```cpp
#include <cstdlib>
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
    return EXIT_SUCCESS;
}
```

Introduction to C++

C++ Fundamentals

Agenda

A Bit about C++

C++ Background
- Timeline
- Core Design Principles

C++ Design
- Undefined Behavior
- Turning C++ into Machine Code
- Language Type
- Static Types and Zero-cost Abstractions

Tools

A Bit about C++

What is C++?

C++ is a compiled, statically-typed, general-purpose programming language.

What is C++?

C++ is a compiled, statically-typed, general-purpose programming language.

What is C++?

C++ is a compiled, statically-typed, general-purpose programming language.

What is C++?

C++ is a compiled, statically-typed, general-purpose programming language.

What is C++?

C++ is a compiled, statically-typed, general-purpose programming language.

What is C++?

C++ is a compiled, statically-typed, general-purpose programming language.

What is C++?

C++ is a compiled, statically-typed, general-purpose programming language.

Fixed data types

What is C++?

C++ is a compiled, statically-typed, general-purpose programming language.

What is C++?

C++ is a compiled, statically-typed, general-purpose programming language.

Create many types of applications

What is C++?

C++ is a compiled, statically-typed, general-purpose programming language.

Create many types of applications

- Object-oriented programming (OOP)

What is C++?

C++ is a compiled, statically-typed, general-purpose programming language.

Create many types of applications

- Object-oriented programming (OOP)
- Imperative programming

What is C++?

C++ is a compiled, statically-typed, general-purpose programming language.

Create many types of applications

- Object-oriented programming (OOP)
- Imperative programming
- Functional programming

C++ Background

C++ Development (1979 – 1983)

C++ Development (1979 – 1983)

- 1979: Developed by Danish computer scientist Bjarne Stroustrup at Bell Labs

C++ Development (1979 – 1983)

- 1979: Developed by Danish computer scientist Bjarne Stroustrup at Bell Labs
- Extension for “The C Programming Language”

C++ Development (1979 – 1983)

- 1979: Developed by Danish computer scientist Bjarne Stroustrup at Bell Labs
- Extension for “The C Programming Language”
  - “C with Classes”

C++ Development (1979 – 1983)

- 1979: Developed by Danish computer scientist Bjarne Stroustrup at Bell Labs
- Extension for “The C Programming Language”
  - “C with Classes”
  - “New C”

C++ Development (1979 – 1983)

- 1979: Developed by Danish computer scientist Bjarne Stroustrup at Bell Labs
- Extension for “The C Programming Language”
  - “C with Classes”
  - “New C”
- Heavily influenced by Simula

C++ Development (1979 – 1983)

- 1979: Developed by Danish computer scientist Bjarne Stroustrup at Bell Labs
- Extension for “The C Programming Language”
  - “C with Classes”
  - “New C”
- Heavily influenced by Simula
- 1983: Released under the name “C++”

Early C++ (1985 – 1989)

Early C++ (1985 – 1989)

- 1985: First edition of “The C++ Programming Language” released

Early C++ (1985 – 1989)

- 1985: First edition of “The C++ Programming Language” released
- 1989: C++ 2.0 released

Standardized C++ (1998 – 2014)

Standardized C++ (1998 – 2014)

- 1998: C++98 released

Standardized C++ (1998 – 2014)

- 1998: C++98 released
  - First standardized version

Standardized C++ (1998 – 2014)

- 1998: C++98 released
  - First standardized version
  - “Standard template library” (STL)

Standardized C++ (1998 – 2014)

- 1998: C++98 released
  - First standardized version
  - “Standard template library” (STL)
- 2003: C++03 released (minor update)

Standardized C++ (1998 – 2014)

- 1998: C++98 released
  - First standardized version
  - “Standard template library” (STL)
- 2003: C++03 released (minor update)
- 2011: C++11 released (major update)

Standardized C++ (1998 – 2014)

