



# **WHAT EVERY C++ PROGRAMMER SHOULD KNOW ABOUT MODERN COMPILERS**

**SŁAWOMIR ZBOROWSKI**  
ACCU 2016, BRISTOL, UK

Welcome to my short presentation with a long title :)

---

**SŁAWEK ZBOROWSKI**

**WROCŁAW, POLAND**

**NOKIA**



I'm currently working for Nokia where we're moving part of software from base station into cloud – hence the name of our team – Miotacze Piorunów.

I think that direct translation would be “lighting throwers”, but I'm not sure.

We even have a flag! :)

# OUTLINE

## overview

architecture, inputs, targets

## standard, compilers and reality

undefined behavior

## optimizations

outsmarting compiler

## ecosystem

tooling, further optimizations

# OUTLINE

## overview

architecture, inputs, targets

## standard, compilers and reality

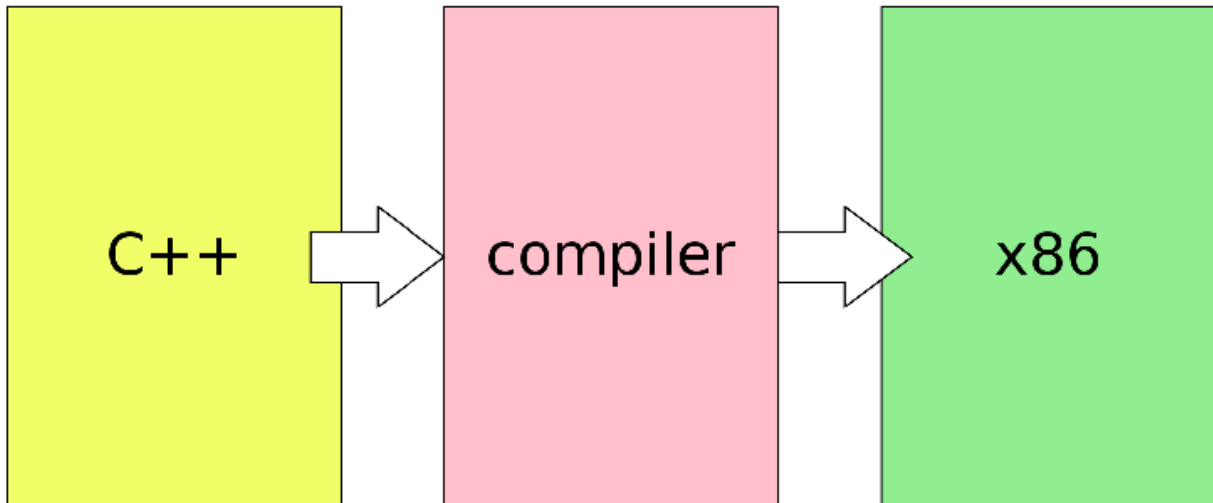
undefined behavior

## optimizations

outsmarting compiler

## ecosystem

tooling, further optimizations

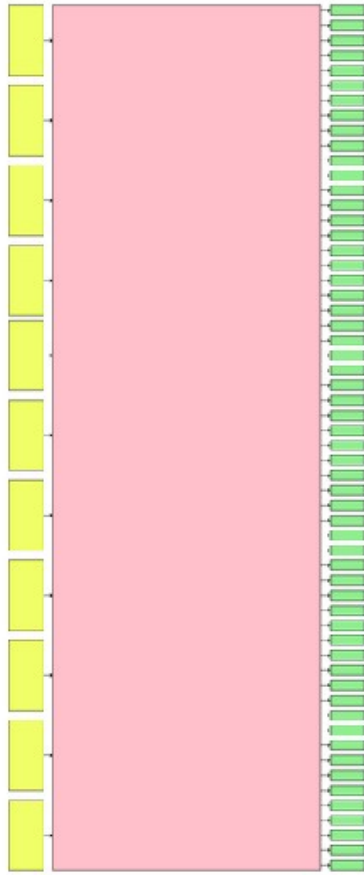


How do we perceive compilers?

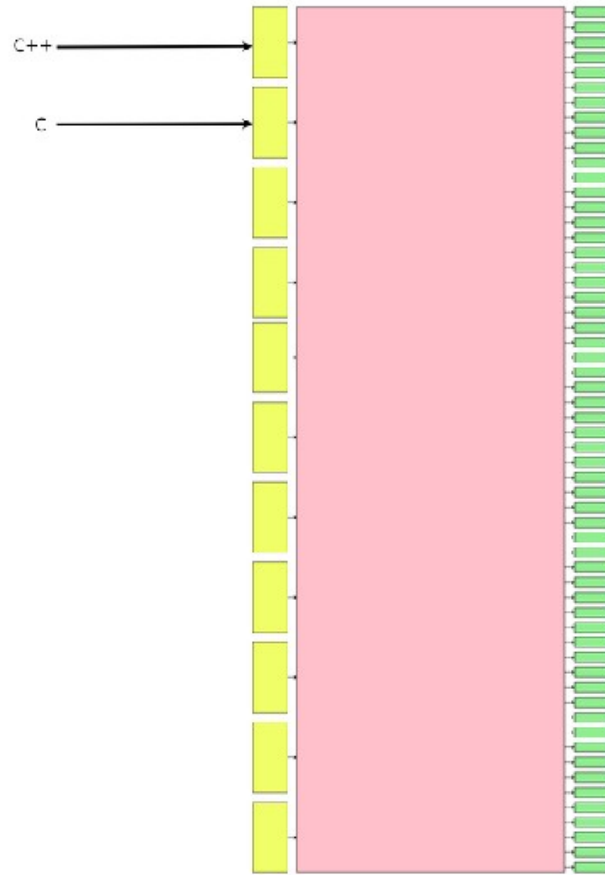
Most of the time when we think about them we have similar picture in our minds.

There's a back box, which is being feed with C++ in order to produce some binary - presumably for x86 machines.

