A Very Quick View Into a Compiler

Arvid Gerstmann
@ArvidGerstmann

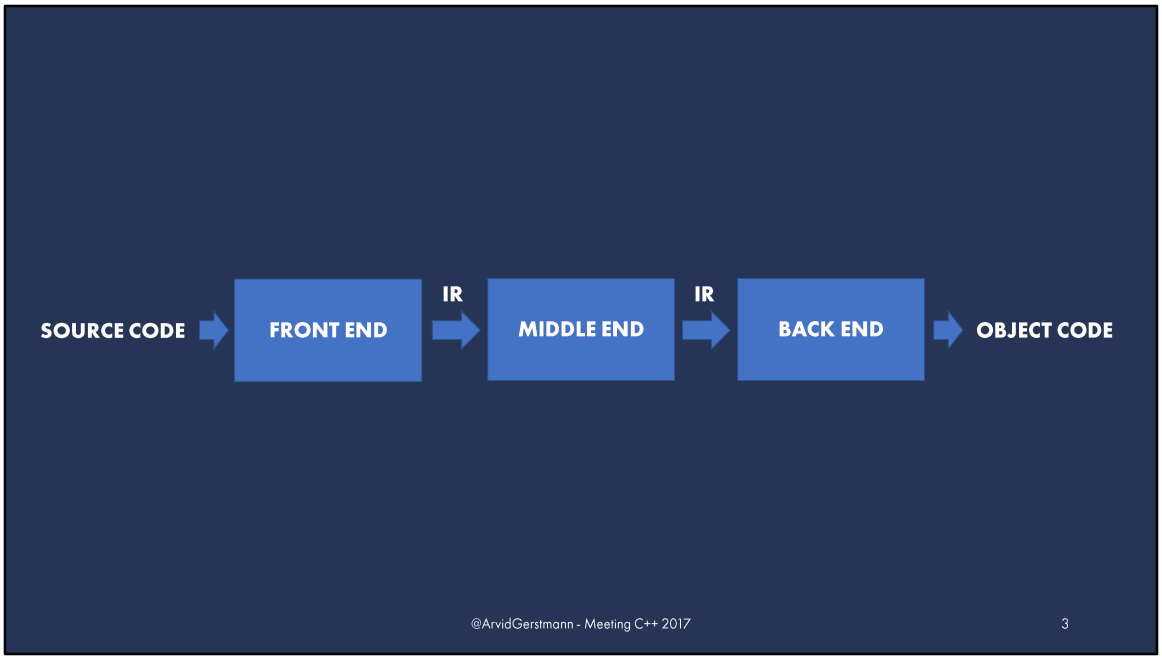
Hey, my name is Arvid Gerstmann. And this is „A Very Quick View Into a Compiler“. Let‘s get started!

What is a Compiler?

What is a compiler? A compiler is a program transforming code written in one language, the source language, into another language, the target language.

In terms of C++, we primarily think of compilers as programs translating our human-readable C++ source code, into machine-readable object code.

Famous compilers include gcc, clang and Microsoft Visual C++.



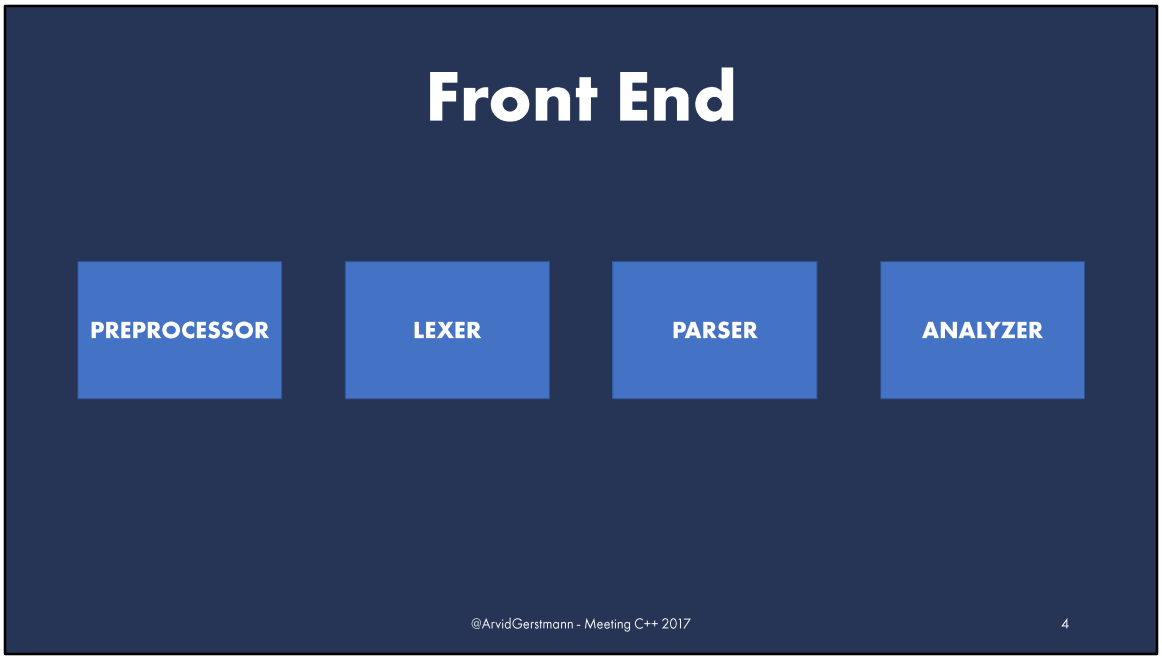
A compiler is usually represented as consisting of multiple, separate, stages. (*next slide*)

