

多文件编程
Programming in
Multiple Files

现代C++基础 Modern C++ Basics

Jiaming Liang, undergraduate from Peking University

- **Preprocessor**
- **Declaration and Definition**
 - **Header files and source files**
- **Namespace**
- **Inline**
- **Linkage**
- **XMake & how to make a library**
- **Modules**

Before that...

Modules	P1103R3 	11 (partial)	8 (partial)	19.0 (2015)* (partial) 19.10* (TS only) 19.28 (16.8)*	10.0.1* (partial)														
---------	---	--------------	-------------	---	-------------------	--	--	--	--	--	--	--	--	--	--	--	--	--	--

- C++20 has already introduced modules, which is supposed to accelerate compilation time dramatically.
- However, we still use header files in our lectures because:
 - Header files are widely used in previous C++ code in real world, and it's very likely that you still need to maintain them in your future career.
 - The progress of supporting modules for compilers is still forging ahead; compiler developers code day and night, but it's really a complex and huge project and yet not finished.
 - The standard library is fully "modularized" only in C++23 (which is also in progress); before that, gcc even doesn't support direct import of standard library.

Standard Library Header Units

The Standard Library is not provided as importable header units. If you want to import such units, you must explicitly build them first. If you do not do this with care, you may have multiple declarations, which the

Source: https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Modules.html

Before that...

- There are still many problems in modules, no matter bugs or performance issues.
 - Like in Clang: [modules increased building time](#), etc.
 - By the way, a major developer of modules in Clang is Chinese, [许传奇](#).
- Finally, IDE auto-completion / prompts are still miserable for modules; even in Visual Studio, modules may trigger false squiggles and be wrongly analyzed, which makes you feel like coding in Vim with no plug-ins.
- Anyway, I believe that modules will be significantly important in the future...
 - Isn't it cool to code in C++ like:
- But now it's not the time to use them.
 - You may try it in small experimental projects.

```
import std;

int main()
{
    std::println("Hello, world!");
    return 0;
}
```

Programming in multiple files

Preprocessor

Preprocessor

- Remember the compilation procedures of C learnt in ICS?
- 1. Preprocess, where comments are stripped and **macros are processed** (so that your `#include` can get the file correctly).
- 2. Compile, where each source file is compiled independently.
 - Def/Decl will cover it!
- 3. Assemble; this is not something we care since it just translate assembly to object file.
- 4. Link, to locate symbols that are referred in other TUs.
 - Linkage will cover it!

Preprocessor

Specified by `-finput-charset=xx` in gcc, and `/source-charset:xx` in msvc.

- To be specific, preprocessing can be divided into 6 phases:
 - 1. Read files and map them to translation character set, which guarantees UTF-8 to be supported (we'll cover UTF-8 in *String and Stream*; it's a superset encoding of ASCII).
 - Before C++23 / In C, it's implementation-defined to support Unicode or not.
 - 2. Backslashes at the end of line are processed to concatenate two physical lines into a single logical line.
 - 3. Comments and macros are extracted, and the whole file is parsed.
 - You'll learn how to parse a language in *Compiler Principles*.
 - Comments are then be stripped out.
 - 4. Preprocessor runs to process all macros.
 - 5. String literals are encoded as specified by the prefix.
 - Ones without prefix will be determined by execution character set.
 - 6. Adjacent string literals are concatenated (e.g. `"1" "2" -> "12"`).

Specified by `-fexec-charset=xx` in gcc, and `/execution-charset:xx` in msvc.

Preprocessor

- A preprocessing directive begins with **#**.
 - There are four kinds of directives.
 - 1. **#include** ..., which copies all file content into the current file.
 - **#include** <...> will find the file at system path or paths specified by compiler options (e.g. **-I**some/path).
 - **#include** “...” will find the file locally (e.g. relative to the path of current file); if it fails to do so, then falls back to <...>.
 - 2. **#define**, i.e. **macros**, which does pure text replacement.
 - For example: `#define WHAT "what?" << std::endl; std::cout << WHAT;`
 - Macros shouldn't recursively refer to each other, since preprocessor will suppress macros at the current substitution chain.
 - For example, **#define A B**, **#define B C** and **#define C A**, then **A** will expand to **B**, **B** will to **C**, but **C** will never to **A** since current chain is **A->B->C**, so any expansion of **A/B/C** will all be suppressed.

Macros

- Beyond that, macros can act as “functions”.

- e.g. `#define FUNC(a, b) a + b` `int a = FUNC(1, 2);`

- Still remember: it's pure text replacement, so: `#define SQR(a) a * a` `SQR(1 + 2);`

- Usually, you need to add squares everywhere, like:

`#define SQR(a) ((a) * (a))`

▶ `#define SQR(a) a * a`
扩展到: `1 + 2 * 1 + 2`

- This may also cause performance issue:

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

`MIN(SomeExpensiveFunc(1), SomeExpensiveFunc(2));`

▶ `#define MIN(a, b) ((a) < (b) ? (a) : (b))`

扩展到: `((SomeExpensiveFunc(1)) < (SomeExpensiveFunc(2)) ? (SomeExpensiveFunc(1)) : (SomeExpensiveFunc(2)))`

联机搜索

- The function will be called one more time!

Macros

- Note1: parameters of macros can be blank (e.g. `SQR()` will get `(())*(()`, `Func(,)` will get `+`).
- Note2: parameters of macros are directly parsed by commas when no parentheses surrounded, so e.g. `FUNC(SomeFunc<int, double>())` -> `SomeFunc<int + double>()` etc.
 - You need to add an additional pair of parentheses to help parse, e.g. `FUNC((SomeFunc<int, double>()), 3)`.
- Note3: You can use `...` for parameters of any number, and reference them by `__VA_ARGS__`. For example:

```
int foo(int, double, float) {};  
#define Func(...) foo(1, __VA_ARGS__)  
auto m = Func(1.0, 2.0f);
```

▶ #define Func(__VA_ARGS__) __VA_ARGS__
扩展到: foo(1, 1.0, 2.0f)

Macros

- Note4: since C++20, you can use `__VA_OPT__(content)`, which will be enabled only when `__VA_ARGS__` is not empty.

- Example:

```
#define F(...) f(0 __VA_OPT__(,) __VA_ARGS__)  
F(a, b, c) // replaced by f(0, a, b, c)  
F()        // replaced by f(0)
```

- Note5: `#` can be used to turn parameters to strings, and `##` can be used to concatenate parameters to make them a whole token.

- Example:

```
#define NameToStr(a, b) #a #b  
  
auto str = NameToStr(1, aaa);  
  
▶ #define NameToStr(a, b) #a #b  
扩展到: "1" "aaa"
```

```
#define ConcatName(a, id) a ## id  
  
int ConcatName(a, 1) = 0;  
std::cout << a1;
```

Macro replacement sequence*

- First, preprocessor of MSVC is non-standard until VS2019, and after that, you need to enable the standard one by `/Zc:preprocessor`.
- This is **optional**, and I really recommend you to **skip** this part; in case you may read some smelly code in the future, you can go back to this page at that time.
- The expansion can be divided into four steps:
 - 1. If a macro is function-like, then parse and expand its arguments first.
 - Macros not followed by `()` will be delayed to step 4 to expand.
 - 2. Parameters are substituted into body of it.
 - 3. Add the expanded macro to the suppress chain.
 - 4. Rescan from the substituted text to see whether macros that are not in suppress chain can be processed.

Macro replacement sequence*

- For example:

```
#define EMPTY
#define SCAN(x)      x
#define EXAMPLE_()   EXAMPLE
#define EXAMPLE(n)   EXAMPLE_ EMPTY()(n-1) (n)
```

- Then **EXAMPLE(5)** will be expanded to:
 - **EXAMPLE_ EMPTY()(5-1)(5)**.
 - Then **EXAMPLE** is added to suppress chain.
 - Then rescan the substituted result:
 - **EXAMPLE_** isn't followed by **()**, so it isn't substituted (conflict with definition).
 - **EMPTY** will be substituted to get **EXAMPLE_()(5-1)(5)**.
 - Rescan, but notice that the newly substituted one starts from blank (i.e. rescan **()(5-1)(5)**), so nothing happens.
- What about **SCAN(EXAMPLE(5))**?

Macro replacement sequence*

- Since `SCAN` is a function-like macro followed by `()`, so expand argument `EXAMPLE(5)` first.
 - Get `EXAMPLE_ EMPTY()(5-1)(5)`.
 - `EXAMPLE` is added to suppress chain.
 - Rescan with same analysis as before, you'll just get `EXAMPLE_()(5-1)(5)`.
 - Current expansion ends, thus `EXAMPLE` is popped up in suppress chain.
- Then `EXAMPLE_()(5-1)(5)` is substituted into body of `SCAN`, and `SCAN` is added to suppress chain (currently, only `SCAN` is in the chain).
- Then rescan the substituted result:
 - `EXAMPLE_` is followed by `()`, no arguments and so expands to `EXAMPLE`.
 - Then `EXAMPLE_` is added to suppress chain.
 - Rescan `EXAMPLE(5-1)(5)`, so expands `EXAMPLE(5-1)` to `EXAMPLE_ EMPTY()(5-1-1)(5-1)`; since `EXAMPLE_` is in suppress chain, so the final result is just `EXAMPLE_()(5-1-1)(5-1)(5)`.