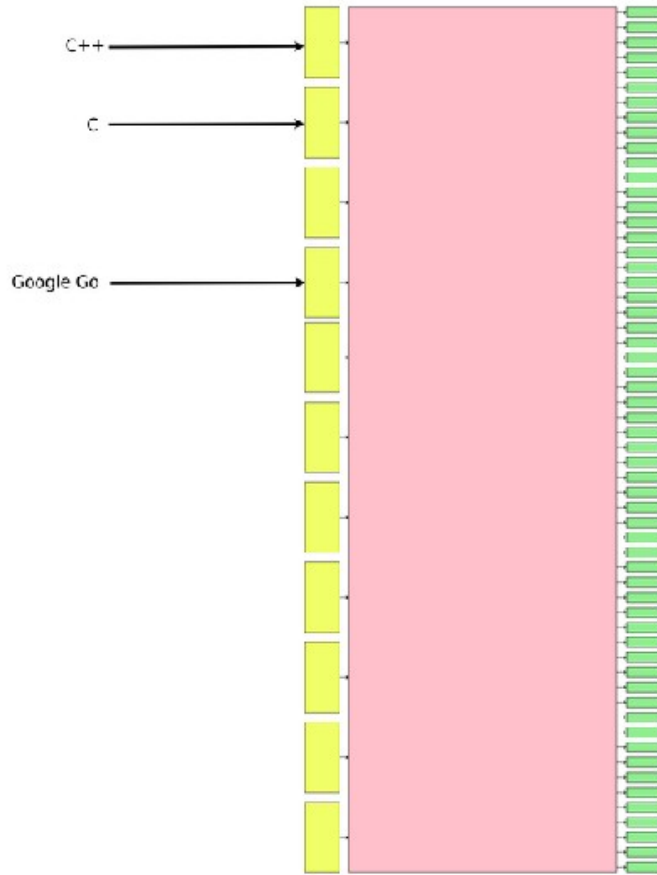
Besides the fact the picture is oversimplified, there's nothing wrong with it. However, if we make a little step towards reality...



... we get something like this.

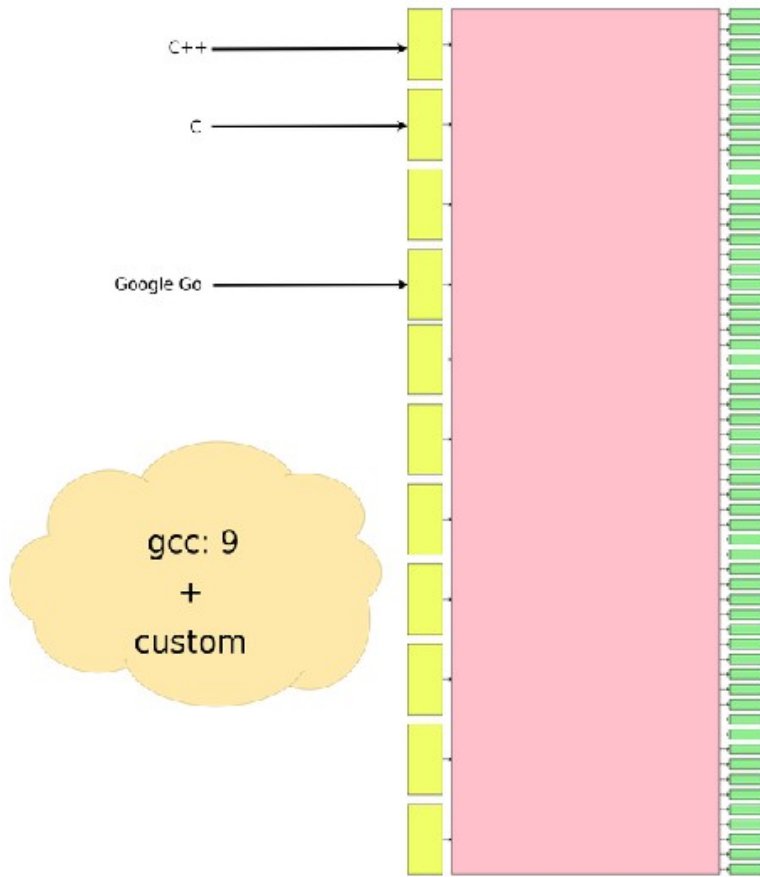


Not only C and C++ are supported by C++ compilers.



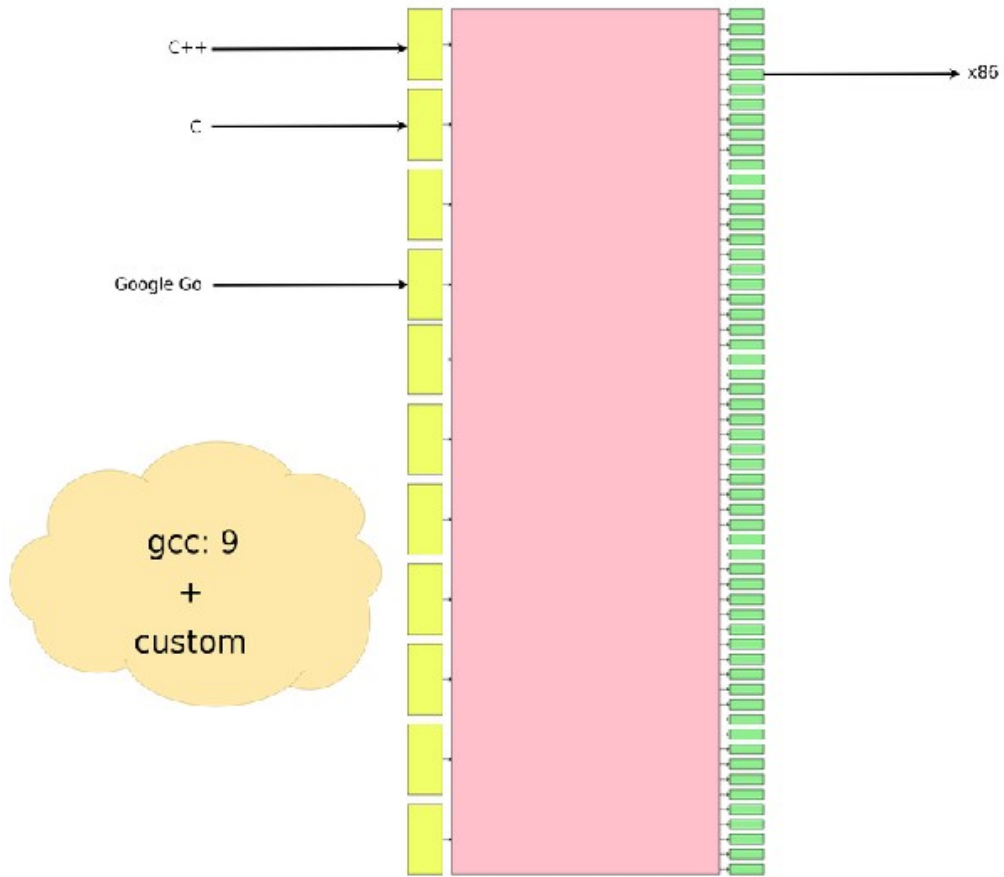
Other languages like Google Go or Objective-C are also supported.



