C, C++, Rust

Simon Pohmann

February 13, 2022

Contents

What makes a good programming language?

To compare programming languages, we first need find some criteria according to which we want to compare them. One criterion, which is especially relevant in the context of the languages C, C++ and Rust is performance. However, all of those three languages allow - in principle - to write code that is as fast as tuned assembly. One might now compare the speed of non-tuned (i.e. naively implemented) algorithms. This is not really an objective comparison, as we do not have a definition of "naively implemented". Still, one can find a lot of comparisons along these lines online [1] seem to indicate that still all three are approximately equal. Hence, we will not compare performance in this essay.

[1] https://devetry.com/blog/c-v-rust-speed-safety-community-comparison/, https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust-clang.html, https://github.com/ixy-languages/ixy-languages/blob/master/Rust-vs-C-performance.md

Instead, we focus on how well the language allows you to implement ideas in a straightforward way. This is absolutely crucial, as straightforward implementations make it easier to avoid bugs and produce more readable code. For software engineering, this is of course by far the most important metric, although it is again very hard too evaluate. Because of this, in the following, we mainly present examples that illustrate situations in which complicated, unwieldy or hard-to-understand implementations are the way to go, or in which some language features behave in unexpected ways.

1 C

As one of the oldest and languages still in common use, C has a history full of famous projects and a gigantic influence on the development of programming languages in general. Slightly surprisingly, unlike C++, the language has not changed too much since its beginning, and still preserves some of its original simplicity and clarity. However, the other side of this minimalistic approach is that C does not offer many ways to build sophisticated abstractions. Mainly, I consider those to be the more OOP-style typedef struct and function pointers, whose style is more "functional". And finally, there are also macros, which certainly deserve to be mentioned explicitly.

1.1 Typedef struct

In fact, as typedef struct is already a combination of a typedef and a struct. In other words, one often writes

```
typedef struct S {
    int x;
} S;
```

instead of

```
// define the struct
struct S {
    int x;
};
// define the type corresponding to an element of the struct
typedef struct S S;
```

Separating the type and struct seems quite unnatural from an OOP perspective, and is not really useful in general. However, except for the ugly syntax, there are more serious drawbacks of this construction.

- First of all, **members of a struct are always public**. While this is not a problem in small projects, visibility is a fundamental part of abstraction is very valuable to build bigger projects. Some people claim that programmer's discipline should be sufficient to prevent using things in ways they are not supposed to be used to, and others oppose them with the argument that programmer's are not disciplined enough in reality. Both make valid points, but the fundamental part is whether the compiler forbids you to access a member, but that the notion of visibility allows you to document in your code which parts are internal and which are external. This makes your code much easier to read and understand, and hence better. Of course, one can fake this documentation e.g. by prepending _ to internal members, but this makes it much easier for mistakes to happen, and is definitely not straightforward.

- There are **no interfaces**. In OOP, classes are not just abstractions of collections of data. In fact, their most important feature is not inheritance, not the contained data and not visibility. The real power comes from subtyping (which is **not** inheritance), i.e. interfaces [2].

  Interfaces are needed whenever an algorithm or operation should work with different kinds of objects that share features, but also if it is just about different representations of the same object. Furthermore, even if there is just one kind of object and representation, interfaces are still extremely useful, by limiting the amount of information the algorithm works with, which introduces no unnecessary and confusing information for a reader. This way, code becomes much more readable.

  In fact, interfaces are so important that functional languages implement it (e.g. Haskell in form of traits), that C++ has recently introduced the notion of a "compile-time interface" (called concept), there is the term "interface-driven programming/design", most modern approaches to web engineering fundamentally rely on interfaces (e.g. REST, SOAP), interfaces are also heavily used in hardware design, ... In other words, they are the core of modern software engineering, and also so important for C programming that it is a common pattern to simulate them using function pointers. While this does work, it is error-prone and far away from straightforward. Honestly, the argument that one can simulate it in complicated way is just nonsense, otherwise we all could just write assembly.

