# How a real-world C++ compiler works

**Martin Cejp**

mentor: **Javier Lopez Gomez**

14th Inverted CERN School of Computing

7 March 2023

rev. 1

CAUTION
May contain traces of assembly code and toolchain authors' dirty secrets

# Background

- High expectations of compilers
  - Past:
    - Take this source file and compile it into correct and efficient code
  - Now:
    - Take this source file and compile it into correct and efficient code
      - + apply generic (but also machine-dependent) optimizations
      - + excellent diagnostics of syntax/semantic error, ideally with fix-it hints
      - + static analysis (detect buggy code)
      - + automatic instrumentation (AddressSanitizer, ...)
      - + compiler as a library powering code completions
    - ... all of that while keeping up with the constant language evolution

# Background

- How do compiler authors manage the complexity?
  - Wizardry?

# Compilation pipeline

- Chosen example – Clang (nice modular design, top 3 compilers along with GCC & MSVC)
- Compilation pipeline
  1. Preprocessing
  2. Lexical, syntactic and semantic analysis
  3. Intermediate code generation and optimization
  4. Code generation for target architecture
  5. Linking
- Orchestrated by **compiler driver** (this is what the `clang` program actually is)

# Compilation pipeline

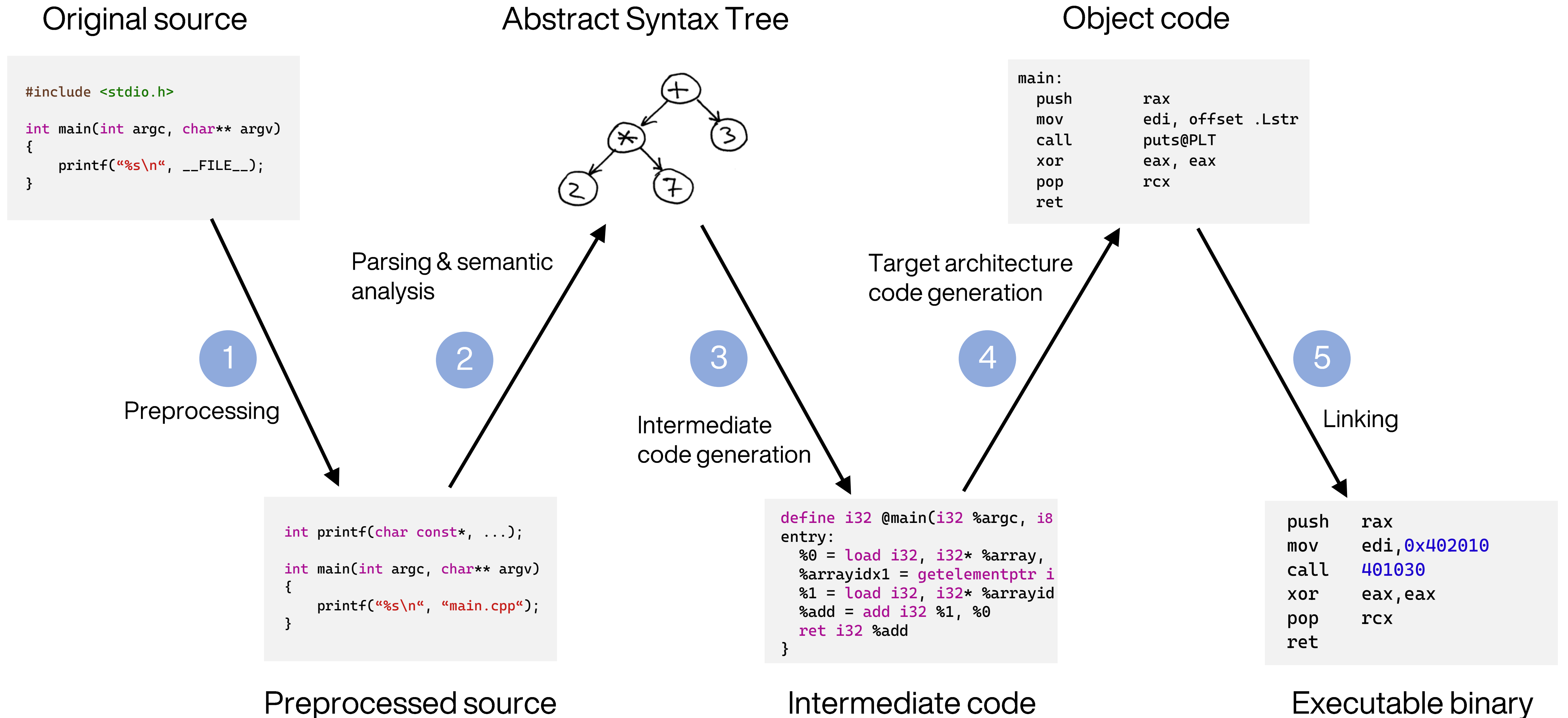
- Chosen example – Clang (nice modular design, top 3 compilers along with GCC & MSVC)
- Compilation pipeline
  1. Preprocessing
  2. Lexical, syntactic and semantic analysis
  3. Intermediate code generation and optimization
  4. Code generation for target architecture
  5. Linking
- Orchestrated by **compiler driver**
  (this is what the `clang` program actually is)

Front-end

Back-end

Linker

# Compilation pipeline



1

Preprocessing

2

Lexical, syntactic & semantic analysis

3

Intermediate code generation

4

Code generation for target architecture

5

Linking

# 1 | Preprocessing

**Input:** original source code
**Output:** preprocessed source code

- **Q:** How does the compiler learn about functions, types and other symbols defined in other compilation units? (ignoring C++20 modules)
  - **A: #include** directive – equivalent to verbatim inclusion of given "header" file
- Other facilities:
  - **#define** directive – macros
    - Constants that can be used anywhere, defined in code, build system, or compiler built-in
    - "Functions" (superficially) that can be used anywhere
    - Custom syntactic constructs
  - Conditional compilation – **#if** directive
- Preprocessor *expands* these directives away
  - Weird exception: **#pragma**
- <u>Limited understanding of syntax, no understanding of semantics</u>