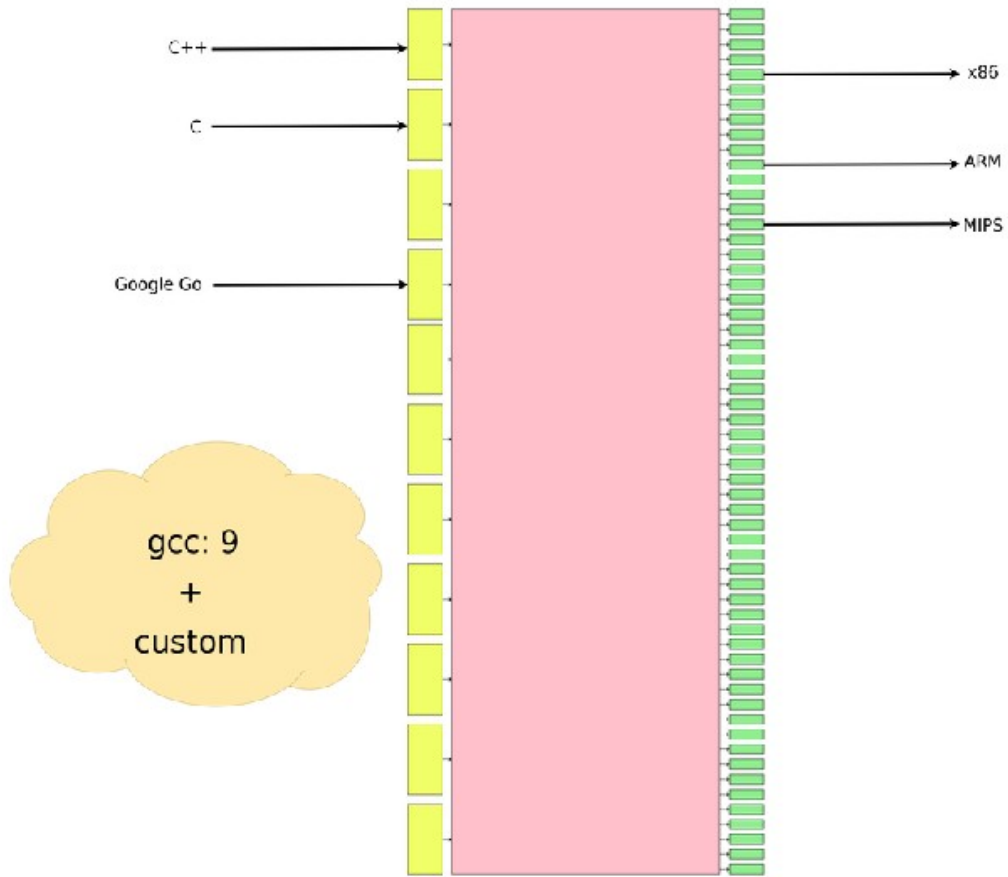


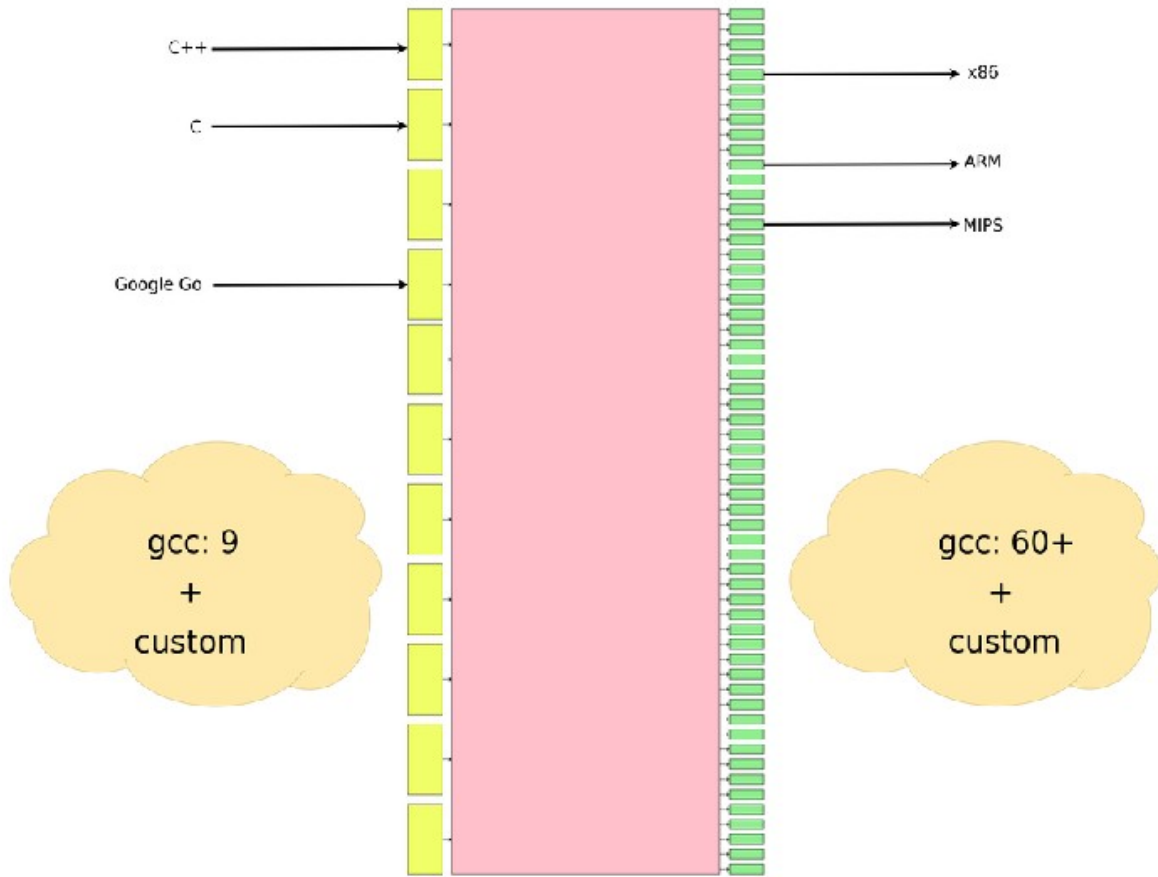
Afaik GCC support 9 languages out of the box.

