Knockin' on Header's Door

An Overview of C++ Modules

ALEXSANDRO THOMAS

CppCon
The C++ Conference
2025
September 13 - 19

About me

Alexsandro Thomas

Senior Software Engineer

Languages nerd, GPGPU enthusiast

alexsandrothomas

northy

Slides URL: northy.github.io/knockin-on-headers-door

Why Modules?

Let’s step back

main.cpp

```cpp
int foo(const int x)
{
    return x - 42;
}

int main()
{
    return foo(42);
}
```

Single source C++ programs


```
g++ -c main.cpp
g++ main.o -o main
```

main.cpp

```cpp
int foo(const int x)
{
    return x - 42;
}

int main()
{
    return foo(42);
}
```

main.cpp

```cpp
int main()
{
    return foo(42);
}
```

lib.cpp

```cpp
int foo(const int x)
{
    return x - 42;
}
```

main.cpp

```cpp
int main()
{
    return foo(42);
}
```

lib.cpp

```cpp
int foo(const int x)
{
    return x - 42;
}
```

```
error: ‘foo’ was not declared in this scope
```

main.cpp

```cpp
int foo(const int x);

int main()
{
    return foo(42);
}
```

lib.cpp

```cpp
int foo(const int x)
{
    return x - 42;
}
```

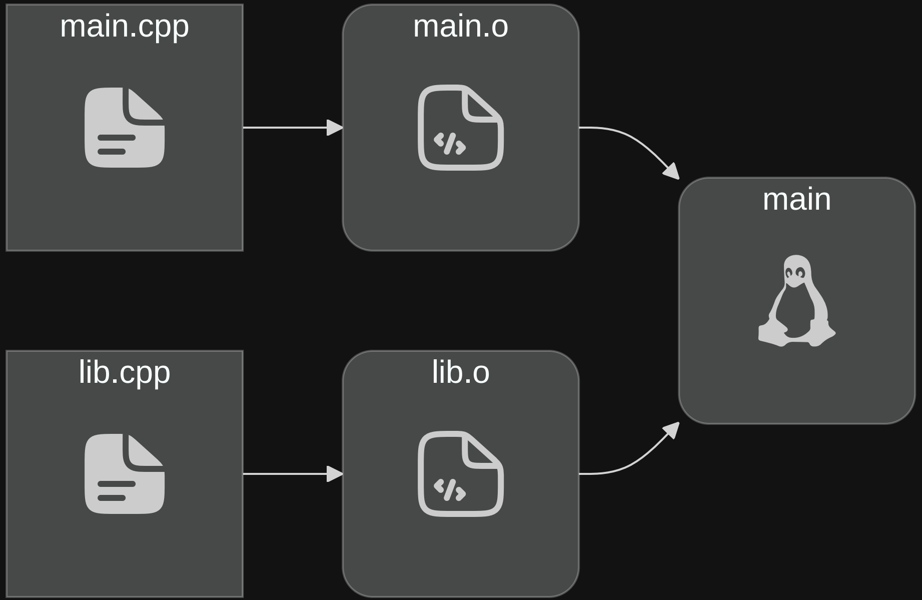
main.cpp

```cpp
int foo(const int x);

int main()
{
    return foo(42);
}
```

lib.cpp

```cpp
int foo(const int x);

int foo(const int x)
{
    return x - 42;
}
```

Multiple source C++ programs




```
g++ lib.cpp -c
g++ main.cpp -c
g++ main.o lib.o -o main
```

Multiple source C++ programs

- Source files → Translation Units (TUs)
- Each name is declared in every TU it is used
- Linker finds suitable definitions
- Inconsistencies will cause errors or unintended behavior

lib.cpp

```cpp
int foo(const int x);

int foo(const int x)
{
    return x - 42;
}
```

main.cpp

```cpp
int foo(const int x);

int main()
{
    return foo(42);
}
```

lib.cpp

```cpp
int foo(const int x, const int y = 42);

int foo(const int x, const int y)
{
    return x - y;
}
```

main.cpp

```cpp
int foo(const int x);

int main()
{
    return foo(42);
}
```

lib.cpp

```cpp
int foo(const int x, const int y = 42);

int foo(const int x, const int y)
{
    return x - y;
}
```

main.cpp

```cpp
int foo(const int x);

int main()
{
    return foo(42);
}
```

```
error LNK2019: unresolved external symbol "int __cdecl foo(int)" referenced in function main
```

lib.cpp

```cpp
int foo(const int x);

int foo(const int x)
{
    return x - 42;
}
```

main.cpp

```cpp
int foo(const int x);

int main()
{
    return foo(42);
}
```

lib.h

```cpp
int foo(const int x);
```

lib.cpp

```cpp
int foo(const int x);

int foo(const int x)
{
    return x - 42;
}
```

main.cpp

```cpp
int foo(const int x);

int main()
{
    return foo(42);
}
```

lib.h

```cpp
int foo(const int x);
```

lib.cpp

```cpp
#include "lib.h"

int foo(const int x)
{
    return x - 42;
}
```

main.cpp

```cpp
#include "lib.h"

int main()
{
    return foo(42);
}
```

lib.h

```cpp
int foo(const int x);
```

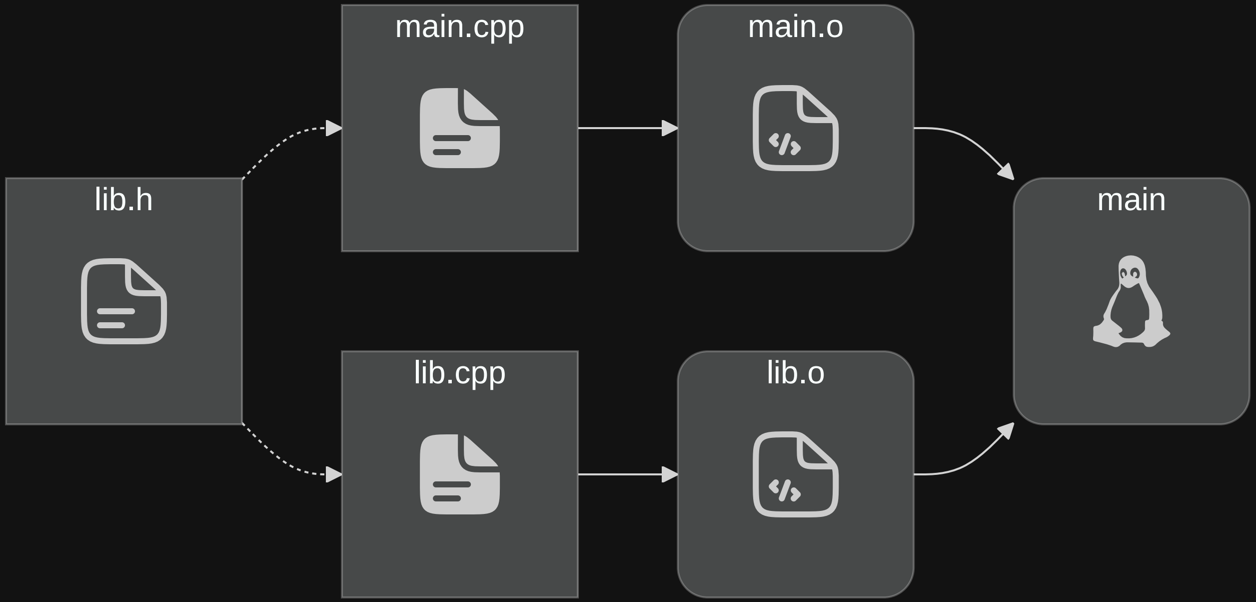
lib.cpp

```cpp
#include "lib.h"

int foo(const int x)
{
    return x - 42;
}
```

main.cpp

```cpp
#include "lib.h"

int main()
{
    return foo(42);
}
```

Compilation of many source files with headers




```
g++ lib.cpp -c
g++ main.cpp -c
g++ main.o lib.o -o main
```

One Definition Rule (ODR)

- Each name must be declared in every TU it is used

One Definition Rule (ODR)

- Each name must be declared in every TU it is used
- Any entity defined exactly once

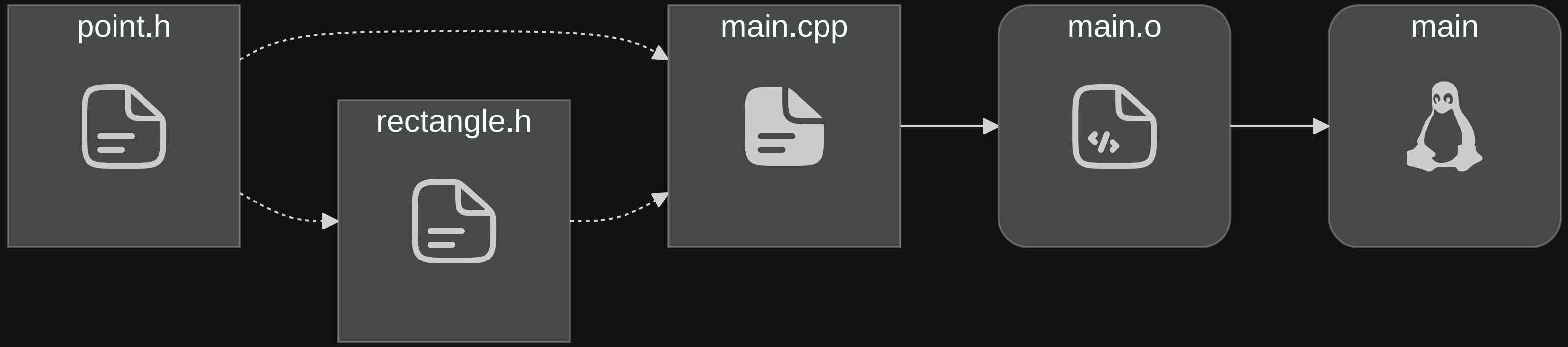
One Definition Rule (ODR)

- Each name must be declared in every TU it is used
- Any entity defined exactly once
  - Only one definition is allowed in any one TU

"Every word in the C and C++ definitions about 'ODR' are there to work around the fact that we cannot identity the one true definition and have to compare definitions instead"

- Bjarne Stroustrup (qtd. in Dos Reis, Hall, and Nishanov)

Compilation of many source files with headers



point.h

```
struct Point
{
    int x, y;
};
```

point.h

```c
struct Point
{
    int x, y;
};
```

rectangle.h

```c
#include "point.h"

struct Rectangle
{
    Point topLeft, bottomRight;
};
```

point.h

```c
struct Point
{
    int x, y;
};
```

rectangle.h

```c
#include "point.h"

struct Rectangle
{
    Point topLeft, bottomRight;
};
```

point.h

```c
struct Point
{
    int x, y;
};
```

rectangle.h

```c
#include "point.h"

struct Rectangle
{
    Point topLeft, bottomRight;
};
```

point.h

```cpp
struct Point
{
    int x, y;
};
```

rectangle.h

```cpp
#include "point.h"

struct Rectangle
{
    Point topLeft, bottomRight;
};
```

main.cpp

```cpp
#include "point.h"
#include "rectangle.h"

int main()
{
    const Point p1{1, 2}, p2{2, 4};
    const Rectangle r{p1, p2};
}
```

point.h

```cpp
struct Point
{
    int x, y;
};
```

rectangle.h

```cpp
#include "point.h"

struct Rectangle
{
    Point topLeft, bottomRight;
};
```

main.cpp

```cpp
#include "point.h"
#include "rectangle.h"

int main()
{
    const Point p1{1, 2}, p2{2, 4};
    const Rectangle r{p1, p2};
}
```

point.h

```cpp
struct Point
{
    int x, y;
};
```

rectangle.h

```cpp
#include "point.h"

struct Rectangle
{
    Point topLeft, bottomRight;
};
```

main.cpp

```cpp
#include "point.h"
#include "rectangle.h"

int main()
{
    const Point p1{1, 2}, p2{2, 4};
    const Rectangle r{p1, p2};
}
```

main.cpp

```cpp
#include "point.h"
#include "rectangle.h"

int main()
{
    const Point p1{1, 2}, p2{2, 4};
    const Rectangle r{p1, p2};
}
```

main.cpp

```cpp
#include "point.h"
#include "rectangle.h"

int main()
{
    const Point p1{1, 2}, p2{2, 4};
    const Rectangle r{p1, p2};
}
```

Pre-processing `main.cpp` ...

main.cpp

```cpp
#include "point.h"
#include "rectangle.h"

int main()
{
    const Point p1{1, 2}, p2{2, 4};
    const Rectangle r{p1, p2};
}
```

Pre-processing `main.cpp` ...

main.cpp

```cpp
#include "point.h"
#include "point.h"

struct Rectangle
{
    Point topLeft, bottomRight;
};

int main()
{
    const Point p1{1, 2}, p2{2, 4};
    const Rectangle r{p1, p2};
}
```

Pre-processing `main.cpp` ...

main.cpp

```cpp
#include "point.h"
#include "point.h"

struct Rectangle
{
    Point topLeft, bottomRight;
};

int main()
{
    const Point p1{1, 2}, p2{2, 4};
    const Rectangle r{p1, p2};
}
```

Pre-processing `main.cpp` ...

main.cpp

```cpp
struct Point
{
    int x, y;
};
struct Point
{
    int x, y;
};

struct Rectangle
{
    Point topLeft, bottomRight;
};

int main()
{
    const Point p1{1, 2}, p2{2, 4};
    const Rectangle r{p1, p2};
}
```

main.cpp

```cpp
struct Point
{
    int x, y;
};
struct Point
{
    int x, y;
};

struct Rectangle
{
    Point topLeft, bottomRight;
};

int main()
{
    const Point p1{1, 2}, p2{2, 4};
    const Rectangle r{p1, p2};
}
```

Compiling `main.cpp` ...

main.cpp

```cpp
struct Point
{
    int x, y;
};
struct Point
{
    int x, y;
};

struct Rectangle
{
    Point topLeft, bottomRight;
};

int main()
{
    const Point p1{1, 2}, p2{2, 4};
    const Rectangle r{p1, p2};
}
```

Compiling `main.cpp` … `point.h:1:8: error: redefinition of 'struct Point'`

point.h

```cpp
struct Point
{
    int x, y;
};
```

rectangle.h

```cpp
#include "point.h"

struct Rectangle
{
    Point topLeft, bottomRight;
};
```

main.cpp

```cpp
#include "point.h"
#include "rectangle.h"

int main()
{
    const Point p1{1, 2}, p2{2, 4};
    const Rectangle r{p1, p2};
}
```

point.h

```cpp
#ifndef POINT_H
#define POINT_H
struct Point
{
    int x, y;
};
#endif
```

rectangle.h

```cpp
#ifndef RECTANGLE_H
#define RECTANGLE_H
#include "point.h"

struct Rectangle
{
    Point topLeft, bottomRight;
};
#endif
```

main.cpp

```cpp
#include "point.h"
#include "rectangle.h"

int main()
{
    const Point p1{1, 2}, p2{2, 4};
    const Rectangle r{p1, p2};
}
```

point.h

```cpp
#pragma once

struct Point
{
    int x, y;
};
```

rectangle.h

```cpp
#pragma once

#include "point.h"

struct Rectangle
{
    Point topLeft, bottomRight;
};
```

main.cpp

```cpp
#include "point.h"
#include "rectangle.h"

int main()
{
    const Point p1{1, 2}, p2{2, 4};
    const Rectangle r{p1, p2};
}
```

Let’s pretend…

Let's pretend...

- `point.h` is now the de-facto point header!

Let’s pretend…

- `point.h` is now the de-facto point header!

```
main.cpp
third_party/
├─ pointlib/
│  ├─ point.h
├─ mathlib/
│  ├─ mathlib.h
│  ├─ third_party/
│  │  ├─ pointlib/
│  │  │  ├─ point.h
```

Let's pretend...

- `point.h` is now the de-facto point header!

```
main.cpp
third_party/
├─ pointlib/
│  ├─ point.h
├─ mathlib/
│  ├─ mathlib.h
│  ├─ third_party/
│  │  ├─ pointlib/
│  │  │  ├─ point.h
```

Let's pretend...

- `point.h` is now the de-facto point header!

```
main.cpp
third_party/
├─ pointlib/
│  ├─ point.h
├─ mathlib/
│  ├─ mathlib.h
│  ├─ third_party/
│  │  ├─ pointlib/
│  │  │  ├─ point.h
```

main.cpp

```cpp
#include "third_party/mathlib/mathlib.h"
#include "third_party/pointlib/point.h"
```

mathlib.h

```cpp
#pragma once
#include "third_party/pointlib/point.h"
```

Let’s pretend...

- `point.h` is now the de-facto point header!

```
main.cpp
third_party/
├ pointlib/
│  ├ point.h
├ mathlib/
│  ├ mathlib.h
│  ├ third_party/
│  │  ├ pointlib/
│  │  │  ├ point.h
```

main.cpp

```cpp
#include "third_party/mathlib/mathlib.h"
#include "third_party/pointlib/point.h"
```

mathlib.h

```cpp
#pragma once
#include "third_party/pointlib/point.h"
```

Compiling...

Let's pretend...