# 1 | Preprocessing

**Input:** original source code
**Output:** preprocessed source code

```
1  #include <iostream>
2
3  // silly macro for illustration
4  #define STD(x) std::x
5
6  int main() {
7      STD(cout) << "Hello, world" << STD(endl);
8  }
```

Included files with their respective dependencies are effectively inserted into the code in full

```
31446
31447  }
31448  # 2 "hello.cpp" 2
31449
31450
31451
31452  int main() {
31453      std::cout << "Hello, world" << std::endl;
31454  }
```

special markers allow the compiler to reconstruct original source locations (for diagnostics / debug info)

comments and macro definitions are discarded

macros are expanded without a trace*

Note: during normal compilation, some information about macros is in fact retained, allowing the compiler to produce better error messages

# 1 | Preprocessing

**Input:** original source code
**Output:** preprocessed source code

*Try this at home!*

```
clang -E main.cpp -o main.ii
```

```
1  #include <iostream>
2
3  // silly macro for illustration
4  #define STD(x) std::x
5
6  int main() {
7      STD(cout) << "Hello, world" << STD(endl);
8  }
```

Included files with their respective dependencies are effectively inserted into the code in full

```
31446
31447  }
31448  # 2 "hello.cpp" 2
31449
31450
31451
31452  int main() {
31453      std::cout << "Hello, world" << std::endl;
31454  }
```

special markers allow the compiler to reconstruct original source locations (for diagnostics / debug info)

comments and macro definitions are discarded

macros are expanded without a trace*

Note: during normal compilation, some information about macros is in fact retained, allowing the compiler to produce better error messages

# 1 | Preprocessing

**Input:** original source code
**Output:** preprocessed source code

- Dangers of macros
  - No regard for language syntax rules

```
#define product(a, b)  a * b

int foo = product(2, 1 + 3);   ➔ 2 * 1 + 3 ➔ WRONG!
```

  - Literal substitution of arguments – risk of multiple evaluation

```
#define min(a, b)      (a < b ? a : b)

int amount_to_withdraw = min(user_input(),
                             account_balance);


 ➔ (user_input() < account_balance ? user_input()
                                   : account_balance) ➔ WRONG!
```

Preprocessing

Lexical, syntactic & semantic analysis

Intermediate code generation

Code generation for target architecture

Linking

# 2 | Lexical analysis

**Input:** preprocessed source code
**Output:** token stream

- Language grammar is defined in terms of tokens – pieces of source code that cannot be meaningfully subdivided
- Examples:

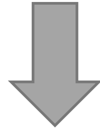
| | |
|---|---|
| `while` | keyword token |
| `main` | identifier token |
| `-` | minus operator token |
| `>` | greater-than operator token |
| `->` | dereference (arrow) operator token |

- Tokenization generally ignorant of parser grammar – independent sets of rules
- One interesting corner case in C++: '`>>`' character sequence
  - `int mask = (0x100 >> 8)`
  - `std::array<int, 1024>>2>` ← valid until C++03
  - `std::vector<std::array<int, 4>>` ← valid from C++11
  - In Clang: always greedily tokenized as '`>>`' operator and special-case handled in parser

# 2 | Lexical analysis

**Input:** preprocessed source code
**Output:** token stream

```
31452   int main() {
31453       std::cout << "Hello, world" << std::endl;
31454   }
```

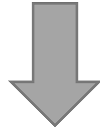


```
int 'int'                         [StartOfLine]                Loc=<hello.cpp:6:1>
identifier 'main'                 [LeadingSpace]               Loc=<hello.cpp:6:5>
l_paren '('                                                    Loc=<hello.cpp:6:9>
r_paren ')'                                                    Loc=<hello.cpp:6:10>
l_brace '{'                       [LeadingSpace]               Loc=<hello.cpp:6:12>
identifier 'std'                  [StartOfLine] [LeadingSpace] Loc=<hello.cpp:7:5 <Spelling=hello.cpp:4:16>>
coloncolon '::'                                                Loc=<hello.cpp:7:5 <Spelling=hello.cpp:4:19>>
identifier 'cout'                                              Loc=<hello.cpp:7:5 <Spelling=hello.cpp:7:9>>
lessless '<<'                     [LeadingSpace]               Loc=<hello.cpp:7:15>
string_literal '"Hello, world"'   [LeadingSpace]               Loc=<hello.cpp:7:18>
lessless '<<'                     [LeadingSpace]               Loc=<hello.cpp:7:33>
identifier 'std'                  [LeadingSpace]               Loc=<hello.cpp:7:36 <Spelling=hello.cpp:4:16>>
coloncolon '::'                                                Loc=<hello.cpp:7:36 <Spelling=hello.cpp:4:19>>
identifier 'endl'                                              Loc=<hello.cpp:7:36 <Spelling=hello.cpp:7:40>>
semi ';'                                                       Loc=<hello.cpp:7:45>
r_brace '}'                       [StartOfLine]                Loc=<hello.cpp:8:1>
eof ''                                                         Loc=<hello.cpp:8:2>
```

# 2 | Lexical analysis

**Input:** preprocessed source code
**Output:** token stream

```
31452   int main() {
31453       std::cout << "Hello, world" << std::endl;
31454   }
```



*Try this at home!*

```
clang -fsyntax-only -Xclang -dump-tokens main.cpp
```

```
int 'int'                          [StartOfLine]                  Loc=<hello.cpp:6:1>
identifier 'main'                  [LeadingSpace]                 Loc=<hello.cpp:6:5>
l_paren '('                                                       Loc=<hello.cpp:6:9>
r_paren ')'                                                       Loc=<hello.cpp:6:10>
l_brace '{'                        [LeadingSpace]                 Loc=<hello.cpp:6:12>
identifier 'std'                   [StartOfLine] [LeadingSpace]   Loc=<hello.cpp:7:5 <Spelling=hello.cpp:4:16>>
coloncolon '::'                                                   Loc=<hello.cpp:7:5 <Spelling=hello.cpp:4:19>>
identifier 'cout'                                                 Loc=<hello.cpp:7:5 <Spelling=hello.cpp:7:9>>
lessless '<<'                      [LeadingSpace]                 Loc=<hello.cpp:7:15>
string_literal '"Hello, world"'    [LeadingSpace]                 Loc=<hello.cpp:7:18>
lessless '<<'                      [LeadingSpace]                 Loc=<hello.cpp:7:33>
identifier 'std'                   [LeadingSpace]                 Loc=<hello.cpp:7:36 <Spelling=hello.cpp:4:16>>
coloncolon '::'                                                   Loc=<hello.cpp:7:36 <Spelling=hello.cpp:4:19>>
identifier 'endl'                                                 Loc=<hello.cpp:7:36 <Spelling=hello.cpp:7:40>>
semi ';'                                                          Loc=<hello.cpp:7:45>
r_brace '}'                        [StartOfLine]                  Loc=<hello.cpp:8:1>
eof ''                                                            Loc=<hello.cpp:8:2>
```