See [here](#) for more examples.

Macros

- Macros are not controlled by namespace (all are seen as global), which makes it easy to pollute names out of the scope.
- For example, some libraries may indirectly include macro `max`, which will interfere with `std::max` in standard library.
 - For example, in `<Windows.h>/<minwindef.h>` of Windows.
 - Take care of your IDE color; function and macros have different colors.
- Then, you need `#undef` to drop the definition of the macro, i.e. disable it in the current scope.
 - Here it's `#undef max`.

Macros

- As a final word of function-like macros:
 - It's really **discouraged** to use function-like macros; just use functions instead!
 - Many cons:
 - 1. Its expansion is oblique, especially when macros are nested.
 - msvc doesn't support the standard one directly either.
 - 2. Don't support recursion.
 - 3. Parameter parsing by comma directly, which needs additional paratheses to help parse.
 - 4. May cause performance loss and need additional paratheses when writing definition, since it's pure text replacement.
 - 5. Operations of variadic arguments are limited (we'll tell you how to code variadic arguments in normal functions in the future, which is far more powerful).
 - 6. Not controlled by namespace, possible conflicts.

Preprocessor

- The third functionality is conditional choice of code...
- By whether the definition of macro exists:
 - `#ifdef`, (`#elifdef` since C++23), `#else`;
 - `#ifndef`, (`#elifndef` since C++23), `#else`;
 - `#if defined xx` is equivalent to `#ifdef xx`.
- You need to end them by an `#endif`.
- A classical usage is to have a `DEBUG` macro, and then:
 - If you don't define `DEBUG`, then this part of code will be omitted and not compiled.
 - We'll also use them in header files sooner.

```
#ifdef DEBUG
    std::cout << "Log something.";
#endif

#ifdef DEBUG
    std::cout << "Logs";
#else
    // ...
#endif
```

Preprocessor

- Or controlled by the value.

- `#if`, `#elif`, `#else`;

- They only judge pure literals instead of variables that can be determined at compilation time, since compilation happens after preprocessing.

- For example:

```
const int a = 1;
#if a
const int b = 1;
#endif
```

Instead:

```
#define a 1
#if a
const int b = 1;
#endif
```

- All non-macro names will be 0.
- There are some special tokens/macros:
 - `__has_include(...)` since C++17, which will detect whether the current file includes (no matter directly or indirectly) the file.

Preprocessor

See [here](#) for more predefined macros.

- Attribute testing macros, by `__has_cpp_attribute(...)`; e.g. `#if __has_cpp_attribute(nodiscard);`
- Language feature macros, e.g. `#if __cpp_lambdas` for lambda expressions.
- Library feature macros defined in `<version>` since C++20, e.g. `#if __cpp_lib_bit_cast` to check whether `std::bit_cast` is implemented.
- `__cplusplus`, which specifies C++ version (msvc needs `/Zc:__cplusplus` to enable it, otherwise always C++98 macro).
- However, these macros are usually not used, since as long as you specify standards, any normal compiler will implement 99% features.
- The final thing is to emit errors or warnings explicitly, by `#error ...` or `#warning ...`.
 - `#warning` is only officially supported in C++23, but before that many compilers support it as an extension.
 - For example:

```
#if __cplusplus < 201100L
#error At least C++11 should be used.
#endif
#error 指令: At least C++11 should be used.
```

Preprocessor

- Besides, you may have seen `#pragma` before.
 - For example, if you've taken HPC courses, you may know `#pragma unroll` to force the compiler to unroll the loop.
 - `#pragma` is in fact a macro for implementation-defined usage.
 - Check compiler docs or tools docs for pragma directives.
 - E.g. before C++11 & OpenMP 5.1, you need to use `#pragma` instead of attributes to specify parallel policies.

```
#pragma omp parallel for schedule(dynamic, 1) private(radiance)
[[omp::sequence(directive(parallel),
| directive(for, schedule(dynamic, 1), private(radiance))))]]
```

- We'll cover `#pragma once` and `#pragma pack` in the future, since these two are basically supported by all compilers.
- BTW, C++11 also allows you to write like `_Pragma("...")`.

Preprocessor

- Finally, you can use `#line` to specify the line number manually.

```
#define FNAME "test.cc"
int main()
{
#line 777 FNAME
    assert(2+2 == 5);
}
```

```
test: test.cc:777: int main(): Assertion `2+2 == 5' failed.
```

- This is widely used in files that generates another file automatically, whose error should be fixed in the current file.
 - E.g. you'll use Bison in Compiler Principle, which uses `.y` to generate a `.cpp`, but the error prompts are still in `.y`.

```
case 4:
#line 92 "/root/compiler/src/Parser/grammar.y"
```

Preprocessor

- Final word: you may usually see do-while in macro definitions.

- For example:

```
#define Func(a, b) do{\n    func(a);\n    func(b);\n} while(0)
```

Rather than:

```
#define Func(a, b) func(a); func(b);
```

- This is to prevent counter-intuitive cases like:

```
if (condition)\n    Func(1, 2);
```

- The second case will only wrap `func(1)` into the `if` clause, `func(2)` will always be executed.

Programming in multiple files

Declaration and Definition

Translation unit

- Now it comes to compilation time...
 - In ICS, we know that all source files are translated **separately**, and we finally link them to eliminate unknown symbols.
 - Such a unit is called translation unit (TU).
- However, since C++ needs to declare before use, each TU needs to see declaration first.
 - That's why you need to e.g. `#include <vector>` before using `std::vector`.
 - Yes, header files are used to expose common declarations, so others who include it will know "how to use them".

```
int func(int); // declaration
int main() { return func(0); }
// definition, can be in another .cpp.
int func(int a) { return a; }
```


Declaration and Definition

- So why do we need declaration, instead of just using definition?
 - C/C++ has One-Definition Rule (ODR), meaning that each entity should have only one definition in a TU or even in a program.
 - So, we can expose declarations as much as we can, but there should only be a single definition.
 - Though one may give others many business cards (名片), but he/she is always him/herself, the unique one.
- Several typical declarations & definitions:
 - Function prototype & one with function body, as we've seen before.
 - This includes methods in a class.
 - `class A;` & `class A{ ... }; ; struct` is the same.
 - `enum class A (: Type);` & with its enumerators.

Strictly speaking, a definition is also a declaration, “declaration” afterwards excludes this case.

Declaration and Definition

- An example first:

```
func.h
example1 > C func.h > Hello()
1 void Hello();

func.cpp
example1 > G+ func.cpp > Hello()
1 #include <iostream>
2
3 void Hello(){
4     std::cout << "Hello, world!\n";
5 }

main.cpp
example1 > G+ main.cpp > main()
1 #include "func.h"
2
3 int main()
4 {
5     Hello();
6     return 0;
7 }
```

C:\WINDOWS\system32\...
D:\111\University\Cou...
Hello, world!

- What if we put the whole definition in **func.h**?

```
func.h
example1 > C func.h > Hello()
1 #include <iostream>
2 void Hello(){
3     std::cout << "Hello, world!\n";
4 }

func.cpp
example1 > G+ func.cpp > Hello()
1 #include "func.h"
2
3 void Hello(){
4     std::cout << "Hello, world!\n";
5 }

main.cpp
example1 > G+ main.cpp > main()
1 #include "func.h"
2
3 int main()
4 {
5     Hello();
6     return 0;
7 }
```

error: main.cpp.obj : error LNK2005: "void __cdecl Hello(void)" (?Hello@@YAXXZ) 已经在 func.cpp.obj 中定义
build\windows\x64\release\example.exe : fatal error LNK1169: 找到一个或多个多重定义的符号

Declaration and Definition

- Reason: func.cpp defines a `Hello()`, while main.cpp defines one too (by `#include`, since `#include` copies everything to the current file).
 - You can check it by `/P` in msvc, or `-E` in clang/gcc.

```
#line 2 "D:\\111\\University\\Course\\main.cpp"
void Hello(){
    std::cout << "Hello, world!\n";
}
#line 2 "main.cpp"

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

msvc preprocessed result

```
# 2 "func.h"
void Hello(){
    std::cout << "Hello, world!\n";
}
# 2 "main.cpp" 2

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

g++ preprocessed result

Declaration and Definition

- We've said declaration of `class` is like `class A;;` so should we put it in the header and definition in the source file?
- Consider: we need to use methods and data members, so we should also expose their declarations to users.
 - Function prototype is enough for others to know how to call it; but class declaration is far more not enough!
 - Thus, though ODR requires the function to be defined only once in a program...
- It only requires the class to be defined only once **in a TU**.
 - And, different TUs should have the same definition of a class.
 - By defining the class in the header file and `#include` it, this is always true.
 - For member methods, if they're split out of class, it still needs to be defined only once in a program.