- No **syntax for functions**. As opposed to all object-oriented and many other languages, C does not provide a .-syntax for member functions of structs. While this is just a cosmetic issue, it still can make code much harder to read and understand. Personally, since I do a lot of mathematical programming, I especially miss operators. Namely

```
FloatVector x; // some custom vector implementation
FloatVector space;
FloatVector proj = space * space.transpose() * x;
```

[2] It is easy to miss the difference between interfaces and superclasses, as inheritance relies on subtyping. However, subtyping is much more fundamental, and a very powerful way to build abstractions. On the other hand, inheritance is mostly a convenient way to reuse code. In fact, inheritance can even be hard to understand (see https://en.wikipedia.org/wiki/Composition_over_inheritance), but subtyping is probably the most important concept in programming, also mirrored in other paradigms in similar ways, e.g. in functional programming.

is just so much more readable than

```
FloatVector x;
FloatVector space;
FloatVector proj = multiply(
    multiply(space, transpose(space)),
    x
);
```

Please note that I am not a fan of inheritance, and certainly will not complain that structs cannot be derived from each other.

1.2 Function pointers

I have already talked a lot about function pointers above, in the point about interfaces. The main problem is that it is very hard for them to contain state. Here I want to note that we do not really want them to contain mutable state (this would again give objects and classes), but at least **immutable** state is required very often. For example, assume we have some collection data list that supports a function

```
for_each(IntCollection* list, void(*function)(int*))
```

Then we can certainly multiply every element by 2 by calling for_each(list, &double) where double is some custom function. However, if we want to make the multiplication constant variable (e.g. depending on user input), then there is no way to achieve this, except for using global variables. This option is so terrible for so many reasons, including readability, multithreading (multiple threads cause race conditions on the global variable) and recursive calls (if, in the pointed-to-function, one calls another function that uses the global variable again, you get a very evil bug). More reasons why global variables are bad are easy to find online.

1.3 No generics or similar things

It is a very common requirement in programming to have "infrastructure functionality" that should do things with different kinds of objects, independent of what kind of object they are, or what functionality they support. The most common example are of course collections, like std::vector<T> in C++ or ArrayList<T> in Java. Usually, this kind of abstraction is implemented using generics/template/whatever they are called [3]. Note that this is not subtyping, as it does not use common features of the considered object, but another (less fine-grained) kind of abstraction. On the other hand, generics allow more control over the type of objects *returned* from functions (this closely related to the concept of variance).

[3]Strictly speaking, templates and generics are very different, but for the purpose of this discussion, we will treat them as equal.

The baseline is however that C has no good way to implement this kind of abstraction. For this discussion, assume for simplicity we want to build a list that contains objects of a type T. We have the following options

- Use a void pointer void∗. This is the most commonly used technique, as it is relatively simple. However, there is again the problem that whenever you really want to do something with an object you get out of the collection, you have to cast it to the concrete type. The compiler has no way to check whether the type is correct, so this is a dangerous source of errors and undefined behavior. Furthermore, a reader of the code will have to spend much more concentration on finding out what types certain objects have, which makes the code significantly harder to understand (in other words, there is no type documentation in the code).
- Use macros. This is a very clever solution that Michael Pusl told originally told me about. The obvious drawback is that it requires heavy use of macros. More about this in the next section.

Finally, there is a very subtle performance problem if one uses void pointers to simulate generics *with additional functionality* (sometimes also called compile-time polymorphism), i.e. assuming that the abstract type T has special properties (in Java, done by

```
<T extends MyInterface>
```

in C++ done by the way how templates work). Namely, if done in the C++ or Rust context, the compiler knows the concrete type of each object of type T when compiling, and so can produce much more efficient code (i.e. inline functions, no virtual function call cost, ...).

1.4 Macros

Many problems and shortcomings of C can be worked around by the use of macros. However, this is incurs quite serious problems (i.e. makes the code hard to read and error-prone), and thus should not be considered a solution, only a **hacky workaround for more fundamental shortcomings**. A non-exhaustive list of problems follows, for more search online [4]

- Macros are very hard to debug, and usually completely break intellisense.
- Macros can cause very subtle bugs, the most famous one being the "forgotten bracket"

```
#define TWO 1 + 1
...
printf("%i", TWO * TWO) // prints 3
```

[4]https://stackoverflow.com/questions/14041453/why-are-preprocessor-macros-evil-and-what-are-the-alternatives, https://scienceblogs.com/goodmath/2007/12/17/macros-why-theyre-evil, https://github.com/FeanorTheElf/crazy-experiments

However there are many other, subtle problems in connection with variable scopes and shadowing, naming conflicts and “call-by-name issues”, like the following:

```
#define MUL_ADD_ASSIGN(var, mul, add) \
var *= mul; var += add;
...
int x = 2;
MUL_ADD_ASSIGN(x, 2, x);
printf("%i", x); // prints 8, not 6
```

This example is trivial, but it is very easy to create very nasty bugs this way. The fundamental problem is that the internals of the macro are hidden, but not abstracted from (i.e. they cannot be ignored when calling the macro).

