e.g. pointers - are used as truth values. Even if this is correct according to the C language definition, it does not lead to readable programs. Values should be explicitly compared with zero or `NULL` as the case may be.

Example (incorrect)

```
{  int a;

    a = lic[4](..);
    if (a)
        ...
}
```

Example (correct)

```
typedef enum { False = 0, True = 1 } Bool;
...

{  Bool a;

    a = lic(..);
    if (a)              /* if the lic computes a truth value */
        ...
}
```

Example (correct)

```
{  int a;

    a = lic(..);
    if (a != 0)         /* if the lic computes an integer */
        ...
}
```

[4] long involved computation

Rule 2.3.7: Arrays should be abstract

An array is a composite data type (size + elements). Also, an array of a particular type can sometimes be represented in a better way that as a straightforward C array. E.g. an array of truth values can be represented as a bit string.

Rule 2.3.8: Don't make an object a part of another structure.

We want to get individual objects that are entirely separate from a containing structure. Also, we want to share objects between structures.

Example (incorrect)

```
typedef struct PersonList_T *PersonList;
struct PersonList_T
{
    int          age;
    char         *name;
    PersonList   rest;
};
```

Comment: Age and name are abstract properties of a specific person - not of a list of persons.

Example (correct)

```
typedef struct
{
    int          age;
    char         *name;
} Person;

typedef struct PersonList_T *PersonList;
struct PersonList_T
{
    Person       first;
    PersonList   rest;
};
```

2.4 Macros

Rule 2.4.1: Don't forget that macros use *call-by-name*.

call-by-name semantics can cause problems.

Example (incorrect)

```
/* Macro to square first argument and assign to y */
#define square(x,y) { float t; t = (x)*(x); y = t; }

  .. foo(t, z) ..                          /* expands to */
  ..  { float t; t = (t)*(t); z = t; } .. /* something undefined */
```

Rule 2.4.2: Enclose formal parameters in parentheses (in macros)
Macros use *call-by-name* at the textual level, so the semantics is not always what you think. If the macro expands to an expression this is true for the entire macro body.

Example (incorrect)

```
#define silly(x,y) x*y

  .. z = silly(a+b, c-d)*t ..   /* expands to */
  .. z = a+b*c-d*t ..           /* != (a+b)*(c-d)*t */
```

Example (correct)

```
#define silly(x,y) ((x)*(y))

  .. z = silly(a+b, c-d)*t ..   /* expands to */
  .. z = ((a+b)*(c-d))*t ..     /* == (a+b)*(c-d)*t */
```

2.5 Variable definitions

Rule 2.5.1: Avoid variable definitions without initial values
You might forget to initialise the variable, leading to undefined behaviour.

You can give the variable an initial value *at the same time* as it is defined. You also shall not reuse variables for different computations. Variable memory reuse is better left to the compiler.

2.6 Return values

Rule 2.6.1: Avoid returning values using pointer arguments
The code becomes difficult to read since (1) you have to introduce unnecessary temporary variables or (2) it is not obvious that a side-effect it performed on one or more of the arguments. Also, it prevents the use of functional composition.

This is (unfortunately) a quite common idiom in C programming, but you should avoid it for the sake of readability. There are situations, though, when it is reasonable to use this technique.

2.7 Rules for stack vs. heap allocated objects

A property of an ADT which unfortunately can not be abstracted away in C is where/how an object is stored. An object is allocated on the heap or on the stack. A heap object is allocated memory through the use of `malloc` (or a similar function) and at the end of its life, the memory must be explicitly deallocated. To avoid memory leaks we must know what objects are active and when they are no longer used. Also, we must not deallocate memory for an object more than once. Memory for a stack object is allocated only while the procedure that created it is active. No explicit memory management need be done. A stack object is a ordinary value.

Since the semantics of stack and heap objects differ so much, we can not disregard the question of how they are allocated. Heap objects will be represented by a reference (a "pointer") while stack objects are represented as values. A function call will copy a stack object, so changes will not be visible outside the function (attempting to) changing it.

Rule 2.7.1: Stack and heap objects should have different types.
Stack objects has a different (copying) semantics compared to heap objects.

Also, the primitives used with the object will reveal what kind of object it is.

