Programming and Algorithms

C: Useful tips.

Jean-Bernard Hayet

CIMAT, A.C.

Writing C code

Is C really necessary?

- C is a rather **low level, very powerful** language;
- but the power comes with a cost: **high level of responsibility**;
- Before starting a programming project,

Structure your code

- A good structure of your project is a **help for the reader**.
- **Separate** the code in **portions, related to semantic units** (objects, concepts).
- One .c for each portion. Prototypes, constant declarations in .h
- Separate functionalities and useful structures from their final use: **libraries** (static/dynamic).

Structure your code

Example:

```
image.h
image.c
imageDerivative.h
imageDerivative.c
interestPoints.h
interestPoints.c
------------------------
imageLib.dll (.so,.a,.dylib)
prog.c with main() dedicated to a specific task
```

Stick to the standard

- If possible, do no use some features that are specific to your OS or your compiler.
- Your code may be compiled by people that have a **completely different system.**
- Stick to a **well-defined standard** (C99 or C11).

For example:

```
#include <conio.h>
```

Control the quantity of code

- Any line of code is a **potential source of bugs**!
- Avoid **repetitions**: find a way to write your functions/structures in the **most generic way** to avoid to re-write them for several types.
- Few mechanisms allow genericity (compared to C++)
  1. **void*** pointers.
  2. Preprocessor **macros**.
  3. **Pointers to functions**.

External libraries

- For many programming problems, libraries **already done and bulletproof**: time gains and (sorry guys...), they are probably written better than your code.
- This may involve not writing your own structures but use those of the external library.
- Example: Lapack, written in Fortran, is a reference for numerical computing.
- Example: in C++, Boost proposes many useful structures, containers, algorithms...
- Example: OpenCV, libMagick in image processing and computer vision.

Readability

- Use **variable names that help in understanding their purpose**.
- Avoid very short/very long names.
- Stick with a **convention**.
- In general: capital letters for constants defined through define; smaller letters for variables.
- One instruction per line.
- **Indentation**: Again, stick to a rule (most editors do it automatically).

Readability

Be **cautious with pointers**.

- Do not use pointers when not necessary.
- Makes the code harder to read.
- Source of errors.

Readability

- Avoid the **repetition** of **numeric constants**: define them once for all, visible from all your code through **some meaningful name**.
- In C, use defines (with capital letters). Example from libdc1394

```
/* Capture flags */
#define DC1394 CAPTURE_FLAGS_CHANNEL_ALLOC 0x00000
#define DC1394 CAPTURE_FLAGS_BANDWIDTH_ALLOC 0x00
/* a reasonable default value : do alloc of bandwidth and chann
#define DC1394 CAPTURE_FLAGS_DEFAULT 0x00000004U
```

- In C++: static const.
- Advice: **ban numerical values** from all your .c files.

Readability

Even though they have caveats in C, enums are practical to make the code more visible, as they introduce semantics:

```c
/* Operation modes */
typedef enum {
        DC1394_OPERATION_MODE_LEGACY = 480,
        DC1394_OPERATION_MODE_1394B
} dc1394_operation_mode_t;
/* Format 7 sensor layouts */
typedef enum {
        DC1394_COLOR_FILTER_RGGB = 512,
        DC1394_COLOR_FILTER_GBRG,
        DC1394_COLOR_FILTER_GRBG,
        DC1394_COLOR_FILTER_BGGR
} dc1394_color_filter_t;
```

Readability

Although you may be technically correct, try not to use numerical values, even for obvious data such as sizes:

```c
int* ptr = (int*)calloc(n, 4);
int* ptr = (int*)calloc(n, sizeof(int));
int* ptr = (int*)calloc(n, sizeof(*ptr));
```

The last two are preferable, but **the last one is even better**: If, later on, you change the type of the pointed variable, the size will adapt to the new type length.

sizeof is **not costly**: it is resolved at compile time.

Readability: Restrict visibility

- **Limit local variables to the smallest scope possible** (just before their use).
- This may have some overhead but it improves readibility a lot.
- You may also apply the same principle to functions, by **declaring them as static** (they will be visible only within the file).

Readability

- Your code will be read by people that will want to **understand** what it is about.
- **Comment the critical parts**, those that are not trivial to read/understand.
- Use comments to describe what the function/program/library is doing as an introduction.
- You may also describe information about the authors and the license.