# 2 | Syntactic and semantic analysis

**Input:** token stream

**Output:** Abstract Syntax Tree

- Syntactic analysis:

```
auto a = 3;
```

➔ (define variable *a:* type *auto*, initialize with literal *3*)

```
auto b = (“hello” * a);
```

➔ (define variable *b:* type *auto*, initialize with (multiplication of literal “hello” with identifier *a*))

➔ syntactically **valid** program

# 2 | Syntactic and semantic analysis

**Input:** token stream
**Output:** Abstract Syntax Tree

- Syntactic analysis:

```
auto a = 3;
```

➔ (define variable *a:* type *auto*, initialize with literal *3*)

```
auto b = (“hello” * a);
```

➔ (define variable *b:* type *auto*, initialize with (multiplication of literal “hello” with identifier *a*))

➔ syntactically **valid** program

- Semantic analysis:

```
auto a = 3;
```

➔ (define variable *a:* type *auto*, initialize with literal *3*)
➔ effective type of *a* is *int*

```
auto b = (“hello” * a);
```

➔ (define variable *b:* type *auto*, initialize with (multiplication of literal “hello” with identifier *a*))
➔ effective type of *b* is... **error!** cannot multiply *string literal* with *int*

➔ semantically **invalid** program

# 2 | Syntactic and semantic analysis

**Input:** token stream
**Output:** Abstract Syntax Tree

- Ideally: two distinct steps
- C++: intertwined processes

Context matters for parsing! Different ways to parse the same expression:

A.

```
int a, b;

auto c = (a) * b;
```

➔ multiply *a* and *b*

B.

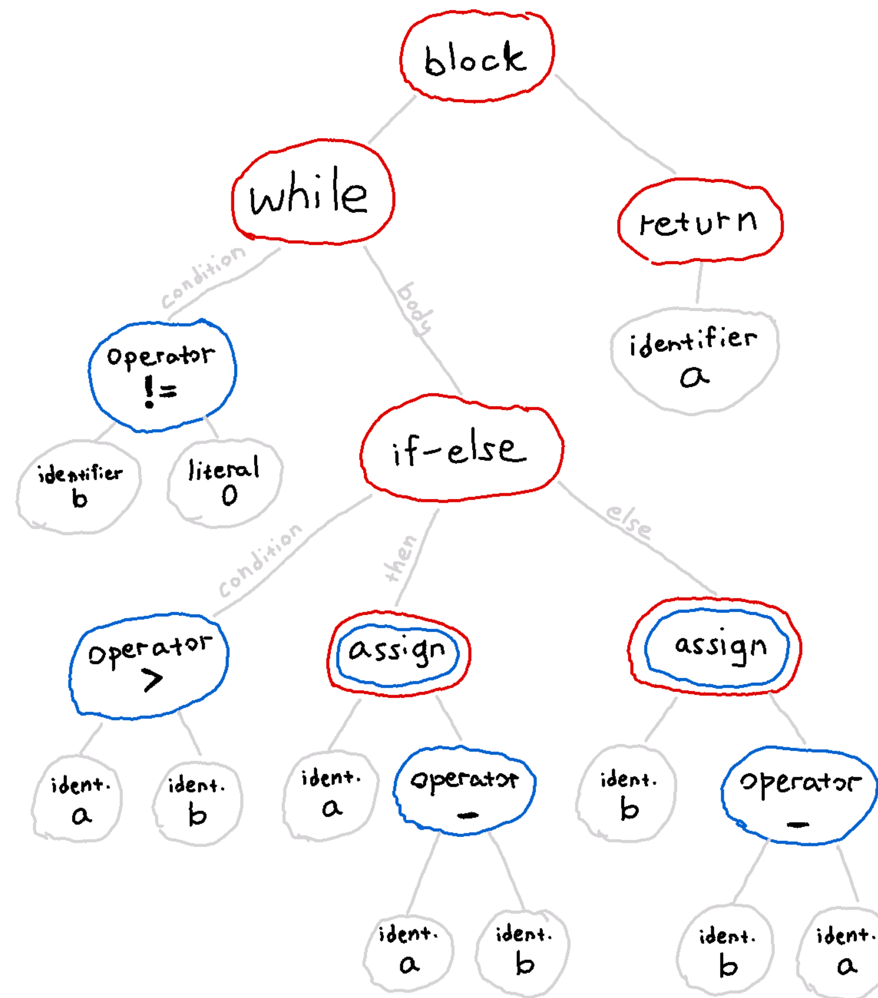
```
typedef short a;
int* b;

auto c = (a) * b;
```

➔ dereference *b*, cast to type *a*

# 2 | Abstract Syntax Tree – textbook example

```
while (b != 0) {
    if (a > b)
        a = a - b;
    else
        b = b - a;
}
return a;
```

(Euclidean algorithm)






Example adapted from https://en.wikipedia.org/wiki/Abstract_syntax_tree

# 2 | Abstract Syntax Tree – real example

```
1   #include <stdio.h>
2
3   int main(int argc, char** argv) {
4       puts("Hello, world!");
5   }
```

```
TranslationUnitDecl <<invalid sloc>> <invalid sloc>
`-FunctionDecl <hello-stdio.cpp:3:1, line:5:1> line:3:5 main 'int (int, char **)'
  |-ParmVarDecl <col:10, col:14> col:14 argc 'int'
  |-ParmVarDecl <col:20, col:27> col:27 argv 'char **'
  `-CompoundStmt <col:33, line:5:1>
    `-CallExpr <line:4:5, col:25> 'int'
      |-ImplicitCastExpr <col:5> 'int (*)(const char *)' <FunctionToPointerDecay>
      | `-DeclRefExpr <col:5> 'int (const char *)' lvalue Function 'puts' 'int (const char *)'
      `-ImplicitCastExpr <col:10> 'const char *' <ArrayToPointerDecay>
        `-StringLiteral <col:10> 'const char[14]' lvalue "Hello, world!"
```

Not shown here: parts of AST generated by the included header (hundreds of lines!)

