Effizientes Programmieren in C, C++ und Rust

Compilation Pipeline

Colin Bretl, Nikolai Maas, Peter Maucher | 24.10.2024

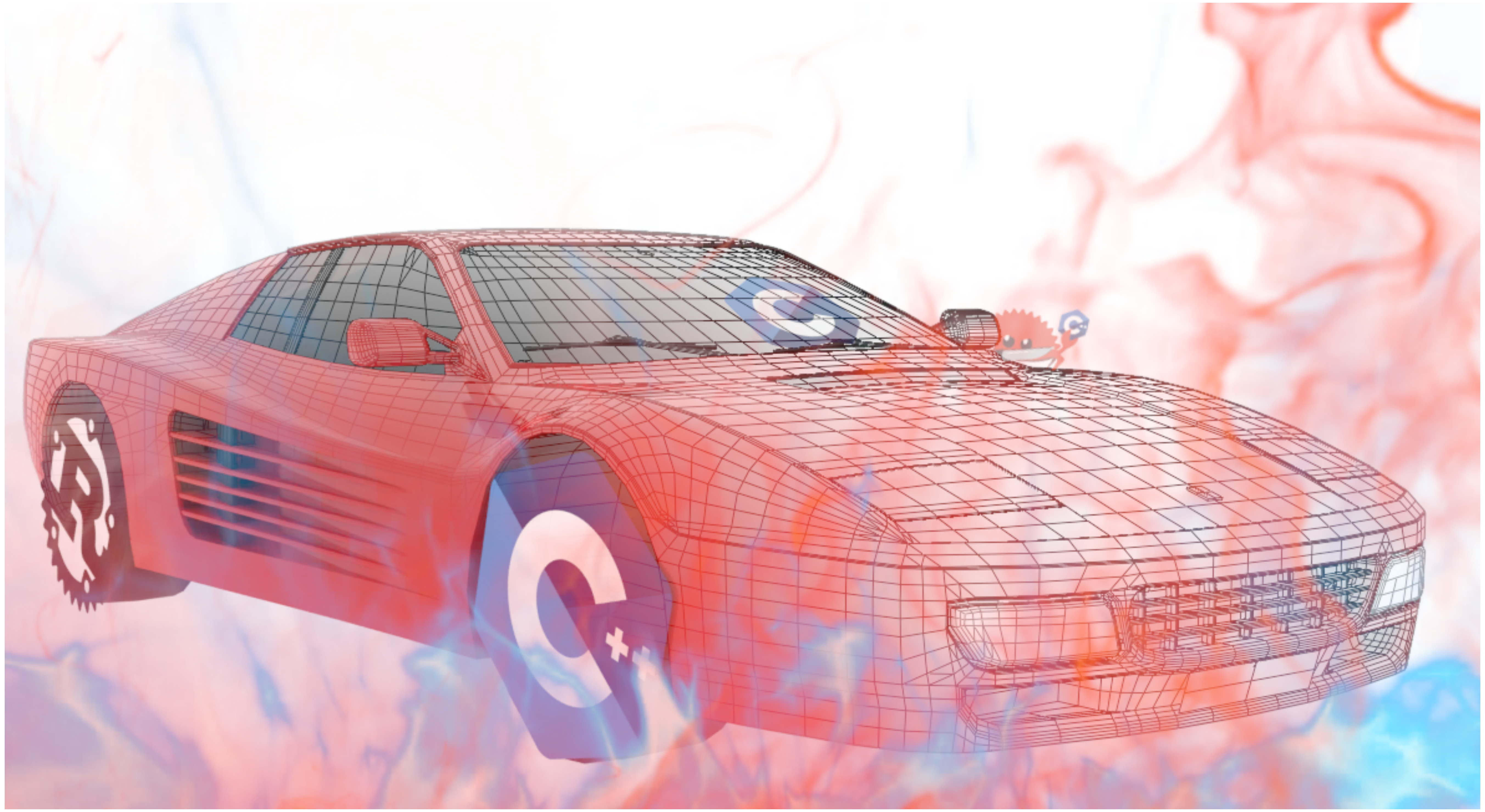


Table of Contents

C vs C++

Motivation

WHAT THE "PERFECT PROGRAM" SHOULD LOOK LIKE

- Knows exactly what to do *(machine readable!)*
- Knows exactly when to do it *(in sequence, little to no branching)*🚴
- Does only the things it needs to do *(no overhead)*🚴
- Does its job as fast as possible *(processing efficiency)*🚴
- Has everything it needs to do for its job at hand *(memory efficiency)*🚴

Motivation

WHAT C++ PROGRAMS/ PROJECTS LOOK LIKE

- (Many) **human-readable** source file(s) (.cpp)
- (Many) **human-readable** header file(s) (.h)
  - `#include "somelib"`
  - `#include <somestdlib>`
- Source code, `#pragma once`, ...
- (Many) directories

  src/, include/, lib/, tests/, doc/, .vscode/, build/
- ...

Motivation

Motivation

Godbolt

- Compiler Explorer https://godbolt.org/
- Developed by Matt Godbolt
- Incredibly useful to compare different compilers, compiler options, ...

Features

- different languages (C, C++, Rust, ...)
- preprocessor output
- different std versions
- different compilers
- target architectures
- libraries (e.g., boost)
- filtering options
- control flow graphs
- stack usage
- ctrl + s to save your code
- additional tools (e.g., Sonar)
- ...

Motivation

Motivation

The Compilation Pipeline

The Compilation Pipeline: IDE

The Compilation Pipeline: IDE

- Usually not considered part of the pipeline
- Performance aspect...? 🚴
  - Software-Engineerish reasoning
  - More important in larger projects
  - Must not **slow** work down!
- Examples
  - Show type of auto variables, show errors, auto-completion, fast definition lookup, tell you where you can use const
  - Pipeline orchestration (for large projects)
  - Tooling (tests, debugging, version control, coding guidelines, performance analyzer, ...)
  - ...