Declaration and Definition

- For example:

```
Vector3.h
example1 > C Vector3.h > Vector3 > Vector3(float, float, float)
1  class Vector3
2  {
3  public:
4      Vector3(float x, float y, float z);
5      float GetLength();
6  private:
7      float x_, y_, z_;
8  };

Vector3.cpp
example1 > C Vector3.cpp > ...
1  #include "Vector3.h"
2  #include <cmath>
3
4  Vector3::Vector3(float x, float y, float z)
5      : x_{x}, y_{y}, z_{z} {}
6
7  float Vector3::GetLength(){
8      return std::sqrt(x_ * x_ + y_ * y_ + z_ * z_);
9  }
```

```
main.cpp
example1 > C main.cpp > ...
1  #include "Vector3.h"
2  #include <iostream>
3
4  int main()
5  {
6      Vector3 v{ 1.0f, 2.0f, 3.0f };
7      std::cout << v.GetLength();
8      return 0;
9  }
```

C:\W...
D:\111\l...
3.74166
D:\111\l...

Declaration and Definition

- Besides, class definition requires to **fully** see all definitions of its members & base classes.
 - You cannot `class Vector3; class A { Vector3 v; };`
 - Formally, we say class cannot have members of **incomplete type**.
 - This is because compilers need to determine the layout of the class; if definitions of some members are unknown, its `sizeof` is unclear.
 - Otherwise how can it know `Vector3` has 3 `float` or 3 `double` when compiling separately?
- So why do we need class declarations?
 - You need to `#include` the header to see definitions.
 - So when the header changes, all other headers that `#include` the header needs will be indirectly changed too, i.e. the change propagate everywhere.
 - Then large re-compilation is needed...

Declaration and Definition

- But if we only use `class Vector3;` in headers, then changes in “`Vector3.h`” won’t affect them, which blocks the propagation.
 - We may still `#include “Vector3.h”` in source files, but since they’re compiled separately, re-compilation is much faster than changing everywhere.
- So typically, we can use mere class declarations in these cases:
 - 1. As an argument of prototype; its members and layout are not used.
 - 2. When you only needs a pointer or reference as members.
 - `Vector3` is incomplete type, but `Vector3*` is determined; it doesn’t have members, and the layout is just same as a pointer.
 - You may `#include “Vector3.h”` in source files to access members further.
 - Question: if we don’t use the class in function body, do we need definition?

```
void Hello2(Vector3 vec){  
    std::cout << "Hello2.\n";  
};
```

Yes, since parameters will be allocated on stack, so its layout needs to be known.

So what about `Vector3&`?

Layout determined, member unused, so no definition is OK.

Declaration and Definition

- Real case: if you need to use two classes that have member of each other...
 - This possibly indicates a bad design.
 - `#include "B.h"` in `A.h` & `#include "A.h"` in `B.h`?
 - No, e.g. in `B.h`, `A` needs to see full definition of `B`, but since `B` is after the `#include`, then `A` uses a member of incomplete type.
 - Solution: declare `class B;` in `A.h`, and declare `class A;` in `B.h`, then uses pointer as member.
 - i.e. `A` contains `B*`, `B` contains `A*`.
 - Then `#include "A.h"` in `B.cpp` and `#include "B.h"` in `A.cpp` to use methods and members.
 - If you find loop-include cases, then splitting declaration and definition is the solution.

Declaration and Definition

- Note1: it's allowed to put definition of methods into class definition (as we always do before).
- Note2: return types and parameter types of declarations and definitions of methods should be the same.
 - Particularly, all specifiers before prototype, **final** and **override** only need to appear in declaration; other specifiers need to appear at both declarations and definitions.
 - In other words, this includes cv-qualifier and **noexcept** specifier.

```
class Vector3
{
public:
    Vector3(float x, float y, float z);
    float GetLength() const noexcept;
private:
    float x_, y_, z_;
};
```

```
float Vector3::GetLength() const noexcept{
    return std::sqrt(x_ * x_ + y_ * y_ + z_ * z_);
}
```

Declaration and Definition

- Note3: **friend** will implicitly declares the class or function.

- For example: `friend void Hello2(Vector3);`
- No `#include "func.h"` in main.cpp!
- There are also other implicit ways, like `void Hello2(class Vector3)` to declare both `Vector3` and `Hello2`; but they're rarely used.

```
#include <iostream>
#include "Vector3.h"

int main()
{
    Hello2(Vector3{1,2,3})
    return 0;
}
```

- Note4: class members in class definition is in fact definition instead of just simple declaration, so we don't need to define it again in .cpp (e.g. `float Vector3::x_`).

- But **static** data members are just declaration, so you need to define them.

```
static auto GetDim(){ return dim_; }
private:
float x_, y_, z_;
static int dim_;
```

```
int Vector3::dim_ = 3;
```

Vector3.cpp

So static variables can be of incomplete type in headers, and only complete in sources...

Declaration and Definition

- Note5: default parameters of functions should be put into declaration, and shouldn't be put into definition.
 - Unless there is no declaration but just definition.
- Note6: type alias (like `using`) and `static_assert` are also declarations, i.e. it's safe to put them in headers.
- Note7: though it's discouraged to use global variables...
 - Still mention that their declaration and definition should also be split.
 - For example, `extern int x;` in header files, and `int x = 0;` in source files.
 - Reason: `int x` can be definition (i.e. an integer with random value), so `extern` should be added to denote that it's an declaration.
 - This requires the variable to have no initializer (i.e. not `extern int x = 1;`); otherwise it still becomes a definition and then may cause multiple definitions.
 - We'll cover `extern` sooner...

Declaration and Definition

- There is one more thing...
- We've said that a class definition should only appear once in a TU; so what if we `#include "a.h"` and `#include "b.h"`, while `b.h` indirectly `#include "a.h"`?
 - Then `a.h` will be copied twice, so a class definition will appear twice...
 - How to solve it?
- By **header guard**!

```
#ifndef VECTOR3_H_
#define VECTOR3_H_
class Vector3
{
public:
    Vector3(float x,
            float GetLength(
            static auto GetD
private:
    float x_, y_, z_
    static int dim_;
};

#endif // VECTOR3_H_
```

Reason: The first time it's included, `VECTOR3_H` isn't defined, so contents between `#ifndef` and `#endif` are preserved.

For the following times, since `VECTOR3_H` is already defined, all contents will be stripped so that redefinition won't happen.

Declaration and Definition

- Header guards are usually composed of namespace with the file name or class name.
- However, it may be troublesome to code a guard manually in a header, so most of compilers support `#pragma once`.
 - It'll automatically generate a header guard, and maybe faster.
 - The only flaw: headers cannot share their guards, so if some files are copied at many places, then headers will still be included multiple times.
 - But it's usually discouraged to copy a file everywhere, so not a big problem.
 - It's usually enough to just use `#pragma once`; but if you really needs to support compilers that may not have this feature, like writing a very basic library, you may use the original header guard.

```
#pragma once  
class Vector3
```

Template

- The final thing is template; should we also code like:

```
template<typename T>  
void Func(const T&);
```

In header.

```
template<typename T>  
void Func(const T&) { /* ... */ }
```

In source file.

- Hint: template will not preserve its information to object file.
 - It just specifies a rule to instantiate the code.
- So answer is no, because:
 - For other files that include the header, since they don't know the implementation, they'll only generate an instantiated **symbol** and hope to find it when linking.
 - For the file that has its implementation, it'll only instantiate ones it uses **in current TU**, since compilation happens separately!
 - Particularly, if you don't use it locally, no instantiation will be generated.

Template

It's also valid for function template to not specify `<int>` (i.e. `template void Func(const int&)`) to instantiate; template parameters will be deduced automatically.

- So linking will fail since symbols cannot be found.
- Thus, we need to put the template implementation into the header file, and those who use it will instantiate it directly, without delaying to linking stage.
- If you really want to put it into source files, then instances of templates are limited to what you've explicitly instantiated.
 - i.e. by `template void Func<int>(const int&);` in source file, this instance will be preserved in the object file.
 - No `<>` after `template`, no function body since it's generated by the template.
 - Then you can use `Func<int>` in other files, but still cannot `Func<float>`.

```
#pragma once
// declares the template
template<typename T> void Func(const T&);
```

```
#include "template.h"
|
// define the template
template<typename T>
void Func(const T&){ /* ... */ }

// instantiate explicitly.
template void Func<int>(const int&);
```

Template in class

- Function template in class should also be put into header.
 - But if you want, you can split declaration and definition in the header, like:

```
class A
{
public:
    template<typename T> void Func(T); // declaration
};

template<typename T>
void A::Func(T x){ /* ... */ } // definition
```

Functions have no `<...>`,
while classes have!

- Template parameters at different layers shouldn't be put together.

```
template<typename T>
class B
{
public:
    template<typename U> void Func(U);
};

template<typename T>
    template<typename U>
void B<T>::Func(U){ /* ... */ }
```

Of course, you can instantiate by
e.g. `template class B<void>;`
`template void B<int>::Func(int);`
to instantiate explicitly.

Summary

- To sum up:
 - Function declarations (prototype), class definitions and enum definitions should be put into header files.
 - Function definitions should be put into source files.
 - Definition of methods in class can be in-class or be split to source file, while the latter is friendlier to future changes since changes of headers will propagate.
 - Sometimes, if you don't need members and layout, class declaration is enough.
 - Templates should usually be put into header files unless you want to limit the instances of template code.
 - Header guards to maintain ODR.