- 1998: C++98 released
  - First standardized version
  - “Standard template library” (STL)
- 2003: C++03 released (minor update)
- 2011: C++11 released (major update)
  - Start of three-year cycle

Standardized C++ (1998 – 2014)

- 1998: C++98 released
  - First standardized version
  - “Standard template library” (STL)
- 2003: C++03 released (minor update)
- 2011: C++11 released (major update)
  - Start of three-year cycle
- 2014: C++14 released (minor update)

Modern C++ (2017 – Present)

Modern C++ (2017 – Present)

- 2017: C++17 released (major update)

Modern C++ (2017 – Present)

- 2017: C++17 released (major update)
- 2020: C++20 released (major update)

Modern C++ (2017 – Present)

- 2017: C++17 released (major update)
- 2020: C++20 released (major update)
    - Modules

Modern C++ (2017 – Present)

- 2017: C++17 released (major update)
- 2020: C++20 released (major update)
  - Modules
- 2023: C++23 released (major update)

Modern C++ (2017 – Present)

- 2017: C++17 released (major update)
- 2020: C++20 released (major update)
  - Modules
- 2023: C++23 released (major update)
  - Better printing

Modern C++ (2017 – Present)

- 2017: C++17 released (major update)
- 2020: C++20 released (major update)
  - Modules
- 2023: C++23 released (major update)
  - Better printing
- 2026: C++26 will be released (major update)

Modern C++ (2017 – Present)

- 2017: C++17 released (major update)
- 2020: C++20 released (major update)
  - Modules
- 2023: C++23 released (major update)
  - Better printing
- 2026: C++26 will be released (major update)
  - Currently in development (C++2c)

Modern C++ (2017 – Present)

- 2017: C++17 released (major update)
- 2020: C++20 released (major update)
  - Modules
- 2023: C++23 released (major update)
  - Better printing
- 2026: C++26 will be released (major update)
  - Currently in development (C++2c)
  - Reflection

Core Design Principles

Core Design Principles

- It must be driven by actual problems, and its features should be immediately useful in real world programs.

Core Design Principles

- It must be driven by actual problems, and its features should be immediately useful in real world programs.
- Every feature should be implementable (with a reasonably obvious way to do so).

Core Design Principles

- It must be driven by actual problems, and its features should be immediately useful in real world programs.
- Every feature should be implementable (with a reasonably obvious way to do so).
- Programmers should be free to pick their own programming style, and that style should be fully supported by C++.

Core Design Principles

- It must be driven by actual problems, and its features should be immediately useful in real world programs.
- Every feature should be implementable (with a reasonably obvious way to do so).
- Programmers should be free to pick their own programming style, and that style should be fully supported by C++.
- Allowing a useful feature is more important than preventing every possible misuse of C++.

Core Design Principles

- It should provide facilities for organizing programs into separate, well-defined parts, and provide facilities for combining separately developed parts.

Core Design Principles

- It should provide facilities for organizing programs into separate, well-defined parts, and provide facilities for combining separately developed parts.
- No implicit violations of the type system (but allow explicit violations; that is, those explicitly requested by the programmer).

Core Design Principles

- It should provide facilities for organizing programs into separate, well-defined parts, and provide facilities for combining separately developed parts.
- No implicit violations of the type system (but allow explicit violations; that is, those explicitly requested by the programmer).
- User-created types need to have the same support and performance as built-in types.

Core Design Principles

- It should provide facilities for organizing programs into separate, well-defined parts, and provide facilities for combining separately developed parts.
- No implicit violations of the type system (but allow explicit violations; that is, those explicitly requested by the programmer).
- User-created types need to have the same support and performance as built-in types.
- Unused features should not negatively impact created executables (e.g., in lower performance).

Core Design Principles

- There should be no language beneath C++ (except Assembly language).

Core Design Principles

- There should be no language beneath C++ (except Assembly language).
- C++ should work alongside other existing programming languages, rather than fostering its own separate and incompatible programming environment.