On the left, we have our source code. It is passed to the compilers „front end“. (*next slide*) The job of the front end is to convert the human-readable source code into a so called „Intermediate Representation“, from now on referred to as the „IR“.

The „IR“ is than passed to the „middle end“, (*next slide*) commonly called the „optimizer“. The optimizers job is to analyze and transform the literal source code into a more „optimized“ version of itself. It is where most of the compilers time is spent.

Last, but not least, comes the „back end“. (*next slide*) Converting the optimized IR received from the last stage into machine-dependent object-code. (*next slide*)

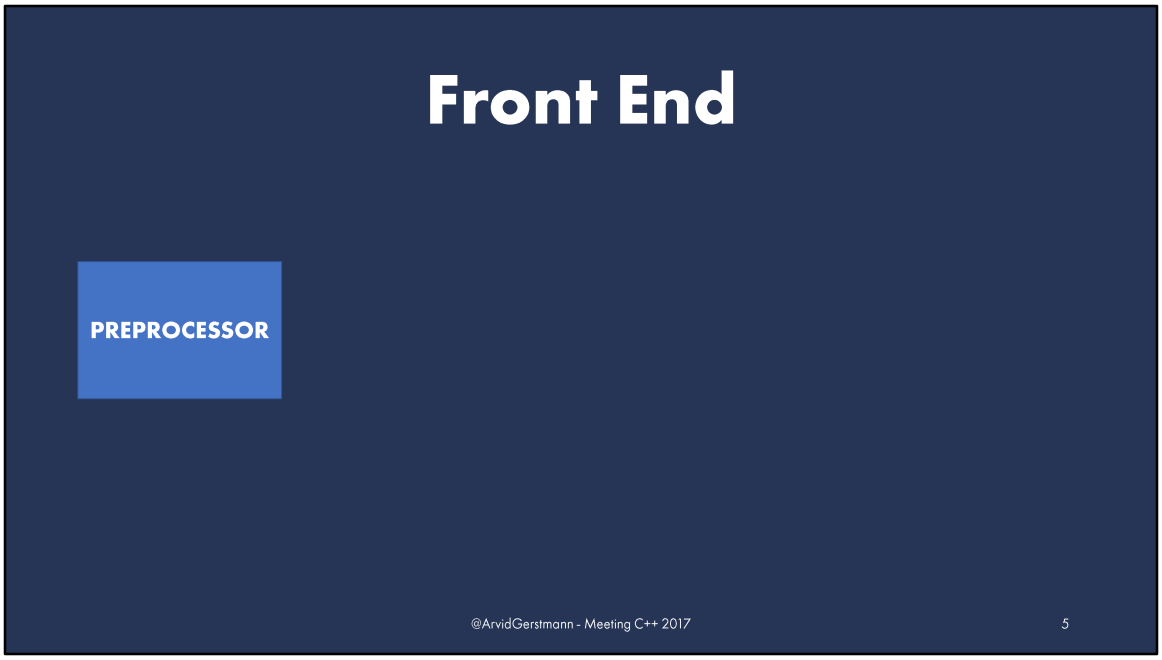
This object-code can than later be combined into an executable.