Programming in multiple files

Namespace

Namespace

- When code base is large, name conflicts usually happen.
 - Namespace is quite like prefix of variables, which prevents such conflict.
- So you just need to `namespace XXX { ... }`, and put contents that are originally global into ...
 - For example:

```
namespace.h ×
example1 > C namespace.h > ...
1  #pragma once
2  namespace Test
3  {
4
5  class A
6  {
7  public:
8      A(int x): x_{x}{}
9      int GetX() const noexcept;
10 private:
11     int x_;
12 };
13
14 void Func();
15
16 } // namespace Test

namespace.cpp ×
example1 > G+ namespace.cpp > ...
1  #include "namespace.h"
2  #include <iostream>
3
4  namespace Test
5  {
6
7  int A::GetX() const noexcept { return x_; }
8
9  void Func(){ std::cout << "Wow.\n"; }
10
11 } // namespace Test
```

```
main.cpp ×
example1 > G+ main.cpp > main()
1  #include "namespace.h"
2
3  int main()
4  {
5      Test::A a{1};
6      Test::Func();
7      auto b = a.GetX();
8      return 0;
9  }
```

Namespace

- Namespace can also be nested:

```
namespace Test2
{
    namespace Test3
    {
        class B { public: static void Output(); };
    } // namespace Test2::Test3
} // namespace Test2
```

```
namespace Test2
{
    namespace Test3
    {
        void B::Output() { std::cout << "Output.\n"; };
    } // namespace Test2::Test3
} // namespace Test2

Test2::Test3::B::Output();
```

- Since C++17, you can also nest directly:

```
namespace Test4::Test5 // since C++17
{
    class C { public: static void Output(); };
}
```

```
namespace Test4::Test5{
    void C::Output() { std::cout << "Output C.\n"; };
}
```

- This is totally equivalent to `namespace Test4 { namespace Test5{ ... } }`.

Namespace

- Note1: You can also e.g. `namespace Test4 { void Test5::C::Output(){...} };`
 - Anyway, as long as the total name is same, they'll be seen as the same entity.
- Note2: Namespace content can be extended, i.e. you can add another `namespace Test4::Test5 { class D{...}; }`, and `C/D` are seen as in the same namespace.
- Note3: you can use namespace alias, e.g. `namespace Test45 = Test4::Test5;` it's similar to type alias.
 - As we have used in `namespace stdr = std::ranges;`
- Note4: you can add attribute between `namespace` and name, e.g.

```
namespace [[deprecated]] util  
{
```

But nested namespace doesn't support to add attributes directly.

Namespace

If you've learnt basic things about Python,
`using namespace xx` is like `from xx import *`,
`using yy::xx` is like `from yy import xx`.

- You can also `using namespace xxx`, so that you can use methods without namespace prefix.
 - Like `using namespace std`;
- You can also only introduce some entities by `using xx`; e.g. `using std::vector` so you can directly `vector<int>`, but `map` should still `std::map`.
 - Note: You should never put `using namespace xxx` or `using xxx` into header files!
 - Header files will be included everywhere, then names are polluted...
 - For example, I hope to use `bind` in socket library; if you `using namespace std` in header files, and `<functional>` is also included, then ambiguity between `bind` and `std::bind` appears!
 - It's Okay to do so in source files, though personally I rarely `using namespace std`.

Inline namespace

- Finally we introduce inline namespace.
 - Inline namespace has nothing to do with `inline` that we'll cover sooner; it's just reuse of the keyword `inline`.
 - It'll expose contents into the parent namespace, as if there is a `using namespace xx;`
 - For example:

```
namespace Test6
{
    inline namespace Implv1
    {
        void Func();
    }
}

namespace Test6::Implv1{
    void Func(){ std::cout << "This is v1.\n"; };
}

Test6::Func();
return 0;
```

No `Implv1::!`

Inline namespace

- This can be used for version control.
 - For example, if v2 is updated...

```
namespace Test6::Implv2{  
void Func(){ std::cout << "This is v2.\n"; }  
}
```

- Then, users can remain their code unchanged, as long as API interfaces aren't changed.
 - Otherwise, if users want to update, they have to change all `Test6::Implv1` to `Test6::Implv2`.
 - If users really want to fix the version, then it's Okay to use `Test6::Implv2::Func`, so future update won't affect it.
- If inline namespace has conflict name with parent namespace, compile error, which is safer than direct `using namespace xx`.

Of course, this version update needs re-compilation; if you don't want to re-compile, only shared library can work. But this isn't something `inline` can solve.

```
namespace Test6  
{  
    namespace Implv1  
    {  
        void Func();  
    }  
  
    inline namespace Implv2  
    {  
        void Func();  
    }  
}
```

You can strip out `inline` in `namespace Implv1`, add `inline` `namespace Implv2`.

Programming in multiple files

Inline

inline

- Recall procedures of function calls:
 - Caller saves registers on the stack;
 - Jump to the calling position;
 - Callee saves registers, execute code, and restores registers.
 - Jumping back by popping up the return address;
 - Caller restores registers on the stack.
- If we can fuse function body into the caller and optimize together, then register saving / restoring and jumping will be almost eliminated.
- Long long ago, `inline` is used to indicate the compiler to do so for a function.

inline

- To achieve this goal, every source file needs to be able to see its function body, thus:
 - inline function should be put into header files, and
 - Every TU will generate a symbol for inline function, and finally linker will recognize them and merge into one, so that ODR isn't violated.
 - This is same as class definition; it lowers ODR of function from the whole program to each TU.
- However, `inline` also has its own problems:
 - The code size may bloat, since the function body is inserted everywhere.
 - Utility of instruction cache may be lower, since the same function has different addresses.
 - Inline function won't create their function address, so you cannot "jump to the next" when debugging.

BTW, taking address of function to get the function pointer will make function always generate a function body address.

inline

Compilers basically preserve a dialect to “force” inline, e.g. `__forceinline` in MSVC and `__attribute__((always_inline))` in GCC;
But this is just a stronger suggestion and compiler may still omit it.

- So, `inline` is just an suggestion to compiler, and compiler may choose to not inline the body.
 - Recursive call cannot be inline, obviously.
 - And functions that are too complex won't be inlined.
 - It's hard for programmer to know whether it's proper to use `inline`.
- With the rapid development of compiler optimization, compilers today basically don't care whether you use `inline` or not.
 - Even though some definition of functions are put in the source file so they cannot be inlined when compiling...
 - Link-Time Optimization (LTO) is widely implemented, which makes inline when linking possible.
 - Thus, `inline` only preserve the meaning that you only need to obey ODR in a TU, not necessarily in a program like non-inline functions.

程序员：建议内联，当然我只是建议。

编译器：我知道，但没办法，宣布吧，我也不是猪，所以你的建议无效。

inline

- For example:

```
func.h
inline-example > C func.h > ...
1  #pragma once
2  #include <iostream>
3
4  inline void Hello()
5  {
6      std::cout << "Hello, world!\n";
7  }
8
9  void Hello2();

func.cpp
inline-example > G+ func.cpp > Hello2()
1  #include "func.h"
2  // Then func.cpp and main.cpp will
3
4  void Hello2() { Hello(); }

main.cpp
inline-example > G+ main.cpp > ...
1  #include "func.h"
2
3  int main()
4  {
5      Hello();
6      Hello2();
7      return 0;
8  }
```

inline

- Since C++17, inline variables are introduced.
 - We've said that global variables should also split declaration and definition;
 - So inline variables can be directly put into headers, and only needs to be ODR in each TU.
 - This mainly facilitates header-only libraries, since they hope to use some global variables without source files.
 - Inline variables can also be in class definitions, so that **static** variable can be unnecessary to split definition and declaration.
 - This thus needs **static** variables to be of complete type.
- Before C++17, you possibly can initialize a **static const** data member in class definition, but you still need a e.g. **const MemberClass A::a** in source file (but without initializer). Not discussed anymore.

```
struct A{  
    static inline int a = 1;  
};
```

inline

- However, inline functions / variables may cause double symbols in shared libraries.
 - We've said that inline functions will create symbols in every TU, and they will be merged as one when linking.
 - Introducing static library needs full re-link, so symbols can be merged.
 - However, for shared library:
 - So to load shared library quickly, many procedures are omitted, which includes resolving different RTTI symbols.
 - i.e. If you don't | `RTLD_GLOBAL` in `dlopen`, it'll be `RTLD_LOCAL` **by default**.
 - Then, symbols are not relocated against other symbols, then you get two different RTTI symbol addresses!