- Macros are hard to read. This is less trivial that it sounds, since we should not consider it a problem that the definition of the macro is somewhere else than the calling point (this is also the case for functions!). However, the problem is that macros hide the definition, but it still must be known by the caller to avoid problems (i.e. does not abstract from it).

  An acquaintance of myself had an absolutely terrible bug in a professional, huge C project, which was caused by hiding a reference-counting pointer behind a macro. In some places, the release()-function was not called as the programmer was not aware that the macro hid a shared pointer. This caused a very expensive and nasty bug.

- The compiler cannot check macro definitions, which makes errors occur at the calling position. Just a small point, but it can be very inconvenient in practice.

And now we have bashed C enough, let us go to C++.

2 C++

In some sense, C++ is almost the opposite of C. Where C is minimalistic and clear (and sometimes not very powerful), C++ is an incredibly powerful big mess of historical, interfering features. I really do not find any problem of C++ that is the result of not having any language feature, but instead, I can list tons of examples where different language features interfere to create strange, unexpected effects. To illustrate this, I just present some wildly chosen examples. This list is by **no means** exhaustive.

2.1 Virtual and destructors

Inheritance is very hard to use correctly in C++. First of all, there is private and public inheritance, there are virtual, pure virtual and non-virtual functions, and a lot of things one has to consider to avoid memory leaks and UB. The most classical example is that if you have a class that should be suitable for subclassing

```
class Base {
    ~Base() { ... } // should be virtual!
}
class Subclass : public Base {
    ~Subclass() { ... } // should be virtual!
}
```

both destructors should be virtual (this is much more important for Base, though). Otherwise, delete a; for Base∗ a filled with an object of Subclass will only delete the base class part in memory, causing a memory leak.

Of course, this is also true for any other method. Consider the following code

```
struct Base {
    void a() { std::cout << "Base.a()"; }
}
struct Subclass : public Base {
    virtual void a() { std::cout << "Subclass.a()"; }
}
...
Base* a = new Subclass();
a->a(); // prints "Base.a()"
```

As it is so typical for C++, this problem is fixed (or at least decreased) by adding more language features, namely the override keyword. However, IMO making the language more and more and more complicated to fix problems resulting from the language to be too complicated seems not to be a successful way.

2.2 Initializer lists and constructors

Again, an example is the best way to demonstrate this:

```
#include <vector>

struct Foo {
    Foo(std::initializer_list<int> list) {
        std::cout << "init␣list" << std::endl;
    }
    Foo(int a, int b) {
        std::cout << "ints" << std::endl;
    }
    Foo(std::vector<int> a) {
        std::cout << "vector" << std::endl;
    }
};

struct Bar {
    Bar(int a, int b) {
        std::cout << "ints" << std::endl;
    }
    Bar(std::vector<int> a) {
        std::cout << "vector" << std::endl;
    }
};

struct Foobar {
    Foobar(std::vector<int> a) {
        std::cout << "vector" << std::endl;
    }
};

int main() {
    Foo a(1, 2); // ints
    Foo b{ 1, 2 }; // init list
    Foo c{ {1, 2} }; // init list
    Bar i{ {1, 2} }; // vector
    Foo d({ 1, 2 }); // init list
    Foo e = { 1, 2 }; // init list
    Bar f(1, 2); // ints
    Bar g{ 1, 2 }; // ints
    Bar h = { 1, 2 }; // ints
    Foobar j({ 1, 2 }); // vector
    // Bar k({1, 2}); error: "ambigious call of overloaded method"
    return 0;
}
```

I will not even try to explain this. Honestly, who should understand this? What is the use of a standard, if it standardizes chaos? Why do you need four (or more, depending on your way of counting) subtly different ways of constructing an object?