Core Design Principles

- There should be no language beneath C++ (except Assembly language).
- C++ should work alongside other existing programming languages, rather than fostering its own separate and incompatible programming environment.
- If the programmer's intent is unknown, allow the programmer to specify it by providing manual control.

Does anything stand out?

C++ Design

Undefined Behavior

Undefined Behavior

Behavior that has not been defined by the language specification.

Undefined Behavior

> Behavior that has not been defined by the language specification.

Implications?

Undefined Behavior

> Behavior that has not been defined by the language specification.

Implications?

- Code may compile, and it may seem like it's working, but it may crash unexpectedly

Undefined Behavior

> Behavior that has not been defined by the language specification.

Implications?

- Code may compile, and it may seem like it's working, but it may crash unexpectedly
- Compiler makes no guarantees about what will happen

Undefined Behavior

> Behavior that has not been defined by the language specification.

Implications?

- Code may compile, and it may seem like it's working, but it may crash unexpectedly
- Compiler makes no guarantees about what will happen
- Must be avoided

Why allow this?

Performance

Why Undefined Behavior?

Why Undefined Behavior?

It’s faster for the compiler to not worry about every little thing that could go wrong.

Why Undefined Behavior?

It's faster for the compiler to not worry about every little thing that could go wrong.

The C motto

Trust the programmer.

Why Undefined Behavior?

It's faster for the compiler to not worry about every little thing that could go wrong.

The C motto

Trust the programmer.

Is this acceptable?

Undefined Behavior and Modern C++

Undefined Behavior and Modern C++

- Modern solutions which avoid undefined behavior

Undefined Behavior and Modern C++

- Modern solutions which avoid undefined behavior
- Undefined behavior can be opted into

Undefined Behavior and Modern C++

- Modern solutions which avoid undefined behavior
- Undefined behavior can be opted into
- Compiler warnings

Compiled Languages

Compiled Languages

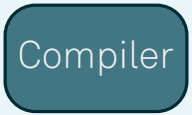
C++ is compiled.

Compiled Languages

C++ is compiled. What does the compiler do?

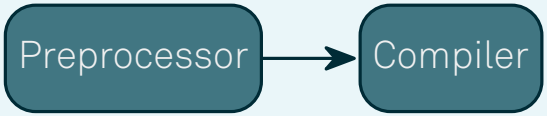
Compiled Languages

C++ is compiled. What does the compiler do?



Compiled Languages

C++ is compiled. What does the compiler do?

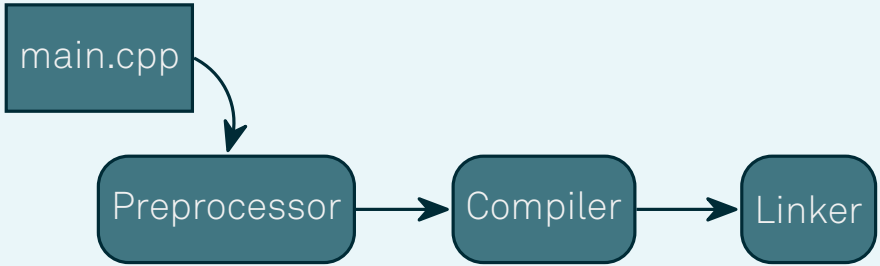


Compiled Languages

C++ is compiled. What does the compiler do?