- `point.h` is now the de-facto point header!

```
main.cpp
third_party/
├ pointlib/
│  ├ point.h
├ mathlib/
│  ├ mathlib.h
│  ├ third_party/
│  │  ├ pointlib/
│  │  │  ├ point.h
```

main.cpp

```cpp
#include "third_party/mathlib/mathlib.h"
#include "third_party/pointlib/point.h"
```

mathlib.h

```cpp
#pragma once
#include "third_party/pointlib/point.h"
```

Compiling... `pointlib/point.h(3): error C2011: 'Point': 'struct' type redefinition`

Avoiding multiple definitions through headers

Some considerations:

Avoiding multiple definitions through headers

Some considerations:

- `#pragma once`
  - Not standard, but widely supported
  - Compilers use heuristics:
    - File size
    - Modification time
    - File content

Avoiding multiple definitions through headers

Some considerations:

- `#pragma once`
  - Not standard, but widely supported
  - Compilers use heuristics:
    - File size
    - Modification time
    - File content
- Header guards
  - Compilers optimize invoking the preprocessor

"Header files are a major source of complexity, errors caused by dependencies, and slow compilation"

- Bjarne Stroustrup

point.h

```cpp
#pragma once

struct Point
{
    int x, y;
};
```

rectangle.h

```cpp
#pragma once

#include "point.h"

struct Rectangle
{
    Point topLeft, bottomRight;
};
```

main.cpp

```cpp
#include "point.h"
#include "rectangle.h"

int main()
{
    const Point p1{1, 2}, p2{2, 4};
    const Rectangle r{p1, p2};
}
```

point.h

```cpp
#pragma once

struct Point
{
    int x, y;
};
```

rectangle.h

```cpp
#pragma once

struct Rectangle
{
    Point topLeft, bottomRight;
};
```

main.cpp

```cpp
#include "point.h"
#include "rectangle.h"

int main()
{
    const Point p1{1, 2}, p2{2, 4};
    const Rectangle r{p1, p2};
}
```

point.h

```cpp
#pragma once

struct Point
{
    int x, y;
};
```

rectangle.h

```cpp
#pragma once

struct Rectangle
{
    Point topLeft, bottomRight;
};
```

main.cpp

```cpp
#include "rectangle.h"
#include "point.h"

int main()
{
    const Point p1{1, 2}, p2{2, 4};
    const Rectangle r{p1, p2};
}
```

What shouldn’t be in a header?

According to Microsoft Learn:

- Built-in type definitions at namespace or global scope
- Non-inline function definitions
- Non-const variable definitions
- Aggregate definitions
- Unnamed namespaces
- Using directives

Built-in type definitions at namespace or global scope

Aliasing built in types isn't allowed:

header.h

```cpp
#pragma once

struct MyInt{ ... };

using int = MyInt; // error: expected unqualified-id

typedef MyInt int; // error: multiple types in one declaration
```

Built-in type definitions at namespace or global scope

`define` directives are bad practices in headers:

header.h

```cpp
#pragma once

#include <cstdint>

#define int int64_t
```

main.cpp

```cpp
#include "header.h"

int main()
{
    return 0;
}
```

Built-in type definitions at namespace or global scope

`define` directives are bad practices in headers:

header.h

```cpp
#pragma once

#include <cstdint>

#define int int64_t
```

main.cpp

```cpp
#include "header.h"

int main()
{
    return 0;
}
```

```
error: 'main' must return 'int'
```

Built-in type definitions at namespace or global scope

Aliases can produce unexpected behavior:

header.h

```
#pragma once

using UserId = int;
using ProductId = int;

void process(UserId user, ProductId product);
```

Built-in type definitions at namespace or global scope

Aliases can produce unexpected behavior:

header.h

```cpp
#pragma once

using UserId = int;
using ProductId = int;

void process(UserId user, ProductId product);
```

Built-in type definitions at namespace or global scope

Aliases can produce unexpected behavior:

header.h

```
#pragma once

using UserId = int;
using ProductId = int;

void process(UserId user, ProductId product);
```

Built-in type definitions at namespace or global scope

Aliases can produce unexpected behavior:

header.h

```cpp
#pragma once

using UserId = int;
using ProductId = int;

void process(UserId user, ProductId product);
```

main.cpp

```cpp
#include "header.h"

int main()
{
    ProductId product = 42;
    UserId user = 24;
    process(product, user);
}
```

Built-in type definitions at namespace or global scope

Aliases can produce unexpected behavior:

header.h

```cpp
#pragma once

using UserId = int;
using ProductId = int;

void process(UserId user, ProductId product);
```

main.cpp

```cpp
#include "header.h"

int main()
{
    ProductId product = 42;
    UserId user = 24;
    process(product, user);
}
```

Built-in type definitions at namespace or global scope

Aliases can produce unexpected behavior:

header.h

```cpp
#pragma once

using UserId = int;
using ProductId = int;

void process(UserId user, ProductId product);
```

main.cpp

```cpp
#include "header.h"

int main()
{
    ProductId product = 42;
    UserId user = 24;
    process(product, user);
}
```

Non-inline function definitions

lib.h

```
#pragma once

struct Point { ... };

void printPoint(const Point& point) { ... }

Point addPoints(const Point& p1, const Point& p2);
```

Non-inline function definitions

lib.h

```cpp
#pragma once

struct Point { ... };

void printPoint(const Point& point) { ... }

Point addPoints(const Point& p1, const Point& p2);
```

Non-inline function definitions

lib.h

```cpp
#pragma once

struct Point { ... };

void printPoint(const Point& point) { ... }

Point addPoints(const Point& p1, const Point& p2);
```

Non-inline function definitions

lib.h

```
#pragma once

struct Point { ... };

void printPoint(const Point& point) { ... }

Point addPoints(const Point& p1, const Point& p2);
```

Non-inline function definitions

lib.h

```cpp
#pragma once

struct Point { ... };

void printPoint(const Point& point) { ... }

Point addPoints(const Point& p1, const Point& p2);
```

lib.cpp

```cpp
#include "lib.h"

Point addPoints(const Point& p1, const Point& p2)
{
    ...
}
```

Non-inline function definitions

lib.h

```cpp
#pragma once

struct Point {…};

void printPoint(const Point& point) { … }

Point addPoints(const Point& p1, const Point& p2);
```

lib.cpp

```cpp
#include "lib.h"

Point addPoints(const Point& p1, const Point& p2)
{
    ...
}
```

Non-inline function definitions

lib.h

```cpp
#pragma once

struct Point { ... };

void printPoint(const Point& point) { ... }

Point addPoints(const Point& p1, const Point& p2);
```

lib.cpp

```cpp
#include "lib.h"

Point addPoints(const Point& p1, const Point& p2)
{
    ...
}
```

Non-inline function definitions

lib.h

```cpp
#pragma once

struct Point { ... };

void printPoint(const Point& point) { ... }

Point addPoints(const Point& p1, const Point& p2);
```

lib.cpp

```cpp
#include "lib.h"

Point addPoints(const Point& p1, const Point& p2)
{
    ...
}
```

main.cpp

```cpp
#include "lib.h"

int main()
{
    Point p = addPoints({1, 2}, {3, 4});
    printPoint(p);
}
```

Non-inline function definitions

lib.h

```cpp
#pragma once

struct Point {…};

void printPoint(const Point& point) { … }

Point addPoints(const Point& p1, const Point& p2);
```

lib.cpp

```cpp
#include "lib.h"

Point addPoints(const Point& p1, const Point& p2)
{
    ...
}
```

main.cpp

```cpp
#include "lib.h"

int main()
{
    Point p = addPoints({1, 2}, {3, 4});
    printPoint(p);
}
```

Non-inline function definitions

lib.h

```cpp
#pragma once

struct Point {…};

void printPoint(const Point& point) { … }

Point addPoints(const Point& p1, const Point& p2);
```

lib.cpp

```cpp
#include "lib.h"

Point addPoints(const Point& p1, const Point& p2)
{
    …
}
```

main.cpp

```cpp
#include "lib.h"

int main()
{
    Point p = addPoints({1, 2}, {3, 4});
    printPoint(p);
}
```

Non-inline function definitions

lib.h

```cpp
#pragma once

struct Point { … };

void printPoint(const Point& point) {  …  }

Point addPoints(const Point& p1, const Point& p2);
```

lib.cpp

```cpp
#include "lib.h"

Point addPoints(const Point& p1, const Point& p2)
{
    …
}
```

main.cpp

```cpp
#include "lib.h"

int main()
{
    Point p = addPoints({1, 2}, {3, 4});
    printPoint(p);
}
```

Compiling…

Non-inline function definitions

lib.h

```cpp
#pragma once

struct Point { … };

void printPoint(const Point& point) {  …  }

Point addPoints(const Point& p1, const Point& p2);
```

lib.cpp

```cpp
#include "lib.h"

Point addPoints(const Point& p1, const Point& p2)
{
    …
}
```

main.cpp

```cpp
#include "lib.h"

int main()
{
    Point p = addPoints({1, 2}, {3, 4});
    printPoint(p);
}
```

Compiling… ✓

Non-inline function definitions

lib.h

```cpp
#pragma once

struct Point {…};

void printPoint(const Point& point) { … }

Point addPoints(const Point& p1, const Point& p2);
```

lib.cpp

```cpp
#include "lib.h"

Point addPoints(const Point& p1, const Point& p2)
{
    …
}
```

main.cpp

```cpp
#include "lib.h"

int main()
{
    Point p = addPoints({1, 2}, {3, 4});
    printPoint(p);
}
```

Compiling... ✓

Linking...

Non-inline function definitions

lib.h

```cpp
#pragma once

struct Point { … };

void printPoint(const Point& point) {  …  }

Point addPoints(const Point& p1, const Point& p2);
```

lib.cpp

```cpp
#include "lib.h"

Point addPoints(const Point& p1, const Point& p2)
{
    …
}
```

main.cpp

```cpp
#include "lib.h"

int main()
{
    Point p = addPoints({1, 2}, {3, 4});
    printPoint(p);
}
```

Compiling... ✓

Linking... `ld: multiple definition of 'printPoint(Point const&)'`

Non-inline function definitions

lib.h

```
#pragma once

struct Point {…};

void printPoint(const Point& point) { … }

Point addPoints(const Point& p1, const Point& p2);
```

lib.cpp

```
#include "lib.h"

Point addPoints(const Point& p1, const Point& p2)
{
    …
}
```

main.cpp

```
#include "lib.h"

int main()
{
    Point p = addPoints({1, 2}, {3, 4});
    printPoint(p);
}
```

Non-inline function definitions

lib.h

```cpp
#pragma once

struct Point {...};

inline void printPoint(const Point& point) { ... }

Point addPoints(const Point& p1, const Point& p2);
```

lib.cpp

```cpp
#include "lib.h"

Point addPoints(const Point& p1, const Point& p2)
{
    ...
}
```

main.cpp

```cpp
#include "lib.h"

int main()
{
    Point p = addPoints({1, 2}, {3, 4});
    printPoint(p);
}
```

Non-inline function definitions

lib.h

```cpp
#pragma once

struct Point { ... };

void printPoint(const Point& point);

Point addPoints(const Point& p1, const Point& p2);
```

lib.cpp

```cpp
#include "lib.h"

void printPoint(const Point& point)
{
    ...
}

Point addPoints(const Point& p1, const Point& p2)
{
    ...
}
```

main.cpp

```cpp
#include "lib.h"

int main()
{
    Point p = addPoints({1, 2}, {3, 4});
    printPoint(p);
}
```

Non-const variable definitions

lib.h

```c
#pragma once

int variable = 0;
```

Non-const variable definitions

lib.h

```c
#pragma once

int variable = 0;
```

```
ld: main.o:(.bss+0x0): multiple definition of `variable'; lib.o:(.bss+0x0): first defined here
```

Non-const variable definitions

h lib.h

```cpp
#pragma once

int variable = 0;
```

```
ld: main.o:(.bss+0x0): multiple definition of `variable'; lib.o:(.bss+0x0): first defined here
```

- Every TU that includes this header gets its own copy of the variable

Non-const variable definitions

lib.h

```cpp
#pragma once

int variable = 0;
```

```
ld: main.o:(.bss+0x0): multiple definition of `variable'; lib.o:(.bss+0x0): first defined here
```

- Every TU that includes this header gets its own copy of the variable
- ODR violation

Aggregate definitions

lib.h

```c
#pragma once

int aggregate[] = {10, 20, 30};
```

Aggregate definitions

h lib.h

```c
#pragma once

int aggregate[] = {10, 20, 30};
```

```
ld: main.o:(.bss+0x0): multiple definition of `aggregate'; lib.o:(.bss+0x0): first defined here
```

Aggregate definitions

lib.h

```c
#pragma once

int aggregate[] = {10, 20, 30};
```

```
ld: main.o:(.bss+0x0): multiple definition of `aggregate'; lib.o:(.bss+0x0): first defined here
```

- Every TU that includes this header gets its own copy of the variable

Aggregate definitions

lib.h

```
#pragma once

int aggregate[] = {10, 20, 30};
```

```
ld: main.o:(.bss+0x0): multiple definition of `aggregate'; lib.o:(.bss+0x0): first defined here
```

- Every TU that includes this header gets its own copy of the variable
- ODR violation

Unnamed namespaces

lib.h

```cpp
#pragma once

namespace
{
    void doSomething() { ... }
}
```

Unnamed namespaces

h lib.h

```cpp
#pragma once

namespace
{
    void doSomething() { ... }
}
```

- Common copy-paste mistake when moving from a `.cpp` file

Unnamed namespaces

lib.h

```cpp
#pragma once

namespace
{
    void doSomething() { ... }
}
```

- Common copy-paste mistake when moving from a `.cpp` file
- Every TU that includes this header gets its own copy of the function definition

Unnamed namespaces

h lib.h

```cpp
#pragma once

namespace
{
    void doSomething() { ... }
}
```

- Common copy-paste mistake when moving from a `.cpp` file
- Every TU that includes this header gets its own copy of the function definition
- Binary bloat!

Using directives

header.h

```
#pragma once

using namespace std;
```

Using directives

header.h

```cpp
#pragma once

using namespace std;
```

- Pollutes the global namespace

Using directives

header.h

```cpp
#pragma once

using namespace std;
```

- Pollutes the global namespace
- Every transitive header becomes problematic

Using directives

header.h

```cpp
#pragma once

using namespace std;
```

- Pollutes the global namespace
- Every transitive header becomes problematic
- Increases the chance of name collisions

Modules: A novel approach to code organization in C++

A strong statement from Microsoft Learn:

> Note:
>
> In Visual Studio 2019, the C++20 modules feature is introduced as an improvement and eventual replacement for header files.

What Are C++ Modules?

Disambiguiation

The term `module` can be ambiguous:

Disambiguiation

The term `module` can be ambiguous:

- Clang Header Module

Disambiguiation

The term `module` can be ambiguous:

- Clang Header Module
- Objective-C Module

Disambiguiation

The term `module` can be ambiguous:

- Clang Header Module
- Objective-C Module
- Standard C++ Module

Disambiguiation

The term `module` can be ambiguous:

- Clang Header Module
- Objective-C Module
- Standard C++ Module
  - Named Modules

Disambiguiation

The term `module` can be ambiguous:

- Clang Header Module
- Objective-C Module
- Standard C++ Module
  - Named Modules
  - Header Units

Disambiguiation

The term `module` can be ambiguous:

- Clang Header Module
- Objective-C Module
- Standard C++ Module
  - Named Modules
  - Header Units

Modules in C++

Modern tool for consuming external code:

- Packages components and encapsulate their implementations
- Designed to co-exist but minimize reliance on the preprocessor

Modules in C++

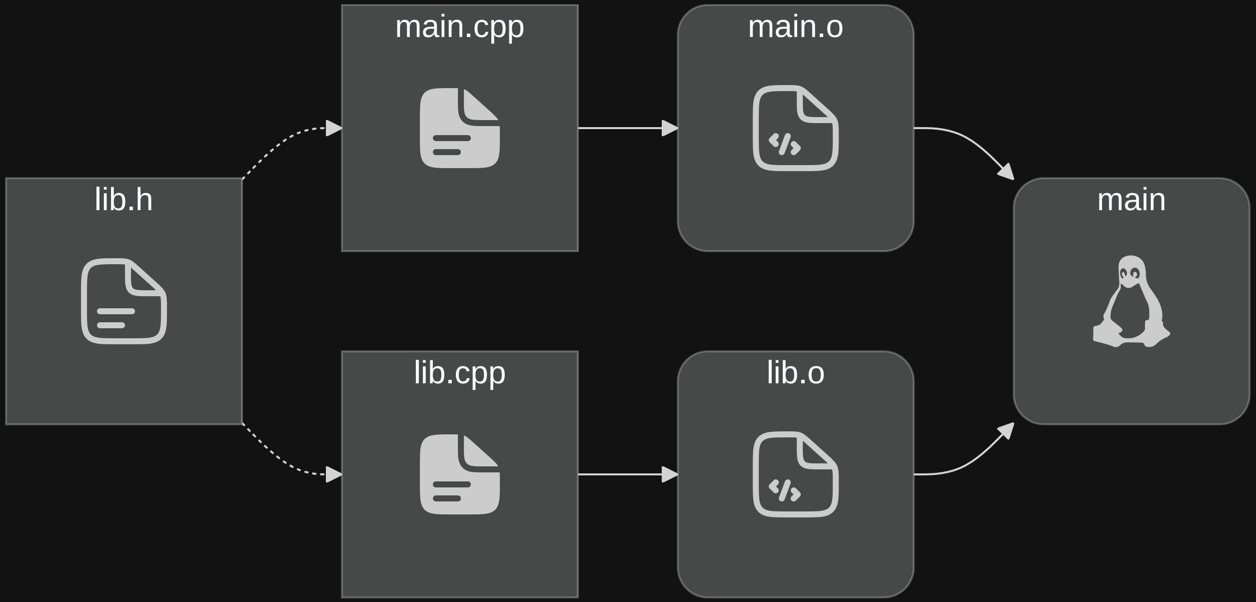
Modern tool for consuming external code:

- Packages components and encapsulate their implementations
- Designed to co-exist but minimize reliance on the preprocessor