Let‘s start with the first stage, the **„front end“.** The front end is usually split into four sub-stages: (*next slide*)

- Preprocessor
- Lexer
- Parser
- And analyzer

Let‘s start with:



The **preprocessor**. The preprocessors job is to handle source file inclusion, macro replacement and conditional inclusion.

Preprocessor

```
#include "file.h"
#define FOO foo
#define BAR bar
#define BAZ 1
FOO BAR
#if BAZ
BAZ IS TRUE
#else
BAZ IS FALSE
#endif
```

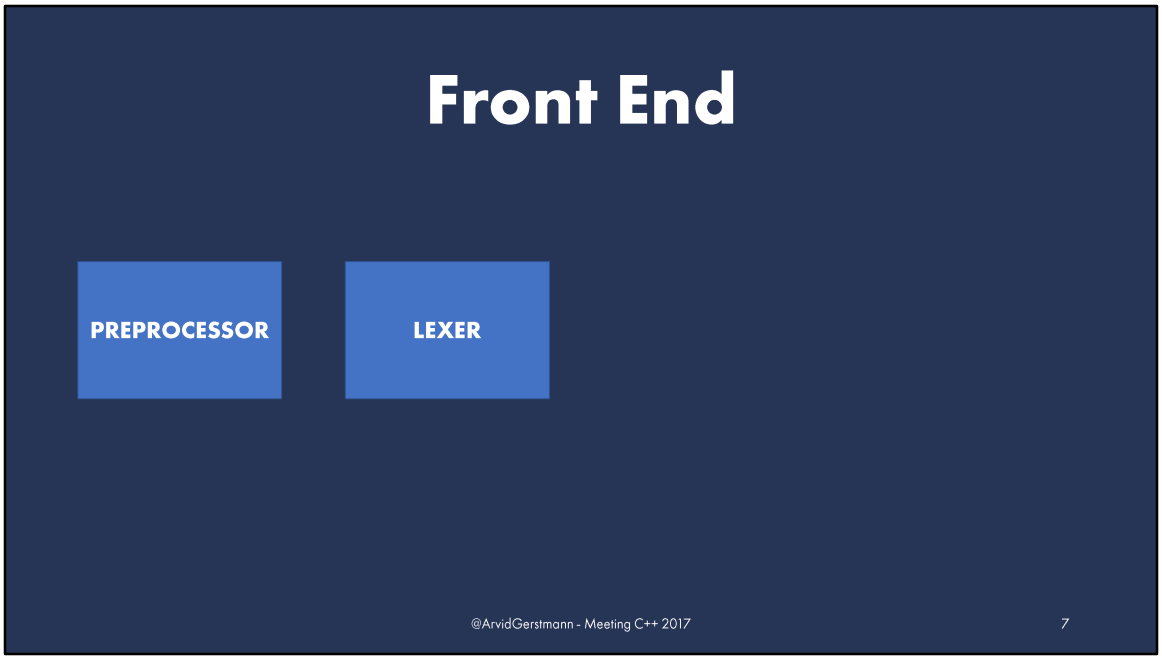
```
/* included file.h */
#define FOO foo
#define BAR bar
#define BAZ 1
foo bar
#if BAZ
BAZ IS TRUE
#else
BAZ IS FALSE
#endif
```


He is going to transform our input on the left, to our fully preprocessed output on the right, by going through a few well defined steps.
As we can see we can see. (*go through steps*)
The preprocessor is a left-over from C, and is a very C and C++ specific step in a compiler. It‘s unlikely to be found in other languages.

In this example, we will go through the steps the preprocessor would take to transform the source code on the left, into it‘s preprocessed form on the right.

- We start doing that by replacing all „#include“ (pound-include) lines with the contents of their respective files. (*next slide*)
- Next, we register all macros described by „#define“ (pound-define), and remove their lines. (*next slide*)
- Following that, all occurences of those just defined macros will now be replaced by it‘s replacement text. (*next slide*)
- Last, but not least, we evaluate all „#if“ statements. (*next slide*)

Leaving us with the fully preprocessed file.



Next is the **lexer.** The lexer is splitting our source code into „tokens“: snippets of text categorized into different, pre-defined, categories, like „identifier“, „literal“ or „punctuator“.