And of course you can provide an implementation for your own (custom) language.



Same thing can be said about possible targets..





It's kind of insane, but GCC supports more than 60 targets! That's just amazing.



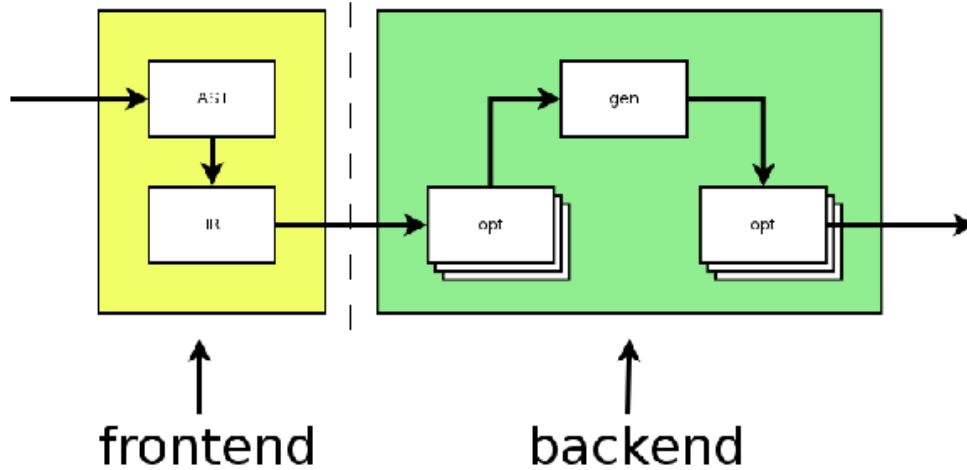
<http://leaningtech.com/cheerp/>

Cheerp is an interesting example of how flexible compilers can be.

It's modified clang and it allows you to write isomorphic web apps in C++!

The funny thing is that instead of producing a binary it produces... JavaScript.

# ARCHITECTURE OVERVIEW



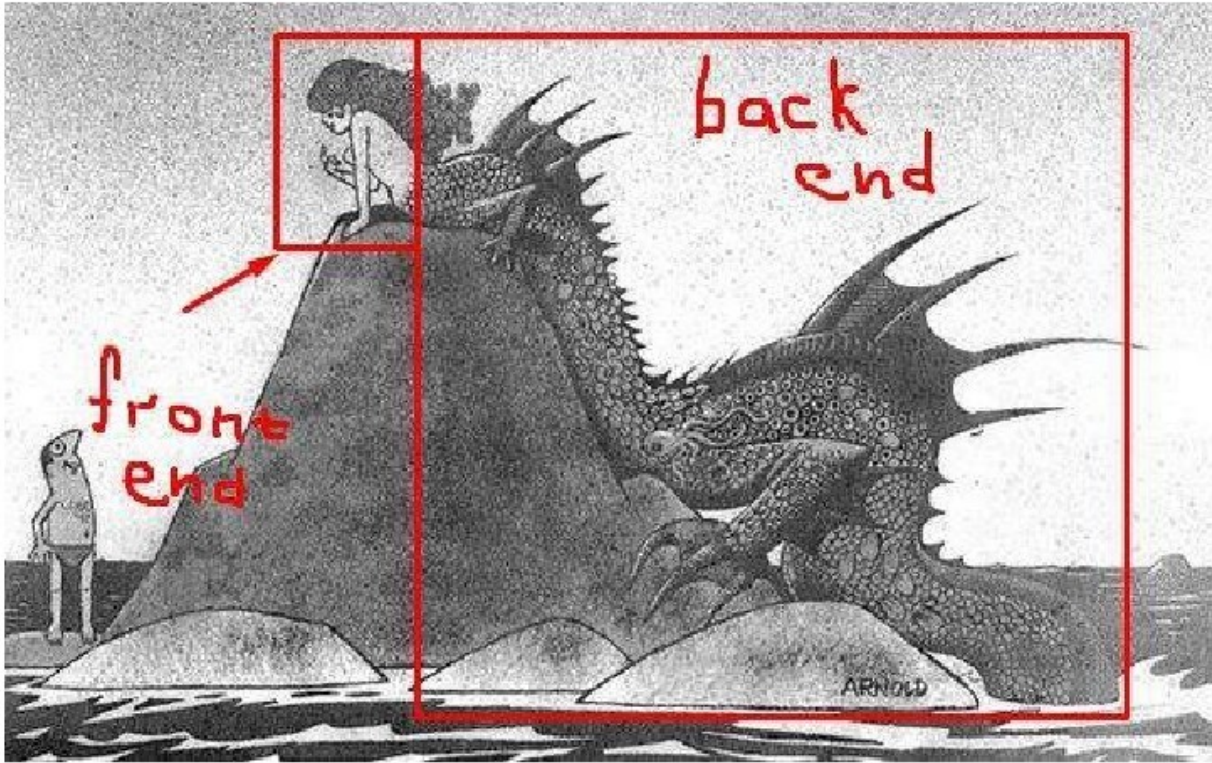
How it's even possible? Mainly because modular architecture.

The picture is of course oversimplified, as almost everything in the presentation.

There's clear split between the front-end and the back-end.

Front-end responsibility is to convert input language into “an internal representation”.

The IR is an interface between the two guys.








<https://devhumor.com/content/uploads//images/October2015/Front-end-x-Back-end.jpg>

Most of the things happen in the backend – optimizations, code generation, more optimizations etc. Frontend is relatively simple, when compared to the backend.

Lots of problems (NP-hard, NP-complete) are solved by compilers. They're huge and complex.

How do we control their behavior, then?