2.3 Templates

Templates in C++ are incredibly powerful, but because only the instantiations are type-checked, and because of crazy things like SFINAE, share many problems with macros. Apart from that, generating useful error message usually is too much for any compiler. I have an unhealthy fascination with heavy use of templates, and I have encountered situations where just forgetting a small detail (say brackets) in one place, yields over a hundred error messages in completely different locations. Furthermore, how clear can a language feature be that sometimes requires code as

```
template< class T >
void f( T &x ) {
    x->template variable < T::constant < 3 >;
}
```

Things like that are also heavily used in the standard library, which makes very simple (non-templated) code sometimes hard to use. Honestly, the push_back-function of std::vector is already declared multiple times, one of the declarations being

```
template<typename _Up = _Tp>
    typename __gnu_cxx::__enable_if<
        !std::__are_same<_Up, bool>::__value,
        void
    >::__type
    push_back(_Tp&& __x)
{ emplace_back(std::move(__x)); }
```

Well, this might be extremely clever and fast, but it is certainly not easy to understand. And that is one of the most simple functions I can think of...

2.4 References

References are a very strange concept, slightly like pointers, slightly less powerful, and just so unintuitive. It is definitely no wonder that no other language has copied this notion from C++. References behave slightly like pointers

```
int a = 2;
int& b = a;
b = 3;
std::cout << a; // prints 3
```

but you cannot do arithmetic or assign them. Also, dereferencing happens automatically. In fact, the assignment operator = can mean two different things for references.

```
int a = 2;
int b = 3;
int& c = a; // c is a reference to a
c = b; // assigns the value of b to a
// we cannot store a reference to a in c
// every further call of = will assign to a, not to c
```

There are so many more strange things in the C++ language, for more info read e.g. the book "Effective modern C++" by Scott Meyers.

3 Rust

Rust has been designed in a much more careful way, to avoid any of the above problems. The main motto is “zero-cost abstractions”, i.e. allow the user to build nice, powerful and easy-to-use abstractions, without sacrificing performance. This is reached (almost) completely [5]. More concretely, Rust completely eliminates any kind of undefined behavior, by having a very strong compile-time code analysis. In very rare, tricky situations, these compile-time constraints can be released by using the "unsafe"-keyword, which then allows you to do things that might be UB. However, I have written many thousand lines of rust already, and only used unsafe twice so far. For less technical applications, it is even completely avoidable.

This careful design allows many things you can do in C++ (like generics, compile-time polymorphism, interfaces, subtyping, ...) without any of the unexpected situations (these are sometimes called "gotchas", and googling "Rust gotchas" yielded only one very sparse result [6], so I guess it is not the case that I just didn't find them...). Furthermore, Rust has some very interesting, powerful features derived from functional programming (like Haskell), in particular traits and blanket implementations. Another advantage is that Rust comes with a very convenient standard library. that allows many basic tasks to be done very elegantly. For example, summing the squares of numbers in a vector is as easy as

```
let a: Vec<i64> = ...;
let sum: i64 = a.iter().map(|x| x * x).sum();
```

which has a much nicer and clearer syntax than the C++ std::accumulate variant.

In fact, IMO there is only one problem with Rust.

3.1 Not very mature

Rust is a very new language, and some very useful language mechanics are currently only available when using nightly builds of the compiler, or only available in feature-reduced form. That is why currently all my projects can only be compiled with nightly Rust, and I even sometimes get warnings like

```
warning: the feature `specialization` is incomplete and may
not be safe to use and/or cause compiler crashes
 --> src\lib.rs:2:12
  |
2 | #![feature(specialization)]
  |            ^^^^^^^^^^^^^^
  |
  = note: see issue #31844 <https://github.com/rust-lang/rust/issues/31844>
  for more information
  = help: consider using `min_specialization` instead,
  which is more stable and complete
```

[5]There is a little bit of cheating involved with bound check in arrays, ... and delete checking, but apart from that, I do not know any situation where Rust is slower than a directly comparable concept in C++.

[6]https://www.reddit.com/r/rust/comments/feyfxh/what_are_the_gotchas_in_rust/

However, there are many professional and open source projects out there that do important stuff, and run completely on stable Rust. In the end, if you are ok with a slightly more conventional programming style (as a C-programmer certainly is), you will not have to deal with this problem. Another, connected point is that the ecosystem of Rust is not yet very big, and there are no nice IDE's or more niche libraries available. However, the big Rust Hype at the moment and its drastic increase in popularity will probably fix that in the close future.