The Compilation Pipeline: Preprocessor

The Compilation Pipeline: Preprocessor

- Text replacement (glorified `'s/foo/bar/g'`)
- Driven by preprocessor directives (#)

demo.cpp

```
#define PI 3.1415
#define circleArea(r) (PI *(r)*(r))
int main() {
    // first circle
    auto a1 = circleArea(5);
    // second circle
    auto a2 = circleArea(10);

    return 0;
}
```

g++ -E demo.cpp

```
# 0 "demo.cpp"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "demo.cpp"

int main() {
    auto a1 = (3.1415 *(5)*(5));
    auto a2 = (3.1415 *(10)*(10));

    return 0;
}
```

The Compilation Pipeline: Preprocessor

Use Cases in C

- Symbolic constants: `#define PI 3.1415`
- (Pre-defined) function-like Macros 🚴
  `#define circleArea(r) (PI *(r)*(r))`
- (Recursive) header file inclusions: `#include`
- Include guards: `#pragma once`
- Conditional compilation:
  `#if _WIN32 ... #else ... #endif`
- Passing configuration options
- Compatibility macros
- ...

Why did (do...) people use function-like macros?

- Inlining!

The Compilation Pipeline: Preprocessor

demo.cpp

```cpp
#define PI 3.1415
#define circleArea(r) (PI *(r)*(r))
int main() {
    // first circle
    auto a1 = circleArea(5);
    // second circle
    auto a2 = circleArea(10);

    return 0;
}
```

g++ -S demo.cpp

```
circleArea(double):
    [10 lines omitted]
    ret
main:
    [3 lines omitted]
    mov rax, QWORD PTR .LC1[rip]
    movq xmm0, rax
    call circleArea(double)
    movq rax, xmm0
    mov QWORD PTR [rbp-8], rax
    mov rax, QWORD PTR .LC2[rip]
    movq xmm0, rax
    call circleArea(double)
    movq rax, xmm0
    mov QWORD PTR [rbp-16], rax
    [2 lines omitted]
    ret
    .LC0: ... [2 lines omitted]
    .LC1: ... [2 lines omitted]
    .LC2: ... [2 lines omitted]
```

Potentially expensive function calls!

The Compilation Pipeline: Preprocessor

Calling a function (calling convention) requires...

- ...pushing the return address on the stack
- ...pushing arguments onto the stack/ registers
- ...jumping to the function body
- ...executing the function
- ...executing a return instruction when finished

g++ -S demo.cpp

```
circleArea(double):
    [10 lines omitted]
    ret
main:
    [3 lines omitted]
    mov rax, QWORD PTR .LC1[rip]
    movq xmm0, rax
    call circleArea(double)
    movq rax, xmm0
    mov QWORD PTR [rbp-8], rax
    mov rax, QWORD PTR .LC2[rip]
    movq xmm0, rax
    call circleArea(double)
    movq rax, xmm0
    mov QWORD PTR [rbp-16], rax
    [2 lines omitted]
    ret
    .LC0: ... [2 lines omitted]
    .LC1: ... [2 lines omitted]
    .LC2: ... [2 lines omitted]
```

Potentially expensive function calls!

The Compilation Pipeline: Preprocessor

demo.cpp

```
#define PI 3.1415
#define circleArea(r) (PI *(r)*(r))
int main() {
    // first circle
    auto a1 = circleArea(5);
    // second circle
    auto a2 = circleArea(10);

    return 0;
}
```

g++ -S demo.cpp

```
main:
    push  rbp
    mov   rbp, rsp
    movsd xmm0, QWORD PTR .LC0[rip]
    movsd QWORD PTR [rbp-8], xmm0
    movsd xmm0, QWORD PTR .LC0[rip]
    movsd QWORD PTR [rbp-16], xmm0
    mov   eax, 0
    pop  rbp
    ret
.LC0: ... [2 lines omitted]
.LC1: ... [2 lines omitted]
```

- For 3 multiplications, function call overhead seems to be disproportionate
- Macro forces area caluation "in line" (inlining)

- Macros have certain disadvantages ↖
- *Side quest: what type do a1 and a2 have?*

The Compilation Pipeline: Preprocessor

Use Cases in C

- Symbolic constants: `#define PI 3.1415`
- (Pre-defined) function-like Macros 🚴
  `#define circleArea(r) (PI *(r)*(r))`
- (Recursive) header file inclusions: `#include`
- Include guards: `#pragma once`
- Conditional compilation:
  `#if _WIN32 ... #else ... #endif`
- Passing configuration options
- Compatibility macros
- ...

SYMBOLIC CONSTANTS

- `constexpr double PI = 3.1415;`
- Declares that the evaluation of the variable (or function) is possible at compile time
- Wherever compile time constant expressions are allowed
- Implies constness

HEADER FILE MANAGEMENT

- See C++20 modules ↑

CONDITIONAL COMPILATION

- `if constexpr (DEBUG_MODE) { ... }`
- Discards branches of an if statement at compile time
- Since C++17

The Compilation Pipeline: Preprocessor

Use Cases in C

- Symbolic constants: `#define PI 3.1415`
- (Pre-defined) function-like Macros 🚴
  `#define circleArea(r) (PI *(r)*(r))`
- (Recursive) header file inclusions: `#include`
- Include guards: `#pragma once`
- Conditional compilation:
  `#if _WIN32 ... #else ... #endif`
- Passing configuration options
- Compatibility macros
- ...

INLINING VIA `inline`...?