# FLAGS

|   |                                       |
|---|---------------------------------------|
|  Newbie  | -o                                    |
|  Junior  | -c, -Wl,-shared, -Wl,-static, ...     |
|  Average | -Wall, -Wextra, -Werror, -O2, ...     |
|  Expert  | -finline-limit, -ftemplate-depth, ... |
|  Ninja   | -B, -nostdlib, ...                    |

With command line switches!

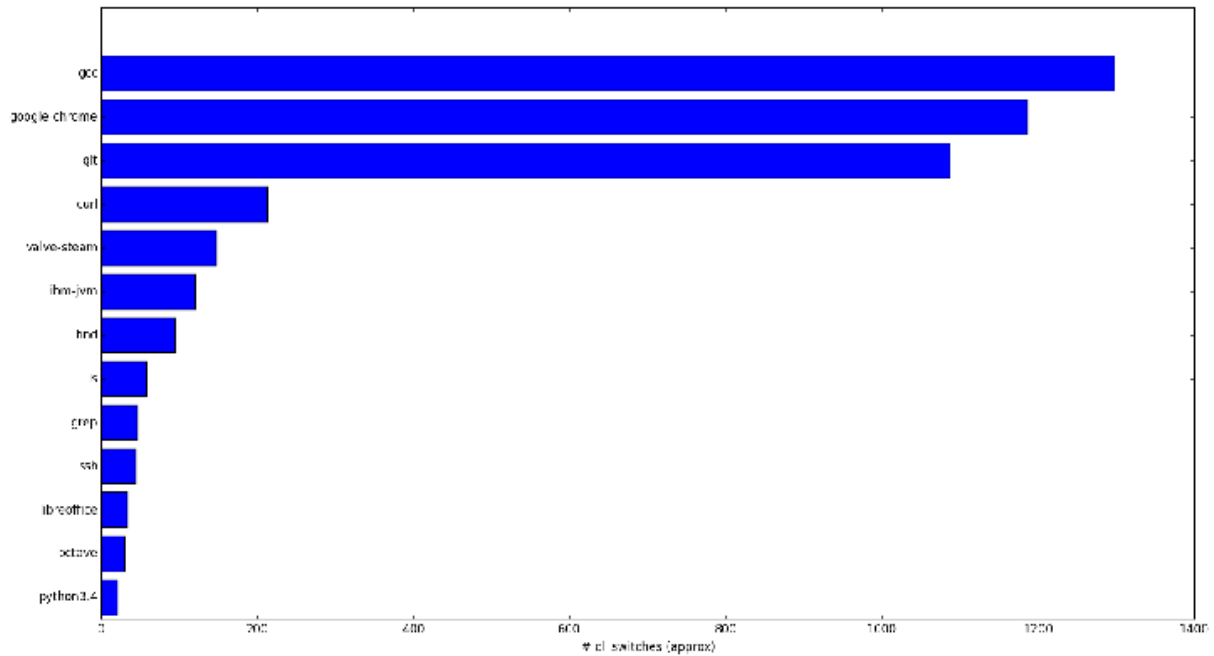
When we're learning C++ we are simultaneously learning how to use our favorite compiler. We start with knowing little things like -o.

Then we discover how to do much more: static libs, shared libs.

Then how to control verbosity, optimizations and so on.

Eventually we become ninjas. Detaching STD lib or preprocessor? Why not :)





I have made small research to see how customizable GCC is compared to other common tools.

There's huge gap between GCC and other Linux tools.

The only two players that are close are Google Chrome (which is like a virtual machine btw) and git. I wasn't surprised about git :)

I know it's not a perfect measure, but it definitely tells us something – compilers are highly customizable.

# OUTLINE

overview

architecture, inputs, targets

**standard, compilers and reality**

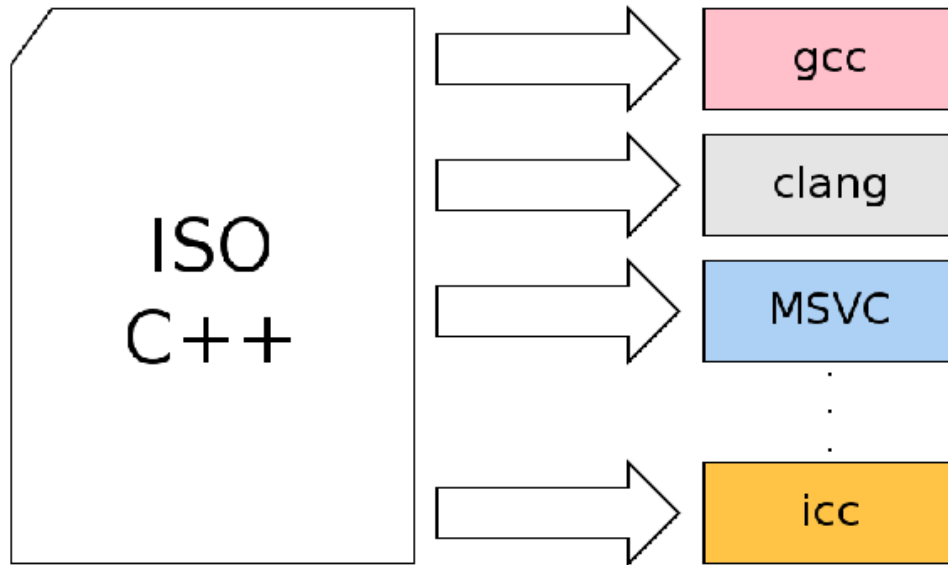
undefined behavior

optimizations

outsmarting compiler

ecosystem

tooling, further optimizations



Similarly to previous image, the relation between C++ standard and compilers seems to be simple.

