C++ for dinosaurs

Guide for readable, maintainable, reusable and faster code

Nick Economidis

C++ for dinosaurs

Guide for readable, maintainable, reusable and faster code

Nick Economidis

This book is for sale at http://leanpub.com/cpp_for_dinosaurs

This version was published on 2014-07-25

ISBN 978-1-312-36766-1

Contents

Preface

Habits are the brain's actions in power-saving mode. Experienced programmers evolve habits so that they can save energy in common tasks, and spend it in solving difficult problems.

Dinosaurs are the experienced programmers in the company who have evolved a style of programming featuring a certain pattern of habits. The older the dinosaurs are, the more obstinate they are. They aren't willing to break these habits, because they work fine for them.

From the aspect of development, this argument is actually right; development is usually faster when you use habits and don't stand to think for simple things. For the whole program though, development only accounts for about 15% of the temporal effort, while maintenance accounts for nearly 80%. There are habits, especially old ones, that have a serious impact in reading, understanding, and therefore maintaining code. This includes both other people's code, or even one's own, two-month-old code.

In this book I am addressing such habits, and provide alternative ones, based on contemporary instruments provided by C++.

This book is not about teaching for a paradigm-shift. It is about writing C++ using native idioms, not like C, Java, or Fortran.

The book makes use of features in *C++03* and *C++ Technical Report 1 libraries* (`tr1`). So, any compiler after 2005 should support the contents of this book. Certain features of C++11 are also discussed as an alternative, where applicable.

Who is this book for

- This book was originally aimed for *engineers* and *programmers* who strongly believe in C. Most bad habits stem from practices developed by the low-level nature of C. The problems of these habits are demonstrated and alternatives are offered. The book contains all the necessary material, so that C programmers can get the kickstart they need to embrace C++.
- *C++ programmers* will benefit from the style described in this book. Their style is often filled with impurities inherited from Java and C books.
- Students will find a guide to creating professional looking coding style. Colleges very rarely provide any incentive for writing quality code. This book bridges the gap between knowing C++ and writing good code.

If you care about software quality but fear that it takes huge organisational changes to achieve it, this book is the missing link towards your goal.

I am describing steps that you can take to write clear code. This is an important quality because:

- Clear code increases readability.
- Readable code allows your partner to have an opinion about your code.
- Readable code is the prerequisite for code reviews.
- It is other people's eyes that spot bugs, not tests[1].
- Clear code also promotes maintenance.

Prerequisite knowledge

- This book will *not* teach you C++. I expect that you can write C or C++ without much difficulty. I suppose that at least six months of working experience or a couple of years of exercises at college should be enough in order to appreciate this book.
- You do not need to know C++, but you should be able to write in C.
- *No* object oriented programming knowledge is required.

How to read this book

The first half of the book can be read sequentially. I demonstrate the characteristics of poor code that I am addressing. Then I describe the common arguments of those who resist adopting C++ and explain how I am going to invert them. Finally, I describe the six steps that you can take to write clearer code.

Throughout the first chapters, there are references to the second part of the book. All examples and demonstrations are studied in separate chapters. This gives me enough space to go into greater depths. It makes reading the whole concept smoother, too.

Why this book was written

I studied Electrical Engineering in the mid 90's, and I was taught C and Matlab for problem solving. I only used C for production code.

I have worked on programs that already had a lifetime of more than 10 years. This means that they carried a lot of legacy code. The features had to be delivered fast and there was hardly ever enough time for proper programming.

Most of my career I had to use C and C++. However, there was always an underlying rule that C was to be preferred. There was not enough trust that C++ had a significant advantage – C++ was thought

[1]tests are written to spot bugs that you know they exist, so that they do not appear again. But tests do not discover bugs.

of as an object-oriented language. Indeed, I have made a lot of regrettable errors in my projects, as a result of experimenting with features of C++, especially object-oriented programming. However, I always felt a strong distaste whenever I had to switch to C.