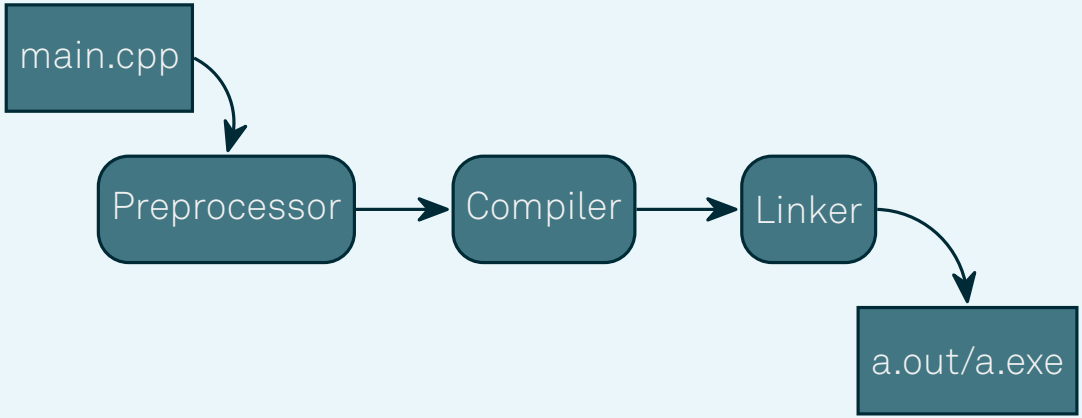
Compiled Languages

C++ is compiled. What does the compiler do?



Compiled Languages

C++ is compiled. What does the compiler do?



Machine Code

```
cffa edfe 0700 0001 0300 0000 0200 0000
1100 0000 d805 0000 8580 2100 0000 0000
1900 0000 4800 0000 5f5f 5041 4745 5a45
524f 0000 0000 0000 0000 0000 0000 0000
...
6c65 3500 4743 435f 6578 6365 7074 5f74
6162 6c65 3700 4743 435f 6578 6365 7074
5f74 6162 6c65 3336 0047 4343 5f65 7863
6570 745f 7461 626c 6534 3100 0000 0000
```

Machine code which prints “Hello, world!”.

Assembly Language

```
global _start

section .text

_start:
    mov rax, 1          ; write(
    mov rdi, 1          ;   STDOUT_FILENO,
    mov rsi, msg        ;   "Hello, world!\n",
    mov rdx, msglen     ;   sizeof("Hello, world!\n")
    syscall             ; );

    mov rax, 60         ; exit(
    mov rdi, 0          ;   EXIT_SUCCESS
    syscall             ; );

section .rodata
    msg: db "Hello, world!", 10
    msglen: equ $ - msg
```

x86_64 Assembly which prints “Hello, world!”

Assembly Language

- Language with close correspondence to machine code

```
global _start

section .text

_start:
    mov rax, 1          ; write(
    mov rdi, 1          ;   STDOUT_FILENO,
    mov rsi, msg        ;   "Hello, world!\n",
    mov rdx, msglen     ;   sizeof("Hello, world!\n")
    syscall             ; );

    mov rax, 60         ; exit(
    mov rdi, 0          ;   EXIT_SUCCESS
    syscall             ; );

section .rodata
    msg: db "Hello, world!", 10
    msglen: equ $ - msg
```

x86_64 Assembly which prints “Hello, world!”

Assembly Language

- Language with close correspondence to machine code
- Human-readable machine code

```
global _start

section .text

_start:
    mov rax, 1          ; write(
    mov rdi, 1          ;   STDOUT_FILENO,
    mov rsi, msg        ;   "Hello, world!\n",
    mov rdx, msglen     ;   sizeof("Hello, world!\n")
    syscall             ; );

    mov rax, 60         ; exit(
    mov rdi, 0          ;   EXIT_SUCCESS
    syscall             ; );

section .rodata
    msg: db "Hello, world!", 10
    msglen: equ $ - msg
```

x86_64 Assembly which prints “Hello, world!”

What Type of Language Is C++?

What Type of Language Is C++?

C++ is a *systems programming language*.

What Type of Language Is C++?

C++ is a *systems programming language*.

What is a systems programming language?

A programming language designed for programming systems rather than applications.

What Type of Language Is C++?

C++ is a *systems programming language*.

What is a systems programming language?

A programming language designed for programming systems rather than applications.

- Close to the hardware

What Type of Language Is C++?

C++ is a *systems programming language*.

What is a systems programming language?

A programming language designed for programming systems rather than applications.

- Close to the hardware
- More control over program details

What Type of Language Is C++?

C++ is a *systems programming language*.