# 2 | Abstract Syntax Tree – real example

```
1   #include <stdio.h>
2
3   int main(int argc, char** argv) {
4       puts("Hello, world!");
5   }
```

*Try this at home!*

```
clang -fsyntax-only -Xclang -ast-dump main.cpp
```

```
TranslationUnitDecl <<invalid sloc>> <invalid sloc>
`-FunctionDecl <hello-stdio.cpp:3:1, line:5:1> line:3:5 main 'int (int, char **)'
  |-ParmVarDecl <col:10, col:14> col:14 argc 'int'
  |-ParmVarDecl <col:20, col:27> col:27 argv 'char **'
  `-CompoundStmt <col:33, line:5:1>
    `-CallExpr <line:4:5, col:25> 'int'
      |-ImplicitCastExpr <col:5> 'int (*)(const char *)' <FunctionToPointerDecay>
      | `-DeclRefExpr <col:5> 'int (const char *)' lvalue Function 'puts' 'int (const char *)'
      `-ImplicitCastExpr <col:10> 'const char *' <ArrayToPointerDecay>
        `-StringLiteral <col:10> 'const char[14]' lvalue "Hello, world!"
```

Not shown here: parts of AST generated by the included header (hundreds of lines!)

1

Preprocessing

2

Lexical, syntactic & semantic analysis

3

Intermediate code generation

4

Code generation for target architecture

5

Linking

# 3 | Intermediate code generation

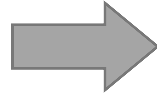
**Input:** Abstract Syntax Tree
**Output:** intermediate code

- Why intermediate code?
  - First compilers: parse one language, emit code for one CPU architecture
  - How to deal with *M* languages and *N* architectures?
  - One possibility: implement a special compiler for every combination ➔ $M \times N$
  - Alternatively: Implement a front-end for every language, a common IR and a backend for each CPU architecture ➔ $M + N$
- LLVM Intermediate Representation – LLVM IR
- Static Single Assignment form (SSA)
  - Every variable in SSA is assigned exactly once
- Advantageous also for implementation of optimization passes
  - Easier to track value lifetime in SSA

# 3 | Intermediate code generation

**Input:** Abstract Syntax Tree
**Output:** intermediate code

```
int sum_els(int const* array) {
    return array[0] + array[1];
}
```



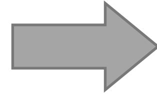
```
define i32 @sum_els(i32* %array) {
entry:
  %0 = load i32, i32* %array, align 4
  %arrayidx1 = getelementptr i32, i32* %array, i64 1
  %1 = load i32, i32* %arrayidx1, align 4
  %add = add i32 %1, %0
  ret i32 %add
}
```

- Functions are still functions
- Most other high-level structure is lost
  - Nested expressions are decomposed into a series of operations
    - Introduction of additional temporary variables
  - Control flow expressed in terms of basic blocks (not shown here)

# 3 | Intermediate code generation

**Input:** Abstract Syntax Tree
**Output:** intermediate code

```
int sum_els(int const* array) {
    return array[0] + array[1];
}
```



```
define i32 @sum_els(i32* %array) {
entry:
  %0 = load i32, i32* %array, align 4
  %arrayidx1 = getelementptr i32, i32* %array, i64 1
  %1 = load i32, i32* %arrayidx1, align 4
  %add = add i32 %1, %0
```

*Try this at home!*

```
clang -S -emit-llvm -fno-discard-value-names main.cpp -o main.ll
```

- Functions are still functions
- Most other high-level structure is lost
  - Nested expressions are decomposed into a series of operations
    - Introduction of additional temporary variables
  - Control flow expressed in terms of basic blocks (not shown here)

# 3 | Optimization

```
int deref(int const* pointer) {
    return *pointer;
}
```

**–O0 (no optimization)**

```
define i32 @deref(i32* %pointer) {
entry:
  %pointer.addr = alloca i32*, align 8
  store i32* %pointer, i32** %pointer.addr, align 8
  %0 = load i32*, i32** %pointer.addr, align 8
  %1 = load i32, i32* %0, align 4
  ret i32 %1
}
```

Notice the pointless copy of the parameter

**–O1 (basic optimization)**

```
define i32 @deref(i32* %pointer) {
entry:
  %0 = load i32, i32* %pointer, align 4
  ret i32 %0
}
```

It’s a feature, not a bug!

- By design, compiler generates naïve code with the assumption that it can be optimized easily
- This lets the compiler implementation be simpler and better maintainable