Comments

Use comments in header files **to describe, before the prototype of each function, what it does**.

```
/* Retrieves next contour */
CVAPI (CvSeq*)cvFindNextContour(CvContourSc anner scanner);
/* Substitutes the last retrieved contour with the new one
( if the substitutor is null , the last retrieved contour is
removed from the tree) */
CVAPI (void)cvSubstituteContour (CvContourScanner scanner,
                                 CvSeq* newcontour);
/* Releases contour scanner and returns pointer to the first
outer contour */
CVAPI (CvSeq *)cvEndFindContours(CvContourScanner* scanner);
```

Comments

In control structures (with nested loops) comments are welcome with closing } to remind which loop it is:

```
for (int i=0;i<cascade->count;i++)
{
  for (int j=0;j<cascade->stageclassifier[i].count;j++)
  {
        for (int l=0;l<cascade->stageclassifier[i]. classifier [j].cou
        {
                CvHaarFeature *feature =
                  &scascade->stageclassifier[i]. classifier [j]. haarfea
                 [...]
                hidfeature->rect[0].weight = (float )(-sum0/area0);
        } /* l */
  } /* j */
} /* i */
```

Documentation

- A software like **doxygen** uses the comments in the headers of your code to generate documentation automatically (HTML, PDF,...).
- In C++, it also gives the class structure, relation graphics etc.

http://www.doxygen.org

assert

- Many functions have **pre-conditions** to work correctly.
- Examples are non-null pointers, limits to indices. . .
- When developing, and to debug more easily, you can use the **assert** function that will quits the program execution if the condition is not met.

```
void assert ( int expression );
```

Even in deployment versions, it improves the understanding (the reader sees well the preconditions); you can also use it for **post-conditions**.

Pointers and dynamical allocation

- Remind that pointers are not necessarily equivalent to dynamic allocation.
- free() applies only to memory on the heap.
- A pointer can **point to any place in the memory**: stack, heap, global variables.
- In the case of memory on the heap, they are the **only way** to refer to this memory space.

Track the memory allocated on the heap

- In C, there is **no garbage collector** as in other languages: Freeing allocated memory is done explicitly.
- Often the allocation/liberation functions appear **in the same block**.
- This is **not always that trivial**: The malloc/free calls may be very far apart.
- There are some functions from the C standard library that **allocates on the heap** and it needs to be freed:

```c
char *strdup(const char *s);
```

Identify clearly pointers to freed memory

- After a call to free(), the **pointer value and the memory content do not change**.
- However, later on, the program may allocate new memory at that location and use it. So it is a **bad idea to keep a valid pointer to freed memory**: It could lead to compromising the content of the heap.
- A common practice is to **set these pointers to NULL**.

Aliasing

As the only way to deal with memory on the heap is through pointers, it is possible to get into this kind of situations:

```
int *data = (int *) calloc(nData, sizeof(*data));
...
int *dataCopy = data;
...
free(data);
```

This is called **aliasing**, i.e. several pointers point at the same place on the heap. This makes very difficult to manage the liberation, to debug etc. (because, yes, you *freed* data).

Common errors

Typos and syntax

- Misuse of "=" and "==".
- Forget a **break** in a **switch**.
- **Operator** aliasing.

```
double ratio = 1/3;
```

- Difficult-to-see spurious syntax items

```
int i = 10;
while (i>0);
        i--;
```

Typos and syntax: strings

- Misuse 'a' and "a".
- Use of "==" or any other relation operator between strings.
- Forget the ending "\0" or lacking space for this character in memory allocation.

Wrong assumptions about what the compiler does

Order of instructions

data1[j++]=data2[j++];

Assuming **initialization** to 0 for local variables.

Missing prototypes

When the compiler finds a call to a function which prototype is not given (yet), it **only generates Warnings** and compiles.

The compiler deduces which arguments its takes and the default return type is int.

Ill-defined blocks

With nested conditionals: better **define your blocks clearly with {}**.

```
if (something)
        if (somethingElse)
                doThis();
else
        doThat();
```

Memory

- **Out of range** indices.
- **Uninitialized** pointer values.
- Pointers read with **another type** than the one they should be.
- Access to **already free**d memory.

Memory

Scanf-like functions may be source of errors:

- They need a **pointer to the structure** that will hold the read data;
- The **formatting string should** use the proper flag.

```c
long data;
scanf("%ld",data); // Syntactically correct!
```

Caution with %s: the programmer should ensure that the memory is enough.

Memory

Do not forget to check the returned pointer from dynamic allocation.

```
int* data = (int*)malloc(1000*sizeof(*data));
if (data==NULL) {
...
```

Data initialization

- Never forget that the simple declarations of local variables and the allocation on the heap through malloc let the memory **uninitialized**.
- In case of using this garbage data, prefer systematically:
  - **Initialize** all your local variables.
  - Prefer calloc.