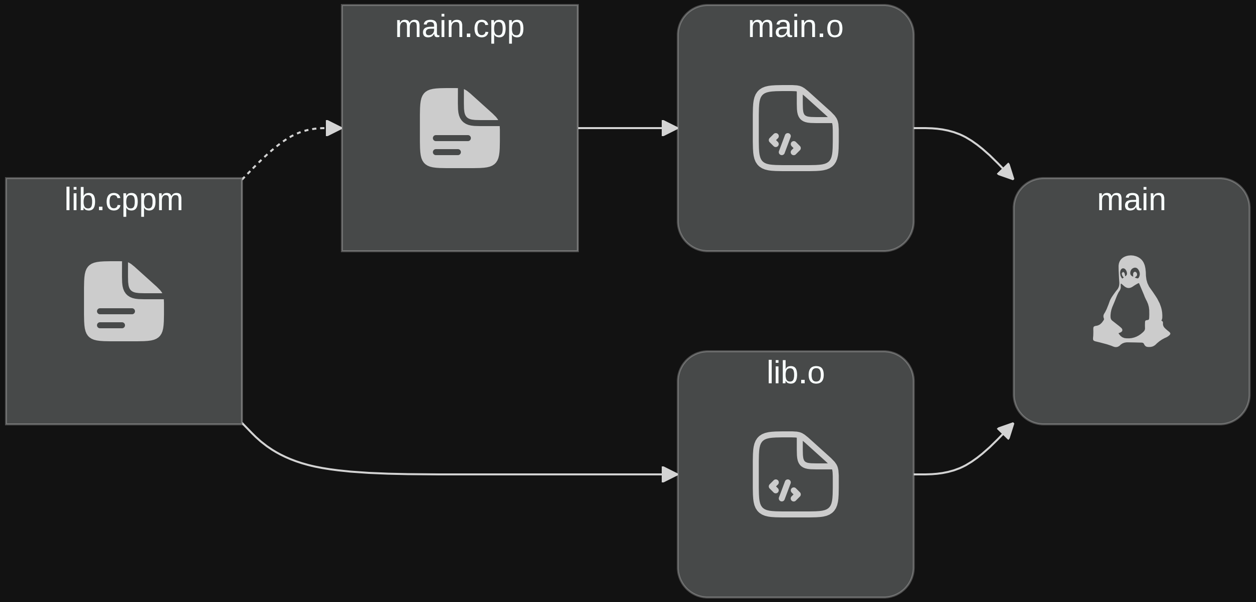
Fundamental goals:

- Componentization
- Isolation from macros
- Scalable build
- Support for modern semantics-aware developer tools
- Reduce opportunities for violating the ODR

Compilation of many source files with headers



Compilation of many source files with modules



Modules in C++

lib.h

```cpp
#pragma once

struct Point
{
    int x, y;
};

Point addPoints(Point const& a, Point const& b);
```

Modules in C++

lib.h

```cpp
#pragma once

struct Point
{
    int x, y;
};

Point addPoints(Point const& a, Point const& b);
```

Modules in C++

lib.h

```cpp
#pragma once

struct Point
{
    int x, y;
};

Point addPoints(Point const& a, Point const& b);
```

Modules in C++

lib.h

```cpp
#pragma once

struct Point
{
    int x, y;
};

Point addPoints(Point const& a, Point const& b);
```

lib.cpp

```cpp
#include "lib.h"

Point addPoints(Point const& a, Point const& b)
{
    return { a.x + b.x, a.y + b.y};
}
```

Modules in C++

lib.h

```cpp
#pragma once

struct Point
{
    int x, y;
};

Point addPoints(Point const& a, Point const& b);
```

lib.cpp

```cpp
#include "lib.h"

Point addPoints(Point const& a, Point const& b)
{
    return { a.x + b.x, a.y + b.y};
}
```

Modules in C++

lib.h

```cpp
#pragma once

struct Point
{
    int x, y;
};

Point addPoints(Point const& a, Point const& b);
```

lib.cpp

```cpp
#include "lib.h"

Point addPoints(Point const& a, Point const& b)
{
    return { a.x + b.x, a.y + b.y};
}
```

Modules in C++

lib.h

```cpp
#pragma once

struct Point
{
    int x, y;
};

Point addPoints(Point const& a, Point const& b);
```

lib.cpp

```cpp
#include "lib.h"

Point addPoints(Point const& a, Point const& b)
{
    return { a.x + b.x, a.y + b.y};
}
```

lib.cppm

```cpp
export module lib;

export struct Point
{
    int x, y;
};

export Point addPoints(Point const& a, Point const& b)
{
    return { a.x + b.x, a.y + b.y};
}
```

Modules in C++

lib.h

```cpp
#pragma once

struct Point
{
    int x, y;
};

Point addPoints(Point const& a, Point const& b);
```

lib.cpp

```cpp
#include "lib.h"

Point addPoints(Point const& a, Point const& b)
{
    return { a.x + b.x, a.y + b.y};
}
```

lib.cppm

```cpp
export module lib;

export struct Point
{
    int x, y;
};

export Point addPoints(Point const& a, Point const& b)
{
    return { a.x + b.x, a.y + b.y};
}
```

Modules in C++

h lib.h

```cpp
#pragma once

struct Point
{
    int x, y;
};

Point addPoints(Point const& a, Point const& b);
```

lib.cpp

```cpp
#include "lib.h"

Point addPoints(Point const& a, Point const& b)
{
    return { a.x + b.x, a.y + b.y};
}
```

lib.cppm

```cpp
export module lib;

export struct Point
{
    int x, y;
};

export Point addPoints(Point const& a, Point const& b)
{
    return { a.x + b.x, a.y + b.y};
}
```

Modules in C++

lib.h

```cpp
#pragma once

struct Point
{
    int x, y;
};

Point addPoints(Point const& a, Point const& b);
```

lib.cpp

```cpp
#include "lib.h"

Point addPoints(Point const& a, Point const& b)
{
    return { a.x + b.x, a.y + b.y};
}
```

lib.cppm

```cpp
export module lib;

export struct Point
{
    int x, y;
};

export Point addPoints(Point const& a, Point const& b)
{
    return { a.x + b.x, a.y + b.y};
}
```

An overview of module units

Every module consists of one or more Module Units (MUs)

An overview of module units

Every module consists of one or more Module Units (MUs)

- A module unit (typically) starts with a module declaration:

```
[export] module module_name[:partition_name];
```

An overview of module units

Every module consists of one or more Module Units (MUs)

- A module unit (typically) starts with a module declaration:

  ```
  [export] module module_name[:partition_name];
  ```

- MU is a special kind of TU

An overview of module units

Every module consists of one or more Module Units (MUs)

- A module unit (typically) starts with a module declaration:

```
[export] module module_name[:partition_name];
```

- MU is a special kind of TU
- MUs aren't exposed outside the module

An overview of module units

Every module consists of one or more Module Units (MUs)

- A module unit (typically) starts with a module declaration:

```
[export] module module_name[:partition_name];
```

- MU is a special kind of TU
- MUs aren't exposed outside the module
- A MU is either an interface or implementation unit

An overview of module units

Every module consists of one or more Module Units (MUs)

- A module unit (typically) starts with a module declaration:

```
[export] module module_name[:partition_name];
```

- MU is a special kind of TU
- MUs aren't exposed outside the module
- A MU is either an interface or implementation unit
- Modules can be broken down into partitions

An overview of module units

Every module consists of one or more Module Units (MUs)

- A module unit (typically) starts with a module declaration:

  ```
  [export] module module_name[:partition_name];
  ```

- MU is a special kind of TU
- MUs aren't exposed outside the module
- A MU is either an interface or implementation unit
- Modules can be broken down into partitions

All MUs are compiled, generating a Built Module Interface (BMI)

Interface and implementation units

- Interface units have an `export` -ed module declaration

example.cppm

```cpp
export module myModule;

export int foo();
```

example.impl.cpp

```cpp
module myModule;

int foo() { return 42; }
```

Interface and implementation units

- Interface units have an `export` -ed module declaration

example.cppm

```cpp
export module myModule;

export int foo();
```

example.impl.cpp

```cpp
module myModule;

int foo() { return 42; }
```

Interface and implementation units

- Interface units have an `export` -ed module declaration

example.cppm

```
export module myModule;

export int foo();
```

example.impl.cpp

```
module myModule;

int foo() { return 42; }
```

Module partitions

- Partitions allow splitting the interface and/or implementation

geometry.cppm

```cpp
export module geometry;

export import :circle;
```

geometry-circle.cppm

```cpp
export module geometry:circle;

import :helpers;

// ...
```

geometry-helpers.cppm

```cpp
module geometry:helpers;

// ...
```

Module partitions

- Partitions allow splitting the interface and/or implementation

geometry.cppm

```
export module geometry;

export import :circle;
```

geometry-circle.cppm

```
export module geometry:circle;

import :helpers;

// ...
```

geometry-helpers.cppm

```
module geometry:helpers;

// ...
```

Module partitions

- Partitions allow splitting the interface and/or implementation

geometry.cppm

```cpp
export module geometry;

export import :circle;
```

geometry-circle.cppm

```cpp
export module geometry:circle;

import :helpers;

// ...
```

geometry-helpers.cppm

```cpp
module geometry:helpers;

// ...
```

Module partitions

- Partitions allow splitting the interface and/or implementation

geometry.cppm

```cpp
export module geometry;

export import :circle;
```

geometry-circle.cppm

```cpp
export module geometry:circle;

import :helpers;

// ...
```

geometry-helpers.cppm

```cpp
module geometry:helpers;

// ...
```

Module partitions

- Partitions allow splitting the interface and/or implementation

geometry.cppm

```cpp
export module geometry;

export import :circle;
```

geometry-circle.cppm

```cpp
export module geometry:circle;

import :helpers;

// ...
```

geometry-helpers.cppm

```cpp
module geometry:helpers;

// ...
```

Module partitions

- Partitions allow splitting the interface and/or implementation

geometry.cppm

```cpp
export module geometry;

export import :circle;
```

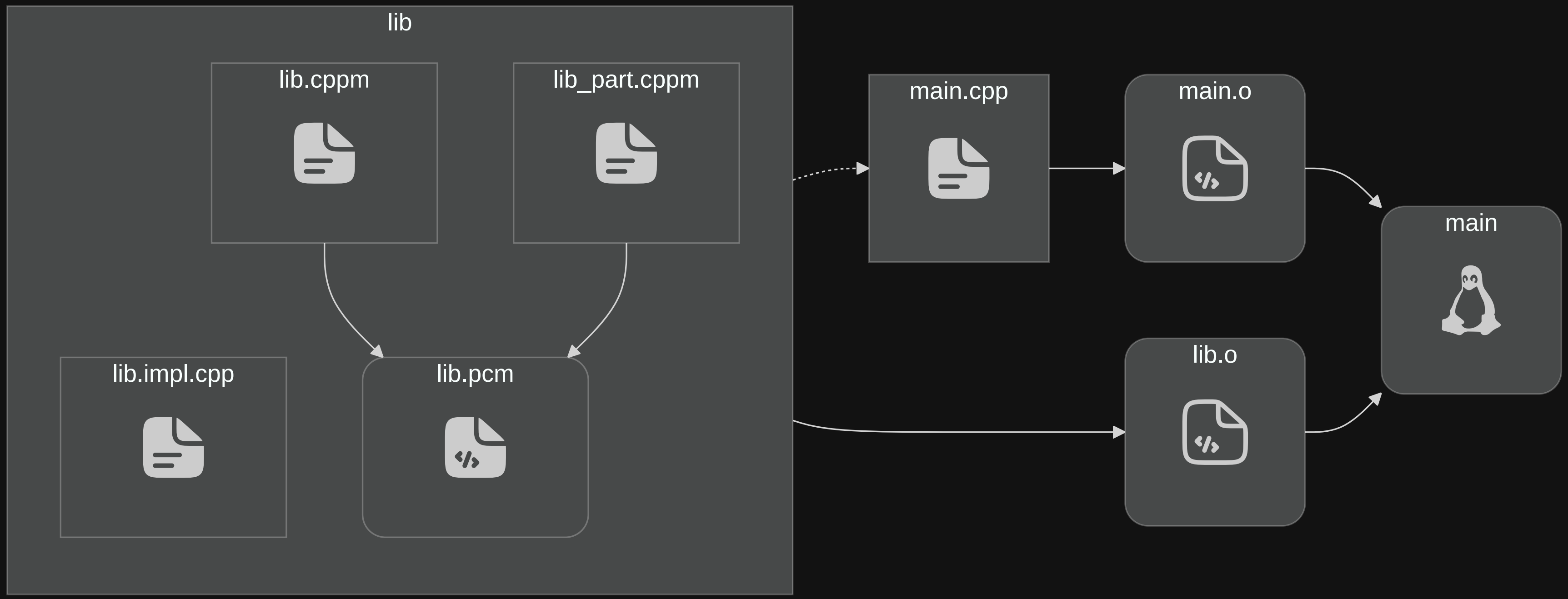
geometry-circle.cppm

```cpp
export module geometry:circle;

import :helpers;

// ...
```

geometry-helpers.cppm

```cpp
module geometry:helpers;

// ...
```

Compilation of many source files with modules



An overview of module units

Module units can be classified as:

- Primary module interface unit
- Module implementation unit
- Module partition interface unit
- Internal module partition unit

Primary module interface unit

Interface module units without a partition:

```
export module module_name;
```

Primary module interface unit

Interface module units without a partition:

```
export module module_name;
```

- Every module must contain exactly one of these

Module implementation unit

Implementation module units without a partition:

```
module module_name;
```

Module implementation unit

Implementation module units without a partition:

```
module module_name;
```

- Useful for splitting interface / implementation

Module implementation unit

Implementation module units without a partition:

```
module module_name;
```

- Useful for splitting interface / implementation
- Implicitly performs `import module_name;`

Module partition interface unit

Interface module units with a partition:

```
export module module_name:partition_name;
```

Module partition interface unit

Interface module units with a partition:

```
export module module_name:partition_name;
```

- Tool for separating large interfaces into multiple files

Module partition interface unit

Interface module units with a partition:

```
export module module_name:partition_name;
```

- Tool for separating large interfaces into multiple files
- Must be (transitively) exported by the primary module interface unit:

  ```
  export module module_name;

  export import :partition_name;
  ```

Internal module partition unit

Implementation module units with a partition:

```
module module_name:partition_name;
```

Internal module partition unit

Implementation module units with a partition:

```
module module_name:partition_name;
```

- Useful for utility functions

Internal module partition unit

Implementation module units with a partition:

```
module module_name:partition_name;
```

- Useful for utility functions
- Not part of the external interface of its module

Private module fragment

Facilitates tidy interfaces:

module.cppm

```cpp
export module myModule;

export void foo() { ... }

void helper() { ... }

export void bar() { helper(); ... }
```

Private module fragment

Facilitates tidy interfaces:

module.cppm

```
export module myModule;

export void foo() { ... }

void helper() { ... }

export void bar() { helper(); ... }

module : private;
```

Private module fragment

Facilitates tidy interfaces:

module.cppm

```cpp
export module myModule;

export void foo();
export void bar();

module : private;

void foo() { ... }

void helper() { ... }

void bar() { helper(); ... }
```

Private module fragment

Facilitates tidy interfaces:

module.cppm

```cpp
export module myModule;

export void foo();
export void bar();

module : private;

void foo() { ... }

void helper() { ... }

void bar() { helper(); ... }
```

Private module fragment

Facilitates tidy interfaces:

module.cppm

```cpp
export module myModule;

export void foo();
export void bar();

module : private;

void foo() { ... }

void helper() { ... }

void bar() { helper(); ... }
```

- Can only be in the `Primary module interface unit`

Private module fragment

Facilitates tidy interfaces:

module.cppm

```cpp
export module myModule;

export void foo();
export void bar();

module : private;

void foo() { … }

void helper() { … }

void bar() { helper(); … }
```

- Can only be in the `Primary module interface unit`
- Only for modules with a single MU

Global module fragment

Preprocessor usage in modules:

module.cppm

```cpp
export module myModule;

#include <vector>
#include <print>

export void printSize(std::vector<int> const &vec)
{
    std::println("Size: {}", vec.size());
}
```

Global module fragment

Preprocessor usage in modules:

module.cppm

```cpp
export module myModule;

#include <vector>
#include <print>

export void printSize(std::vector<int> const &vec)
{
    std::println("Size: {}", vec.size());
}
```

```
warning C5244: '#include <vector>' in the purview of module 'm' appears erroneous.
Consider moving that directive before the module declaration
```

Global module fragment

Preprocessor usage in modules:

module.cppm

```cpp
module;

#include <vector>
#include <iostream>

export module myModule;

export void printSize(std::vector<int> const &vec)
{
    std::println("Size: {}", vec.size());
}
```

Global module fragment

Preprocessor usage in modules:

module.cppm

```cpp
module;

#include <vector>
#include <iostream>

export module myModule;

export void printSize(std::vector<int> const &vec)
{
    std::println("Size: {}", vec.size());
}
```

Global module fragment

Preprocessor usage in modules:

module.cppm

```cpp
module;

#include <vector>
#include <iostream>

export module myModule;

export void printSize(std::vector<int> const &vec)
{
    std::println("Size: {}", vec.size());
}
```

- Non-referenced entities are discarded

Global module fragment

Preprocessor usage in modules:

module.cppm

```cpp
module;

#include <vector>
#include <iostream>

export module myModule;

export void printSize(std::vector<int> const &vec)
{
    std::println("Size: {}", vec.size());
}
```

- Non-referenced entities are discarded
- Useful when it is not possible to import the headers

Header Units

Allows importing headers:

```
import "header.h";
```

Header Units

Allows importing headers:

```
import "header.h";
```

- Translation unit formed by synthesizing an importable header

Header Units

Allows importing headers:

```
import "header.h";
```

- Translation unit formed by synthesizing an importable header

Header Units

Allows importing headers:

```
import "header.h";
```

- Translation unit formed by synthesizing an importable header
- All declarations are implicitly exported

Header Units

Allows importing headers:

```
import "header.h";
```

- Translation unit formed by synthesizing an importable header
- All declarations are implicitly exported
- Provides module benefits for legacy headers

Header Units

No external definitions:

foo_legacy.h

```
#pragma once

void foo() { ... }
```

Header Units

No external definitions:

foo_legacy.h

```cpp
#pragma once

void foo() { ... }
```

```
error: non-inline external definitions are not permitted in C++ header units
```

Header Units

No external definitions:

foo_legacy.h

```
#pragma once

inline void foo() { ... }
```

Header Units

Macros are made visible:

foo_legacy.h

```cpp
#pragma once

#define FOO_VERSION 42
inline void foo() { ... }
```

foo.cppm

```cpp
export module foo;

export import "foo_legacy.h";

export bool check_version(int const version)
{
    return version == FOO_VERSION;
}
```