1
Preprocessing

2
Lexical, syntactic & semantic analysis

3
Intermediate code generation

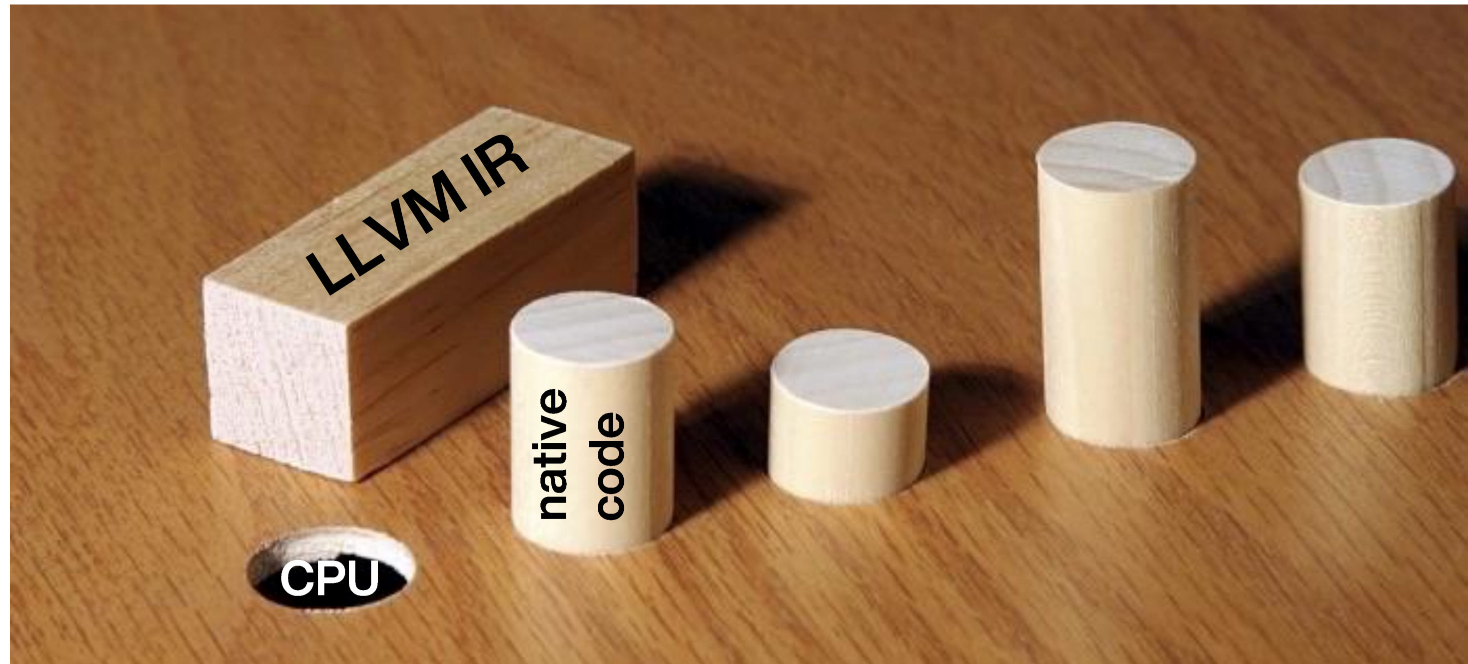
4
Code generation for target architecture

5
Linking

# 4 | Code generation for target architecture

**Input:** intermediate code
**Output:** object code for a specific combination of architecture + OS

- Intermediate code has many advantages, but ultimately, we need to run on a real CPU
- Architecture-specific optimizations may be applied during this phase
- Intermediate code is still abstract (even naïve) in many ways
  - Instruction set
  - Register allocation
  - Local variables
  - Calling convention



# 4 | Code generation for target architecture

- Intermediate code has many advantages, but ultimately, we need to run on a real CPU
- Architecture-specific optimizations may be applied during this phase
- Intermediate code is still abstract (even naïve) in many ways
  - Instruction set
  - Register allocation
  - Local variables
  - Calling convention

**Input:** intermediate code
**Output:** object code for a specific combination of architecture + OS

```
void foo(int a, int* b) {
    bar(a, b[1]);
}
```

LLVM IR

```
define void @foo(i32 %a, i32* %b) {
entry:
  %arrayidx = getelementptr i32, i32* %b, i64 1
  %0 = load i32, i32* %arrayidx, align 4
  call void @bar(i32 %a, i32 %0)
  ret void
}
```

Generally higher-level instructions operating on variables

x86_64

```
foo:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    mov     dword ptr [rbp - 4], edi
    mov     qword ptr [rbp - 16], rsi
    mov     edi, dword ptr [rbp - 4]
    mov     rax, qword ptr [rbp - 16]
    mov     esi, dword ptr [rax + 4]
    call    bar
    add     rsp, 16
    pop     rbp
    ret
```

Operating on registers or memory locations

# 4 | Code generation for target architecture

**Input:** intermediate code
**Output:** object code for a specific combination of architecture + OS

- Intermediate code has many advantages, but ultimately, we need to run on a real CPU
- Architecture-specific optimizations may be applied during this phase
- Intermediate code is still abstract (even naïve) in many ways
  - Instruction set
  - Register allocation
  - Local variables
  - Calling convention

```
void foo(int a, int* b) {
    bar(a, b[1]);
}
```

LLVM IR

```
define void @foo(i32 %a, i32* %b) {
entry:
  %arrayidx = getelementptr i32, i32* %b, i64 1
  %0 = load i32, i32* %arrayidx, align 4
  call void @bar(i32 %a, i32 %0)
  ret void
}
```

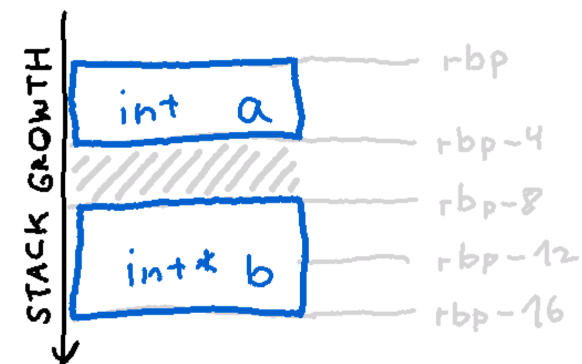
Arbitrary number of temporary variables, assigned only once

x86_64

```
foo:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     dword ptr [rbp - 4], edi
        mov     qword ptr [rbp - 16], rsi
        mov     edi, dword ptr [rbp - 4]
        mov     rax, qword ptr [rbp - 16]
        mov     esi, dword ptr [rax + 4]
        call    bar
        add     rsp, 16
        pop     rbp
        ret
```

Fixed set of CPU registers; must be re-used

# 4 | Code generation for target architectu




**Input:** intern
**Output:** objec
comb

- Intermediate code has many advantages, but ultimately, we need to run on a real CPU
- Architecture-specific optimizations may be applied during this phase
- Intermediate code is still abstract (even naïve) in many ways
  - Instruction set
  - Register allocation
  - Local variables
  - Calling convention

```
void foo(int a, int* b) {
    bar(a, b[1]);
}
```