What is a systems programming language?

A programming language designed for programming systems rather than applications.

- Close to the hardware
- More control over program details

C++ is a *middle-level language* compared to C and a *high-level language* compared to Assembly language.

Language-level Breakdown

Language-level Breakdown

- Relative to Assembly language

Language-level Breakdown

- Relative to Assembly language
  - Low level (machine code, Assembly)

Language-level Breakdown

- Relative to Assembly language
  - Low level (machine code, Assembly)
    - “Close to the hardware”

Language-level Breakdown

- Relative to Assembly language
  - Low level (machine code, Assembly)
    - “Close to the hardware”
    - Specific control over hardware (stack)

Language-level Breakdown

- Relative to Assembly language
  - Low level (machine code, Assembly)
    - “Close to the hardware”
    - Specific control over hardware (stack)
    - Hard to read

Language-level Breakdown

- Relative to Assembly language
  - Low level (machine code, Assembly)
    - “Close to the hardware”
    - Specific control over hardware (stack)
    - Hard to read
  - High level (BASIC, C, C++, COBOL, Fortran)

Language-level Breakdown

- Relative to Assembly language
  - Low level (machine code, Assembly)
    - “Close to the hardware”
    - Specific control over hardware (stack)
    - Hard to read
  - High level (BASIC, C, C++, COBOL, Fortran)
    - Add abstractions

Language-level Breakdown

- Relative to Assembly language
  - Low level (machine code, Assembly)
    - “Close to the hardware”
    - Specific control over hardware (stack)
    - Hard to read
  - High level (BASIC, C, C++, COBOL, Fortran)
    - Add abstractions
    - Automate some control over hardware (stack)

Language-level Breakdown

- Relative to Assembly language
  - Low level (machine code, Assembly)
    - “Close to the hardware”
    - Specific control over hardware (stack)
    - Hard to read
  - High level (BASIC, C, C++, COBOL, Fortran)
    - Add abstractions
    - Automate some control over hardware (stack)
    - Use natural-language elements

Language-level Breakdown

- Relative to Assembly language
  - Low level (machine code, Assembly)
    - “Close to the hardware”
    - Specific control over hardware (stack)
    - Hard to read
  - High level (BASIC, C, C++, COBOL, Fortran)
    - Add abstractions
    - Automate some control over hardware (stack)
    - Use natural-language elements
  - Very high level (Java, JavaScript, Python, Ruby, C#)

Language-level Breakdown

- Relative to Assembly language
  - Low level (machine code, Assembly)
    - “Close to the hardware”
    - Specific control over hardware (stack)
    - Hard to read
  - High level (BASIC, C, C++, COBOL, Fortran)
    - Add abstractions
    - Automate some control over hardware (stack)
    - Use natural-language elements
  - Very high level (Java, JavaScript, Python, Ruby, C#)
    - Add many abstractions

Language-level Breakdown

- Relative to Assembly language
  - Low level (machine code, Assembly)
    - “Close to the hardware”
    - Specific control over hardware (stack)
    - Hard to read
  - High level (BASIC, C, C++, COBOL, Fortran)
    - Add abstractions
    - Automate some control over hardware (stack)
    - Use natural-language elements
  - Very high level (Java, JavaScript, Python, Ruby, C#)
    - Add many abstractions
    - Remove details about hardware control

Language-level Breakdown

- Relative to C

Language-level Breakdown

- Relative to C
    - Low level (C)

Language-level Breakdown

- Relative to C
  - Low level (C)
    - A lot of hardware control

Language-level Breakdown

- Relative to C
  - Low level (C)
    - A lot of hardware control
    - Not many abstractions

Language-level Breakdown

- Relative to C
  - Low level (C)
    - A lot of hardware control
    - Not many abstractions
  - Middle level (C++)

Language-level Breakdown

- Relative to C
  - Low level (C)
    - A lot of hardware control
    - Not many abstractions
  - Middle level (C++)
    - A blend of abstractions and hardware control