Header Units

Macros are made visible:

h foo_legacy.h

```cpp
#pragma once

#define FOO_VERSION 42
inline void foo() { ... }
```

foo.cppm

```cpp
export module foo;

export import "foo_legacy.h";

export bool check_version(int const version)
{
    return version == FOO_VERSION;
}
```

Header Units

Macros are made visible:

foo_legacy.h

```cpp
#pragma once

#define FOO_VERSION 42
inline void foo() { ... }
```

foo.cppm

```cpp
export module foo;

export import "foo_legacy.h";

export bool check_version(int const version)
{
    return version == FOO_VERSION;
}
```

Header Units

Macros are made visible:

foo_legacy.h

```
#pragma once

#define FOO_VERSION 42
inline void foo() { ... }
```

foo.cppm

```
export module foo;

export import "foo_legacy.h";

export bool check_version(int const version)
{
    return version == FOO_VERSION;
}
```

Header Units

Macros are made visible:

h foo_legacy.h

```cpp
#pragma once

#define FOO_VERSION 42
inline void foo() { ... }
```

foo.cppm

```cpp
export module foo;

export import "foo_legacy.h";

export bool check_version(int const version)
{
    return version == FOO_VERSION;
}
```

Header Units

Macros are made visible:

foo_legacy.h

```cpp
#pragma once

#define FOO_VERSION 42
inline void foo() { ... }
```

foo.cppm

```cpp
export module foo;

export import "foo_legacy.h";

export bool check_version(int const version)
{
    return version == FOO_VERSION;
}
```

- Macros are not re-exported

STL header units

C++20 defines *importable C++ library headers*

STL header units

C++20 defines *importable C++ library headers*

main.cpp

```cpp
#include <iostream>
#include <map>
#include <vector>
#include <algorithm>
#include <chrono>
#include <random>
#include <memory>
#include <cmath>
#include <thread>
```

STL header units

C++20 defines *importable C++ library headers*

main.cpp

```cpp
import <iostream>;
import <map>;
import <vector>;
import <algorithm>;
import <chrono>;
import <random>;
import <memory>;
import <cmath>;
import <thread>;
```

STL module

C++23 exposes the STL as modules

STL module

C++23 exposes the STL as modules

main.cpp

```cpp
#include <iostream>
#include <map>
#include <vector>
#include <algorithm>
#include <chrono>
#include <random>
#include <memory>
#include <cmath>
#include <thread>
```

STL module

C++23 exposes the STL as modules

main.cpp

```cpp
import std;
```

STL module

C++23 exposes the STL as modules

main.cpp

```cpp
import std;
```

- `std` exposes all symbols in the `std` namespace

STL module

C++23 exposes the STL as modules

main.cpp

```cpp
import std;
```

- `std` exposes all symbols in the `std` namespace
- `std.compat` additionally exposes C global namespace declarations
  - e.g. `::printf` from `<cstdio>`

STL module

C++23 exposes the STL as modules

main.cpp

```cpp
import std;
```

- `std` exposes all symbols in the `std` namespace
- `std.compat` additionally exposes C global namespace declarations
  - e.g. `::printf` from `<cstdio>`
- Macros are not exposed
  - e.g. `assert` from `<cassert>`

Traditional Headers vs. Modules

Let’s circle back to headers

Dependency completeness

point.h

```cpp
#pragma once

struct Point
{
    int x, y;
};
```

rectangle.h

```cpp
#pragma once

#include "point.h"

struct Rectangle
{
    Point topLeft, bottomRight;
};
```

main.cpp

```cpp
#include "point.h"
#include "rectangle.h"

int main()
{
    const Point p1{1, 2}, p2{2, 4};
    const Rectangle r{p1, p2};
}
```

Dependency completeness

point.cppm

```cpp
export module point;

export struct Point
{
    int x, y;
};
```

rectangle.cppm

```cpp
export module rectangle;

export import point;

export struct Rectangle
{
    Point topLeft, bottomRight;
};
```

main.cpp

```cpp
import point;
import rectangle;

int main()
{
    const Point p1{1, 2}, p2{2, 4};
    const Rectangle r{p1, p2};
}
```

Built-in type definitions at namespace or global scope

header.h

```cpp
#pragma once

#include <cstdint>

#define int int64_t
```

main.cpp

```cpp
#include "header.h"

int main() // error: 'main' must return 'int'
{
    return 0;
}
```

Built-in type definitions at namespace or global scope

module.cppm

```cpp
module;

#define int int64_t

export module myModule;
```

main.cpp

```cpp
import myModule;

int main()
{
    return 0;
}
```

Non-inline function definitions

header.h

```
#pragma once

void doSomething()
{
    ...
}
```

Non-inline function definitions

module.cppm

```cpp
export module myModule;

export void doSomething()
{
    ...
}
```

Non-const variable definitions

header.h

```
#pragma once

int variable = 0;
```

Non-const variable definitions

module.cppm

```cpp
export module myModule;

export int variable = 0;
```

Aggregate definitions

header.h

```
#pragma once

int aggregate[] = {10, 20, 30};
```

Aggregate definitions

module.cppm

```cpp
export module myModule;

export int aggregate[] = {10, 20, 30};
```

Unnamed namespaces

header.h

```cpp
#pragma once

namespace
{
    void doSomething() { ... }
}
```

Unnamed namespaces

module.cppm

```cpp
export module myModule;

namespace
{
    void doSomething() { ... }
}
```

Unnamed namespaces

module.cppm

```cpp
export module myModule;

namespace
{
    export void doSomething() { ... }
}
```

Unnamed namespaces

module.cppm

```cpp
export module myModule;

namespace
{
    export void doSomething() { ... }
}
```

```
error C2294: cannot export symbol '`anonymous-namespace'::doSomething' because it has internal linkage
```

Using directives

h header.h

```
#pragma once

using namespace std;
```

Using directives

module.cppm

```cpp
export module myModule;

using namespace std;
```

Encapsulation

h Private.h

```cpp
#define SECRET 42
namespace Private
{
    inline int secret() {  return SECRET; }
}
```

Encapsulation

h Private.h

```
#define SECRET 42
namespace Private
{
    inline int secret() {  return SECRET; }
}
```

Encapsulation

Private.h

```cpp
#define SECRET 42
namespace Private
{
    inline int secret() {  return SECRET; }
}
```

Encapsulation

h Private.h

```cpp
#define SECRET 42
namespace Private
{
    inline int secret() {  return SECRET; }
}
```

h UserFacing.h

```cpp
using Pimpl = shared_ptr<struct UserFacingImpl>;
class UserFacing
{
public:
    UserFacing();
    int getNumber() const;

private:
    Pimpl m_pimpl;
};
```

Encapsulation

Private.h

```cpp
#define SECRET 42
namespace Private
{
    inline int secret() {  return SECRET; }
}
```

UserFacing.h

```cpp
using Pimpl = shared_ptr<struct UserFacingImpl>;
class UserFacing
{
public:
    UserFacing();
    int getNumber() const;

private:
    Pimpl m_pimpl;
};
```

Encapsulation

h Private.h

```
#define SECRET 42
namespace Private
{
    inline int secret() {  return SECRET; }
}
```

h UserFacing.h

```
using Pimpl = shared_ptr<struct UserFacingImpl>;
class UserFacing
{
public:
    UserFacing();
    int getNumber() const;

private:
    Pimpl m_pimpl;
};
```

Encapsulation

Private.h

```cpp
#define SECRET 42
namespace Private
{
    inline int secret() {  return SECRET; }
}
```

UserFacing.h

```cpp
using Pimpl = shared_ptr<struct UserFacingImpl>;
class UserFacing
{
public:
    UserFacing();
    int getNumber() const;

private:
    Pimpl m_pimpl;
};
```

UserFacing.cpp

```cpp
#include "UserFacing.h"
#include "Private.h"

struct UserFacingImpl
{ int number = Private::secret(); };

UserFacing::UserFacing()
    : m_pimpl(new UserFacingImpl()) {}

int UserFacing::getNumber() const
{ return m_pimpl->number; }
```

Encapsulation

Private.h

```cpp
#define SECRET 42
namespace Private
{
    inline int secret() {  return SECRET; }
}
```

UserFacing.h

```cpp
using Pimpl = shared_ptr<struct UserFacingImpl>;
class UserFacing
{
public:
    UserFacing();
    int getNumber() const;

private:
    Pimpl m_pimpl;
};
```

UserFacing.cpp

```cpp
#include "UserFacing.h"
#include "Private.h"

struct UserFacingImpl
{ int number = Private::secret(); };

UserFacing::UserFacing()
    : m_pimpl(new UserFacingImpl()) {}

int UserFacing::getNumber() const
{ return m_pimpl->number; }
```

Encapsulation

Private.h

```cpp
#define SECRET 42
namespace Private
{
    inline int secret() {  return SECRET; }
}
```

UserFacing.h

```cpp
using Pimpl = shared_ptr<struct UserFacingImpl>;
class UserFacing
{
public:
    UserFacing();
    int getNumber() const;

private:
    Pimpl m_pimpl;
};
```

UserFacing.cpp

```cpp
#include "UserFacing.h"
#include "Private.h"

struct UserFacingImpl
{ int number = Private::secret(); };

UserFacing::UserFacing()
    : m_pimpl(new UserFacingImpl()) {}

int UserFacing::getNumber() const
{ return m_pimpl->number; }
```

Encapsulation

Private.h

```cpp
#define SECRET 42
namespace Private
{
    inline int secret() {  return SECRET; }
}
```

UserFacing.h

```cpp
using Pimpl = shared_ptr<struct UserFacingImpl>;
class UserFacing
{
public:
    UserFacing();
    int getNumber() const;

private:
    Pimpl m_pimpl;
};
```

UserFacing.cpp

```cpp
#include "UserFacing.h"
#include "Private.h"

struct UserFacingImpl
{ int number = Private::secret(); };

UserFacing::UserFacing()
    : m_pimpl(new UserFacingImpl()) {}

int UserFacing::getNumber() const
{ return m_pimpl->number; }
```

Encapsulation

Private.h

```cpp
#define SECRET 42
namespace Private
{
    inline int secret() {  return SECRET; }
}
```

UserFacing.h

```cpp
using Pimpl = shared_ptr<struct UserFacingImpl>;
class UserFacing
{
public:
    UserFacing();
    int getNumber() const;

private:
    Pimpl m_pimpl;
};
```

UserFacing.cpp

```cpp
#include "UserFacing.h"
#include "Private.h"

struct UserFacingImpl
{ int number = Private::secret(); };

UserFacing::UserFacing()
    : m_pimpl(new UserFacingImpl()) {}

int UserFacing::getNumber() const
{ return m_pimpl->number; }
```

Encapsulation with modules

Private.cppm

```cpp
module;
#define SECRET 42
export module Private;
namespace Private {
    export inline int secret() {  return SECRET; }
}
```

Encapsulation with modules

Private.cppm

```cpp
module;
#define SECRET 42
export module Private;
namespace Private {
    export inline int secret() {  return SECRET; }
}
```

Encapsulation with modules

Private.cppm

```
module;
#define SECRET 42
export module Private;
namespace Private {
    export inline int secret() {  return SECRET; }
}
```

Encapsulation with modules

Private.cppm

```cpp
module;
#define SECRET 42
export module Private;
namespace Private {
    export inline int secret() {  return SECRET; }
}
```

Encapsulation with modules

Private.cppm

```cpp
module;
#define SECRET 42
export module Private;
namespace Private {
    export inline int secret() {  return SECRET; }
}
```

UserFacing.cppm

```cpp
export module UserFacing;
import Private;

struct UserFacingImpl
{ const int number = Private::secret(); };

export class UserFacing
{
public:
    UserFacing()
      : m_pimpl(std::make_shared<UserFacingImpl>())
    {}

    int getNumber() const
    {
        return m_pimpl->number;
    }

private:
    std::shared_ptr<UserFacingImpl> m_pimpl;
};
```

Encapsulation with modules

Private.cppm

```cpp
module;
#define SECRET 42
export module Private;
namespace Private {
    export inline int secret() {  return SECRET; }
}
```

UserFacing.cppm

```cpp
export module UserFacing;
import Private;

struct UserFacingImpl
{ const int number = Private::secret(); };

export class UserFacing
{
public:
    UserFacing()
      : m_pimpl(std::make_shared<UserFacingImpl>())
    {}

    int getNumber() const
    {
        return m_pimpl->number;
    }

private:
    std::shared_ptr<UserFacingImpl> m_pimpl;
};
```

Encapsulation with modules

UserFacing.cppm

```cpp
export module UserFacing;
import Private;

struct UserFacingImpl
{ const int number = Private::secret(); };

export class UserFacing
{
public:
    UserFacing()
      : m_pimpl(std::make_shared<UserFacingImpl>())
    {}

    int getNumber() const
    {
        return m_pimpl->number;
    }

private:
    std::shared_ptr<UserFacingImpl> m_pimpl;
};
```

Private.cppm

```cpp
module;
#define SECRET 42
export module Private;
namespace Private {
    export inline int secret() {  return SECRET; }
}
```

Encapsulation with modules

UserFacing.cppm

```
export module UserFacing;
import Private;

struct UserFacingImpl
{ const int number = Private::secret(); };

export class UserFacing
{
public:
    UserFacing()
      : m_pimpl(std::make_shared<UserFacingImpl>())
    {}

    int getNumber() const
    {
        return m_pimpl->number;
    }

private:
    std::shared_ptr<UserFacingImpl> m_pimpl;
};
```

Private.cppm

```
module;
#define SECRET 42
export module Private;
namespace Private {
    export inline int secret() {  return SECRET; }
}
```

Encapsulation with modules

Private.cppm

```cpp
module;
#define SECRET 42
export module Private;
namespace Private {
    export inline int secret() {  return SECRET; }
}
```

UserFacing.cppm

```cpp
export module UserFacing;
import Private;

struct UserFacingImpl
{ const int number = Private::secret(); };

export class UserFacing
{
public:
    UserFacing()
      : m_pimpl(std::make_shared<UserFacingImpl>())
    {}

    int getNumber() const
    {
        return m_pimpl->number;
    }

private:
    std::shared_ptr<UserFacingImpl> m_pimpl;
};
```

Encapsulation with modules (two units)

Private.cppm

```cpp
module;
#define SECRET 42
export module Private;
namespace Private {
    export inline int secret() {  return SECRET; }
}
```

UserFacing.cppm

```cpp
export module UserFacing;

using Pimpl = shared_ptr<struct UserFacingImpl>;
export class UserFacing
{
public:
    UserFacing();
    int getNumber() const;
private:
    Pimpl m_pimpl;
};
```

Encapsulation with modules (two units)

Private.cppm

```cpp
module;
#define SECRET 42
export module Private;
namespace Private {
    export inline int secret() {  return SECRET; }
}
```

UserFacing.cppm

```cpp
export module UserFacing;

using Pimpl = shared_ptr<struct UserFacingImpl>;
export class UserFacing
{
public:
    UserFacing();
    int getNumber() const;
private:
    Pimpl m_pimpl;
};
```

Encapsulation with modules (two units)

Private.cppm

```
module;
#define SECRET 42
export module Private;
namespace Private {
    export inline int secret() {  return SECRET; }
}
```

UserFacing.cppm

```
export module UserFacing;

using Pimpl = shared_ptr<struct UserFacingImpl>;
export class UserFacing
{
public:
    UserFacing();
    int getNumber() const;
private:
    Pimpl m_pimpl;
};
```

Encapsulation with modules (two units)

Private.cppm

```cpp
module;
#define SECRET 42
export module Private;
namespace Private {
    export inline int secret() {  return SECRET; }
}
```

UserFacing.cppm

```cpp
export module UserFacing;

using Pimpl = shared_ptr<struct UserFacingImpl>;
export class UserFacing
{
public:
    UserFacing();
    int getNumber() const;
private:
    Pimpl m_pimpl;
};
```

UserFacing.impl.cpp

```cpp
module UserFacing;
import Private;

struct UserFacingImpl
{ int number = Private::secret(); };

UserFacing::UserFacing()
    : m_pimpl(new UserFacingImpl()) {}

int UserFacing::getNumber() const
{ return m_pimpl->number; }
```

Encapsulation with modules (two units)

Private.cppm

```cpp
module;
#define SECRET 42
export module Private;
namespace Private {
    export inline int secret() {  return SECRET; }
}
```

UserFacing.cppm

```cpp
export module UserFacing;

using Pimpl = shared_ptr<struct UserFacingImpl>;
export class UserFacing
{
public:
    UserFacing();
    int getNumber() const;
private:
    Pimpl m_pimpl;
};
```

UserFacing.impl.cpp

```cpp
module UserFacing;
import Private;

struct UserFacingImpl
{ int number = Private::secret(); };

UserFacing::UserFacing()
    : m_pimpl(new UserFacingImpl()) {}

int UserFacing::getNumber() const
{ return m_pimpl->number; }
```

Encapsulation with modules (two units)

Private.cppm

```cpp
module;
#define SECRET 42
export module Private;
namespace Private {
    export inline int secret() {  return SECRET; }
}
```

UserFacing.cppm

```cpp
export module UserFacing;

using Pimpl = shared_ptr<struct UserFacingImpl>;
export class UserFacing
{
public:
    UserFacing();
    int getNumber() const;
private:
    Pimpl m_pimpl;
};
```

UserFacing.impl.cpp

```cpp
module UserFacing;
import Private;

struct UserFacingImpl
{ int number = Private::secret(); };

UserFacing::UserFacing()
    : m_pimpl(new UserFacingImpl()) {}

int UserFacing::getNumber() const
{ return m_pimpl->number; }
```

Compile-time performance

Let's compare compilation times:

- `hello_world.cpp`: Needs just `<iostream>`.
- `mix.cpp`: Requires including 9 standard headers.