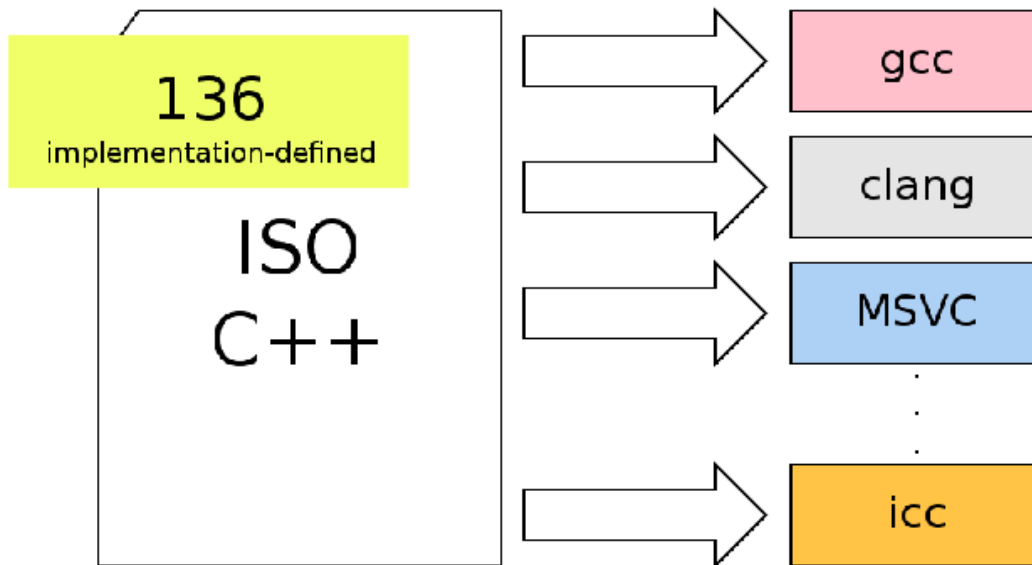
There's one international specification which is being implemented by multiple vendors.

Hence we have different compilers.

Easy, isn't it?



[http://community.fansshare.com/pic108/w/traducianism/1200/1752\\_caution\\_this\\_is\\_sparta\\_jex.jpg](http://community.fansshare.com/pic108/w/traducianism/1200/1752_caution_this_is_sparta_jex.jpg)

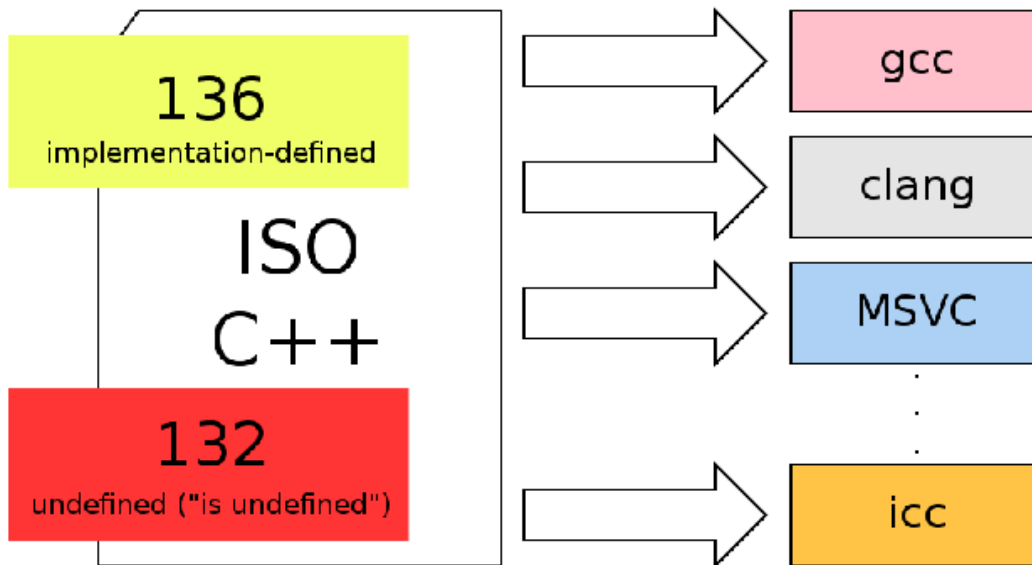


If you look for implementation-defined stuff in the standard, you're going to find more than 100 items.

This leaves room for compilers to differ.

However, there's one more evil thing out there.

Yes, you're right.



C++ standard (C++14 working draft to be precise) contains over 100 occurrences of “is undefined” phrase. Crazy.



[http://facstaff.cbu.edu/seisen/CadFa0913\\_files/image032.jpg](http://facstaff.cbu.edu/seisen/CadFa0913_files/image032.jpg)

Even if you're seasoned programmer you are likely to introduce bugs to applications. The worst thing is that you're not gonna notice that until something breaks in production.

Programming in C++ used to be like going into dark cave without a flashlight, knowing that you may be bitten by a snake.

Fortunately, the situation is getting better.

# UNDEFINED BEHAVIOR SANITIZER

Modern compilers are shipped with tools that help to find different kind of problems.

One of these tools is undefined behavior sanitizer.

The idea behind UBSan is that the compiler instruments the code so that in runtime it can detect Ubs.

This isn't without cost. It slows down execution ~ 2-3x. It is definitely good to give it a try in tests, though.

Programmer, for different kinds of UB can choose the behavior – trap instruction, print & continue, or print & exit.





<https://www.youtube.com/watch?v=Hf-znKiVwyk>

With UBSan there's simply more control.

# OUTLINE

## overview

architecture, inputs, targets

## standard, compilers and reality

undefined behavior

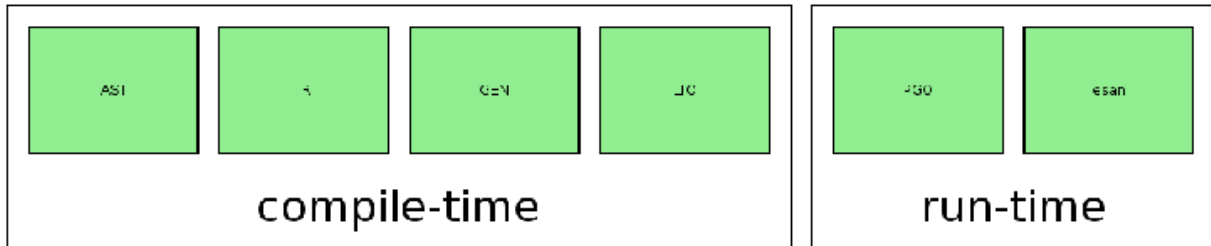
## optimizations

outsmarting compiler

## ecosystem

tooling, further optimizations

# OPTIMIZATIONS



Optimizations are everywhere.

They do start in the frontend (AST), then they are apparent at IR phase, GEN phase. Even after creating object files we have optimizations.

Special information is encoded within object files so the link time optimizer can kick in and optimize at the top level.