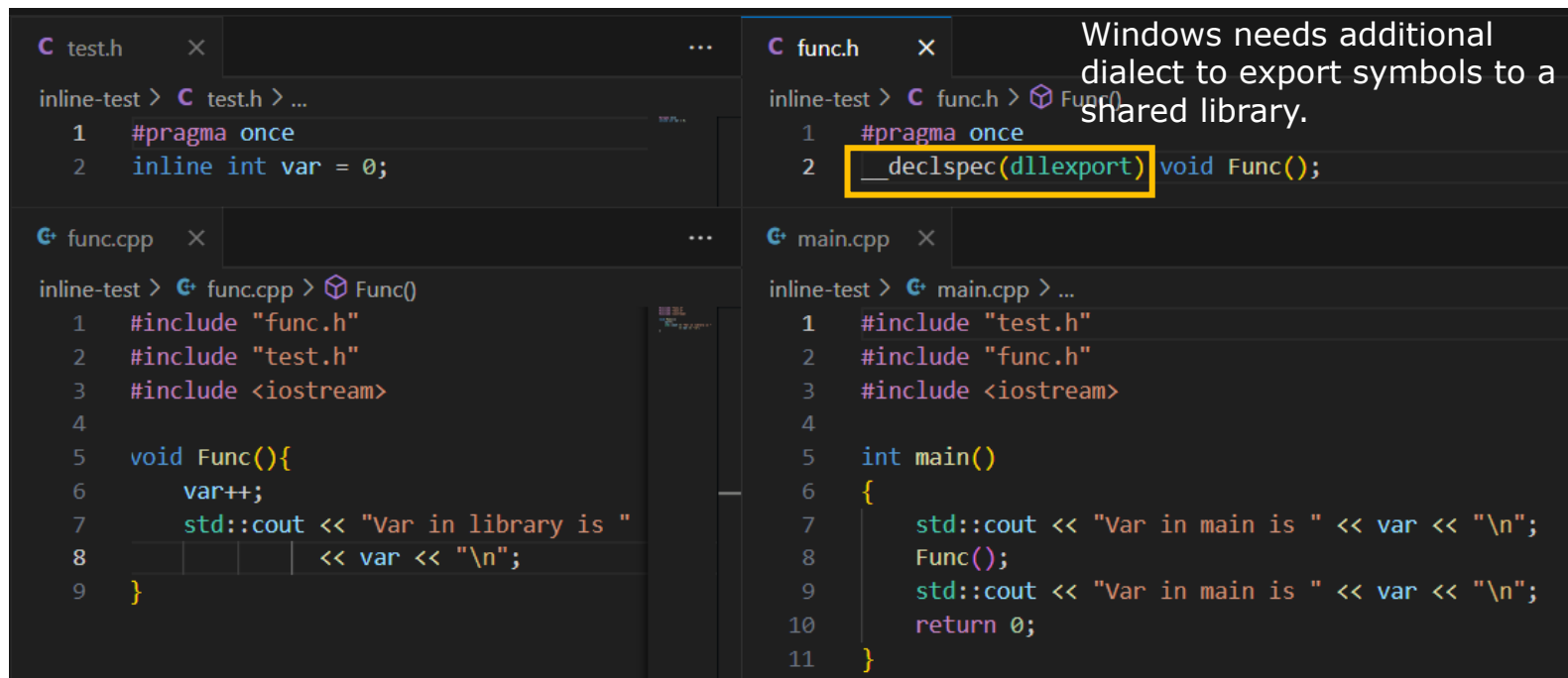
In *Lifetime and Type Safety*, RTTI section.

inline

- Thus, two versions of functions / variables are generated;
 - Methods in the library will call one...
 - While methods out of the library will call another.
 - This is a severe problem for non-const inline variables, or static variable in functions, since updates from different places are inconsistent.
- For example:

```
Var in main is 0
Var in library is 1
Var in main is 0
```

Solution: if some states may be modified, just declare in headers and put definition in source files.



```
test.h
1 #pragma once
2 inline int var = 0;

func.h
1 #pragma once
2 __declspec(dllexport) void Func();

func.cpp
1 #include "func.h"
2 #include "test.h"
3 #include <iostream>
4
5 void Func(){
6     var++;
7     std::cout << "Var in library is "
8     << var << "\n";
9 }

main.cpp
1 #include "test.h"
2 #include "func.h"
3 #include <iostream>
4
5 int main()
6 {
7     std::cout << "Var in main is " << var << "\n";
8     Func();
9     std::cout << "Var in main is " << var << "\n";
10    return 0;
11 }
```

Windows needs additional dialect to export symbols to a shared library.

Programming in multiple files

Linkage

Linkage

- We've discussed a lot about splitting declarations and definition;
 - But it seems that we always assume one thing – linker can always find the symbol in other TUs!
 - This is in fact **external linkage**; things below all have external linkage by default:
 - Class members;
 - Functions;
 - Functions declared by `friend`;
 - Enumerations;
 - Templates;
 - Non-const variables, volatile variables and inline variables.
 - Yes, almost all entities will have external linkage if no additional keyword is used.

Linkage

- But sometimes, we don't want the entity exposed.
 - For example, we implement a function in source file without declaration in header files, which means we only want to use it in the current TU.
 - But, if someone knows the prototype of function, then he can just declares the prototype at its own TU and linker will find it!
 - In other words, linker "steals" the privacy of our implementation.
- Thus, **static** is introduced to force **internal linkage**.
 - Notice that you can still split declaration and definition in a single file; it's a good code style to put all declarations at the beginning (so it's convenient to view all functions) and implementation at the end.
 - For example:

```
static int b = 1;
static void Func(); // declarations

static void Func(){
    // Cannot be referenced by other TUs.
}
```

Unlike static member functions, **static** that denotes linkage is necessary **in definition**.


Linkage

- Another way is to define things in an anonymous namespace.
 - All anonymous namespaces in a single TU are seen as the same one;
 - While anonymous namespaces from different TUs are seen as different.
 - Even though you give a named namespace in an anonymous namespace, it still has internal linkage.
 - For example:

```
namespace{  
  
void Func2(){  
    // Cannot be referenced by other TUs.  
}  
  
}
```

Linkage

- We have said that template should be put into header files...
 - Otherwise others cannot instantiate code directly.
 - So is it contradictory with the fact that template has external linkage?
 - No! Example:

linkage-test >  template.cpp > ...

Instantiated code has
external linkage.

```
1  #include <iostream>
2
3  template<typename T>
4  void NonStaticFunc() { std::cout << "Non static func.\n"; }
5
6  template<typename T>
7  static void StaticFunc() { std::cout << "Static func.\n"; }
8
9  template void NonStaticFunc<int>();
10 template void StaticFunc<int>();
```

```
template<typename T> void NonStaticFunc();
template<typename T> void StaticFunc();

int main()
{
    NonStaticFunc<int>();
    // StaticFunc<int>(); // Link-time error
```

Linkage

- However, there are some special cases where entities are born with internal linkage:
 - `const` global variables;
 - Anonymous unions.
 - Notice that these two cases are only in C++; C still views them as external.
- So if you hope to make them have external linkage, you need explicit `extern`.
 - Any entity with `extern` will have external linkage;
 - E.g. `extern void Func()` is equivalent to `void Func()`.
 - Personally, I'll add an `extern` if the declaration is not in the corresponding header, like refer to `Func` which declares in `a.h` and defines in `a.cpp`, but I hope to refer it in `b.cpp` without the header. This `extern` is a special notice that means "Definition not here".

Linkage

- So for example, if you hope to expose your const variables, you need to:
 - `extern const A xx;` in header or positions to refer it.
 - `extern const A xx{...};` in source file.
 - This is different with non-const variables...
 - If you don't specify `extern` explicitly in definition, it will have internal linkage and thus won't be seen as definition of the external one in header.
 - But `const` variables hardly ever need to be cross-TU, so it's not a big problem to adopt default internal one.
- One more thing to mention...
 - Though we've taught you a lot about linkage of global variables, it's discouraged to use global variables.
 - Here "global" also includes those wrapped into namespace.

Singleton

- Reason:
 - 1. The sequence of initializing global variables across TU isn't determined.
 - It's only determined in the same TU, which guarantees from first to last (as you've used lots of global variables in single file previously).
 - For example, if there is a global memory allocator `alloc` in a.cpp; and a global `Person person = alloc.Alloc(...);` in b.cpp.
 - That is, you're assuming `alloc` is initialized before `person`.
 - This is not always true; for instance, if you add a file c.cpp, then `person` may suddenly be initialized first.
 - But `alloc` has nothing, which causes logical error.
 - Similarly, destruction sequence isn't determined.
 - 2. Side effects caused by global variables may not be executed.
 - For example, if you have a class whose ctor will output "Hello", with a global variable `var` of that class type.

Singleton

- Then you make it a library.
- Sooner, you link another project against this library and expect "Hello"...
- Then surprisingly it disappears!
- This in fact not part of C++ standard, but related to OS & compiler regulation. Compiler may choose to not really link the library when symbols aren't used, and thus side effects of global variable don't happen.
- Solution:
 - Use global variable only when it has no side effects and not utilized by other TUs (i.e. internal linkage at best).
- However, sometimes we really need singleton pattern (单例模式) in our program, which really needs cross-TU.
 - Then, specify sequence manually by ourselves!