- "The original intent of the inline keyword was to serve as an indicator to the optimizer that inline substitution of a function is preferred over a function call [...]"
- So, let's
  `inline double circleArea(double r){ ... }`

The Compilation Pipeline: Preprocessor

demo.cpp

```
#define PI 3.1415

inline double circleArea(double r){
    return PI * r * r;
}

int main() {
    // first circle
    auto a1 = circleArea(5);
    // second circle
    auto a2 = circleArea(10);

    return 0;
}
```

Tell the compiler that **you** think inlining is beneficial

g++ -S demo.cpp

```
circleArea(double):
    [10 lines omitted]
    ret
main:
    [3 lines omitted]
    mov rax, QWORD PTR .LC1[rip]
    movq xmm0, rax
    call circleArea(double)
    movq rax, xmm0
    mov QWORD PTR [rbp-8], rax
    mov rax, QWORD PTR .LC2[rip]
    movq xmm0, rax
    call circleArea(double)
    movq rax, xmm0
    mov QWORD PTR [rbp-16], rax
    [2 lines omitted]
    ret
    .LC0: ... [2 lines omitted]
    .LC1: ... [2 lines omitted]
    .LC2: ... [2 lines omitted]
```

Still function calls!

The Compilation Pipeline: Preprocessor

demo.cpp

```cpp
#define PI 3.1415

double circleArea(double r){
    return PI * r * r;
}

int main() {
    // first circle
    auto a1 = circleArea(5);
    // second circle
    auto a2 = circleArea(10);

    return 0;
}
```

Tell the compiler that **you** don't think inlining is beneficial

g++ -S demo.cpp

```asm
main:
    push  rbp
    mov   rbp, rsp
    movsd xmm0, QWORD PTR .LC0[rip]
    movsd QWORD PTR [rbp-8], xmm0
    movsd xmm0, QWORD PTR .LC0[rip]
    movsd QWORD PTR [rbp-16], xmm0
    mov   eax, 0
    pop  rbp
    ret
.LC0: ... [2 lines omitted]
.LC1: ... [2 lines omitted]
```

Compiler inlines!

No actual compiler output. Used for demonstration purpose only and to build up a meme. Your compiler is even smarter.'

The Compilation Pipeline: Preprocessor

demo.cpp

```
#define PI 3.1415

inline double circleArea(double r){
    return PI * r * r;
}

int main() {
    // first circle
    auto a1 = circleArea(5);
    // second circle
    auto a2 = circleArea(10);

    return 0;
}
```

Let the compiler happily **ignore** your humble opinion :-)

- The original intent of the inline keyword was to serve as an indicator to the optimizer that inline substitution of a function is preferred over function call, [...].
  - Compiler w/o optimization does not inline (debugging)
  - Optimizing compilers decide on their own whether to inline a function call or not *(both calls are good candidates here)*
  - `inline` is almost certainly ignored
  - Takeaway: Inlining is good. Inline is not inlining.
  - `inline __attribute__((alwys_inline))` serves as proper hint to the compiler (widely supported)
- Let's run our example with an actual optimizing compiler now...

The Compilation Pipeline: Preprocessor

demo.cpp

```cpp
#define PI 3.1415

inline double circleArea(double r){
    return PI * r * r;
}

int main() {
    // first circle
    auto a1 = circleArea(5);
    // second circle
    auto a2 = circleArea(10);

    return 0;
}
```

g++ -S -O3 demo.cpp

```
main:
    xor    eax, eax
    ret
```

The Compilation Pipeline: Compiler

The Compilation Pipeline: Compiler

INTERPRETED

- Executes instructions written in a another programming or scripting language

JUST-IN-TIME COMPILATION

- Translates bytecode to machine code and immediately executes it
- Continuosly analyses running code, (re-)compiles to gain speedup

AHEAD-OF-TIME COMPILATION

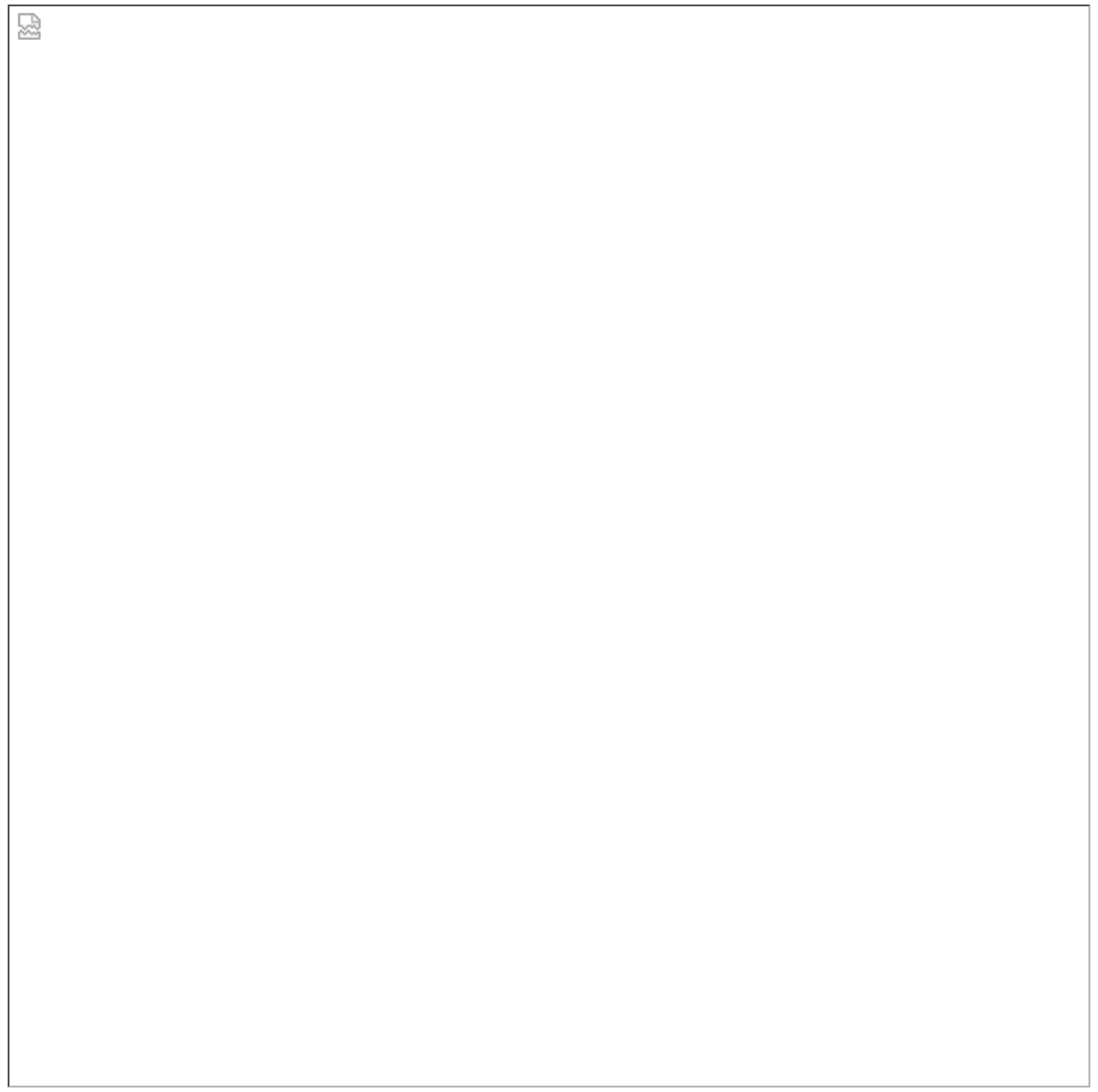
- Compilation happens before execution
- Reduces amount of work needed to be performed at run time