There are also run-time optimizations, like profile guided optimization or recently announced efficiency sanitizer which hopefully will join C++ world.

# OUTSMARTING COMPILER

```
1 auto tmp = a;  
2 a = b;  
3 b = tmp;
```

```
1 a ^= b;  
2 b ^= a;  
3 a ^= b;
```

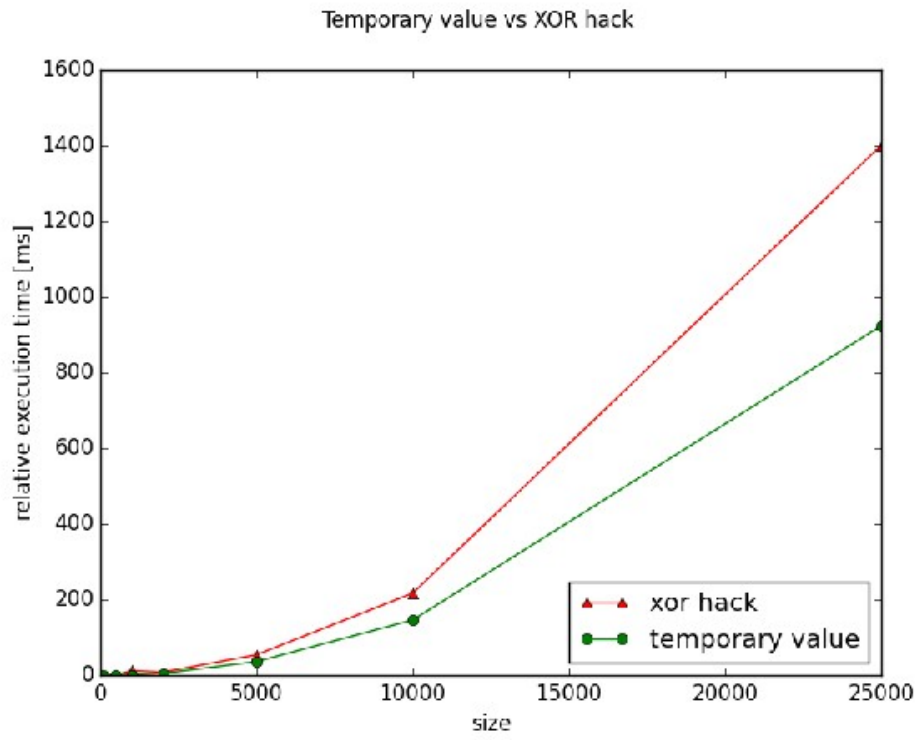


[https://www.youtube.com/watch?v=bS5P\\_LAqiVg](https://www.youtube.com/watch?v=bS5P_LAqiVg)

Other important thing about optimizing is that we tend to underestimate the compiler. We also try to outsmart compiler sometimes.

E.g. instead of traditional approach to swapping one can use three XOR assignments.

Looks hacky, but is it faster indeed?



Apparently not.

# COMPILER'S KNOWLEDGE

## CPU CACHES

## CPU EXTENSIONS

## CPU PERFORMANCE BUGS

## INSTRUCTION SIZES (IN BINARY)

...

A lot of knowledge has been put into compilers by smart people.

They know about CPU caches and extensions (like e.g. SIMD).

Also, not all CPUs are perfect. There may be a performance in some revision of some CPU. It's likely that the compiler will already know about it and will avoid this instruction whilst generating code.

What about us? Do we know? Not necessarily.

Compilers look at the code generations from multiple perspectives. It's simply hard to outsmart them.

And even if you can, then why wouldn't you improve them in a first place anyway?

# OUTLINE

## overview

architecture, inputs, targets

## standard, compilers and reality

undefined behavior

## optimizations

outsmarting compiler

## ecosystem

tooling, further optimizations

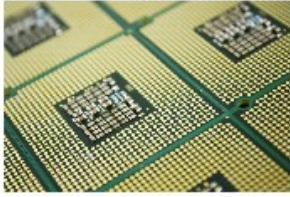
# INSTRUMENTALIZATION



ADDRESS SANITIZER

MEMORY SANITIZER

CONTROL FLOW SANITIZER



THREAD SANITIZER

EFFICIENCY SANITIZER

...

[https://bournetocode.com/projects/GCSE\\_Computing\\_Fundamentals/pages/img/RAM.jpg](https://bournetocode.com/projects/GCSE_Computing_Fundamentals/pages/img/RAM.jpg)

<http://create.pro/blog/wp-content/uploads/2014/12/Mac-Pro-Multi-Core-Processor-3-e1418658689878.jpg>

I've already mentioned UBSan.

We have more than that.

AddressSanitizer and MemorySanitizer allows us to spot errors related to memory (e.g. reading uninitialized memory or ordinary memory leak).

Our apps can be subject of cracking. Some bad people can try to hijack the flow so the program does unintended things. For protection we can use ControlFlowSanitizer. It will look for situations that could possibly be used by hacker to take control over execution (e.g. return-oriented programming).

If our application is multi-threaded we can use ThreadSanitizer to find deadlocks, data races, etc.

And finally, recently announced EfficiencySanitizer will look how the performance can be improved. E.g. it will look for write-after-write, possible reordering of fields in struct to improve cache friendliness etc...

All of the sanitizers are likely to slow down the program.





# CLANG TOOLS

CLANG-FORMAT

CLANG-TIDY

CLANG-COMPLETE

CLANG-ANALYZER

...

[https://upload.wikimedia.org/wikipedia/en/4/4c/LLVM\\_Logo.svg](https://upload.wikimedia.org/wikipedia/en/4/4c/LLVM_Logo.svg)

Modern compilers are also shipped with auxiliary tools, like presented on the slide.