Singleton

- Then we manually `GetInstance()` one by one at the beginning of `int main()`.
- Since it's global instance, it doesn't need to release memory.
 - When the program exits, all resources are recycled by the OS.
 - Or if you need deallocation, then just add an `void DestroyInstance() { delete &GetInstance(); }` and call manually at the end.
- Or you can just `static Singleton instance{};`, which will automatically call dtor.
 - The deallocation order is reversed; so you cannot use it if:
 - The deallocation sequence is special.
 - Or dtor of some global variables need these singletons.

```
C singleton.h X
example1 > C singleton.h > ...
1  #pragma once
2  class Singleton
3  {
4  public:
5      static Singleton& GetInstance();
6  private:
7      Singleton() = default;
8  };

G singleton.cpp X
example1 > G singleton.cpp > GetInstance()
1  #include "singleton.h"
2
3  Singleton& Singleton::GetInstance(){
4      static Singleton* instance = new Singleton{};
5      return *instance;
6  }
```

Linkage

- The final linkage is no linkage.
 - This happens for static local variables (i.e. `static` variables in functions) and normal local variables.
 - You can even define a class inside the block scope, which also has no linkage.
 - All in all, they can only be referred in the current scope.

Programming in multiple files

XMake & How to make a library

XMake

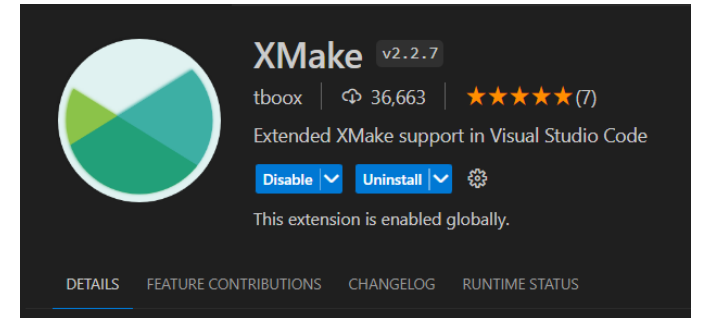
- Big project will have a bunch of headers and source files...
 - We may need to compile them to more than one executable,
 - Or possibly we need to package them to a library instead of executables,
 - Or you may need to specify options to debug different parts.
- All of these require a more powerful building tool.
 - In ICS, you've had a taste of MakeFile; besides...
 - CMake
 - It uses DSL to generate different configurations for different platforms, like generate MakeFile on Linux, and MSBuild on Windows, etc.
 - Pros: most widely used, document is sufficient.
 - Cons: DSL too ugly, not support C++20 modules until 2023, etc.
 - Scons, premake, autotools, bazel, etc...

```
file(GLOB CXX_SOURCES "src/*.cpp")
add_executable(main ${CXX_SOURCES})
target_link_libraries(main pthread)
```

XMake

- Usually, we also need a package manager to import external libraries.
 - In Python, you can just `pip / conda install` a package, like `numpy`.
 - But in C++, you possibly haven't used external libraries...
 - We need a package manager that can get the package easily, and may do some version control.
 - C++ doesn't have an official package manager, and you can use:
 - vcpkg, good on Windows, not good on other platforms...
 - conan (bazel + conan are popular)
 - System library, like apt-get in Linux
- [XMake](#) integrates them all; it acts as both a build tool and a package manager.

XMake



- Pros:
 - Use Lua as configuration language, better than DSL.
 - Developed by Chinese, [ruki](#).
 - Thus have Chinese documents.
- Cons:
 - Documents are not sufficient; you need to ask questions on github etc.
 - May need some Lua knowledge if you want to config some advanced scripts (but knowledge in this course doesn't require you to learn Lua).
- So let's see how to play with it!
 - Check <https://xmake.io/#/zh-cn/guide/installation> to install it first.
 - If you use VSCode, you can install extension.
 - More extensions like Clion can be found [here](#).

```
target("compiler")
  set_kind("binary")
  add_files("src/*.cpp")
  add_links("pthread")
```

XMake

- Particularly, XMake will install packages at home directory; that's usually fine on Linux, but not on Windows.
 - Install your software on D: drive because system won't use it...a full C drive will drag your system and reduce lifetime of your SSD.
- Besides, since some packages may come from Github, while visiting Github is not stable, you may tell xmake to use mirror websites.
 - XMake provides one internally at `scripts/pac/github_mirror.lua`.

I install it on D:\Work\CS\BuildTools\Xmake\Software, so it's in D:\Work\CS\BuildTools\Xmake\Software\scripts\pac\github_mirror.lua; Other two paths are random, just make it in D drive.

```
PS D:\Work\CS> xmake g --pkg_installdir="D:\Work\CS\BuildTools\XMake\Packages"
configure
{
  pkg_installdir = D:\Work\CS\BuildTools\XMake\Packages
  theme = default
  proxy_pac = pac.lua
  network = public
  pkg_cachedir = D:\Work\CS\BuildTools\XMake\Caches\
}
PS D:\Work\CS> xmake g --pkg_cachedir="D:\Work\CS\BuildTools\XMake\Caches"
configure
{
  theme = default
  pkg_installdir = D:\Work\CS\BuildTools\XMake\Packages
  proxy_pac = pac.lua
  pkg_cachedir = D:\Work\CS\BuildTools\XMake\Caches
  network = public
}
PS D:\Work\CS> xmake g --proxy_pac="D:\Work\CS\BuildTools\XMake\Software\scripts\pac\github_mirror.lua"
configure
{
  network = public
  pkg_cachedir = D:\Work\CS\BuildTools\XMake\Caches
  theme = default
  proxy_pac = D:\Work\CS\BuildTools\XMake\Software\scripts\pac\github_mirror.lua
  pkg_installdir = D:\Work\CS\BuildTools\XMake\Packages
}
```


XMake

- You need to configure your project in xmake.lua with:
 - (Optional) Project name, project version and required xmake version.
 - In informal project, they can be omitted.
 - Modes and language version.
 - Usually we only need debug and release modes; there are also [other modes](#), like least code size.
 - Language version, like cxxlatest, cxx20, c17.
 - Some other options.
 - Personally I will add `-Wall` (warnings are as important as errors!).
 - `set_policy("build.warning", true)` means that report warnings even if compile success.
 - And required packages.

```
set_project("Programming in Multiple Files")
set_xmakever("2.8.1")
set_version("0.0.0")
```

```
add_rules("mode.debug", "mode.release")
set_languages("cxx20")
```

```
set_policy("build.warning", true)
set_warnings("all")
```

```
add_requires("ctre 3.8.1", "catch2")
```

XMake

- Then we need to specify the building target.

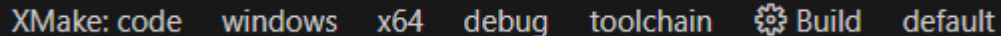
- For example:

```
target("example")
    set_kind("binary")
    add_headerfiles("example1/*.h")
    add_files("example1/*.cpp")
```

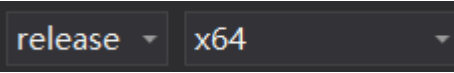
- `target(name)`
- `set_kind(...)`
 - “binary” will compile to the executable;
 - “static” – static library;
 - “shared” – shared library / dynamic-linked library;
 - “phony” – empty, just used to combine targets like libraries;
 - “header_only” – for projects that only have header files.
- `add_files(...)`
 - Add source files, support wildcard like `**` and `*`.
- `add_headerfiles(...)`
 - Add header files; this is optional, but if you hope to generate `.sln` with headers, you need it.

XMake

- So a full project is possibly like:
- Then, just **xmake** to compile all targets...
 - Or **xmake -b xx** to compile specific targets.
 - Or **xmake run xx** to run specific executable.
- You may change debug/release mode at **bottom** bar in VSCode:



XMake: code windows x64 debug toolchain Build default

- Or top bar in Visual Studio,  or other positions in other IDEs.
- You may also use **xmake f -m debug** to manually switch to debug mode, etc..

```
set_project("Programming in Multiple Files")
set_xmakever("2.8.1")
set_version("0.0.0")

add_rules("mode.debug", "mode.release")
set_languages("cxx20")
set_policy("build.warning", true)
set_warnings("all")

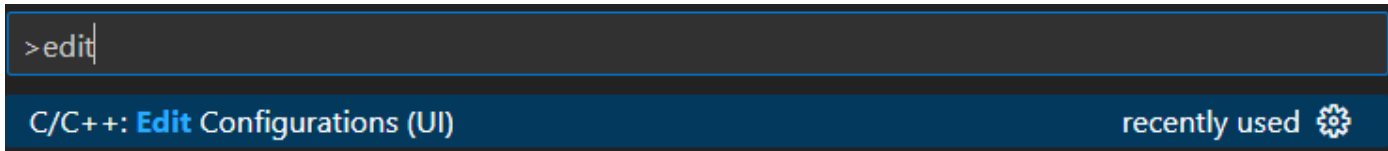
target("example")
    set_kind("binary")
    add_headerfiles("example1/*.h")
    add_files("example1/*.cpp")

target("inline-example")
    set_kind("binary")
    add_headerfiles("inline-example/*.h")
    add_files("inline-example/*.cpp")

target("inline-static-lib")
    set_kind("static")
    add_headerfiles("inline-test/*.h")
    add_files("inline-test/func.cpp")
```