Example (incorrect)

```
typedef struct
{
    int    age;
    char   *name;
} Person;

Bool Person_identify(Person *p);
```

Example (correct)

```
typedef struct
{
    int    age;
    char   *name;
} Person;

Bool Person_identify(Person p);

typedef Person *PersonRef;

Bool PersonRef_identify(PersonRef pref);
```

Rule 2.7.2: Never treat a stack object as a heap object
The converse is also true. An object should keep its particular semantics during its lifetime. There is also a risk that a heap object will point to the stack. Such a pointer will have a short life-span!

If you still (for some reason) want to use reference semantics with a stack object it must be done in a safe way. That is, the stack object should be copied to the heap. Conversely, if you want to use copying semantics with a heap object, it should be copied to the stack. There should be primitives available for this purpose.

Rule 2.7.3: Always document the intended use of an object
The risk for memory leakage is less if it is clearly stated who should release the memory for the object.

Example (correct)

```
/*************************************************************
 *                                                           *
 * ADT         : Bitmap                                      *
 * Description : ... A bitmap is always heap allocated so it *
 *               only uses reference semantics. ...          *
 * Invariant   : The size of the ColourArray should be       *
 *               width*height                                *
 *************************************************************/
typedef struct Bitmap_T *Bitmap;
struct Bitmap_T
{
    int         width, height;
    ColourArray  pixels;
};
```

2.8 Rules for error handling

Rule 2.8.1: Error codes should always be handled.
Continued program execution (as if everything was right) is not meaningful if a function returns an error code. It is not unusual that memory runs out, that the disk becomes full, that a file does not exist etc.

Example (incorrect)

```
{
    char       data[100];
    FILE       *f = fopen("config.dat", "r");
    fread(data, sizeof(char), 100, f);
    fclose(f);
}
```

Example (correct)

```
#define CONFIG_FILE         "config.dat"

{
    char       data[100];
    FILE       *f = fopen(CONFIG_FILE, "r");

    if (f == NULL)
    {
         fprintf(stderr, "File '%s' could not be opened.",
                 CONFIG_FILE);
         exit(-1);
    }

    fread(data, sizeof(char), 100, f);
    fclose(f);
}
```

Rule 2.8.2: Always validate user input.
We don't want the program to crash because of bad input data. The user is part of reality and we all know what reality can do to you.

Rule 2.8.3: Check that preconditions are satisfied before execution.
Continued program execution (as if everything was right) is not meaningful. We risk undefined behaviour.

Using `assert` for this purpose make it easy to find bugs when programs are tested. When you believe in the correct function of your program, you can turn the checking off to save time.

Example (incorrect)

```
...
typedef ObjectList_T *ObjectList;
struct ObjectList_T
{
    Object       *first;
    ObjectList   *rest;
}

/* Return the first element of a non-empty list. */
Object ObjectList_first(ObjectList ol)
{
    return ol->first;
}
```

Example (correct)

```
#include <assert.h>
...

/* True iff list is empty. */
Bool ObjectList_isEmpty(ObjectList ol)
{
    return ol == NULL;
}

/* Return the first element of a non-empty list. */
Object ObjectList_first(ObjectList ol)
{
    assert(!ObjectList_isEmpty(ol));

    return ol->first;
}
```

Rule 2.8.4: Always use sensible error messages.
Facilitates debugging and lets the user know what really went wrong.

Example (incorrect)

```
Bitmap Bitmap_createFromFile(char *filename)
{
    FILE *f = fopen(filename, "r");

    if (f == NULL)
    {
       fprintf(stderr, "Shit!");
       exit(-1);
    }
    ...
}
```

Example (correct)

```
Bitmap Bitmap_createFromFile(char *filename)
{
    FILE *f = fopen(filename, "r");

    if (f == NULL)
    {
       fprintf(stderr,
               "Bitmap_createFromfile: Couldn't open file '%s'",
               filename);
       exit(-1);
    }
    ...
}
```

2.9 Flow of control

Rule 2.9.1: Distinguish between expressions and commands
The readability will benefit if you show what you really mean.

The most noteworthy difference is that C has both conditional expressions (`?:`) and conditional statements (`if .. then .. else ..`). Make a distinction between them and use the right construction at the right time.

Rule 2.9.2: Don't use GOTO
Readability will suffer (to say the least).

2.10 Global variables

Rule 2.10.1: Avoid global variables
Global variables cause unnecessary limitations as they only allow one instance of the concept depending on the global variable. Also, it will introduce dependencies between different parts of the code which are difficult to spot.