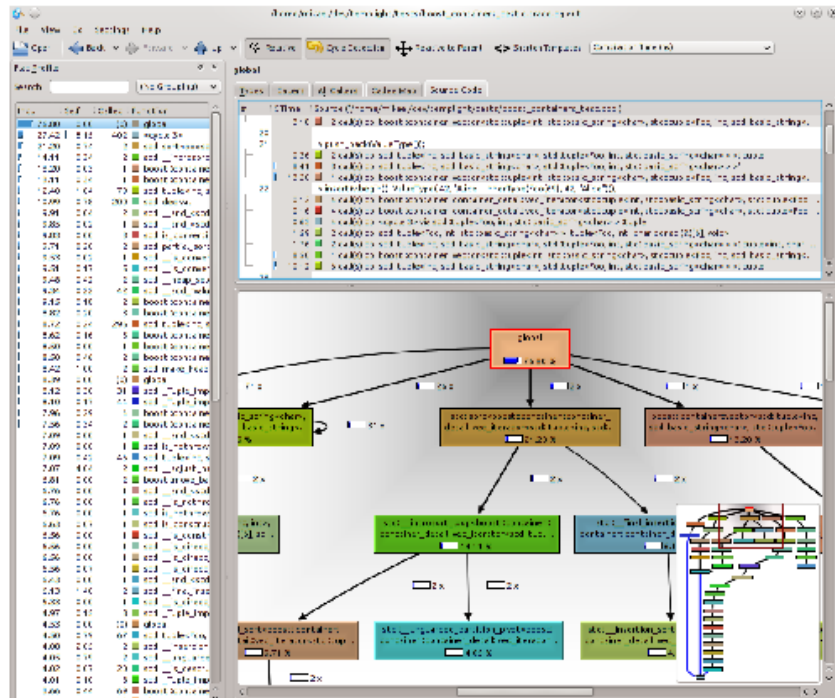
- Reformatting the code → clang-format
- C++ linter → clang-tidy
- Suggestions in IDE → clang-complete
- Static analysis tool → clang-analyzer

All of these tools have common part – they're using compiler as a library.

No C++ parsing is done by these tools (that was always tricky part). All of the information is obtained from the compiler so 100% accuracy is achieved.

The list is not complete! Go read your compiler's documentation :)

# TEMPLIGHT / TEMPLATOR



<https://github.com/mikael-s-person/templight-tools>

It's both funny and sad that we've been having runtime debuggers for years without single compile-time debugger.

Now we have two.

They can be used to see how templates are instantiated and memoized.

Can be quite useful in case we want to e.g. decrease compilation time or see with 100% accuracy how templates are instantiated in our project.

# SYNTH

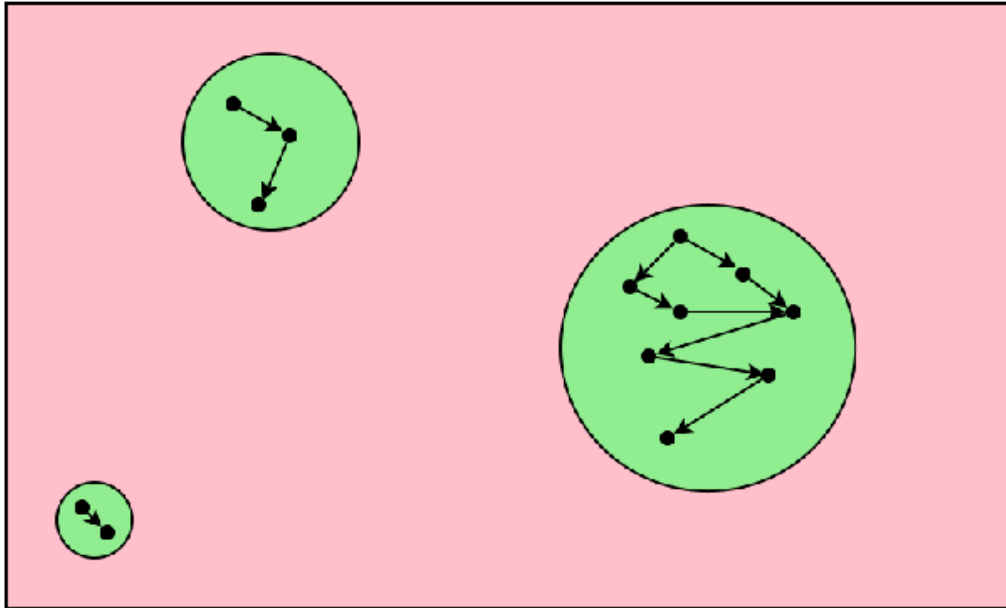
```
static bool processCompileCommand(  
    const CompileCommand& cmd,  
    std::vector<char const*> extraArgs,  
    bool pct,  
    ThreadSharedState& state)  
{  
    const file(const CompileCommand& getFilename(cmd));  
    if (!file.empty() && !state.multiThreadProcessor.isFileIncluded(file.get()))  
        return false;  
  
    std::vector<const char*> clArgsHandles = getClArgs(cmd);  
    std::vector<char const*> clArgs;  
    clArgs.reserve(clArgsHandles.size() + extraArgs.size());  
    for (const char* s : clArgsHandles)  
        clArgs.push_back(s.get());  
    clArgs.insert(clArgs.end(), extraArgs.begin(), extraArgs.end());  
    const dirStr = const CompileCommand& getDirectory(cmd);  
    if (!dirStr.empty())  
        state.workingDirUpdated,  
        state.workingDirChangedOrFree,  
        state.workingDirMut);  
  
    bool dirChanged = false;  
    if (!dirStr.empty()) {  
        fs::path dir = std::move(dirStr).get();  
        bool dirOk;  
        std::unique_lock<std::mutex> lock(state.workingDirMut);  
        state.workingDirChangedOrFree.wait(lock, [&]() {  
            if (state.cancel)  
                return true;  
            return true;  
        });  
    }  
}
```

Other example of how compiler can be used as libraries is Synth tool which was announced few weeks ago.

Purpose of the tool is to convert C++ code into hyper-linked colorful HTML. Hyperlinks of standard containers, classes, etc. redirect to C++ reference page.

# STOKE

Stochastic optimizer (x86\_64)



And finally a tool that isn't actually strictly related to compilers, but it lives near by.

STOKE is a tool that can possibly improve performance of single functions (10, 11 instructions). Functions' codes (little programs in fact) are treated as if they were single points in multi-dimensional space.

Changing instruction order, adding instructions, removing them, or changing order of arguments – it's all a move in such a space.

Compilers start in some area where instruction set composes a valid program. Then they optimize the code within the area.

It's likely than they won't be able to jump to other (distant) areas where the code is still valid and has better performance.

And that's the niche STOKE is focused on... maybe it's not mature now but who knows – maybe it's a future of code optimization?

**THANKS!!**