Finally after ten years, through the advancements of C++ and some experts' ideas, I feel that I can present (without remorse) some ways C programmers can use to create elegant, readable, yet traditional programs.

These ways involve the use of certain C++ features to make good old-fashioned C code read and work better. They have nothing to do with object oriented programming. It is not the most suitable solution for most problems anyway.

I presented these advancements through a series of lectures to my colleagues, and it was the first time that I managed to change the dinosaurs' attitude towards C++. In fact, they were so interested that *they* encouraged me to write this book.

In this book I will demonstrate how C-style makes code unreadable, buggy and slow. I will show the features of C++ that deter people from adopting it, and how to fix this issue. I will present a few basic rules, that will change how you write, not how you think. Ultimately, with this style shift, your code will read nicer, you will reduce the bug count, and your program may even run faster.

Acknowledgements:

This book collects the best ideas and suggestions about readable code, presented recently by Sean Parent, Bjarne Stroustup, James Coplien and Trygve Reenskaug. Their presentations were so inspiring I felt I had to test, present and write about them. This book serves as a testimonial that their proposals actually work for production code, and are also easy to learn and adhere to.

The cover of this book was painted by Chrysa Malama[2].

[2]http://rabbithole2014.blogspot.com/

0.1 Step 4: Algorithms should read like pseudocode

When an algorithm is described like pseudo-code, there is hardly ever any mention about the underlying data-structures. No statement is made on whether a `list`, a `dynamic array`, or a `binary tree` should be used.

With C++, it *is* possible to code in such a way.

Let's compare how you would write an algorithm in C, that accumulates numbers in an array:

```
int accumulate (int numbers[], int sz, int sum)
{
    int i;
    for (i=0; i<sz; i++) {
        sum += numbers[i];
    }
    return sum;
}
```

Now, what would you do if you had to accumulate the entries of a `binary tree`? or a `linked list`?

- would you *re-write* a version of `accumulate()` for the `tree` or the `list`? or
- would you *copy* the `tree` or `list` into an array and then call `accumulate()`?

well, people usually choose the path of least pain:

- if you're in a hurry, you copy the data into suitable data structure, and then move to the next task.
- if you have time, you rewrite the algorithm, find a clumsy name, like `accumulate_list()`, or `accumulate2()`, taking the risk of introducing bugs.
- If the algorithm is simple enough, you rewrite it in-place, where you need it.

> a really conscious programmer might even consider writing a macro for it–for reuse. Try writing a macro for as simple an algorithm as this one! Then, try to provide a nice API to call it!

Done that? Now what would you do if you needed to accumulate over a container of `doubles`?

But really, if you wanted to describe the algorithm to your colleague, you would write it like:

```
T accumulate (IT from, IT to, T sum)
{
    foreach entry in [from:to]
        sum += entry;
    return sum
}
```

well, the equivalent in C++ is:

```
template <typename T, typename IT>
inline
T accumulate (IT from, IT to, T sum)
{
    for (IT iter = from; iter != to; ++iter)
        sum += *iter;
    return sum;
}
```

Look how you can call it:

```
vector<int>  v;   int    sum = 5;   sum = accumulate (v.begin(), v.end(), sum);
list<double> v;   double sum = 5;   sum = accumulate (v.begin(), v.end(), sum);
set<float>   v;   float  sum = 5;   sum = accumulate (v.begin(), v.end(), sum);
```

this allows you to replace the container types without changing user code.