The Compilation Pipeline: Compiler

1. What ProPa teaches you, i.e.,
   1. **Lexical analysis** (turn sequence of symbols into tokens and maintain identifiers)
   2. **Syntactical analysis** (tell if sequence of tokens is a valid word of the context-free language implied by C++)
   3. **Semantic analysis** (context-sensitive analysis of names introduced by declarations, types and language rules)
2. Produce intermediate code, report meaningful errors

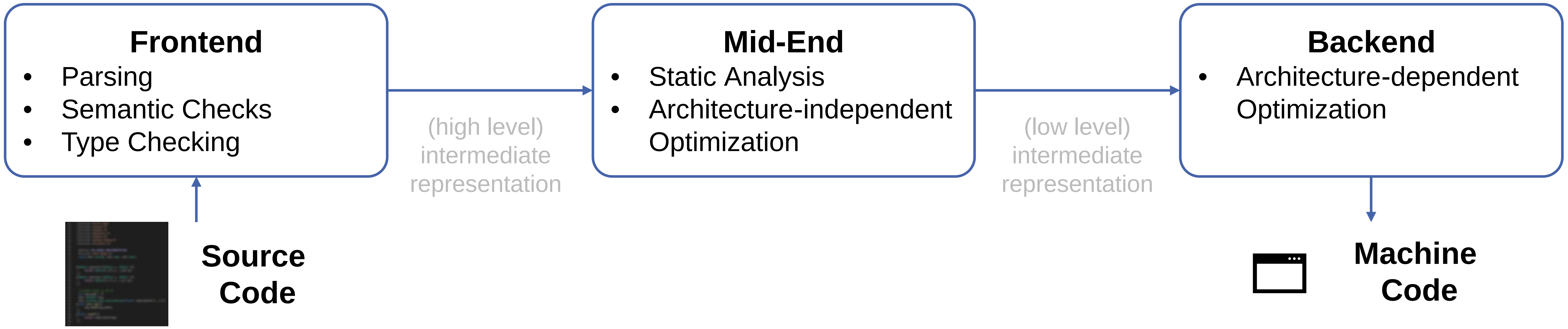
The Compilation Pipeline: Compiler

1. What ProPa teaches you, i.e.,
   1. **Lexical analysis** (turn sequence of symbols into tokens and maintain identifiers)
   2. **Syntactical analysis** (tell if sequence of tokens is a valid word of the context-free language implied by C++)
   3. **Semantic analysis** (context-sensitive analysis of names introduced by declarations, types and language rules)
2. Produce intermediate code, report meaningful errors



Hier schon :-)

The Compilation Pipeline: Compiler



- Mostly irrelevant for optimization
- But you can still have fun with this if that's your cup of tea!
- C++: **Monomorphization** ↗

The Compilation Pipeline: Compiler

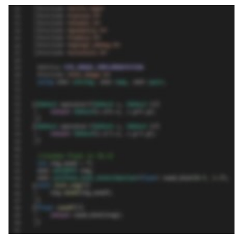
Frontend
- Parsing
- Semantic Checks
- Type Checking