LLVM IR

```
define void @foo(i32 %a, i32* %b) {
entry:
  %arrayidx = getelementptr i32, i32* %b, i64 1
  %0 = load i32, i32* %arrayidx, align 4
  call void @bar(i32 %a, i32 %0)
  ret void
}
```

No regard for storage location of temporaries

```
foo:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     dword ptr [rbp - 4], edi
        mov     qword ptr [rbp - 16], rsi
        mov     edi, dword ptr [rbp - 4]
        mov     rax, qword ptr [rbp - 16]
        mov     esi, dword ptr [rax + 4]
        call    bar
        add     rsp, 16
        pop     rbp
        ret
```

Concrete layout of stack frame

# 4 | Code generation for target architecture

- Intermediate code has many advantages, but ultimately, we need to run on a real CPU
- Architecture-specific optimizations may be applied during this phase
- Intermediate code is still abstract (even naïve) in many ways
  - Instruction set
  - Register allocation
  - Local variables
  - Calling convention

**Input:** intermediate code
**Output:** object code for a specific combination of architecture + OS

```
void foo(int a, int* b) {
    bar(a, b[1]);
}
```

LLVM IR

```
define void @foo(i32 %a, i32* %b) {
entry:
  %arrayidx = getelementptr i32, i32* %b, i64 1
  %0 = load i32, i32* %arrayidx, align 4
  call void @bar(i32 %a, i32 %0)
  ret void
}
```

x86_64

```
foo:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    mov     dword ptr [rbp - 4], edi
    mov     qword ptr [rbp - 16], rsi
    mov     edi, dword ptr [rbp - 4]
    mov     rax, qword ptr [rbp - 16]
    mov     esi, dword ptr [rax + 4]
    call    bar
    add     rsp, 16
    pop     rbp
    ret
```

Arguments explicitly listed in call instruction

Arguments in registers or on stack (per calling conv.)

# 4 | Code generation for target architecture

- Intermediate code has many advantages, but ultimately, we need to run on a real CPU
- Architecture-specific optimizations may be applied during this phase
- Intermediate code is still abstract (even naïve) in many ways
  - Instruction set
  - Register allc
  - Local variab
  - Calling conv

**Input:** intermediate code
**Output:** object code for a specific combination of architecture + OS

*Try this at home!*

```
clang -c -S -masm=intel main.cpp -o main.s
```

```
define void @foo
entry:
  %arrayidx = getelementptr i32, i32* %b, i64 1
  %0 = load i32, i32* %arrayidx, align 4
  call void @bar(i32 %a, i32 %0)
  ret void
}
```

x86_64

```
foo:
        push    rbp
        ...     [rbp - 4], edi
        ...     [rbp - 16], rsi
        ...     ...d ptr [rbp - 4]
        mov     rax, qword ptr [rbp - 16]
        mov     esi, dword ptr [rax + 4]
        call    bar
        add     rsp, 16
        pop     rbp
        ret
```

Arguments explicitly listed in call instruction

Arguments in registers or on stack (per calling conv.)

1

Preprocessing

2

Lexical, syntactic &
semantic analysis

3

Intermediate
code generation

4

Code generation for
target architecture

5

Linking

# 5 | Linking

- Discrepancy between
  1. **Source code:** function and variable names
  2. **Machine code:** addresses within code and data regions
- How to ensure that the entire program is composed meaningfully, and all references are correct?
  - Including dynamically linked libraries
- Job of the linker

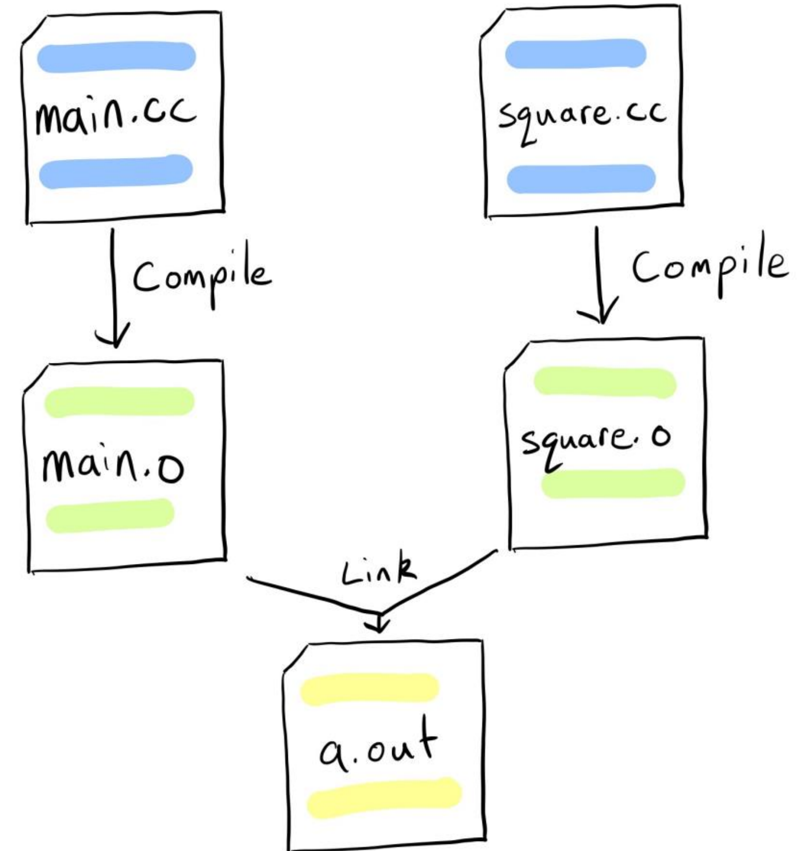
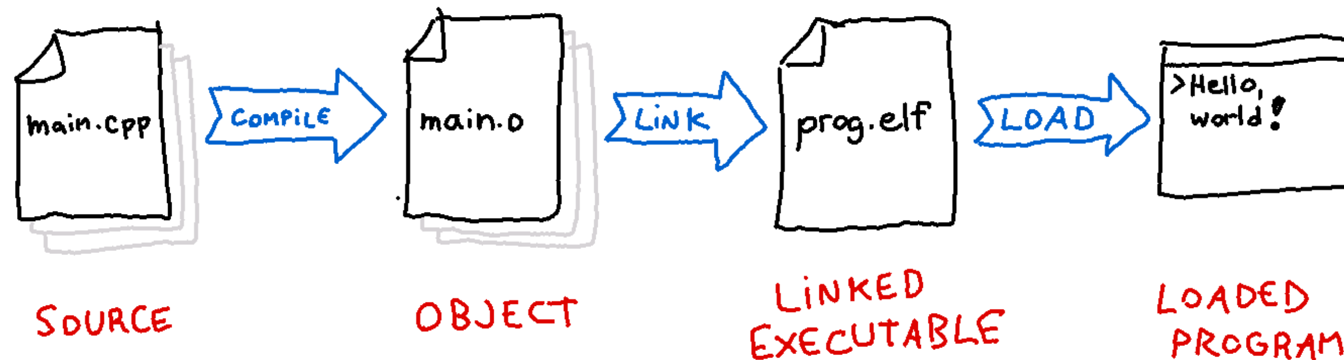