Helpers:

- `all_std.h`: Includes all standard library headers

`#include` necessary headers

hello_world.cpp

```cpp
#include <iostream>

...
```

mix.cpp

```cpp
#include <iostream>
#include <map>
#include <vector>
#include <algorithm>
#include <chrono>
#include <random>
#include <memory>
#include <cmath>
#include <thread>

...
```

Compile-time performance

	#include
Hello world	0.87s
Mix	2.20s

`#include` all headers

hello_world.cpp

```cpp
#include "all_std.h"

...
```

mix.cpp

```cpp
#include "all_std.h"

...
```

Compile-time performance

	#include	#include all
Hello world	0.87s	3.43s
Mix	2.20s	3.53s

`import` necessary header units

hello_world.cpp

```cpp
import <iostream>;

...
```

mix.cpp

```cpp
import <iostream>;
import <map>;
import <vector>;
import <algorithm>;
import <chrono>;
import <random>;
import <memory>;
import <cmath>;
import <thread>;

...
```

Compile-time performance

	#include	#include all	import
Hello world	0.87s	3.43s	0.32s
Mix	2.20s	3.53s	0.77s

`import` all header units

hello_world.cpp

```cpp
import "all_std.h";

...
```

mix.cpp

```cpp
import "all_std.h";

...
```

Compile-time performance

	#include	#include all	import	import all
Hello world	0.87s	3.43s	0.32s	0.62s
Mix	2.20s	3.53s	0.77s	0.99s

`import std;`

C++23 enables:

hello_world.cpp

```cpp
import std;

...
```

mix.cpp

```cpp
import std;

...
```

Compile-time performance

	#include	#include all	import	import all	import std
Hello world	0.87s	3.43s	0.32s	0.62s	0.08s
Mix	2.20s	3.53s	0.77s	0.99s	0.44s

Compile-time performance

	#include	#include all	import	import all	import std
Hello world	0.87s	3.43s	0.32s	0.62s	0.08s
Mix	2.20s	3.53s	0.77s	0.99s	0.44s

Compile-time performance (GCC)

	#include	#include all	import	import all	import std
Hello world	0.67s	1.63s	0.09s	0.28s	0.27s
Mix	1.82s	2.60s	1.58s	1.26s	1.09s

Compile-time performance (GCC)

	#include	#include all	import	import all	import std
Hello world	0.67s	1.63s	0.09s	0.28s	0.27s
Mix	1.82s	2.60s	1.58s	1.26s	1.09s

Compile-time performance (clang)

	#include	#include all	import	import all	import std
Hello world	0.92s	2.02s	0.06s	0.04s	0.07s
Mix	1.62s	2.23s	-	0.49s	0.35s

Compile-time performance (clang)

	#include	#include all	import	import all	import std
Hello world	0.92s	2.02s	0.06s	0.04s	0.07s
Mix	1.62s	2.23s	-	0.49s	0.35s

A small note

"But I could achieve this before with XYZ!"

- Modules are standardized, and accessible by every C++ developer
- Every "getting started" guide can provide this
- Imports are a portable performance gain

Problems solved

- Structured, semantic import mechanism
- Component interfaces compiled independently from the TUs that import them
- Processed only once, into an efficient binary representation (BMI)
- Modules provide strong isolation, no global polution

Compiler Support Landscape

C++20 modules support

According to cppreference.com:

	GCC	Clang	MSVC
Modules support	✓	✓	✓

C++20 modules support

According to cppreference.com:

	GCC	Clang	MSVC
Modules support			

… But what does this mean?

Feature testing

	GCC	Clang	MSVC
Named modules	?	?	?
Header units	?	?	?
Global module fragments	?	?	?
Private module fragments	?	?	?
Module implementation units	?	?	?
Module partition interface units	?	?	?
Internal module partition units	?	?	?

✓: Success | ✓: Success with warnings | ⊗: Failure

Feature testing

Using the following versions:

- `MSVC v143`
- `clang version 20.1.8`
- `g++ (GCC) 15.2.1 20250813`

Feature testing workflow

After defining a module:

1. Compile the code

```
int getAnswer()
{
    return 42;
}
```

Feature testing workflow

After defining a module:

1. Compile the code

```
int getAnswer()
{
    return 42;
}
```

2. Compile a simple test

Feature testing workflow

After defining a module:

1. Compile the code

```
int getAnswer()
{
    return 42;
}
```

2. Compile a simple test
3. Verify that `getAnswer() == 42`

Named modules

test.cppm

```
export module test;

export int getAnswer()
{
    return 42;
}
```

Named modules

test.cppm

```
export module test;

export int getAnswer()
{
    return 42;
}
```

GCC	Clang	MSVC
✔	✔	✔

✔: Success | ✔: Success with warnings | ⊗: Failure

Header units

header.h

```
#pragma once

#define ANSWER 42
```

test.cppm

```
export module test;

import "header.h";

export int getAnswer()
{
    return ANSWER;
}
```

Header units

header.h

```cpp
#pragma once

#define ANSWER 42
```

test.cppm

```cpp
export module test;

import "header.h";

export int getAnswer()
{
    return ANSWER;
}
```

GCC	Clang	MSVC
✔	✔*	✔

* `warning: the implementation of header units is in an experimental phase`

✔: Success | ✔: Success with warnings | ⊗: Failure

Global module fragments

test.cppm

```cpp
module;

#include <cmath>

export module test;

export int getAnswer()
{
    return std::sqrt(1764);
}
```

Global module fragments

test.cppm

```cpp
module;

#include <cmath>

export module test;

export int getAnswer()
{
    return std::sqrt(1764);
}
```

GCC	Clang	MSVC
✔	✔	✔

✔: Success | ✔: Success with warnings | ⊗: Failure

Private module fragments

test.cppm

```cpp
export module test;

export int getAnswer();

module : private;

int getAnswer() {
    return 42;
}
```

Private module fragments

test.cppm

```cpp
export module test;

export int getAnswer();

module : private;

int getAnswer() {
    return 42;
}
```

GCC	Clang	MSVC
⊗*	✓	✓

* `sorry, unimplemented: private module fragment`

✓: Success | ✓: Success with warnings | ⊗: Failure

Module implementation units

test.cppm

```cpp
export module test;

export int getAnswer();
```

test.impl.cpp

```cpp
module test;

int getAnswer()
{
    return 42;
}
```

Module implementation units

test.cppm

```cpp
export module test;

export int getAnswer();
```

test.impl.cpp

```cpp
module test;

int getAnswer()
{
    return 42;
}
```

GCC	Clang	MSVC
✓	✓	✓

✓: Success | ✓: Success with warnings | ⊗: Failure

Module partition interface units

test.cppm

```cpp
export module test;

export import :pinterface;
```

test_pinterface.cppm

```cpp
export module test:pinterface;

export int getAnswer()
{
    return 42;
}
```

Module partition interface units

test.cppm

```cpp
export module test;

export import :pinterface;
```

test_pinterface.cppm

```cpp
export module test:pinterface;

export int getAnswer()
{
    return 42;
}
```

GCC	Clang	MSVC
✓	✓	✓

✓: Success | ✓: Success with warnings | ⊗: Failure

Internal module partition units

test.cppm

```
export module test;

import :answer;

export int getAnswer()
{
    return answer();
}
```

test_answer.impl.cppm

```
module test:answer;

int answer()
{
    return 42;
}
```

Internal module partition units

test.cppm

```cpp
export module test;

import :answer;

export int getAnswer()
{
    return answer();
}
```

test_answer.impl.cppm

```cpp
module test:answer;

int answer()
{
    return 42;
}
```

GCC	Clang	MSVC
✓	✓	✓

✓: Success | ✓: Success with warnings | ⊗: Failure

C++20 modules support

	GCC	Clang	MSVC
Named modules	✔	✔	✔
Header units	✔	✓	✔
Global module fragments	✔	✔	✔
Private module fragments	⊗	✔	✔
Module implementation units	✔	✔	✔
Module partition interface units	✔	✔	✔
Internal module partition units	✔	✔	✔

✔: Success | ✓: Success with warnings | ⊗: Failure

Standard library modules

According to cppreference.com:

	GCC libstdc++	Clang libc++	MSVC STL
STL header units	✓	✗	✓
`std` module	✓	✓	✓

Standard library modules

According to cppreference.com:

	GCC libstdc++	Clang libc++	MSVC STL
STL header units	✓	✗	✓
`std` module	✓	✓	✓

… But what does this mean?

Feature testing

Using `mix.cpp`:

	GCC libstdc++	Clang libc++	MSVC STL
stdc++ header unit	?	?	?
STL header units	?	?	?
`std` module	?	?	?

✓: Success | ✓: Success with warnings | ⊗: Failure

stdc++ header unit

stdcpp.h

```cpp
#pragma once

#include <algorithm>
#include <any>
#include <array>
#include <atomic>
#include <barrier>
// ...
```

mix.cpp

```cpp
import "stdcpp.h"
// ...
```

stdc++ header unit

stdcpp.h

```cpp
#pragma once

#include <algorithm>
#include <any>
#include <array>
#include <atomic>
#include <barrier>
// ...
```

mix.cpp

```cpp
import "stdcpp.h"
// ...
```

GCC libstdc++	Clang libc++	MSVC STL
✔	✔	✔

✔: Success | ✔: Success with warnings | ⊗: Failure

STL header units

Compiling every header unit one by one:

mix.cpp

```cpp
import <algorithm>;
import <any>;
import <array>;
import <atomic>;
import <barrier>;
// ...
```

STL header units

Compiling every header unit one by one:

mix.cpp

```cpp
import <algorithm>;
import <any>;
import <array>;
import <atomic>;
import <barrier>;
// ...
```

GCC libstdc++	Clang libc++	MSVC STL
✓	⊗*	✓

* `error: [ ... ] is ambiguous`

✓: Success | ✓: Success with warnings | ⊗: Failure

`std` module

mix.cpp

```cpp
import std;
// ...
```

`std` module

mix.cpp

```cpp
import std;
// ...
```

GCC libstdc++	Clang libc++	MSVC STL
✔	✔	✔

✔: Success | ✔: Success with warnings | ⊗: Failure

Standard library modules

	GCC libstdc++	Clang libc++	MSVC STL
stdc++ header unit	✔	☑	✔
STL header units	✔	⊗	✔
`std` module	✔	✔	✔

✔: Success | ☑: Success with warnings | ⊗: Failure

Standard library modules

- Libraries provide the module definition files for `std` and `std.compat`
- Users need to build their own BMIs

Standard library modules

- Libraries provide the module definition files for `std` and `std.compat`
- Users need to build their own BMIs

```
clang++ -std=c++23 -stdlib=libc++ /usr/share/libc++/v1/std.cppm -Wno-reserved-module-identifier --precompile
```

Standard library modules

- Libraries provide the module definition files for `std` and `std.compat`
- Users need to build their own BMIs

```
g++ -std=c++23 -fmodules -fsearch-include-path bits/std.cc -c
```

Standard library modules

- Libraries provide the module definition files for `std` and `std.compat`
- Users need to build their own BMIs

```
cl /EHsc /std:c++latest "%VCToolsInstallDir%\modules\std.ixx" /c
```

Compiler interoperability

There are variations among compiler implementations:

Compiler interoperability

There are variations among compiler implementations:

- Filenames:
  - MSVC: `.ixx` → `.ifc`
  - Clang: `.cppm` → `.pcm`
  - GCC: `.cpp` → `.gcm`

Compiler interoperability

There are variations among compiler implementations:

- Filenames:
  - MSVC: `.ixx` → `.ifc`
  - Clang: `.cppm` → `.pcm`
  - GCC: `.cpp` → `.gcm`
- ABI:
  - ABI formats differ per compiler
  - No cross-compiler reuse possible

Consistency requirements

Compilers perform very strict checking for consistency:

- Compiler options consistency
- Source Files Consistency
- Object definition consistency

Build Systems and Tooling

Build ordering

Suppose we have the following components:

Build ordering

Suppose we have the following components:

- `libGeometry`
  - `Point`
  - `Rectangle`
  - `Square`
  - `Circle`

Build ordering

Suppose we have the following components:

- `libGeometry`
  - `Point`
  - `Rectangle`
  - `Square`
  - `Circle`
- `Main`

Build ordering

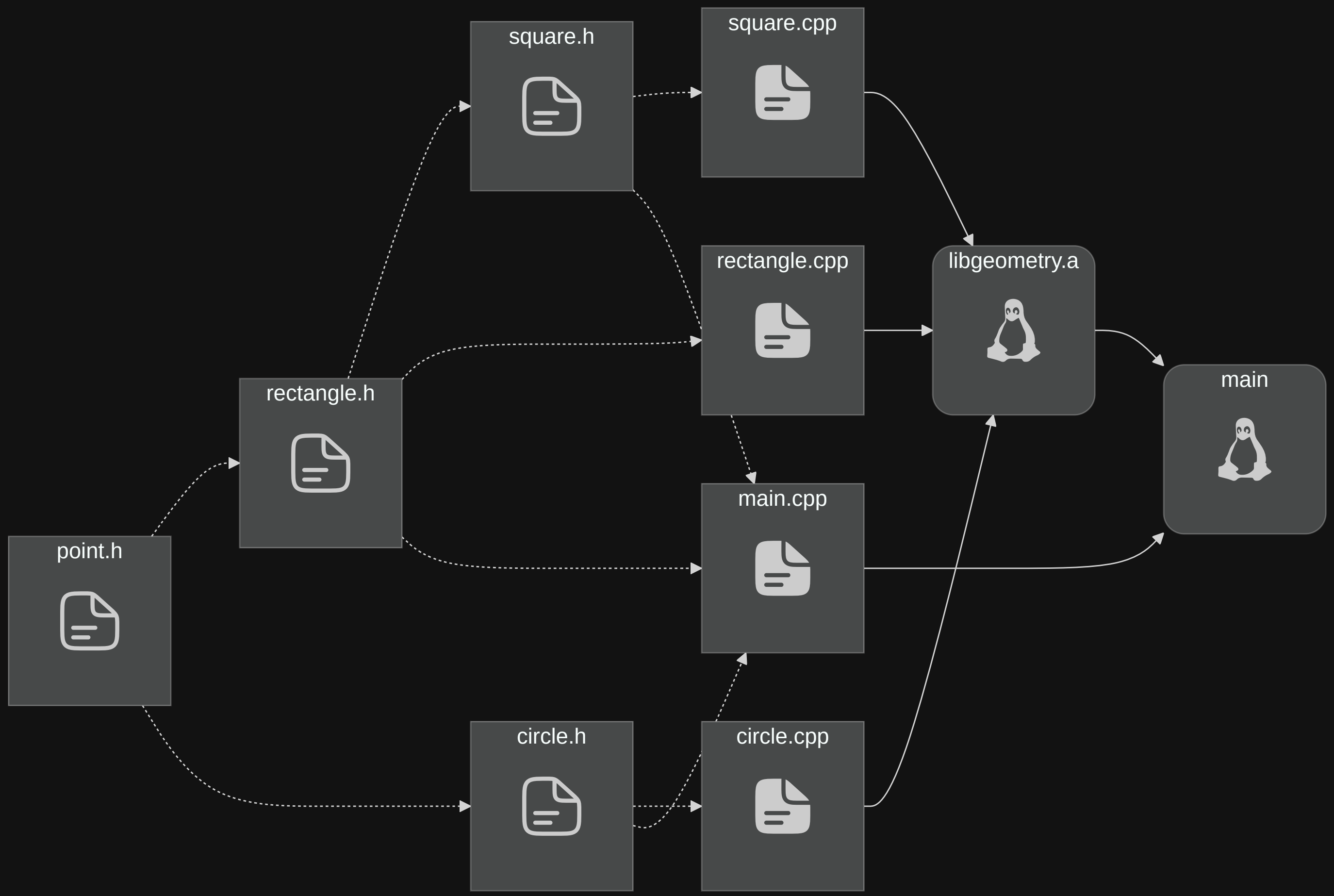
Suppose we have the following components:

- `libGeometry`
  - `Point`
  - `Rectangle`
  - `Square`
  - `Circle`
- `Main`

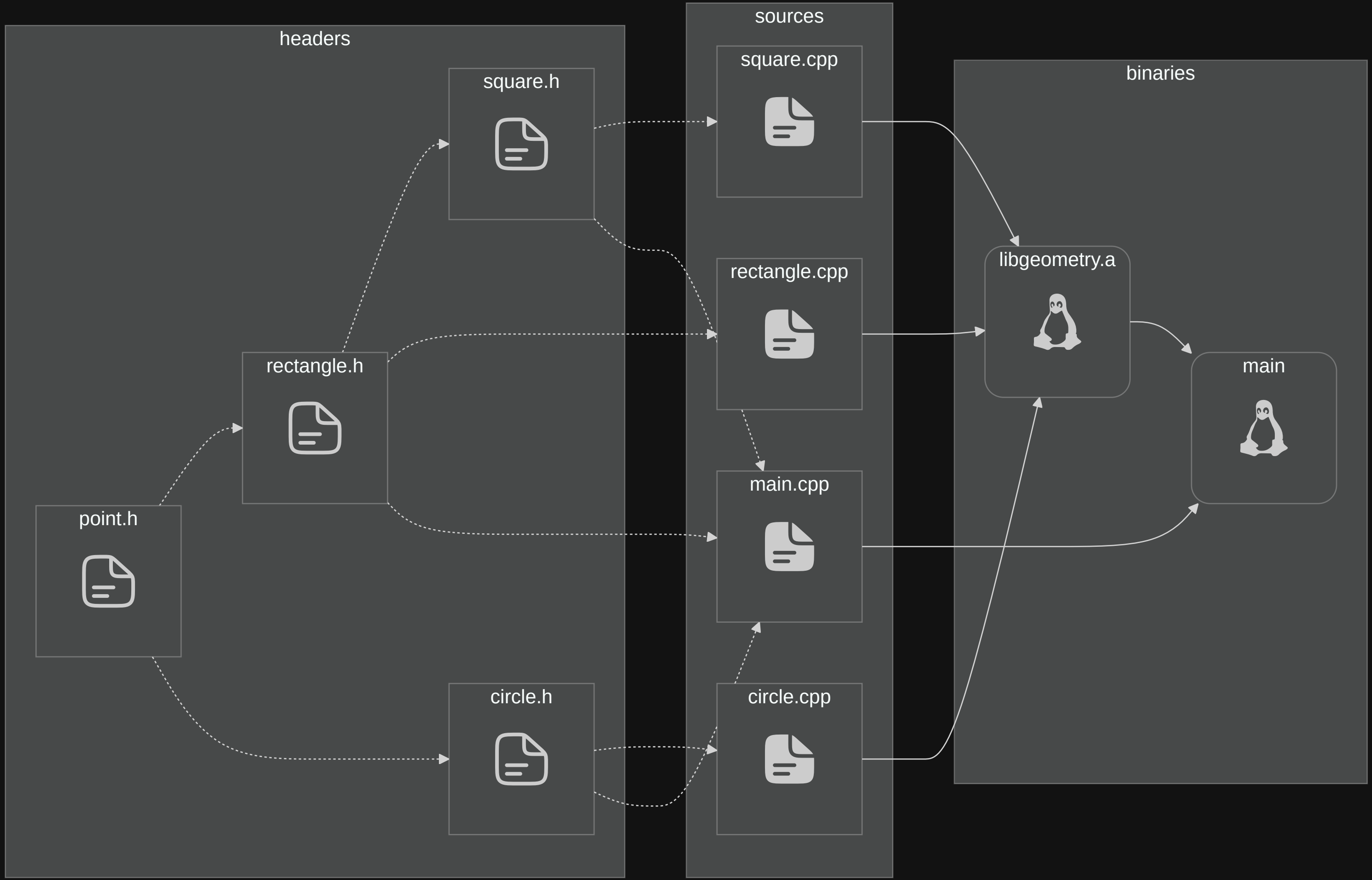
And the following rules:

`libGeometry` → `Main`

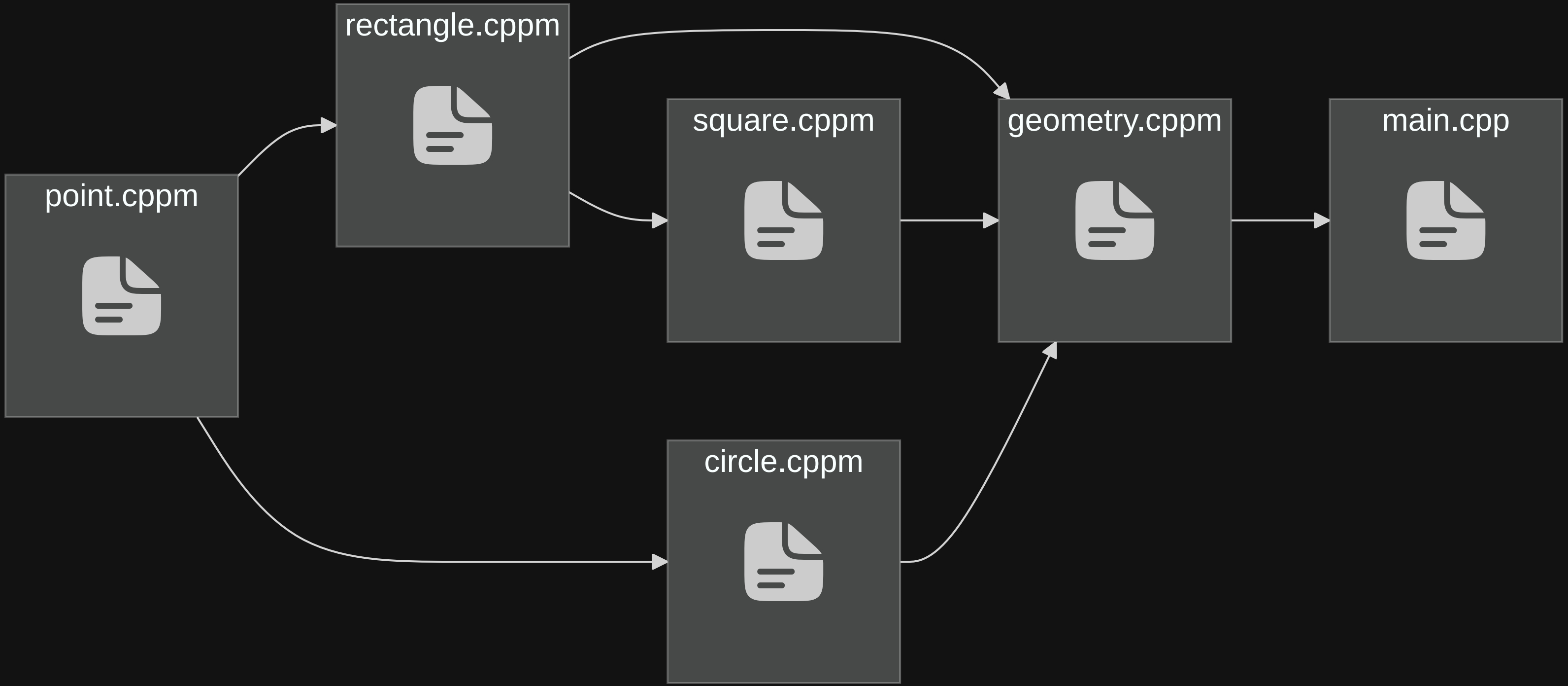
Build ordering



Build ordering



Build ordering with modules



Dependency scanning

Things are more complicated:

square.cppm

```cpp
export module square;

export import rectangle;
```

Dependency scanning

Things are more complicated:

square.cppm

```cpp
export module square;

export import rectangle;
```

We can "scan" the dependencies:

{ } square.cppm.ddi

```json
{
    "primary-output": "square.o",
    "provides": [{
        "compiled-module-path": "square.pcm",
        "logical-name": "square"
    }],
    "requires": [{
        "logical-name": "rectangle"
    }]
}
```

Dependency scanning

Things are more complicated:

square.cppm

```cpp
export module square;

export import rectangle;
```

We can "scan" the dependencies:

{ } square.cppm.ddi

```json
{
    "primary-output": "square.o",
    "provides": [{
        "compiled-module-path": "square.pcm",
        "logical-name": "square"
    }],
    "requires": [{
        "logical-name": "rectangle"
    }]
}
```

Dependency scanning

Things are more complicated:

square.cppm

```
export module square;

export import rectangle;
```

We can "scan" the dependencies:

square.cppm.ddi

```
{
    "primary-output": "square.o",
    "provides": [{
        "compiled-module-path": "square.pcm",
        "logical-name": "square"
    }],
    "requires": [{
        "logical-name": "rectangle"
    }]
}
```

Dependency scanning

Things are more complicated:

square.cppm

```
export module square;

export import rectangle;
```

We can "scan" the dependencies:

square.cppm.ddi

```
{
    "primary-output": "square.o",
    "provides": [{
        "compiled-module-path": "square.pcm",
        "logical-name": "square"
    }],
    "requires": [{
        "logical-name": "rectangle"
    }]
}
```

Modules support in CMake

CMake 3.28: Native support* for named modules (not header units)

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.28 FATAL_ERROR)

set(CMAKE_CXX_STANDARD 20)

project(ModulesExample CXX)

add_library(Lib)
target_sources(Lib
    PUBLIC FILE_SET modules TYPE CXX_MODULES FILES lib.cppm
    PRIVATE lib.impl.cpp
)

add_executable(Main main.cpp)
target_link_libraries(Main PRIVATE Lib)
```

* Supported on the `Ninja` and `Visual Studio` generators

Modules support in CMake

CMake 3.28: Native support* for named modules (not header units)

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.28 FATAL_ERROR)

set(CMAKE_CXX_STANDARD 20)

project(ModulesExample CXX)

add_library(Lib)
target_sources(Lib
    PUBLIC FILE_SET modules TYPE CXX_MODULES FILES lib.cppm
    PRIVATE lib.impl.cpp
)

add_executable(Main main.cpp)
target_link_libraries(Main PRIVATE Lib)
```

* Supported on the `Ninja` and `Visual Studio` generators

Modules support in CMake

CMake 3.28: Native support* for named modules (not header units)

CMakeLists.txt

```cmake
cmake_minimum_required(VERSION 3.28 FATAL_ERROR)

set(CMAKE_CXX_STANDARD 20)

project(ModulesExample CXX)

add_library(Lib)
target_sources(Lib
    PUBLIC FILE_SET modules TYPE CXX_MODULES FILES lib.cppm
    PRIVATE lib.impl.cpp
)

add_executable(Main main.cpp)
target_link_libraries(Main PRIVATE Lib)
```

* Supported on the `Ninja` and `Visual Studio` generators

Modules support in CMake

CMake 3.28: Native support* for named modules (not header units)

CMakeLists.txt

```cmake
cmake_minimum_required(VERSION 3.28 FATAL_ERROR)

set(CMAKE_CXX_STANDARD 20)

project(ModulesExample CXX)

add_library(Lib)
target_sources(Lib
    PUBLIC FILE_SET modules TYPE CXX_MODULES FILES lib.cppm
    PRIVATE lib.impl.cpp
)

add_executable(Main main.cpp)
target_link_libraries(Main PRIVATE Lib)
```

* Supported on the `Ninja` and `Visual Studio` generators

Modules support in CMake

CMake 3.28: Native support* for named modules (not header units)

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.28 FATAL_ERROR)

set(CMAKE_CXX_STANDARD 20)

project(ModulesExample CXX)

add_library(Lib)
target_sources(Lib
    PUBLIC FILE_SET modules TYPE CXX_MODULES FILES lib.cppm
    PRIVATE lib.impl.cpp
)

add_executable(Main main.cpp)
target_link_libraries(Main PRIVATE Lib)
```

* Supported on the `Ninja` and `Visual Studio` generators

Modules support in CMake

```
$ cmake .. -GNinja -DCMAKE_BUILD_TYPE=Release && ninja
```

```
[1/10] Scanning lib.cppm for CXX dependencies
[2/10] Scanning main.cpp for CXX dependencies
[3/10] Scanning lib.impl.cpp for CXX dependencies
[4/10] Generating CXX dyndep file CMakeFiles/Lib.dir/CXX.dd
[5/10] Generating CXX dyndep file CMakeFiles/Main.dir/CXX.dd
[6/10] Building CXX object CMakeFiles/Lib.dir/lib.cppm.o
[7/10] Building CXX object CMakeFiles/Lib.dir/lib.impl.cpp.o
[8/10] Building CXX object CMakeFiles/Main.dir/main.cpp.o
[9/10] Linking CXX static library libLib.a
[10/10] Linking CXX executable Main
```

Modules support in CMake

```
$ cmake .. -GNinja -DCMAKE_BUILD_TYPE=Release && ninja
```

```
[1/10] Scanning lib.cppm for CXX dependencies
[2/10] Scanning main.cpp for CXX dependencies
[3/10] Scanning lib.impl.cpp for CXX dependencies
[4/10] Generating CXX dyndep file CMakeFiles/Lib.dir/CXX.dd
[5/10] Generating CXX dyndep file CMakeFiles/Main.dir/CXX.dd
[6/10] Building CXX object CMakeFiles/Lib.dir/lib.cppm.o
[7/10] Building CXX object CMakeFiles/Lib.dir/lib.impl.cpp.o
[8/10] Building CXX object CMakeFiles/Main.dir/main.cpp.o
[9/10] Linking CXX static library libLib.a
[10/10] Linking CXX executable Main
```

Modules support in CMake

```
$ cmake .. -GNinja -DCMAKE_BUILD_TYPE=Release && ninja
```

```
[1/10] Scanning lib.cppm for CXX dependencies
[2/10] Scanning main.cpp for CXX dependencies
[3/10] Scanning lib.impl.cpp for CXX dependencies
[4/10] Generating CXX dyndep file CMakeFiles/Lib.dir/CXX.dd
[5/10] Generating CXX dyndep file CMakeFiles/Main.dir/CXX.dd
[6/10] Building CXX object CMakeFiles/Lib.dir/lib.cppm.o
[7/10] Building CXX object CMakeFiles/Lib.dir/lib.impl.cpp.o
[8/10] Building CXX object CMakeFiles/Main.dir/main.cpp.o
[9/10] Linking CXX static library libLib.a
[10/10] Linking CXX executable Main
```

Modules support in CMake

```
$ cmake .. -GNinja -DCMAKE_BUILD_TYPE=Release && ninja
```

```
[1/10] Scanning lib.cppm for CXX dependencies
[2/10] Scanning main.cpp for CXX dependencies
[3/10] Scanning lib.impl.cpp for CXX dependencies
[4/10] Generating CXX dyndep file CMakeFiles/Lib.dir/CXX.dd
[5/10] Generating CXX dyndep file CMakeFiles/Main.dir/CXX.dd
[6/10] Building CXX object CMakeFiles/Lib.dir/lib.cppm.o
[7/10] Building CXX object CMakeFiles/Lib.dir/lib.impl.cpp.o
[8/10] Building CXX object CMakeFiles/Main.dir/main.cpp.o
[9/10] Linking CXX static library libLib.a
[10/10] Linking CXX executable Main
```

Modules support in CMake

```
$ cmake .. -GNinja -DCMAKE_BUILD_TYPE=Release && ninja
```

```
[1/10] Scanning lib.cppm for CXX dependencies
[2/10] Scanning main.cpp for CXX dependencies
[3/10] Scanning lib.impl.cpp for CXX dependencies
[4/10] Generating CXX dyndep file CMakeFiles/Lib.dir/CXX.dd
[5/10] Generating CXX dyndep file CMakeFiles/Main.dir/CXX.dd
[6/10] Building CXX object CMakeFiles/Lib.dir/lib.cppm.o
[7/10] Building CXX object CMakeFiles/Lib.dir/lib.impl.cpp.o
[8/10] Building CXX object CMakeFiles/Main.dir/main.cpp.o
[9/10] Linking CXX static library libLib.a
[10/10] Linking CXX executable Main
```

Modules support in CMake: STL module

CMake 3.30: Experimental support for `import std;`

CMakeLists.txt

```cmake
cmake_minimum_required(VERSION 3.30 FATAL_ERROR)

set(CMAKE_CXX_STANDARD 23)
set(CMAKE_EXPERIMENTAL_CXX_IMPORT_STD "0e5b6991-d74f-4b3d-a41c-cf096e0b2508")
set(CMAKE_CXX_MODULE_STD ON)

project(ImportStd LANGUAGES CXX)

add_executable(main main.cpp)
```

Modules support in CMake: STL module

CMake 3.30: Experimental support for `import std;`

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.30 FATAL_ERROR)

set(CMAKE_CXX_STANDARD 23)
set(CMAKE_EXPERIMENTAL_CXX_IMPORT_STD "0e5b6991-d74f-4b3d-a41c-cf096e0b2508")
set(CMAKE_CXX_MODULE_STD ON)

project(ImportStd LANGUAGES CXX)

add_executable(main main.cpp)
```

Modules support in CMake: STL module

CMake 3.30: Experimental support for `import std;`

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.30 FATAL_ERROR)

set(CMAKE_CXX_STANDARD 23)
set(CMAKE_EXPERIMENTAL_CXX_IMPORT_STD "0e5b6991-d74f-4b3d-a41c-cf096e0b2508")
set(CMAKE_CXX_MODULE_STD ON)

project(ImportStd LANGUAGES CXX)

add_executable(main main.cpp)
```

Modules support in CMake: STL module

```
$ cmake .. -GNinja -DCMAKE_BUILD_TYPE=Release && ninja
```

```
[1/10] Scanning std.compat.ixx for CXX dependencies
[2/10] Scanning std.ixx for CXX dependencies
[3/10] Scanning main.cpp for CXX dependencies
[4/10] Generating CXX dyndep file CMakeFiles\__cmake_cxx23.dir\CXX.dd
[5/10] Generating CXX dyndep file CMakeFiles\main.dir\CXX.dd
[6/10] Building CXX object std.ixx.obj
[7/10] Building CXX object std.compat.ixx.obj
[8/10] Linking CXX static library __cmake_cxx23.lib
[9/10] Building CXX object CMakeFiles\main.dir\main.cpp.obj
[10/10] Linking CXX executable main.exe
```

Modules support in CMake: STL module

```
$ cmake .. -GNinja -DCMAKE_BUILD_TYPE=Release && ninja
```

```
[1/10] Scanning std.compat.ixx for CXX dependencies
[2/10] Scanning std.ixx for CXX dependencies
[3/10] Scanning main.cpp for CXX dependencies
[4/10] Generating CXX dyndep file CMakeFiles\__cmake_cxx23.dir\CXX.dd
[5/10] Generating CXX dyndep file CMakeFiles\main.dir\CXX.dd
[6/10] Building CXX object std.ixx.obj
[7/10] Building CXX object std.compat.ixx.obj
[8/10] Linking CXX static library __cmake_cxx23.lib
[9/10] Building CXX object CMakeFiles\main.dir\main.cpp.obj
[10/10] Linking CXX executable main.exe
```

Modules support in CMake: STL module