(high level) intermediate representation

Mid-End
- Static Analysis
- Architecture-independent Optimization

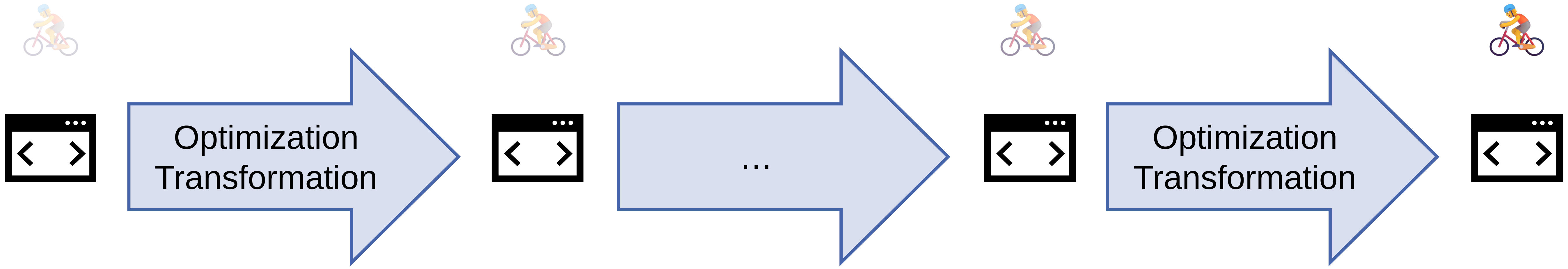
(low level) intermediate representation

Backend
- Architecture-dependent Optimization



Source Code

Machine Code



The Compilation Pipeline: Compiler

- Output must be semantically equivalent to the input (as-if rule)
- C++: Compiler is permitted to perform any changes to the program as long as the "observable behaviour" does not change
  - E.g., regrouping operators (reductions)
  - ```
    auto dummy = (b + c);
    auto assoc = (a + b) + c
    ```
  - Ok if a, b and c are int, forbidden if they are float
- No Elder Wand! *(but a little bit like a magic wand occasionally)*
    ```

The Compilation Pipeline: Compiler

- Output must be semantically equivalent to the input (as-if rule)
- Exceptions to the as-if rule
  - Undefined behaviour ↗
  - Copy elision
  - Return value optimization

```
struct BLOB {        // big large object
    ...
}

BLOB f() {
    return BLOB();  // local, nameless BLOB
}

int main() {
    BLOB b = f();  // local, named BLOB
}
```

The Compilation Pipeline: Compiler

- Output must be semantically equivalent to the input (as-if rule)
- Exceptions to the as-if rule
  - Undefined behaviour ↗
  - Copy elision
  - Return value optimization
- These are **not** semantically equivalent if creation/ destruction of BLOB has sideeffects (custom types!)
  - Explicitly allowed by the standard
  - To disable with gcc: `-fno-elide-constructors`

```
struct BLOB {         // big large object
    ...
}

void f(BLOB* pB) {
    // generate BLOB in f
}

int main() {
    BLOB b = f();   // local, named BLOB
    f(&b);
}
```

The Compilation Pipeline: Compiler

- Local vs. global

```
{
  ...
  double r = 5;

  double a = circleArea(r);

  int a2 = static_cast<int>(a)* 2;
  return a2;
}
```

- All that happens here is transforming (compile-time) constants
- Same as `return 157;` (not refactoring friendly)
- But we cannot necessarily make this assumption if we don't know what circleArea does!

The Compilation Pipeline: Compiler

- Local vs. global
  - Inlining to the rescue!
  - Enables further compiler optimization!
  - Decided per call
- And much, much more! ↗↗↗
- Modern compilers excel at this!

```
{
  ...
  double r = 5;

  double a = circleArea(r);