Language-level Breakdown

- Relative to C
  - Low level (C)
    - A lot of hardware control
    - Not many abstractions
  - Middle level (C++)
    - A blend of abstractions and hardware control
    - “Smart” memory management

Language-level Breakdown

- Relative to C
  - Low level (C)
    - A lot of hardware control
    - Not many abstractions
  - Middle level (C++)
    - A blend of abstractions and hardware control
    - “Smart” memory management
  - High level (Python, Java)

Language-level Breakdown

- Relative to C
  - Low level (C)
    - A lot of hardware control
    - Not many abstractions
  - Middle level (C++)
    - A blend of abstractions and hardware control
    - “Smart” memory management
  - High level (Python, Java)
    - Many abstractions

Language-level Breakdown

- Relative to C
  - Low level (C)
    - A lot of hardware control
    - Not many abstractions
  - Middle level (C++)
    - A blend of abstractions and hardware control
    - “Smart” memory management
  - High level (Python, Java)
    - Many abstractions
    - Automated memory management (garbage collector)

Why Static Types?

Why Static Types?

- Data types are important to programmers

Why Static Types?

- Data types are important to programmers
- Type errors are caught at *compile-time*, not runtime.

Why Static Types?

- Data types are important to programmers
- Type errors are caught at *compile-time*, not runtime.
- Meaning of code is clearer.

Why Static Types?

- Data types are important to programmers
- Type errors are caught at *compile-time*, not runtime.
- Meaning of code is clearer.
- What do Python and JavaScript do?

Why Static Types?

- Data types are important to programmers
- Type errors are caught at *compile-time*, not runtime.
- Meaning of code is clearer.
- What do Python and JavaScript do?
    - mypy

Why Static Types?

- Data types are important to programmers
- Type errors are caught at *compile-time*, not runtime.
- Meaning of code is clearer.
- What do Python and JavaScript do?
  - mypy
  - TypeScript/JSDoc comments

Zero-cost Abstractions

Zero-cost Abstractions

Code using abstractions should be as fast as the same code without abstractions

Zero-cost Abstractions

> Code using abstractions should be as fast as the same code without abstractions

Whether we choose to take advantage of abstraction, our code will still be as efficient.

Tools

Our Tools

Our Tools

Compiler

Our Tools

Compiler

- LLVM (Clang)

Our Tools

Compiler

- LLVM (Clang)
- GCC

Our Tools

Compiler

- LLVM (Clang)
- GCC
- MSVC

Our Tools

Compiler

- LLVM (Clang)
- GCC
- MSVC

Meta build system

Our Tools

Compiler

- LLVM (Clang)
- GCC
- MSVC

Meta build system

- CMake

Our Tools

Compiler

- LLVM (Clang)
- GCC
- MSVC

Meta build system

- CMake
- Meson

Our Tools

Compiler

- LLVM (Clang)
- GCC
- MSVC

Meta build system

- CMake
- Meson
- Bazel

Our Tools

Compiler

- LLVM (Clang)
- GCC
- MSVC

Meta build system

- CMake
- Meson
- Bazel

Build system

Our Tools

Compiler

- LLVM (Clang)
- GCC
- MSVC

Meta build system

- CMake
- Meson
- Bazel

Build system

- Make

Our Tools

Compiler

- LLVM (Clang)
- GCC
- MSVC

Meta build system

- CMake
- Meson
- Bazel

Build system

- Make
- Ninja

Our Tools

Compiler

- **LLVM (Clang)**
- GCC
- MSVC

Meta build system

- CMake
- Meson
- Bazel

Build system

- Make
- Ninja

Our Tools

Compiler

- **LLVM (Clang)**
- GCC
- MSVC

Meta build system

- **CMake**
- Meson
- Bazel

Build system

- Make
- Ninja

Our Tools

Compiler

- **LLVM (Clang)**
- GCC
- MSVC

Meta build system

- **CMake**
- Meson
- Bazel

Build system

- Make
- **Ninja**