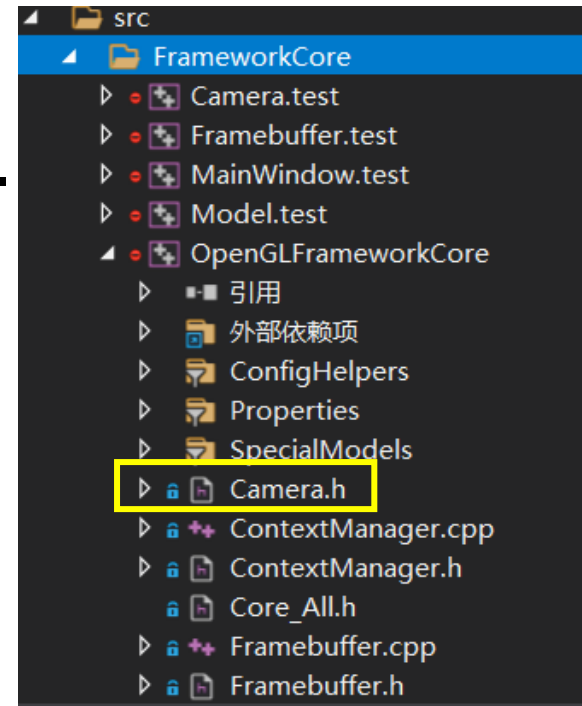
XMake

- Note1: for Visual Studio and XCode, you can use:
 - `xmake project -k xcode.`
 - `xmake project -k [vs2010|vs2013|...|vs2019|vs2022] -m "debug;release".`
 - You may also add a `add_rules("plugin.vsxmake.autoupdate")` to detect your update of xmake.lua, so `.sln` will be updated automatically.
- Note2: for VSCode, to enable correct auto completion, you need to edit `c_cpp_properties.json`.
 - Ctrl + Shift + P to find:



- Click it, and the json file will be generated at `.vscode`.

Headers like this are added to `.sln` only when you specify `add_headerfiles`.

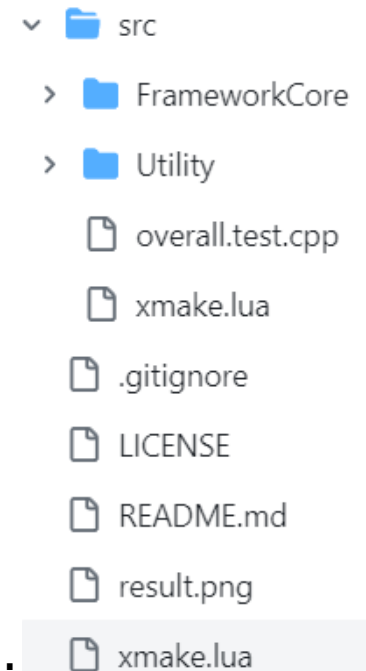


XMake

Every time you save the xmake.lua, `compile_commands.json` will be updated;
If it doesn't work, add `add_rules("plugin.compile_commands.autoupdate", {outputdir = ".vscode"})` in xmake.lua so build will update it.

- Then edit it...
 - Add a `"compileCommands"`, and vscode will automatically find language versions, etc.
 - This is for source files; not enough for header files; anyway, a complete code example is given.
- Note3: you can distribute xmake.lua to sub-directories, and `includes(...)` will process them together.
 - When the project is huge, you may split them into many xmake.lua to keep each one clean.

```
    ],  
    "intelliSenseMode": "windows-msvc-x64",  
    "compileCommands": ".vscode/compile_commands.json"  
  }  
  "version": 4
```



- ▼ src
- > FrameworkCore
- > Utility
- overall.test.cpp
- xmake.lua
- .gitignore
- LICENSE
- README.md
- result.png
- xmake.lua

add things like versions in the root
directory, and finally `includes("src")`
This will make it find `src/xmake.lua`.

XMake

- For package management, XMake integrates `xrepo`.
 - `xrepo search xx` to search the package...
 - And `xrepo install xx` to install it.
 - Some packages, like `cuda`, is only fetched from system; that is, you need to install it manually while `xmake` can find it automatically.
 - And `xrepo remove xx` to remove it.
 - `xrepo update-repo` to update packages from remote.
 - The official packages will be updated frequently.
- In `xmake.lua`, `add_requires("xx", "yy")` is used to specify packages you require users to install.
 - If users don't have, `xmake` will ask them.
 - And when a targets needs that package, just `add_packages("xx")`.
 - You can also `add_packages` globally if all targets need it.

XMake

- For example: `add_requires("ctre 3.8.1", "catch2")`

```
target("lexer")
  set_kind("static")
  add_files("hw-Lexer/Lexer.cpp")
  add_packages("ctre")
```

- There are more things in XMake, and we'll only use a little part of them in this course.
 - To learn more, read the full document, or zhihu articles like [xq114](#) or [Pointer](#).
 - BTW, [xq114](#) is a major contributor of XMake; he is also a PKUer in School of Physics, who won gold medal in IPhO in 2018.

How to make a library

- So with Xmake, it's easy to make a static library.
 - Just `set_kind("static")`, and source files shouldn't contain `int main()`.
- But for shared libraries, it's still a little bit clumsy (no matter in which building tool).
 - Library provider can determine symbols to be export.
 - In msvc, symbols are not exported by default, while gcc/clang are.
 - And msvc specifies visibility by dialect `__declspec(xx)`, while GCC/Clang by `__attribute__((visibility(xx)))`.
 - So to maintain cross-platform, you need a DLLMacro.h.

How to make a library

- For example:

```
#if defined _WIN32 || defined __CYGWIN__
#define DLL_IMPORT __declspec(dllimport)
#define DLL_EXPORT __declspec(dllexport)
#define DLL_LOCAL
#else
#if __GNUC__ >= 4
#define DLL_IMPORT __attribute__((visibility ("default")))
#define DLL_EXPORT __attribute__((visibility ("default")))
#define DLL_LOCAL __attribute__((visibility ("hidden")))
#else
#define DLL_IMPORT
#define DLL_EXPORT
#define DLL_LOCAL
#endif
#endif
```

```
#if defined DLL_MACRO_NEED_IMPORT
#define DLL_PORT DLL_IMPORT
#elif defined DLL_MACRO_NEED_EXPORT
#define DLL_PORT DLL_EXPORT
#else
#define DLL_PORT
#endif

#undef DLL_MACRO_NEED_IMPORT
#undef DLL_MACRO_NEED_EXPORT
```

Constrain the macros to only have effects locally, otherwise `#include` will affect other headers..

How to make a library

- And for the header that is the interface of the library, code like:
- For the shared library, we need to export symbols, so we might as well

`#define FUNC_H_EXPORT_`

```
target("inline-dynamic-lib")
set_kind("shared")
add_files("inline-test/func.cpp")
add_headerfiles("inline-test/*.h")
add_defines("FUNC_H_EXPORT_")
```

- For the user, we need to import symbols, so we might as well `#define FUNC_H_IMPORT_`.

```
target("inline-dynamic-test")
set_kind("binary")
add_deps("inline-dynamic-lib")
add_headerfiles("inline-test/*.h")
add_files("inline-test/main.cpp")
add_defines("FUNC_H_IMPORT_")
```

```
#pragma once

#if defined FUNC_H_IMPORT_
#define DLL_MACRO_NEED_IMPORT
#elif defined FUNC_H_EXPORT_
#define DLL_MACRO_NEED_EXPORT
#endif

#include "DLLMacro.h"

DLL_PORT void Func();
class DLL_PORT A {};

#undef DLL_PORT
```

Don't leak to includers.

How to make a library

- Notice: we don't directly `add_defines("DLL_MACRO_NEED_EXPORT")`, since your shared library may load other shared libraries, and thus you need `DLL_MACRO_NEED_IMPORT` there. But if you can ensure not, it's Okay.
- Now we `xmake`, then your library is made and the executable is linked against the library.
 - Congratulations!
 - Since shared libraries are loaded lazily, you can `xmake -b xx` to rebuild the library, and `xmake run yy` can still get the newest code.
 - Let me show you an example...

How to make a library

- However, you need to ensure two things:
 - Symbols before can find the newest ones; that is, namespace, class, function names and function parameter types shouldn't be changed.
 - Remember?

Function Overloading

Our first review lecture...

- In C++, functions can have the same name with different parameters.
 - This is prohibited in C.
- This is done by compilers using a technique called **name mangling**.
 - Though two functions seem to have the same name, the compiler decorates each one to a unique symbol from the parameters and their types.
 - Return type does **NOT** participate in name mangling!
 - E.g. for function `namespace Namespace {int function(int x);}`, it's mangled in msvc as `?function@NameSpace@@YAHH@Z`, while in gcc as `_ZN9NameSpace8functionEi`.