  int a2 = static_cast<int>(a)* 2;
  return a2;
}
```

The Compilation Pipeline: Compiler

Taming the Beast

- Controlling optimization (gcc)
- -O0: default, no optimization, faster compilation time
- -O1: turn on optimization (predefined set of optimization flags)
- -O2: even more optimization (more flags)
- -O3: even more optimization (more flags)
- -Os: optimize for binary size
- -Ofast: disregards strict standards compliance (e.g., non-associativity of float). You have been warned!
- Fine-tuning with specific options
  - -fno-inline
  - -flop-parallelize-all
  - -fwhole-program
- ...

What could possibly go wrong?

The Compilation Pipeline: Compiler

Taming the Beast

- Takes preprocessor output as input
- Calling the compiler produces an object file (.o)
- Objects (functions, global variables) have symbolic addresses
- Object files can refer to symbols, that are not defined (declaration w/o definition)
- Different "blocks of code" (separate .cpp files) are compiled separately
- Blocks of code = Translation Units
  - Single source file (.cpp)
  - + any header (in-/)directly included (.h)
  - - lines ignored using conditional preprocessing

The Compilation Pipeline: Compiler

Taming the Beast

- Takes preprocessor output as input
- Calling the compiler produces an object file (.o)
- Objects (functions, global variables) have symbolic addresses
- Object files can refer to symbols, that are not defined (declaration w/o definition)
- Different "blocks of code" (separate .cpp files) are compiled separately
- Blocks of code = Translation Units
  - Single source file (.cpp)
  - + any header (in-/)directly included (.h)
  - - lines ignored using conditional preprocessing

The Compilation Pipeline: Compiler

Taming the Beast

- Takes preprocessor output as input
- Calling the compiler produces an object file (.o)
- Objects (functions, global variables) have symbolic addresses
- Object files can refer to symbols, that are not defined (declaration w/o definition)
- Different "blocks of code" (separate .cpp files) are compiled separately
- Blocks of code = Translation Units
  - Single source file (.cpp)
  - + any header (in-/)directly included (.h)
  - - lines ignored using conditional preprocessing

The Compilation Pipeline: Compiler

Taming the Beast

- Takes preprocessor output as input
- Calling the compiler produces an object file (.o)
- Objects (functions, global variables) have symbolic addresses
- Object files can refer to symbols, that are not defined (declaration w/o definition)
- Different "blocks of code" (separate .cpp files) are compiled separately
- Blocks of code = Translation Units
  - Single source file (.cpp)
  - + any header (in-/)directly included (.h)
  - - lines ignored using conditional preprocessing

The Compilation Pipeline: Compiler

Taming the Beast

- Takes preprocessor output as input
- Calling the compiler produces an object file (.o)
- Objects (functions, global variables) have symbolic addresses
- Object files can refer to symbols, that are not defined (declaration w/o definition)
- Different "blocks of code" (separate .cpp files) are compiled separately
- Blocks of code = Translation Units
  - Single source file (.cpp)
  - + any header (in-/)directly included (.h)
  - - lines ignored using conditional preprocessing

The Compilation Pipeline: Compiler

Taming the Beast

- Takes preprocessor output as input
- Calling the compiler produces an object file (.o)
- Objects (functions, global variables) have symbolic addresses
- Object files can refer to symbols, that are not defined (declaration w/o definition)
- Different "blocks of code" (separate .cpp files) are compiled separately
- Blocks of code = Translation Units
  - Single source file (.cpp)
  - + any header (in-/)directly included (.h)
  - - lines ignored using conditional preprocessing

The Compilation Pipeline: Compiler

Taming the Beast

- Takes preprocessor output as input
- Calling the compiler produces an object file (.o)
- Objects (functions, global variables) have symbolic addresses
- Object files can refer to symbols, that are not defined (declaration w/o definition)
- Different "blocks of code" (separate .cpp files) are compiled separately
- Blocks of code = Translation Units
  - Single source file (.cpp)
  - + any header (in-/)directly included (.h)
  - - lines ignored using conditional preprocessing

The Compilation Pipeline: Compiler

Taming the Beast

- Takes preprocessor output as input
- Calling the compiler produces an object file (.o)
- Objects (functions, global variables) have symbolic addresses
- Object files can refer to symbols, that are not defined (declaration w/o definition)
- Different "blocks of code" (separate .cpp files) are compiled separately
- Blocks of code = Translation Units
  - Single source file (.cpp)
  - + any header (in-/)directly included (.h)
  - - lines ignored using conditional preprocessing

The Compilation Pipeline: Compiler

Taming the Beast

- Takes preprocessor output as input
- Calling the compiler produces an object file (.o)
- Objects (functions, global variables) have symbolic addresses
- Object files can refer to symbols, that are not defined (declaration w/o definition)
- Different "blocks of code" (separate .cpp files) are compiled separately
- Blocks of code = Translation Units
  - Single source file (.cpp)
  - + any header (in-/)directly included (.h)
  - - lines ignored using conditional preprocessing

The Compilation Pipeline: Compiler

Taming the Beast

- Takes preprocessor output as input
- Calling the compiler produces an object file (.o)
- Objects (functions, global variables) have symbolic addresses
- Object files can refer to symbols, that are not defined (declaration w/o definition)
- Different "blocks of code" (separate .cpp files) are compiled separately
- Blocks of code = Translation Units
  - Single source file (.cpp)
  - + any header (in-/)directly included (.h)
  - - lines ignored using conditional preprocessing

Header Guards

- Explicitly

```
#ifndef MY_UNIQUE_NAME
#define MY_UNIQUE_NAME

// code goes here

    ...

// endif
```

- Most compilers support `#pragma once`

The Compilation Pipeline: Compiler

Taming the Beast

- Takes preprocessor output as input
- Calling the compiler produces an object file (.o)
- Objects (functions, global variables) have symbolic addresses
- Object files can refer to symbols, that are not defined (declaration w/o definition)
- Different "blocks of code" (separate .cpp files) are compiled separately
- Blocks of code = Translation Units
  - Single source file (.cpp)
  - + any header (in-/)directly included (.h)
  - - lines ignored using conditional preprocessing

Remaining Problem

The Compilation Pipeline: Compiler

Taming the Beast

- Takes preprocessor output as input
- Calling the compiler produces an object file (.o)
- Objects (functions, global variables) have symbolic addresses
- Object files can refer to symbols, that are not defined (declaration w/o definition)
- Different "blocks of code" (separate .cpp files) are compiled separately
- Blocks of code = Translation Units
  - Single source file (.cpp)
  - + any header (in-/)directly included (.h)
  - - lines ignored using conditional preprocessing

Remaining Problem

The Compilation Pipeline: Linker

The Compilation Pipeline: Linker

- Compiler converts source code (.c, .cpp) into (machine-readable) object code (.o)
- Object files are **not** executables
- Example on the right:
  - main.o is from valid C++
  - main.o knows that there is some `void log()` (and how to call it)
  - main.o does **not** know, where to find what `void log()` does (missing binding)
- Linker takes object files and combines them into a single executable (or library file or another object file)
- Maps symbolic addresses to (proper) memory addresses (can involve relocating code)

The Compilation Pipeline: Linker

Static Linking (.a, .lib)

- All objects/ library functions (that are used) are copied into the executable

Dynamic Linking (.so, .dll)

- Executable contains...
  - ...undefined symbols
  - ...list of all objects/ libraries that provide definitions
- Resolving undefined symbols is deferred until runtime

The Compilation Pipeline: Linker

Static Linking (.a, .lib)

- All objects/ library functions (that are used) are copied into the executable
- Advantages
  - Prevents "DLL hell"
  - Not the entire library must be installed if only a few functions are actually needed
  - Portability
- Disadvantages
  - *((Requires more memory))*
- Static linking allows link-time optimization

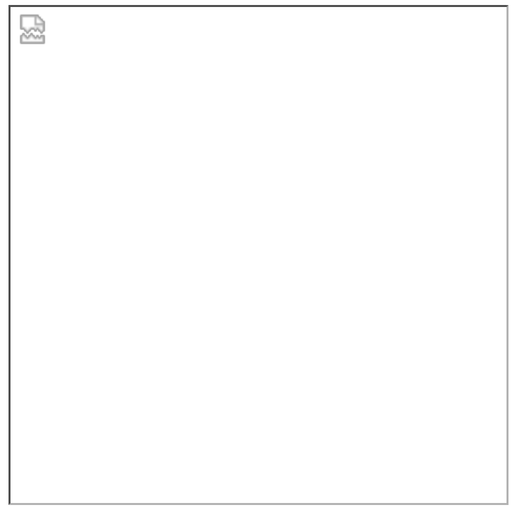
Dynamic Linking (.so, .dll)

- Executable contains...
  - ...undefined symbols
  - ...list of all objects/ libraries that provide definitions
- Resolving undefined symbols is deferred until runtime
- Advantages
  - Often-used libraries (e.g., standard system libraries) are not duplicated
  - No re-linking after, e.g., a bug fix necessary
- Disadvantages
  - Incompatible update libraries can break executables (DLL hell)

The Compilation Pipeline: Linker

Link-time Optimization

- Compiler requires holistic view of involved files to do certain optimizations
  - Inlining across files
  - Removal of unused code across files
- At best: same performance/ optimizations as without linking
- Additionally: information on final object layout can allow micro optimizations (e.g., for jump instructions)



The Compilation Pipeline: Loader

The Compilation Pipeline: Loader

- Why even bother?
- Before execution, a program must be loaded into main memory
- If need be, dynamic linking must be performed
- Binary loaders require absolute binary form of a program (always located in the same memory area)
- Relocating loaders translate addresses
  - Set a relocation bit to each machine language instruction
  - Group all information in a relocation table
- Run program by jumping to the first instruction (main)

The Compilation Pipeline: Summary

ABI

ABI

- API (Application Programming Interface)
  - Source code level
  - Types, structures, functions, ... used to access functionality of external components (libraries, OS functionality, ...)
  - Which functions are part of the library
  - E.g., order of arguments to a function
- ABI (Application Binary Interface)
  - How compiled types, structures, function, ... (binary data) are accessed
  - How code is stored inside the library
  - E.g., how arguments to a function are passed (registers used, stack, ...)
- Whenever ABI changes, all programs using the affected library must be recompiled

ABI

- ABI specifies how C++ compilers compile
  - Example 1: data layout