Figure: https://joellaity.com/2020/01/25/linking.html

# 5 | Linking

Seen in Chapter 4

- Reality: Additional intermediate steps
  1. Source code: function and variable names
  2. **Object code**: relative references where possible (branches within function, local variables), global names otherwise
  3. **Linked program**: addresses within code and data regions
     - If linked **statically**: fully self-contained
     - If linked **dynamically**: includes references to functions in shared libraries
  4. Machine code: addresses within code and data regions



# 5 | Linking

```c
#include <stdio.h>

int main(int argc, char** argv) {
    puts("Hello, world!");
}
```

**Input:** object files
**Output:** linked program (e.g., ELF format)

Object code

```
main:
        push    rax
        mov     edi, offset .Lstr
        call    puts@PLT
        xor     eax, eax
        pop     rcx
        ret

        .section .rodata
.Lstr:
        .asciz  "Hello, world!"
```

Linked program

```
00401130 <main>:
  401130:    50                   push   rax
  401131:    bf 10 20 40 00       mov    edi,0x402010
  401136:    e8 f5 fe ff ff       call   401030
  40113b:    31 c0                xor    eax,eax
  40113d:    59                   pop    rcx
  40113e:    c3                   ret
  ...

  402010:    48 65 6c 6c 6f 2c 20 77 6f 72 6c 64 21 00
```

Symbolic references

Absolute addresses

# 5 | Linking

```c
#include <stdio.h>

int main(int argc, char** argv) {
    puts("Hello, world!");
}
```

**Input:** object files
**Output:** linked program (e.g., ELF format)

**Object code**

```
main:
        push    rax
        mov     edi, offset .Lstr
        call    puts@PLT
        xor     eax, eax
        pop     rcx
        ret

        .section .rodata
.Lstr:
        .asciz   "Hello, world!"
```

**Linked program**

```
00401130 <main>:
  401130:    50                  push   rax
  401131:    bf 10 20 40 00      mov    edi,0x402010
  401136:    e8 f5 fe ff ff      call   401030
  40113b:    31 c0               xor    eax,eax
  40113d:    59                  pop    rcx
  40113e:    c3                  ret
  ...

  402010:    48 65 6c 6c 6f 2c 20 77 6f 72 6c 64 21 00
```

Some references may still be relative (making such code **position-independent**)
In this case: 0xFFFFFEF5 ≙ –267

# 5 | Linking

```c
#include <stdio.h>

int main(int argc, char** argv) {
    puts("Hello, world!");
}
```

**Input:** object files
**Output:** linked program (e.g., ELF format)

**Object code**

```
main:
        push    rax
        mov     edi, offset .Lstr
        call    puts@PLT
        xor     eax, eax
        pop     rcx
        ret

        .section .rodata
.Lstr:
        .asciz  "Hello, world!"
```

**Linked program**

```
00401130 <main>:
  401130:    50                   push   rax
  401131:    bf 10 20 40 00       mov    edi,0x402010
  401136:    e8 f5 fe ff ff       call   401030
  40113b:    31 c0                xor    eax,eax
  40113d:    59                   pop    rcx
  40113e:    c3                   ret
  ...

  402010:    48 65 6c 6c 6f 2c 20 77 6f 72 6c 64 21 00
```

Question for ABI experts in the audience:
What's up with this pair of instructions?

# 5 | Linking

```c
#include <stdio.h>

int main(int argc, char** argv) {
    puts("Hello, world!");
}
```

**Input:** object files
**Output:** linked program (e.g., ELF format)

Object code

```
main:
        push    rax
        mov     edi, of
        call    puts@PL
        xor     eax, ea
        pop     rcx
        ret

        .section .rodata
.Lstr:
        .asciz  "Hello, world!"
```

Linked program

```
00401130 <main>:
                                push   rax
    0                           mov    edi,0x402010
    f                           call   401030
                                xor    eax,eax
                                pop    rcx
  40113e:    c3                 ret
  ...

  402010:    48 65 6c 6c 6f 2c 20 77 6f 72 6c 64 21 00
```

*Try this at home!*

```
objdump -Mintel -D program
```

Question for ABI experts in the audience:
What's up with this pair of instructions?

# 5 | Static vs dynamic linking

**Input:** object files
**Output:** linked program (e.g., ELF format)

- **Static:** we have all the code and include it in the binary
- **Dynamic:** we know where the function resides, but we only include a reference to it
  - Advantages
    - Reduced program size
    - Reduced memory usage (single loaded instance of library for multiple programs)
    - Library can be updated independently
  - Drawbacks
    - May complicate distribution of program to user
    - Library can be updated independently – not always a win
  - Mechanism (Linux/ELF)
    - 2 data structures: **Procedure Linkage Table** and **Global Offset Table**
    - Managed at runtime by the **dynamic linker**
    - By default: lazy linking – a function's address is only resolved when it is called for the first time