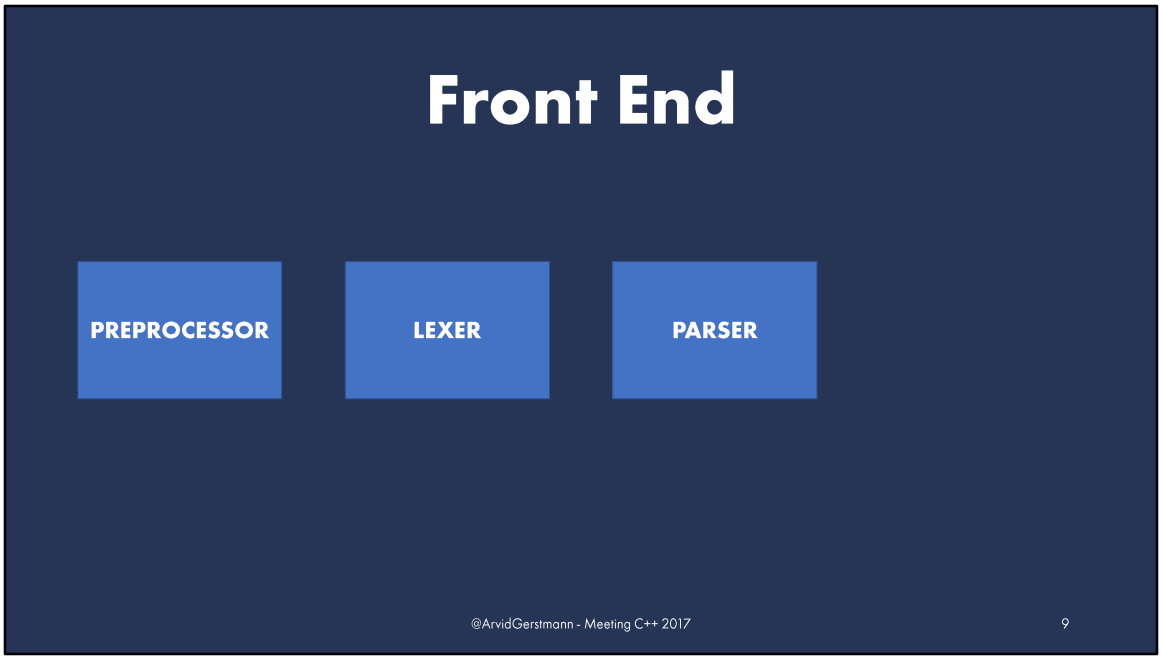
Lexer

```
int add()
{
    return 5 + 10;
}
```