```
class StdLibraryObj {
private:
    char c;
    int* p;

public:
    void foo(); // API
};

int bar(StdLibraryObj lo) { ... }
// ----------------------------------
int main(){ // App

    ...
    return bar(...);
}
```

ABI

- ABI specifies how C++ compilers compile
  - Example 1: data layout

```
class StdLibraryObj {
private:
    char c;
    int* p;
    float f; // new private member
public:
    void foo(); // API does not change!
};

int bar(StdLibraryObj lo) { ... }
// ----------------------------------
int main(){ // App

    ...
    return bar(...);
}
```

ABI

- ABI specifies how C++ compilers compile
  - Example 1: data layout
  - Example 2: calling conventions

- Low-level rules for function calls
- Calling conventions define...
  - ...where arguments and return values are stored (stack/ registers)
  - ...what is the ordering of arguments
  - ...who is responsible for cleaning up after return
- Performance consideration
  - return value (or pointer to it) in register, arguments on stack
  - vs. return value (or pointer to it) in register **and** (first few) arguments on stack
- Prefer register-passed arguments! 🚴

ABI

- ABI specifies how C++ compilers compile
  - Example 1: data layout
  - Example 2: calling conventions
  - Example 3: std::unique_ptr (register-passed arguments)

- Consider

```
int deref(std::unique_ptr<int> ptr){
    *ptr = 7; // assign 7 to pointee
}
```

- What dereferencing a `std::unique_ptr` should look like:

```
mov      DWORD PTR [rdi], 7
```

  - Keep pointee (`int`) in register (fast 🚴)

- What dereferencing a `int` looks like:

```
mov      rax, QWORD PTR [rdi], 7
mov      QWORD PTR [rdi], 7
```

  - Keep **reference** to stack (pointee) in register
  - Required for "non-trivial" objects in C++ by standard

- Must not mistake pointee and reference (otherwise: random access of stack address!)
- Smart pointers are still preferred for many other reasons ↗

ABI

- ABI specifies how C++ compilers compile
  - Example 1: data layout
  - Example 2: calling conventions
  - Example 3: std::unique_ptr (register-passed arguments)
  - Stack layout and register usage
  - Name mangling
  - Managing template instances
  - Passing of hidden function parameters (this)
  - Throwing and catching exceptions
  - Other standard library details
  - ...

- In hindsight, some standard library features that are ABI dependent cannot be changed
- see std::unique_ptr example
- *Maybe there is an ABI break one day...*

ABI

- There are changes (quality of life, performance, ...) that could be unlocked by an ABI break
  - std::regex
  - container layouts
  - fit std::unique_ptr in register
  - ...
- There are problems that could arise from an ABI break
  - Source code lost, only object files from (now bankrupt third party) exist

ABI

C ABI

- No "de-jure" standard but well-defined on a specific OS and architecture (de-facto standard, e.g., System V)
- Target triplet: architecture, OS, compiler
- "Lingua franca" of programming languages

C++

- Faces more "ABI design decisions" (name mangling, ...)
- Guarantees C ABI for subset of C++ compatible with C
- `extern "C" int f(...){...}` to make function f explicitly have C linkage (for plain old datatypes, PODs)

Other Languages (e.g., Python)

- Internal (not necessarily stable) ABI for intra-language calls
- External ABI for inter-language calls (C ABI)
- For a deeper dive from a Python POV: https://azhpushkin.me/posts/python-c-under-the-hood

Build Systems

Build Systems

- Configuring the build pipeline for large projects is tedious *(debug version(s), release version(s), different target machines, multiple libraries, many source files, testing, corresponding header files, linking everything together,...)*
- Common Issues
  - Hard-coded paths
  - No continuous integration
  - No cross-platform support
  - Incomprehensible project structure
  - ...

- Build systems to the rescue!
  - GNU Make on Linux (Makefiles)
  - Visual Studio Generator on Windows (VS projects)
  - Platform/ compiler-dependent
- Meta build systems to the rescue from the rescue!
  - CMake is de-facto standard for C/C++
  - Supported by most/ all (relevant) IDEs *(even VS)*
  - Modern CMake (3.5+, 3.29+, any version that came out after your compiler)

Sample Project Structure