```
$ cmake .. -GNinja -DCMAKE_BUILD_TYPE=Release && ninja
```

```
[1/10] Scanning std.compat.ixx for CXX dependencies
[2/10] Scanning std.ixx for CXX dependencies
[3/10] Scanning main.cpp for CXX dependencies
[4/10] Generating CXX dyndep file CMakeFiles\__cmake_cxx23.dir\CXX.dd
[5/10] Generating CXX dyndep file CMakeFiles\main.dir\CXX.dd
[6/10] Building CXX object std.ixx.obj
[7/10] Building CXX object std.compat.ixx.obj
[8/10] Linking CXX static library __cmake_cxx23.lib
[9/10] Building CXX object CMakeFiles\main.dir\main.cpp.obj
[10/10] Linking CXX executable main.exe
```

Dependency managers

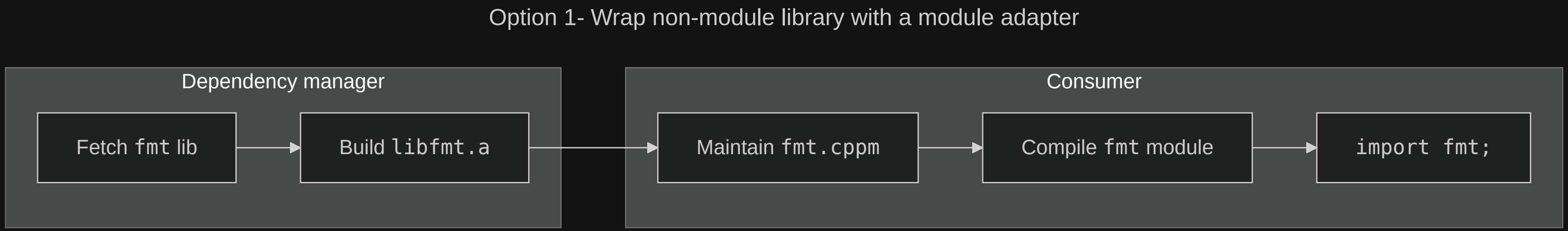
Dependency managers perform:

- Automated retrieval
- Transitive dependency handling
- Cross-platform consistency
- Build integration
- Binary caching

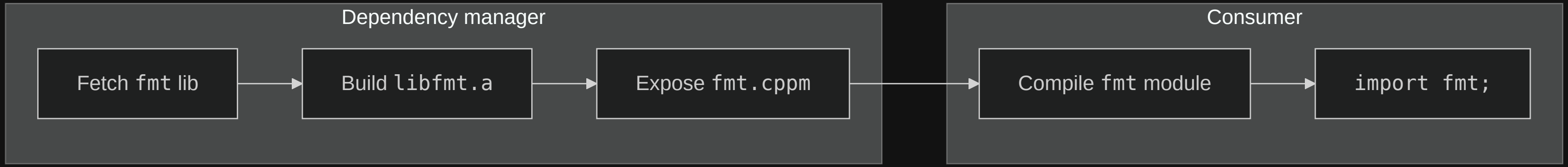
Dependency managers

Dependency managers perform:

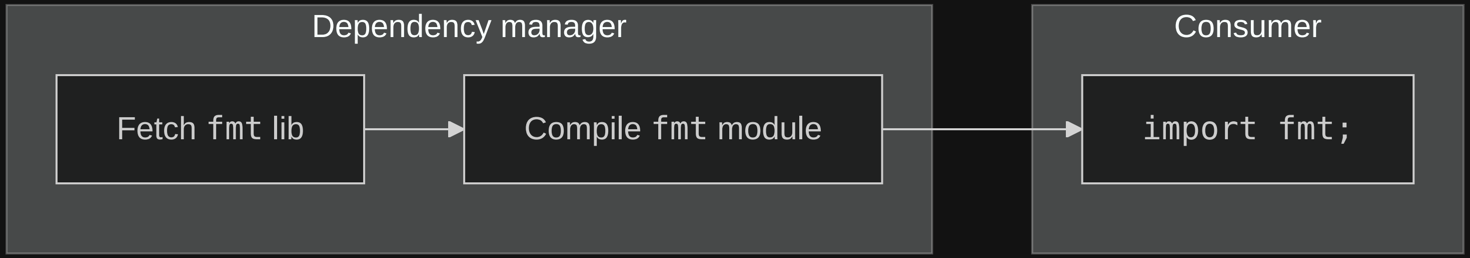
- Automated retrieval
- Transitive dependency handling
- Cross-platform consistency
- Build integration
- Binary caching


Option 1- Wrap non-module library with a module adapter
Dependency manager
Fetch fmt lib
Build libfmt.a
Consumer
Maintain fmt.cppm
Compile fmt module
import fmt;

Option 2- Ship module interface sources, consumer builds BMIs



Option 3- Ship prebuilt BMIs with package

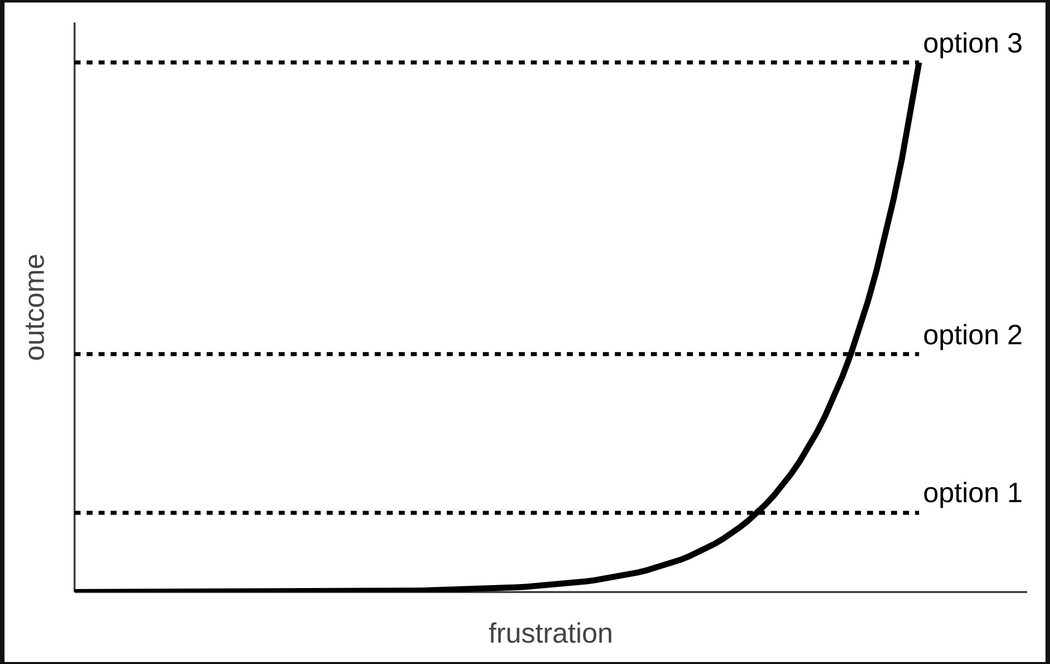


Dependency managers

The process can easily become:

- Platform-specific
- Difficult to maintain
- Error-prone

Better first-class support from tools is greatly awaited


option 3
option 2
option 1
outcome
frustration

Adopting Modules in Projects

Module-first greenfield designing

Starting out with modules is now recommended:

- Clang: "It is best for new code and libraries to use modules from the start if possible"
- Microsoft learn: "We recommend that you use modules in new projects rather than header files as much as possible"

Module-first greenfield designing

Starting out with modules is now recommended:

- Clang: "It is best for new code and libraries to use modules from the start if possible"
- Microsoft learn: "We recommend that you use modules in new projects rather than header files as much as possible"

And there are some real advantages:

- Adopting modules early minimizes transition cost
- Future improvements will be easily achievable
- Code can be structured for modules

Modules in existing projects

The choice is not always clear:

- Most projects will not rewrite everything at once
- Build system support increment is required
- Need for accustomed developers to learn new workflows

Modules in existing projects

The choice is not always clear:

- Most projects will not rewrite everything at once
- Build system support increment is required
- Need for accustomed developers to learn new workflows

Other build-time strategies might be better-suited:

- Precompiled headers (PCH)
- Unity builds

Header units

Using header units can bridge legacy and modern code:

- Modern access to legacy headers
- No changes are required for legacy code

Header units

Using header units can bridge legacy and modern code:

- Modern access to legacy headers
- No changes are required for legacy code

But there are some problems:

- Still experimental on some compilers
- Lack of build system support
- Macro-heavy headers will break

Incremental adoption

Moving step by step is often more practical:

- Start with utility libraries or self-contained components
- Gradually convert commonly used headers into modules
- Experiment with internal-only modules before exposing public ones
- Define strong code-guidelines for modules
- Regularly review build performance metrics

Future and Current Outlook

Modules in 2025

- C++20 introduced modules
- C++23 enhanced modules with the standard library modules
- C++26 proposes further refinements to improve tooling support

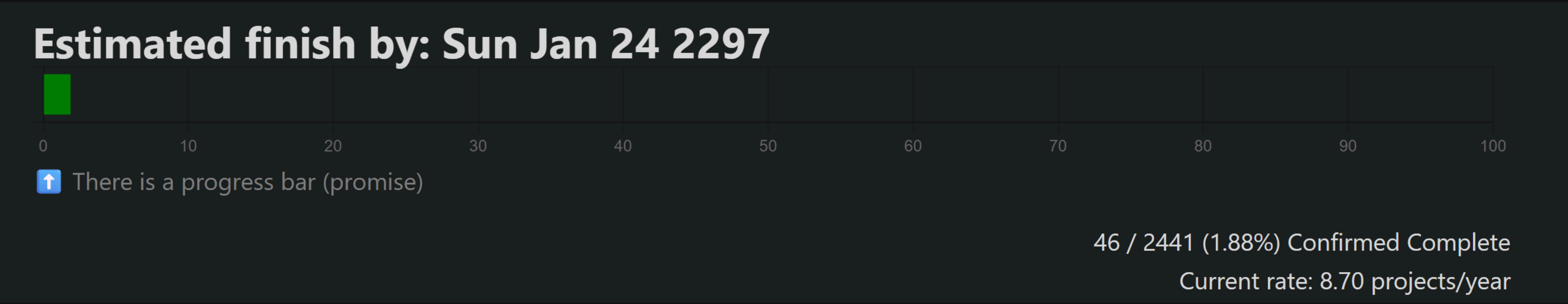
Modules in 2025

- C++20 introduced modules
- C++23 enhanced modules with the standard library modules
- C++26 proposes further refinements to improve tooling support

… But we’re still not quite there

Lack of ecosystem adoption

Source: Are We Modules Yet?



Adoption Challenges

- Chicken-egg: users vs tooling
- Compiler/build support is uneven
- Libraries rarely publish modules

Built Module Interfaces

Current implementations are underwhelming:

- Difficult to utilize
- Not an information hiding mechanism
- Not portable

Built Module Interfaces

Current implementations are underwhelming:

- Difficult to utilize
- Not an information hiding mechanism
- Not portable

That doesn’t have to be forever:

- BMIs could be shipped independently
- Shared libraries could be type-safe and self-descriptive
- Could enable Foreign Function Interfaces (FFIs)
- Cross-compiler compatibility could appear

New features and improvements

- CWG#2732 (2023) DR: Preprocessor definitions don't affect importing headers
- P3034: Forbids macro expansion in the name of module declarations
- P3618: Clarifies `main` usage in named modules
- P2577, P2701, P3286: Providing modules alongside prebuilt C++ libraries

Refactor-less improvements

The standard opens up possibilities:

[cpp.include]: implementation can treat *importable headers*

```
#include "someheader.h"
```

as if it were

```
import "someheader.h";
```

Refactor-less improvements

The standard opens up possibilities:

[cpp.include]: implementation can treat *importable headers*

```
#include "someheader.h"
```

as if it were

```
import "someheader.h";
```

Refactor-less improvements

The standard opens up possibilities:

[cpp.include]: implementation can treat *importable headers*

```
#include "someheader.h"
```

as if it were

```
import "someheader.h";
```

The Future of C++?

A new baseline

C++ modules shape the future of C++:

- Real encapsulation alongside build improvements
- Standardized importing semantics
- Easier boilerplate + migration
- AI can generate module-first code

Are modules the future of C++?

Are modules the future of C++? Yes!

Are modules the future of your current project?

Are modules the future of your current project? Maybe…

Thank you!

Appendix A - Compile-time performance (MSVC)

	#include	#include all	import	import all	import std
Helo world	0.55s	1.68s	0.11s	0.12s	0.12s
Mix	1.03s	1.76s	0.33s	0.26s	0.25s

Appendix A - Compile-time performance (MSVC)

	#include	#include all	import	import all	import std
Helo world	0.55s	1.68s	0.11s	0.12s	0.12s
Mix	1.03s	1.76s	0.33s	0.26s	0.25s

Appendix B - Modules support in Bazel

- Experimental community rules for modules

BUILD

```
load("@rules_cpp23_modules//cc_module:defs.bzl", "cc_module", "cc_module_binary", "cc_compiled_module")

cc_module(
    name = "lib",
    src = "lib.cppm",
    impl_srcs = ["lib.impl.cpp",],
    copts = ["-std=c++23", "-stdlib=libc++"],
)

cc_module_binary(
    name = "main",
    srcs = ["main.cpp",],
    deps = [":lib"],
    copts = ["-std=c++23"],
    linkopts = ["-stdlib=libc++"],
)
```

Appendix B - Modules support in Bazel

- Experimental community rules for modules

BUILD

```
load("@rules_cpp23_modules//cc_module:defs.bzl", "cc_module", "cc_module_binary", "cc_compiled_module")

cc_module(
    name = "lib",
    src = "lib.cppm",
    impl_srcs = ["lib.impl.cpp",],
    copts = ["-std=c++23", "-stdlib=libc++"],
)

cc_module_binary(
    name = "main",
    srcs = ["main.cpp",],
    deps = [":lib"],
    copts = ["-std=c++23"],
    linkopts = ["-stdlib=libc++"],
)
```

Appendix B - Modules support in Bazel

- Experimental community rules for modules

BUILD

```
load("@rules_cpp23_modules//cc_module:defs.bzl", "cc_module", "cc_module_binary", "cc_compiled_module")

cc_module(
    name = "lib",
    src = "lib.cppm",
    impl_srcs = ["lib.impl.cpp",],
    copts = ["-std=c++23", "-stdlib=libc++"],
)

cc_module_binary(
    name = "main",
    srcs = ["main.cpp",],
    deps = [":lib"],
    copts = ["-std=c++23"],
    linkopts = ["-stdlib=libc++"],
)
```

Appendix B - Modules support in Bazel

- Experimental community rules for modules

BUILD

```
load("@rules_cpp23_modules//cc_module:defs.bzl", "cc_module", "cc_module_binary", "cc_compiled_module")

cc_module(
    name = "lib",
    src = "lib.cppm",
    impl_srcs = ["lib.impl.cpp",],
    copts = ["-std=c++23", "-stdlib=libc++"],
)

cc_module_binary(
    name = "main",
    srcs = ["main.cpp",],
    deps = [":lib"],
    copts = ["-std=c++23"],
    linkopts = ["-stdlib=libc++"],
)
```

Appendix B - Modules support in Bazel

- Experimental community rules for modules

BUILD

```
load("@rules_cpp23_modules//cc_module:defs.bzl", "cc_module", "cc_module_binary", "cc_compiled_module")

cc_module(
    name = "lib",
    src = "lib.cppm",
    impl_srcs = ["lib.impl.cpp",],
    copts = ["-std=c++23", "-stdlib=libc++"],
)

cc_module_binary(
    name = "main",
    srcs = ["main.cpp",],
    deps = [":lib"],
    copts = ["-std=c++23"],
    linkopts = ["-stdlib=libc++"],
)
```

Appendix B - Modules support in Bazel

- Experimental community rules for modules

BUILD

```
load("@rules_cpp23_modules//cc_module:defs.bzl", "cc_module", "cc_module_binary", "cc_compiled_module")

cc_module(
    name = "lib",
    src = "lib.cppm",
    impl_srcs = ["lib.impl.cpp",],
    copts = ["-std=c++23", "-stdlib=libc++"],
)

cc_module_binary(
    name = "main",
    srcs = ["main.cpp",],
    deps = [":lib"],
    copts = ["-std=c++23"],
    linkopts = ["-stdlib=libc++"],
)
```

Appendix B - Modules support in Bazel

- Experimental community rules for modules

BUILD

```
load("@rules_cpp23_modules//cc_module:defs.bzl", "cc_module", "cc_module_binary", "cc_compiled_module")

cc_module(
    name = "lib",
    src = "lib.cppm",
    impl_srcs = ["lib.impl.cpp",],
    copts = ["-std=c++23", "-stdlib=libc++"],
)

cc_module_binary(
    name = "main",
    srcs = ["main.cpp",],
    deps = [":lib"],
    copts = ["-std=c++23"],
    linkopts = ["-stdlib=libc++"],
)
```

Appendix B - Modules support in Bazel

- Experimental community rules for modules

BUILD

```
load("@rules_cpp23_modules//cc_module:defs.bzl", "cc_module", "cc_module_binary", "cc_compiled_module")

cc_module(
    name = "lib",
    src = "lib.cppm",
    impl_srcs = ["lib.impl.cpp",],
    copts = ["-std=c++23", "-stdlib=libc++"],
)

cc_module_binary(
    name = "main",
    srcs = ["main.cpp",],
    deps = [":lib"],
    copts = ["-std=c++23"],
    linkopts = ["-stdlib=libc++"],
)
```

Appendix B - Modules support in Bazel

- Experimental community rules for modules

BUILD

```
load("@rules_cpp23_modules//cc_module:defs.bzl", "cc_module", "cc_module_binary", "cc_compiled_module")

cc_compiled_module(name="std", cmi="std.pcm")

cc_module(
    name = "lib",
    src = "lib.cppm",
    impl_srcs = ["lib.impl.cpp",],
    deps = [":std"],
    copts = ["-fmodule-file=std=std.pcm", "-std=c++23", "-stdlib=libc++"],
)

cc_module_binary(
    name = "main",
    srcs = ["main.cpp",],
    deps = [":lib", ":std"],
    copts = ["-fmodule-file=std=std.pcm", "-std=c++23"],
    linkopts = ["-stdlib=libc++"],
)
```

Appendix B - Modules support in Bazel

- Experimental community rules for modules

BUILD

```
load("@rules_cpp23_modules//cc_module:defs.bzl", "cc_module", "cc_module_binary", "cc_compiled_module")

cc_compiled_module(name="std", cmi="std.pcm")

cc_module(
    name = "lib",
    src = "lib.cppm",
    impl_srcs = ["lib.impl.cpp",],
    deps = [":std"],
    copts = ["-fmodule-file=std=std.pcm", "-std=c++23", "-stdlib=libc++"],
)

cc_module_binary(
    name = "main",
    srcs = ["main.cpp",],
    deps = [":lib", ":std"],
    copts = ["-fmodule-file=std=std.pcm", "-std=c++23"],
    linkopts = ["-stdlib=libc++"],
)
```

Appendix B - Modules support in Bazel

- Experimental community rules for modules

BUILD