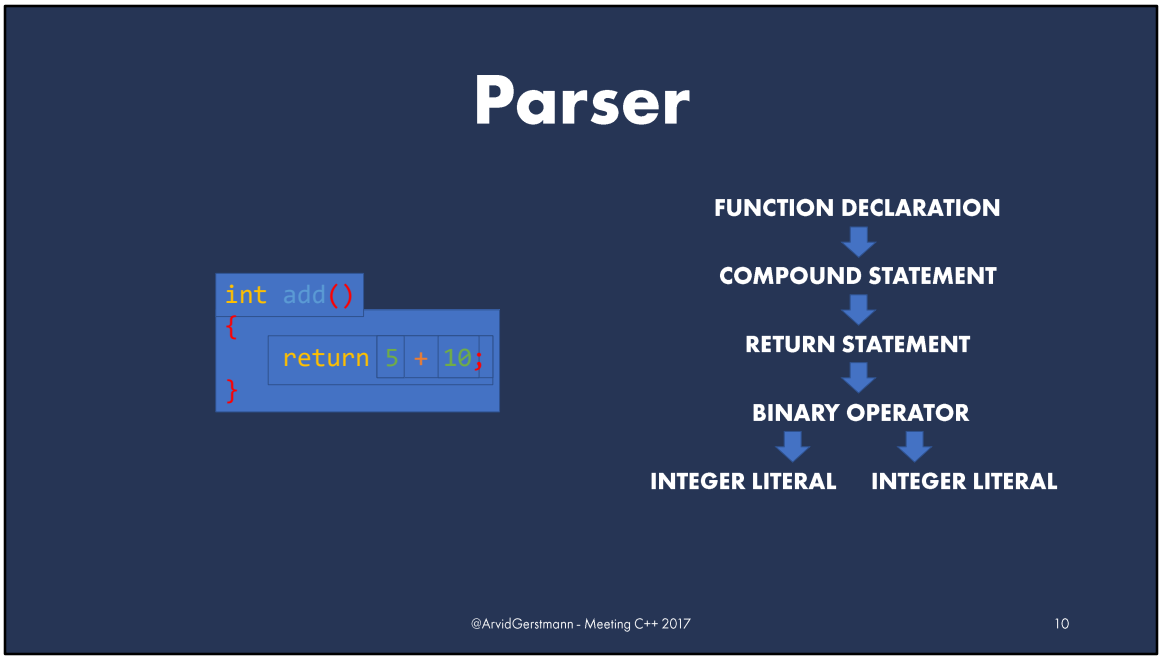
KEYWORD
IDENTIFIER
PUNCTUATOR
LITERAL
OPERATOR

On the left we see a simple C++ function adding to integers together.
Let‘s quickly go through and see how a lexer would split them:

- First, we have the „keywords“: „int“ and „return“. Here represented by the color yellow. (*next slide*)
- Next we have an „identifier“: „add“, seen in blue. (*next slide*)
- These are separated either by whitespace, or by so-called „punctuators“: parantheses, braces, semi-colons, et cetera. As seen here in red. (*next slide*)
- Numbers in our source code are called „literals“. In our case, the „5“ and „10“, shown in green. (*next slide*)
- The „+“ sign in between is called an „operator“, represented in orange.



As the second to last element in our front end, we have the **parser**, turning our just produced „tokens“ into a representation, easily consumable by the steps followed. Nowadays, this is usually an AST, an abstract syntax tree.

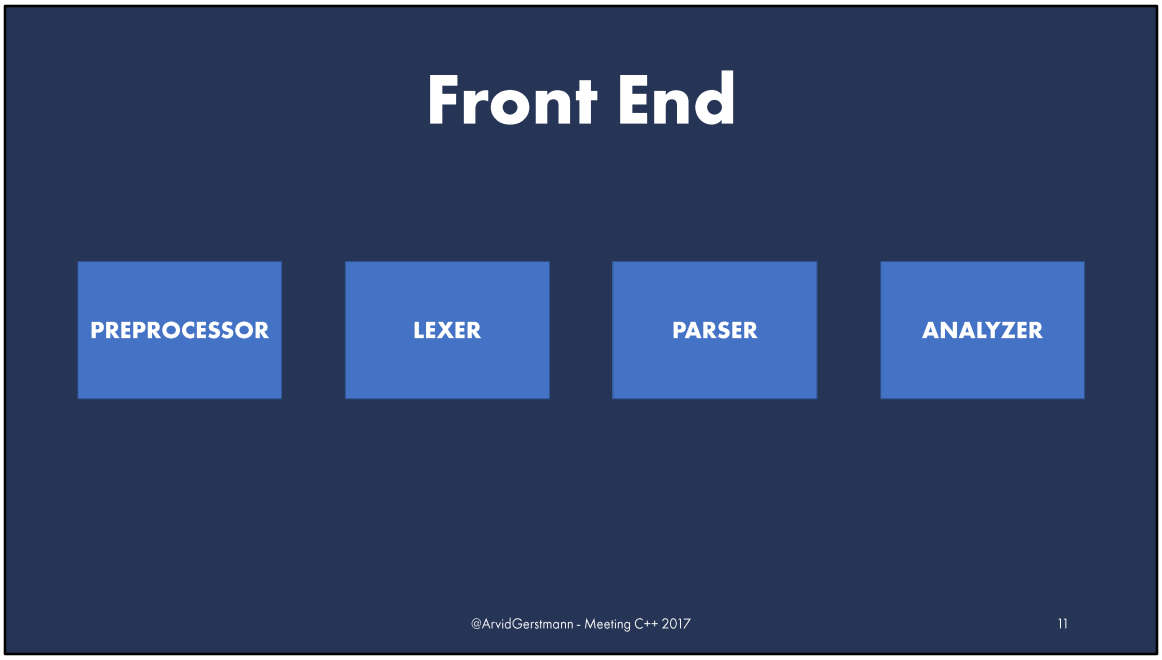


The AST is created by traversing the list of tokens created by the „lexer“.
If the input source code is syntactically correct, the compiler will produce a valid syntax tree.
If the input source code is syntactically incorrect, the compiler will likely produce diagnostics for the user, showing where the errors occurred, and might even stop the compilation process.