0.1.1 How does that compare to a C function ?

readability
apart from the first line that declares the `template` types, the rest is good-old C/C++ code – there is no special syntax, or characters like `\`, `#` or `##` that you see in macros.

speed
execution speed is definitely much faster than copying the container to an array. The function is marked as `inline`, allowing the optimiser to decide whether it should be inlined, or not. In C,

- if `accumulate()` was an `extern` function, the optimiser would not be able to optimise.
- if a *macro* was used, the function would have forcefully be inlined, but that might not be the best action to take, especially for functions longer than 2-3 lines.

size of executable

if `accumulate()` is implemented as an **`extern`** function:

- in C, the code remains in the executable, even if it is not called.
- in C++, the `template` function is instantiated only when it is used.

if `accumulate()` is implemented as **macro**:

- in C, the code is copied in every macro use.
- In C++, *some* calls of the `template` function may be inlined, while others will share a single instance of the function.

code reuse

only by using a macro (a well known bad practice) can you reuse an algorithm with different parameter types. Template provides a much better option.

Learning the details

1 Avoid using low-level strings

C-style strings require too much housekeeping. Code is susceptible to bugs and memory leaks.

1.1 Problem description

You do not know when a function returns a pointer to a string, or to a copy of the string. You simply cannot trust the return value of a function.

Consider the following code, that uses a function, `change_working_directory()`.

```
const char *old_dir;
const char *new_dir;

change_working_directory ("~/books/literature");
old_dir = get_working_directory();

change_working_directory ("~/books/comics");
new_dir = get_working_directory();
...
```

What is the string for `old_dir` and `new_dir` ?

It should be that `old_dir` is `"~/books/literature"` and `new_dir` is `"~/books/comics"`. But it actually depends on the implementation of `get_working_directory()`.

typically, you can get the current working directory using this method.

```
int sz = A_FILENAME_MAX;
char *cwd = malloc(sz);
if( !getcwd( cwd, sz ) ) {
    if (errno == ERANGE) {
        do {
            sz *= 2;
            cwd = realloc(cwd, sz);
        } while (getcwd(cwd, sz) == NULL && errno == ERANGE);
    }
    else {
    }
}
```

If `get_working_directory()` is implemented as:

```
const char* get_working_directory()
{
    int sz = A_FILENAME_MAX;
    static char *cwd = malloc(sz);   // static, to avoid allocations
    ...
    return cwd;
}
```

then `old_dir` and `new_dir` will be the same, namely `"~/books/comics"`. Which is *totally counter intuitive*!

If this is the way that `get_working_directory()` is implemented, you would have to write

```
const char *old_dir;
const char *new_dir;

change_working_directory ("~/books/literature");
old_dir = strdup (get_working_directory());

change_working_directory ("~/books/comics");
new_dir = strdup(get_working_directory());

...
free(new_dir);
free(old_dir);
```

We have guarded ourselves from unexpected behavior, taking a big hit in readability.

Unfortunately, this is not a global solution against this type of problems: the code now runs the risk of *memory leaks*.

What if `get_working_directory()` is implemented like this?

```
const char* get_working_directory()
{
    int sz = A_FILENAME_MAX;
    char *cwd = malloc(sz);   // non-static
    ...
    return cwd;
}
```

In this case, it is the *user* who is responsible of freeing the string.

```
const char *old_dir;
const char *new_dir;

change_working_directory ("~/books/literature");
old_dir = get_working_directory();

change_working_directory ("~/books/comics");
new_dir = get_working_directory();

...

free (new_dir);
free (old_dir);
```

As you can see, there's no way you can compose the user code to guard against any kind of implementation of such functions. Even if one looks at the declaration of the function, he cannot tell whether he owns the string returned, or not.

1.2 The solution

The solution is to use C++'s `string`. In this case, we can implement `get_working_directory()` was so that it returns a `string`:

```
string get_working_directory()
{
    int sz = A_FILENAME_MAX;
    char *cwd = malloc(sz);
    ...
    return string(cwd);
}
```

This leads to a user code which is always safe, and easy to read.

```
change_working_directory ("~/books/literature");
string old_dir = get_working_directory();

change_working_directory ("~/books/comics");
string new_dir = get_working_directory();

...
```

now, this code

- is easy to read
- requires no housekeeping (no calls to `free()` are needed)
- the strings returned *can* be changed - they are *not* `const`
- there are no surprises when it comes to calling `get_working_directory()`