```
project/

  src/
    engine.cpp
    utils.cpp

  include/          // -> /usr/include
    project/        // library
      engine.h
      utils.h
  apps/
    app1.cpp        // int main(){...}
```

Build Systems

Sample Project Structure

```
project/
  CMakeLists.txt
  src/
    engine.cpp
    utils.cpp

  include/            // -> /usr/include
    project/          // library
      engine.h
      utils.h
  apps/
    app1.cpp          // int main(){...}


  .gitignore          // /build*
  READMDE.md
  LICENSE.md
  cmake/              // helper modules
  docs/               // documentation
  tests/              // tests
  scripts/            // helper.py
  extern/             // git submodules
  python/             // python bindings
```

project/CMakeLists.txt

```
#this is a comment
cmake_minimum_required(VERSION 3.29)

#project name, useful settings
project(
  ExampleProject
  VERSION 0.1
  DESCRIPTION "An example project"
  LANGUAGES CXX)

#for main project: set C++ standard
if(CMAKE_PROJECT_NAME STREQUAL PROJECT_NAME)
  set(CMAKE_CXX_STANDARD C++17)
endif()

#add Boost::boost
#cmake -help-module-list shows all modules
find_package(Boost) #find_package(Boost REQUIRED)
if (NOT Boost_FOUND)
  # show output
  message(WARNING "Boost not found")
endif()

add_subdirectory(src)
add_subdirectory(apps)
```

Build Systems

Sample Project Structure

```
project/
  CMakeLists.txt
  src/
    engine.cpp
    utils.cpp
    CMakeLists.txt
  include/          // -> /usr/include
    project/        // library
      engine.h
      utils.h
  apps/
    app1.cpp        // int main(){...}


  .gitignore        // /build*
  REAMDE.md
  LICENSE.md
  cmake/            // helper modules
  docs/             // documentation
  tests/            // tests
  scripts/          // helper.py
  extern/           // git submodules
  python/           // python bindings
```

project/src/CMakeLists.txt

```
#find header files by globbing expression
#globbing expressions are simplified regexes
file(GLOB HEADER_LIST CONFIGURE_DEPENDS
"${ExampleProject_SOURCE_DIR}/include/project/*.h")

#build library (dynamic/ static based on user #setting)
as a first target
add_library(my_lib engine.cpp utils.cpp ${HEADER_LIST}

#include
target_include_directories(my_lib PUBLIC ../include)

#depend on other (header-only) library
target_link_libraries(my_lib PRIVATE Boost::boost)
```

Build Systems

Sample Project Structure

```
project/
  CMakeLists.txt
  src/
    engine.cpp
    utils.cpp
    CMakeLists.txt
  include/          // -> /usr/include
    project/        // library
      engine.h
      utils.h
  apps/
    app1.cpp        // int main(){...}
    CMakeLists.txt

  .gitignore        // /build*
  REAMDE.md
  LICENSE.md
  cmake/            // helper modules
  docs/             // documentation
  tests/            // tests
  scripts/          // helper.py
  extern/           // git submodules
  python/           // python bindings
```

project/apps/CMakeLists.txt

```
add_executable(my_app app1.cpp)

#optional pre-processor definitions
option(USE_EXPERIMENTAL "Enable experimental
stuff" OFF)

#(platform-dependent) compile options
if(CMAKE_SYSTEM_NAME STREQUAL "Windows")
  target_compile_options(my_app PRIVATE /W4)
elseif(CMAKE_SYSTEM_NAME STREQUAL "Linux")
  target_compile_options(my_app PRIVATE -Wall -
                         Wextra -Wpedantic)
endif()

link with previously targeted library
#target_link_options(my_app ...)
target_link_libraries(app1 PRIVATE my_lib
  fmt::fmt)
```

Build Systems

Sample Project Structure

```
project/
  CMakeLists.txt
  src/
    engine.cpp
    utils.cpp
    CMakeLists.txt
  include/          // -> /usr/include
    project/        // library
      engine.h
      utils.h
  apps/
    app1.cpp        // int main(){...}
    CMakeLists.txt
  build/
  .gitignore        // /build*
  REAMDE.md
  LICENSE.md
  cmake/            // helper modules
  docs/             // documentation
  tests/            // tests
  scripts/          // helper.py
  extern/           // git submodules
  python/           // python bindings
```

Build Project

- Run cmake from CMakeLists.txt in root directory and build in current directory
  - `cmake -B build && cmake --build build`
- `cmake -DUSE_EXPERIMENTAL=ON ..`
- `cmake -DCMAKE_BUILD_TYPE=Debug ..`
- `cmake -DCMAKE_BUILD_TYPE=Release ..`
- CMakeCache.txt
  - CMake caches user configurable options after first build configuration
  - `cmake --target clean`
- CMake also features testing, ...

Takeaways

Takeaways

- Compiler turns human-readable code (.cpp files, .h files) into machine-readable code
- Must obey as-(almost)-if rule
- Compilation pipeline consists of
  - (IDE)
  - Preprocessor
  - Compiler: Frontend, Mid-End, Backend
  - Linker
  - (Loader)
- Compiler and Linker are capable of compile-time and link-time optimizations
- CMake as meta-build-system to automate the build process
- ABI stability limits optimization potential (but an ABI break is not trivial)

Thank You!