```
load("@rules_cpp23_modules//cc_module:defs.bzl", "cc_module", "cc_module_binary", "cc_compiled_module")

cc_compiled_module(name="std", cmi="std.pcm")

cc_module(
    name = "lib",
    src = "lib.cppm",
    impl_srcs = ["lib.impl.cpp",],
    deps = [":std"],
    copts = ["-fmodule-file=std=std.pcm", "-std=c++23", "-stdlib=libc++"],
)

cc_module_binary(
    name = "main",
    srcs = ["main.cpp",],
    deps = [":lib", ":std"],
    copts = ["-fmodule-file=std=std.pcm", "-std=c++23"],
    linkopts = ["-stdlib=libc++"],
)
```

Appendix B - Modules support in Bazel

- Experimental community rules for modules

BUILD

```
load("@rules_cpp23_modules//cc_module:defs.bzl", "cc_module", "cc_module_binary", "cc_compiled_module")

cc_compiled_module(name="std", cmi="std.pcm")

cc_module(
    name = "lib",
    src = "lib.cppm",
    impl_srcs = ["lib.impl.cpp",],
    deps = [":std"],
    copts = ["-fmodule-file=std=std.pcm", "-std=c++23", "-stdlib=libc++"],
)

cc_module_binary(
    name = "main",
    srcs = ["main.cpp",],
    deps = [":lib", ":std"],
    copts = ["-fmodule-file=std=std.pcm", "-std=c++23"],
    linkopts = ["-stdlib=libc++"],
)
```

Appendix C - Modules support in Build2

buildfile

```
cxx.std = latest
cxx.features.modules=true

using cxx

./: libue{lib}: mxx{lib.cppm} cxx{lib.impl.cpp}
./: exe{main}: cxx{main.cpp} libue{lib}
```

Appendix C - Modules support in Build2

buildfile

```
cxx.std = latest
cxx.features.modules=true

using cxx

./: libue{lib}: mxx{lib.cppm} cxx{lib.impl.cpp}
./: exe{main}: cxx{main.cpp} libue{lib}
```

Appendix C - Modules support in Build2

buildfile

```
cxx.std = latest
cxx.features.modules=true

using cxx

./: libue{lib}: mxx{lib.cppm} cxx{lib.impl.cpp}
./: exe{main}: cxx{main.cpp} libue{lib}
```

Glossary:

- `libue{x}`: utility library for an executable called `x`

Appendix C - Modules support in Build2

buildfile

```
cxx.std = latest
cxx.features.modules=true

using cxx

./: libue{lib}: mxx{lib.cppm} cxx{lib.impl.cpp}
./: exe{main}: cxx{main.cpp} libue{lib}
```

Glossary:

- `libue{x}` : utility library for an executable called `x`
- `mxx{x.cppm}` : module source `x.cppm`
- `cxx{x.cppm}` : C++ source `x.cppm`

Appendix C - Modules support in Build2

buildfile

```
cxx.std = latest
cxx.features.modules=true

using cxx

./: libue{lib}: mxx{lib.cppm} cxx{lib.impl.cpp}
./: exe{main}: cxx{main.cpp} libue{lib}
```

Glossary:

- `libue{x}` : utility library for an executable called `x`
- `mxx{x.cppm}` : module source `x.cppm`
- `cxx{x.cppm}` : C++ source `x.cppm`
- `exe{x}` : executable called `x`

Appendix C - Modules support in Build2

buildfile

```
cxx.std = latest
cxx.features.modules=true

using cxx

./: libue{lib}: mxx{lib.cppm} cxx{lib.impl.cpp}
./: exe{main}: cxx{main.cpp} libue{lib}
```

Glossary:

- `libue{x}` : utility library for an executable called `x`
- `mxx{x.cppm}` : module source `x.cppm`
- `cxx{x.cppm}` : C++ source `x.cppm`
- `exe{x}` : executable called `x`

Appendix D - Sample code for fmt module adapter

fmt.cppm

```cpp
module;

#include <fmt/core.h>

export module fmt;

export namespace fmt
{
    using ::fmt::print;
    using ::fmt::println;
    using ::fmt::format;
    // ...
}
```

Appendix D - Sample code for fmt module adapter

fmt.cppm

```cpp
module;

#include <fmt/core.h>

export module fmt;

export namespace fmt
{
    using ::fmt::print;
    using ::fmt::println;
    using ::fmt::format;
    // ...
}
```

Appendix D - Sample code for fmt module adapter

fmt.cppm

```cpp
module;

#include <fmt/core.h>

export module fmt;

export namespace fmt
{
    using ::fmt::print;
    using ::fmt::println;
    using ::fmt::format;
    // ...
}
```

Appendix E - VCPKG: Module wrapper

{} vcpkg.json

```json
{
    "dependencies": [
        "fmt"
    ]
}
```

Appendix E - VCPKG: Module wrapper

{ } vcpkg.json

```json
{
    "dependencies": [
        "fmt"
    ]
}
```

CMakeLists.txt

```cmake
find_package(fmt REQUIRED CONFIG)

add_library(fmt-module)
target_sources(
    fmt-module
    PUBLIC
    FILE_SET modules
    TYPE CXX_MODULESFILES "fmt.cppm")

target_link_libraries(
    fmt-module
    PRIVATE
    fmt::fmt-header-only)
```

Appendix E - VCPKG: Module wrapper

vcpkg.json

```json
{
    "dependencies": [
        "fmt"
    ]
}
```

CMakeLists.txt

```cmake
find_package(fmt REQUIRED CONFIG)

add_library(fmt-module)
target_sources(
    fmt-module
    PUBLIC
    FILE_SET modules
    TYPE CXX_MODULESFILES "fmt.cppm")

target_link_libraries(
    fmt-module
    PRIVATE
    fmt::fmt-header-only)
```

fmt.cppm

```cpp
module;

#include <fmt/core.h>

export module fmt;

export namespace fmt
{
    using ::fmt::print;
    using ::fmt::println;
    using ::fmt::format;
    // ...
}
```

Appendix E - VCPKG: Module wrapper

{ } vcpkg.json

```json
{
    "dependencies": [
        "fmt"
    ]
}
```

CMakeLists.txt

```cmake
find_package(fmt REQUIRED CONFIG)

add_library(fmt-module)
target_sources(
    fmt-module
    PUBLIC
    FILE_SET modules
    TYPE CXX_MODULESFILES "fmt.cppm")

target_link_libraries(
    fmt-module
    PRIVATE
    fmt::fmt-header-only)
```

fmt.cppm

```cpp
module;

#include <fmt/core.h>

export module fmt;

export namespace fmt
{
    using ::fmt::print;
    using ::fmt::println;
    using ::fmt::format;
    // ...
}
```

main.cpp

```cpp
import fmt;

// ...
```

Appendix F - Conan: Module wrapper

conanfile.txt

```
[requires]
fmt/11.2.0

[options]
fmt/*:header_only=True
```

Appendix F - Conan: Module wrapper

conanfile.txt

```
[requires]
fmt/11.2.0

[options]
fmt/*:header_only=True
```

CMakeLists.txt

```
find_package(fmt REQUIRED CONFIG)

add_library(fmt-module)
target_sources(
    fmt-module
    PUBLIC
    FILE_SET modules TYPE CXX_MODULES
    FILES "fmt.cppm")

target_link_libraries(
    fmt-module
    PRIVATE
    fmt::fmt-header-only)
```

Appendix F - Conan: Module wrapper

conanfile.txt

```
[requires]
fmt/11.2.0

[options]
fmt/*:header_only=True
```

CMakeLists.txt

```
find_package(fmt REQUIRED CONFIG)

add_library(fmt-module)
target_sources(
    fmt-module
    PUBLIC
    FILE_SET modules TYPE CXX_MODULES
    FILES "fmt.cppm")

target_link_libraries(
    fmt-module
    PRIVATE
    fmt::fmt-header-only)
```

Appendix F - Conan: Module wrapper

conanfile.txt

```
[requires]
fmt/11.2.0

[options]
fmt/*:header_only=True
```

CMakeLists.txt

```
find_package(fmt REQUIRED CONFIG)

add_library(fmt-module)
target_sources(
    fmt-module
    PUBLIC
    FILE_SET modules TYPE CXX_MODULES
    FILES "fmt.cppm")

target_link_libraries(
    fmt-module
    PRIVATE
    fmt::fmt-header-only)
```

Appendix F - Conan: Module wrapper

conanfile.txt

```
[requires]
fmt/11.2.0

[options]
fmt/*:header_only=True
```

CMakeLists.txt

```
find_package(fmt REQUIRED CONFIG)

add_library(fmt-module)
target_sources(
    fmt-module
    PUBLIC
    FILE_SET modules TYPE CXX_MODULES
    FILES "fmt.cppm")

target_link_libraries(
    fmt-module
    PRIVATE
    fmt::fmt-header-only)
```

Appendix F - Conan: Module wrapper

conanfile.txt

```
[requires]
fmt/11.2.0

[options]
fmt/*:header_only=True
```

CMakeLists.txt

```
find_package(fmt REQUIRED CONFIG)

add_library(fmt-module)
target_sources(
    fmt-module
    PUBLIC
    FILE_SET modules TYPE CXX_MODULES
    FILES "fmt.cppm")

target_link_libraries(
    fmt-module
    PRIVATE
    fmt::fmt-header-only)
```

Appendix F - Conan: Module wrapper

conanfile.txt

```
[requires]
fmt/11.2.0

[options]
fmt/*:header_only=True
```

CMakeLists.txt

```
find_package(fmt REQUIRED CONFIG)

add_library(fmt-module)
target_sources(
    fmt-module
    PUBLIC
    FILE_SET modules TYPE CXX_MODULES
    FILES "fmt.cppm")

target_link_libraries(
    fmt-module
    PRIVATE
    fmt::fmt-header-only)
```

fmt.cppm

```
module;

#include <fmt/core.h>

export module fmt;

export namespace fmt
{
    using ::fmt::print;
    using ::fmt::println;
    using ::fmt::format;
    // ...
}
```

Appendix F - Conan: Module wrapper

conanfile.txt

```
[requires]
fmt/11.2.0

[options]
fmt/*:header_only=True
```

CMakeLists.txt

```
find_package(fmt REQUIRED CONFIG)

add_library(fmt-module)
target_sources(
    fmt-module
    PUBLIC
    FILE_SET modules TYPE CXX_MODULES
    FILES "fmt.cppm")

target_link_libraries(
    fmt-module
    PRIVATE
    fmt::fmt-header-only)
```

fmt.cppm

```
module;

#include <fmt/core.h>

export module fmt;

export namespace fmt
{
    using ::fmt::print;
    using ::fmt::println;
    using ::fmt::format;
    // ...
}
```

Appendix F - Conan: Module wrapper

conanfile.txt

```
[requires]
fmt/11.2.0

[options]
fmt/*:header_only=True
```

CMakeLists.txt

```
find_package(fmt REQUIRED CONFIG)

add_library(fmt-module)
target_sources(
    fmt-module
    PUBLIC
    FILE_SET modules TYPE CXX_MODULES
    FILES "fmt.cppm")

target_link_libraries(
    fmt-module
    PRIVATE
    fmt::fmt-header-only)
```

fmt.cppm

```
module;

#include <fmt/core.h>

export module fmt;

export namespace fmt
{
    using ::fmt::print;
    using ::fmt::println;
    using ::fmt::format;
    // ...
}
```

Appendix F - Conan: Module wrapper

conanfile.txt

```
[requires]
fmt/11.2.0

[options]
fmt/*:header_only=True
```

CMakeLists.txt

```
find_package(fmt REQUIRED CONFIG)

add_library(fmt-module)
target_sources(
    fmt-module
    PUBLIC
    FILE_SET modules TYPE CXX_MODULES
    FILES "fmt.cppm")

target_link_libraries(
    fmt-module
    PRIVATE
    fmt::fmt-header-only)
```

fmt.cppm

```
module;

#include <fmt/core.h>

export module fmt;

export namespace fmt
{
    using ::fmt::print;
    using ::fmt::println;
    using ::fmt::format;
    // ...
}
```

Appendix F - Conan: Module wrapper

conanfile.txt

```
[requires]
fmt/11.2.0

[options]
fmt/*:header_only=True
```

CMakeLists.txt

```
find_package(fmt REQUIRED CONFIG)

add_library(fmt-module)
target_sources(
    fmt-module
    PUBLIC
    FILE_SET modules TYPE CXX_MODULES
    FILES "fmt.cppm")

target_link_libraries(
    fmt-module
    PRIVATE
    fmt::fmt-header-only)
```

fmt.cppm

```
module;

#include <fmt/core.h>

export module fmt;

export namespace fmt
{
    using ::fmt::print;
    using ::fmt::println;
    using ::fmt::format;
    // ...
}
```

main.cpp

```
import fmt;

// ...
```

Appendix G - Conan: Packaging BMIs

conanfile.py

```python
class FmtBMI(ConanFile):
    name = "fmt-bmi"
    version="11.2.0"
    url = "https://github.com/fmtlib/fmt"
    settings = "os", "compiler", "build_type", "arch"
    generators = "CMakeToolchain", "CMakeDeps"
    exports_sources = []
    no_copy_source = True

 # ...
```

Appendix G - Conan: Packaging BMIs

conanfile.py

```python
class FmtBMI(ConanFile):
    # ...

    def source(self):
        git = Git(self)
        git.clone(url="https://github.com/fmtlib/fmt.git", target="fmt")
        git.folder="fmt"
        self.folders.source = "fmt"

        git.checkout(self.version)

    # ...
```

Appendix G - Conan: Packaging BMIs

conanfile.py

```python
class FmtBMI(ConanFile):
    # ...

    def build(self):
        cmake = CMake(self)
        cmake.configure(
            variables={
                "FMT_MODULE": True,
            }
        )
        cmake.build(target="fmt")

    # ...
```

Appendix G - Conan: Packaging BMIs

conanfile.py

```python
class FmtBMI(ConanFile):
    # ...

    def build(self):
        cmake = CMake(self)
        cmake.configure(
            variables={
                "FMT_MODULE": True,
            }
        )
        cmake.build(target="fmt")

    # ...
```

Appendix G - Conan: Packaging BMIs

conanfile.py

```python
class FmtBMI(ConanFile):
    # ...

    def package(self):
        cmake = CMake(self)
        cmake.install()

        copy(self, "fmt.pcm", src=".", dst=self.package_folder + "/res")
        copy(self, "src/fmt.cc", src="fmt", dst=self.package_folder + "/res")

        copy(self, "LICENSE.rst", src="fmt", dst=self.package_folder + "/licenses")

    # ...
```

Appendix G - Conan: Packaging BMIs

conanfile.py

```python
class FmtBMI(ConanFile):
    # ...

    def package(self):
        cmake = CMake(self)
        cmake.install()

        copy(self, "fmt.pcm", src=".", dst=self.package_folder + "/res")
        copy(self, "src/fmt.cc", src="fmt", dst=self.package_folder + "/res")

        copy(self, "LICENSE.rst", src="fmt", dst=self.package_folder + "/licenses")

    # ...
```

Appendix G - Conan: Packaging BMIs

conanfile.py

```python
class FmtBMI(ConanFile):
    # ...

    def package_info(self):
        self.cpp_info.components["fmt-bmi"].libs = ["fmt"]
        self.cpp_info.components["fmt-bmi"].set_property("cmake_target_name", "fmt-bmi::fmt-bmi")
        self.cpp_info.components["fmt-bmi"].resdirs = ["res"]

        pcm_path = os.path.join(self.package_folder, "res/fmt.pcm")
         self.cpp_info.components["fmt-bmi"].cxxflags = [f"-fmodule-file=fmt={pcm_path}"]
```

Appendix G - Conan: Packaging BMIs

conanfile.py

```python
class FmtBMI(ConanFile):
    # ...

    def package_info(self):
        self.cpp_info.components["fmt-bmi"].libs = ["fmt"]
        self.cpp_info.components["fmt-bmi"].set_property("cmake_target_name", "fmt-bmi::fmt-bmi")
        self.cpp_info.components["fmt-bmi"].resdirs = ["res"]

        pcm_path = os.path.join(self.package_folder, "res/fmt.pcm")
         self.cpp_info.components["fmt-bmi"].cxxflags = [f"-fmodule-file=fmt={pcm_path}"]
```

Appendix G - Conan: Packaging BMIs

conanfile.py

```python
class FmtBMI(ConanFile):
    # ...

    def package_info(self):
        self.cpp_info.components["fmt-bmi"].libs = ["fmt"]
        self.cpp_info.components["fmt-bmi"].set_property("cmake_target_name", "fmt-bmi::fmt-bmi")
        self.cpp_info.components["fmt-bmi"].resdirs = ["res"]

        pcm_path = os.path.join(self.package_folder, "res/fmt.pcm")
         self.cpp_info.components["fmt-bmi"].cxxflags = [f"-fmodule-file=fmt={pcm_path}"]
```

- Platform-specific

Appendix G - Conan: Packaging BMIs

conanfile.py

```python
class FmtBMI(ConanFile):
    # ...

    def package_info(self):
        self.cpp_info.components["fmt-bmi"].libs = ["fmt"]
        self.cpp_info.components["fmt-bmi"].set_property("cmake_target_name", "fmt-bmi::fmt-bmi")
        self.cpp_info.components["fmt-bmi"].resdirs = ["res"]

        pcm_path = os.path.join(self.package_folder, "res/fmt.pcm")
         self.cpp_info.components["fmt-bmi"].cxxflags = [f"-fmodule-file=fmt={pcm_path}"]
```

- Platform-specific
- Difficult to maintain

Appendix G - Conan: Packaging BMIs

conanfile.py

```python
class FmtBMI(ConanFile):
    # ...

    def package_info(self):
        self.cpp_info.components["fmt-bmi"].libs = ["fmt"]
        self.cpp_info.components["fmt-bmi"].set_property("cmake_target_name", "fmt-bmi::fmt-bmi")
        self.cpp_info.components["fmt-bmi"].resdirs = ["res"]

        pcm_path = os.path.join(self.package_folder, "res/fmt.pcm")
         self.cpp_info.components["fmt-bmi"].cxxflags = [f"-fmodule-file=fmt={pcm_path}"]
```

- Platform-specific
- Difficult to maintain
- Error-prone