Reading files

Test with EOF :

```
int countLinesize(FILE *fp) {
        char ch;
        int cnt = 0;
        while ((ch = fgetc(fp))!=EOF && ch != '\n')
                cnt++;
        return cnt;
}
```

Reading files

Caution with the return value of fgetc, getc, getchar!

```
NAME
fgetc, fgets, getc, getchar, ungetc - input of characte

SYNOPSIS
#include <stdio.h>

int fgetc(FILE *stream);
char *fgets(char *s, int size, FILE *stream);
int getc(FILE *stream);
int getchar(void);
int ungetc(int c, FILE *stream);
```

Reading files

Caution with feof:

```
#include <stdio.h>
int main() {
        FILE *fp = fopen("test.txt","r") ;
        char line[100];
        while (!feof(fp)) {
                fgets(line, sizeof(line), fp);
                fputs(line,stdout) ;
        }
        fclose(fp);
        return 0;
}
```

feof gives the result of the **last reading operation** (fgets).

Error codes

A good practice is to **propagate error codes** through the return values of functions:

```
/* Return values for visible functions */
typedef enum {
        DC1394_SUCCESS = 0,
        /* Success is zero */
        DC1394_FAILURE,
        /* Errors are positive numbers */
        DC1394_NO_FRAME = -2, /* Warnings or info are negative numbers */
        DC1394_NO_CAMERA = 3,
        DC1394_NOT_A_CAMERA,
        DC1394_FUNCTION_NOT_SUPPORTED,
        DC1394_CAMERA_NOT_INITIALIZED,
        DC1394_INVALID_FEATURE ,
        DC1394_INVALID_VIDEO_FORMAT,
        DC1394_INVALID_VIDEO_MODE,
        DC1394_INVALID_FRAMERATE,
        DC1394_INVALID_TRIGGER_MODE,
        DC1394_INVALID_TRIGGER_SOURCE,
        DC1394_INVALID_ISO_SPEED,
        DC1394_INVALID_IIDC_VERSION,
        DC1394_INVALID_COLOR_CODING
```

Warnings

The C compiler may be very permissive and generate executables when there are in fact huge errors.
Example:

```c
int h() {

}
int main() {
        int a=h();
        printf("%d\n",a);
}
```

Use **all the warnings** (-Wall) and consider them as errors.

Debugging

Facing bugs

- Check all **allocations/liberations**.
- Compile for debugging (gcc -g).
- **Reproduce** the crash in the debugger.
- Do not enter in **Panic Mode**.

Debugging

- You may start by using **printf** to debug (before using a debugger), and it is OK.
- Use an **endline character** in your printf (it flushes the buffer). Without it you may think that your program has not reached the printf command while it has (but the impression stayed in the **buffer**).

Debugging

- Important (although not necessarily sufficient) to find bugs.
- The program is **run within the debugger** (ex: GDB), which has control over the memory program.
- Allows **step-by-step execution**, linearly (next) or getting into (step) functions.
- **Breakpoints** where the execution stops (but can be continued).
- **Observation** of variables, of the state of the memory, of the stack.

Debugging

- When your program crashes: re-run it through the debugger.
- **Locate** roughly where the program crashes (with breakpoints or printf).
- Set **breakpoints** close to this area, and execute step-by-step to understand what is going wrong.
- The **state of the stack** at the moment of the crash will give you some hints.

Debugging: gdb

Typical use:

```
Reading symbols from ./myProgram...done.
(gdb) b tata.c:4
Breakpoint 1 at 0x6a5: file tata.c, line 4.
(gdb) run
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x000055555555470a in main () at tata.c:7
7                 a[i*100]=a[i];
(gdb) p i
$1 = 337
```

Debugging: gdb

- r(un): runs the program.
- q(uit): quits.
- b(reak) [file:]function/line: sets a breakpoint (gdb will stop the program each time it arrives at that point, before the instruction).
- c(ontinue): continues after stopped.
- bt: display the function calls stack.

Debugging: gdb

- n(ext): runs the single next instruction linearly (does not get into a function call).
- s(tep): run the single next instruction vertically (gets into a function call, when this applies).
- print: prints (once) an expression or the content of a variable.
- display var: makes that a variable will be displayed at each execution step (e.g. after each next, step).

Debugging

For some delicate bugs such as **memory corruption** (i.e., when some freed area is written through a pointer), you may use the command **watch** to set a **watchpoint**, i.e. detect changes in the value of some specified chunk of memory.

The execution stops when a change is observed!

```
(gdb) watch *(int *) 0x6198212
Watchpoint 1: *(int *) 8916
```