The following will show how an abstract syntax tree might look like:

- At the top, we have our „function declaration“. Consisting of the return type „int“, the identifier „add“ and an empty argument list.
- The body of our function is a single „compound statement“.
- In our example, containing only a single „return statement“.
- The return statement consists of „binary operation“, adding together two „integer literals“: „5“ and „10“

As you can see, the compiler has now build a **new** representation of the source code. It‘s not operating on the level of „tokens“, or source text anymore, but on a more **abstract** level.



Last, but not least, we have the most important part of the front end: **The Analyzer.**

The semantic analyzer will now build the symbol table and annotate the syntax tree. Lots of different information is added: For example, the type of variables, which can later be used for static type checking.
The analyzer will than proceed to analyze our source code for semantically incorrect constructs, by making use of all that new information.
If we would try to return a string from our „add“ function, for example, the analyzer would generate a diagnostic and possibly stop the compilation process.

The now annotated abstract syntax tree is converted into the IR and passed to the next stage:

Middle End

The **„middle end“.** Having now received the IR, the job of the middle end is to analyze and perform optimizations on the IR.
This is often done in so-called „passes“. An optimization pass is transforming IR, without changing the observable side-effects, with the goal of either reducing size, improving run-time performance, or both.

There are many different optimizations a compiler can perform, for example:

- Propagating constants, the compiler would‘ve optimized the „add“ function from the previous slides, by replacing the „5 + 10“ with a literal „15“, since both operands of the „add“ operation are known at compile time.
- Expanding inline functions
- Unrolling loops
- Eliminating dead-code
- Parallizing sections of code
- And many, many more. A compiler has tens, if not hundreds of those small optimizations.

These optimizations are enabled by the following, critical, component: the analysis.

Analysis is crucial, since without the compiler having thorough knowledge about the generated IR, **prior** to applying optimizations, the optimizations would lack fundamental information, and would not be possible. For example, to eliminate dead-code, the compiler has to be certain a branch cannot be reached at all times. And this might even rely on previous optimizations, like constant propagation.

Some of the applied analysis include, but are not limited to:

- Alias Analysis
- Dependency Analysis
- Data Flow Analysis
- And many more

Once the middle end is done, the IR is passed to the last stage:

Back End

The **„back end“.** So far, the compiler was fully architecture independent. The job of the back end is now to convert the architecture-independent IR, into architecture dependent assembly. This is step often called „code generation“. (*next slide*)
To generate efficient assembly from the IR, the backend has to go through a few steps:

First, the back end has to transform the IR, received from the middle end, into real assembly. This procedure is called „Instruction Selection“ (*next slide*) and is done, by mapping operations inside the IR into groups of real instructions. In this step, registers are ignored, as they will be filled later.
Throghout the process, the compiler might perform further, machine-dependant optimizations, when deemed necessary or useful. One example of such an optimization are „peephole optimizations“, replacing one set of instruction, with another more efficient set, while keeping the observeable output the same.

Bringing us to the next step: (*next slide*) Register Allocation.

The formerly left-out registers will now be filled with real registers or memory operands. Since registers are a finite set, the compiler has to decide which to use where. If this is not possible, registers might be „spilled“ on the stack, essentially saving the contents, to be picked up later again.

Once done, we come to the last step: (*next slide*) Instruction Scheduling.

All recent CPU designs feature instruction pipelines and can execute several instructions in parallel. The instruction scheduler tries to avoid stalling the pipeline, by rearranging the order of instructions, while maintaining the meaning of the code. This step can either be done before, or after register scheduling. Resulting in either improved parallelism, but increased possibility of register spilling, or the opposite.

Assembling & Linking

As the very last step, we have assembling and linking, which is, strictly speaking, not part of the compiler anymore, and has traditionally been implemented in different projects.
(*next slide*)

The compiler is now done, and can send the resulting assembly to the assembler.
The assembler is converting our internal assembly representation into the final object code. The zeros and ones. Bits and bytes.

Assembling & Linking

```
add eax, 5
mul edx
```

```
83 c0 05
f7 e2
```


As we can see here, the assembler has now translated the assembly on the left, into a binary representation.
The binary representation is written as a .o or .obj file to disk and is ready to be linked.

Assembling & Linking

The linker will now combine one or more object files, resolve all symbols and create the final executable.

Thank You!

Find me on:

Twitter: twitter.com/ArvidGerstmann

Blog: arvid.io

Thank you!
Slides will be available to download from arvid.io later today.