# Summary & insights

- We scratched the surface of how the compilation pipeline works, using Clang/LLVM as an example
- The rabbit hole goes much deeper, though:
  - parsing the C++ language is sometimes not trivial
  - recovery from syntax errors
  - optimization passes, etc...
- Modular design helps tackle the complexity (*divide and conquer*)
  - e.g., language front-end does not need to produce optimal code
- Impressive amount of code processed to compile a 5-line **Hello World** program
- You can try this at home, all examples generated with off-the-shelf tools
  - Great online tools also available: Compiler Explorer – https://godbolt.org/
  - C++ Insights – https://cppinsights.io/

**Thank you** for your attention!

# References & credits

- LLVM source code (incl. Clang) – https://github.com/llvm/llvm-project
- The Clang AST - a Tutorial – https://youtu.be/VqCkCDFLSsc
- LLVM Language Reference Manual – https://llvm.org/docs/LangRef.html
- Clang design docs – https://clang.llvm.org/docs/Toolchain.html

- Resources used in preparing the presentation
  - Safety sign generator – https://observatory.db.erau.edu/generators/signs/
  - Syntax highlighting in slides – https://stratus3d.com/blog/2022/01/01/syntax-highlighting-in-powerpoint/
  - Cover slide – https://unsplash.com/photos/w95Fb7EEcjE

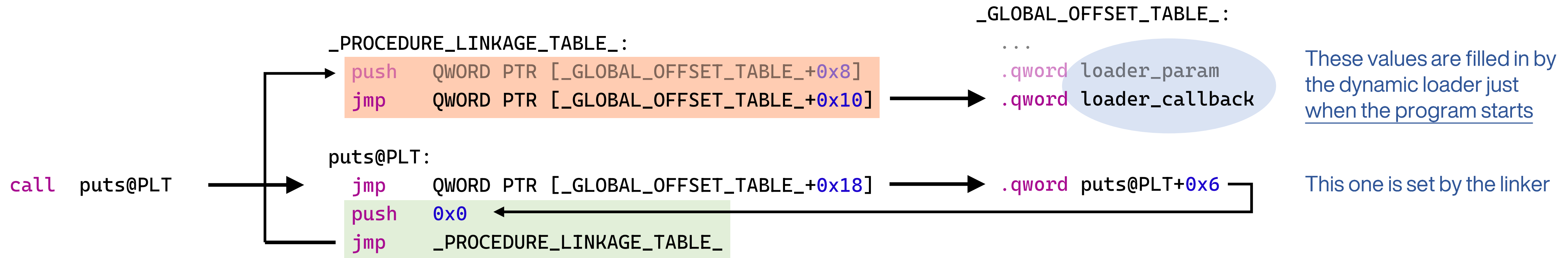

- Thanks to
  - my mentor Javier Lopez Gomez for invaluable feedback and suggestions
  - my home section SY-EPC-CCS and my supervisor Raul Murillo Garcia


- Find me on GitHub: https://github.com/mcejp

# Command cheatsheet

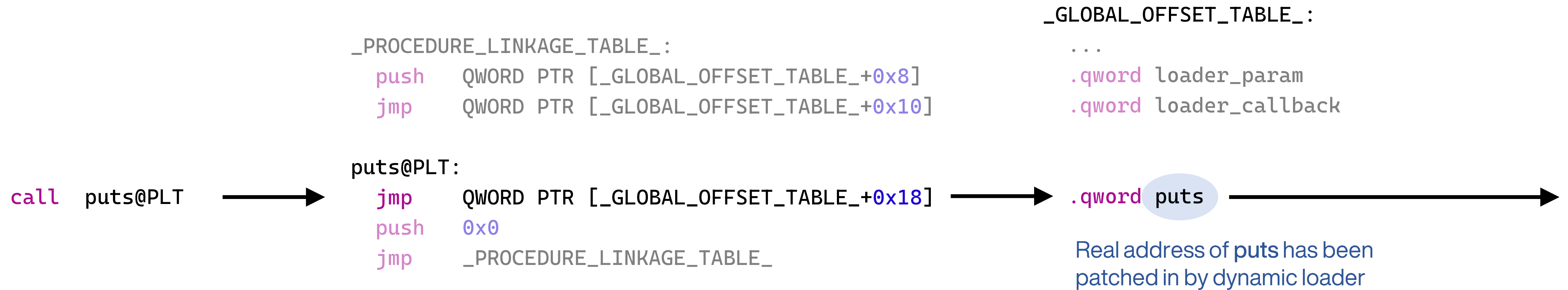
- Preprocess only
  - `clang -E main.cpp -o main.ii`
- Dump tokens
  - `clang -fsyntax-only -Xclang -dump-tokens main.cpp`
- Dump AST
  - `clang -fsyntax-only -Xclang -ast-dump main.cpp`
- Dump LLVM IR
  - `clang -S -emit-llvm -fno-discard-value-names main.cpp -o main.ll`
- Dump assembly
  - `clang -c -S -masm=intel main.cpp -o main.s`
- Disassemble executable
  - `objdump -Mintel -D program`

# Lazy dynamic linking




- First call to **puts@PLT**
  - GOT entry for **puts** is preloaded with address of initialization thunk
  - Jump to initialization thunk, from there jump to the 1st PLT entry, which in turns calls into the dynamic loader
  - Dynamic loader resolves address of **puts** function and places it into the GOT
- Subsequent calls
  - GOT entry for **puts** now points directly to the function in the shared library
    - Initialization thunk for **puts** is never used again
  - 1st PLT entry may be used again by a first call to *another* dynamically linked function

# Lazy dynamic linking




- First call to **puts@PLT**
  - GOT entry for **puts** is preloaded with address of **initialization thunk**
  - Jump to initialization thunk, from there jump to the 1st **PLT entry**, which in turns calls into the dynamic loader
  - Dynamic loader resolves address of **puts** function and places it into the GOT
- Subsequent calls
  - GOT entry for **puts** now points directly to the function in the shared library
    - Initialization thunk for **puts** is never used again
  - 1st PLT entry may be used again by a first call to *another* dynamically linked function

# Q: What's the deal with push eax / pop ecx?

A: Stack alignment before the call to *puts*! See https://stackoverflow.com/a/37774474

# Q: Why does GetElementPtr exist?

A: https://llvm.org/docs/GetElementPtr.html#how-is-gep-different-from-ptrtoint-arithmetic-and-inttoptr