How to make a library

- Layout of the class that is exposed to users aren't changed.
 - For example, if you defines `class A{ int a; int b; };` initially, but change it to `class A{ int b; int a; };` after your update on library.
 - Then the program can run since it can still find symbol of `class A;` but the result is wrong...
 - For users, what it knows is `class A{ int a; int b; };`, which in binary code means two offsets, like `sp + offset_a` and `sp + offset_b`.
 - Say `offset_a` is 0, and `offset_b` is 4.
 - For library, when it uses `class A{ int b; int a; };`, `var.b` will lead to offset 0 `var.a` to offset 4...
 - So now, `var.b` in users will in fact get `var.a` and vice versa;
 - Say you have a member function `int GetSub(){ return b - a; }`
 - Before update, you get 1; but after update, you will get -1...

Thus though return type doesn't participate in name mangling, changing it may still corrupt the function call (user think it's `A`, but library gives `B`).

How to make a library

- Another thing that relates to layout is virtual functions.
 - You shouldn't change the sequence of virtual functions, otherwise old vtable will get the wrong one.
 - You shouldn't add virtual functions to base classes too.
 - But you may enlarge the virtual table, i.e. add virtual functions at back; users cannot call them, but the original methods are still correct.
- ABI compatibility is to maintain correctness of shared library.
- A famous example is GCC 5.1; it changes the definition of `std::string`, which makes many old libraries "unlinkable".
 - To be exact, errors happen silently in runtime.
 - That's also why some stubborn companies doesn't use new g++ versions and thus limit C++ version to C++11...

Check [KDE policy](#) and [this article](#) for more things to keep ABI compatibility when you need it.

We'll also cover a technique called **PImpl** to help in the future.

Language linkage

- Sometimes, you may need to link the library in other languages, which requires you to load from symbols.
 - But name mangling differs from compiler to compiler, and thus it's hard to cross-platform.
- Solution: use C!
 - C doesn't have function overloading, templates, namespace etc., so it can use the function name as symbol directly.
 - For example, `void Func()` will be mangled as `?Func@@YAXXZ` in msvc/C++, but just `Func` in C.
 - But C++ has many features that C doesn't have, which makes the interface restricted to a small set.

Language linkage

- For example:

```
extern-C-test > C a.h > ...
1  #pragma once
2  extern "C" int a;
3  extern "C" void Func();

a.cpp
extern-C-test > a.cpp > Func()
1  #include <iostream>
2  #include "a.h"
3
4  extern "C" int a = 1;
5  extern "C" void Func()
6  {
7      std::cout << "a = " << a << '\n';
8      std::cout << "Hello, world!\n";
9  }
```

Restrictions:

1. All namespaces are omitted.

`namespace A{ int a; }`
refers to the same symbol
as global `int a`.

2. Function overloading will cause symbol conflict.

3. `extern "C"` has no effect on all classes, templates, non-static class methods; but have effects on parameters of class methods.

Language linkage

- Or in a word, symbols that can be represented in C will be affected by `extern "C"`; otherwise it will keep the original symbols in C++.
- Note1: `extern "C"` entities will have external linkage.
- Note2: `extern "C"` doesn't mean you cannot use any C++ feature; you can wrap your C++ features in source files, and users only need to recognize symbols in headers.
- Note3: to be exact, `extern "C"` means C linker can link it; if there is `extern "Java"`, then it'll mean Java can use it.
 - But most compilers only support `extern "C"`.

Language linkage

- For example, if we hope to use C++ library in C.
 - We don't hope to load symbols by ourselves, but use C linker directly.
 - By headers!
 - In procedures of compiling C, header declaration will be compiled to C symbols.
 - When linking C program, it then can use that symbol to find our library.
 - Two ways:

- 1. Make a new header for C, without source files.

- Then your C++ interface header can have C++ features, `#include` C++ libraries, etc..
- In your C header, just write declarations of those `extern "C"` in C++ header.

- For example:

```
Same as
extern "C"
{
    int a;
    void Func();
}
```

```
a.hpp  x
extern-C-test > test2 > a.hpp > ...
1  #pragma once
2  class A{}; // C++ features.
3
4  extern "C" int a;
5  extern "C" void Func();

target("extern-C-lib2")
set_kind("static")
add_headerfiles("extern-C-test/test2/a.hpp")
add_files("extern-C-test/test2/a.cpp")
```

Language linkage

- And for headers for C, only write what you need to link, like:

```
C a.h x
extern-C-test > test2 > C a.h > ...
1  #pragma once
2  extern int a;
3  extern void Func();
4
C main.c x
extern-C-test > test2 > C main.c > main()
1  #include "a.h"
2
3  int main()
4  {
5      Func();
6      return 0;
7  }
```

```
target("extern-C-test2")
    set_kind("binary")
    add_deps("extern-C-lib2")
    add_headerfiles("extern-C-test/test2/a.h")
    add_files("extern-C-test/test2/main.c")
```

Notice that some may prefer .hpp as suffix for C++ headers; use it if you like.

In fact, source files can be named as .cxx, .cc, .cpp, and headers can be .h, .hh, .hpp.

Language linkage

- 2. share header between C and C++.
 - Use `__cplusplus` to detect whether compiled in C++.
 - This is not convenient to use C++ features.

```
target("extern-C-lib1")
  set_kind("static")
  add_headerfiles("extern-C-test/test1/a.h")
  add_files("extern-C-test/test1/a.cpp")

target("extern-C-test1")
  set_kind("binary")
  add_deps("extern-C-lib1")
  add_headerfiles("extern-C-test/test1/a.h")
  add_files("extern-C-test/test1/main.c")
```

- Ahh, you can also link C++ library against other languages, like C#, Go, etc.; but that depends on the language specification.
 - C convention is basically accepted by all languages.
 - Check their manuals if necessary.

```
C a.h
extern-C-test > test1 > C a.h > ...
1  #pragma once
2
3  #define EXTERN_C_BEGIN \
4      extern "C"         \
5      {
6  #define EXTERN_C_END }
7
8  #ifdef __cplusplus
9      EXTERN_C_BEGIN
10 #endif
11
12  int a;
13  void Func();
14
15  #ifdef __cplusplus
16      EXTERN_C_END
17  #endif
18
19  #undef EXTERN_C_BEGIN
20  #undef EXTERN_C_END

C main.c
extern-C-test > test1 > C main.c > ...
1  #include "a.h"
2
3  int main()
4  {
5      Func();
6      return 0;
7  }
```

Programming in multiple files

Modules

Modules

- There are several problems in headers:
 - Non-inline functions cannot be defined to keep ODR.
 - `#include` always requires the preprocessor to copy all contents, which makes the real file huge and drags the compilation stage.
 - Some compilers may support PCH (pre-compiled header), but it has many hard-to-match restrictions.
 - If macros aren't undef, they will be leaked. (Like minmax)
- Modules eliminate these problems!
 - All entities can be defined in "interface file" without worrying about ODR.
 - Modules are always "pre-compiled", so future import will be quite fast.
 - Macros won't affect other modules.

Modules

Module interface files have no determined suffix; msvc – .ixx; clang – .cppm; gcc – don't care. It can also be .mpp, .mxx.

We use .mpp here, and xmake identifies all of .mpp, .mxx, .cppm, .ixx.

- Every module has only one **primary interface unit**, which is like header file.
 - Begin with `export module Name;`.
 - It regulates what entities are visible to other modules by `export`.

Person.mpp

```
export module Person; // denote that it's a module interface.
import std;

export class Person
{
public:
    Person(const std::string& init_name, std::uint64_t init_id):
        name{init_name}, id{init_id} {}

    void Print() const {
        std::println("Person #{}: {}", id, name);
    }

private:
    std::string name;
    std::uint64_t id;
};
```

```
// You still need to declare before use, if it's impossible to define before use.
// Here you can just put the definition, we're just showing an example.
void InnerMethod();

// You can write definition in module interface; it obeys ODR!
export void HelloWorld()
{
    std::println("Hello, world!");
    std::println("Calling non-exported method...");
    InnerMethod();
}

void InnerMethod()
{
    std::println("No export, not visible from other modules.");
}

export template<typename T>
void TemplateMethod()
{
    std::println("This is in template method.");
}
```

The whole function definition can be put here.

Not exported, not visible from other modules.

Modules

- Entities with external linkage but without **export** has module linkage.
 - For example, **class Person** in module A and **class Person** in module B are different entities (i.e. different type).
- Entities with explicit internal linkage (i.e. **static** or anonymous namespace) cannot be **exported**.
- Entities born with internal linkage (i.e. **const** variables) that are **exported** will have external linkage.
- If a module interface imports another module and hopes to make it visible to others, it can use **export import**.
 - Export its imported module!

```
import std;
import Person;

int main()
{
    std::println("-----module-simple-----");
    HelloWorld();
    TemplateMethod<int>();
    // InnerMethod(); // error: cannot find identifier
    //                  // (because it's not exported)

    Person person{ "Haoyang", 6 };
    person.Print();
    return 0;
}
```

main.cpp

Modules

```
export module Customer;
export import Person; // users can also get access to Person.

export class Customer : public Person
{
public:
    Customer(const std::string& init_name, std::uint64_t init_id) :
        Person{ "customer " + init_name, init_id } {}

    void Print() const {
        std::println("This is a customer...");
        Person::Print();
    }
};
```

Customer.mpp

```
import std;
import Customer; // Customer has "export import Person",
                // so all exported things of Person can be used.

int main()
{
    std::println("-----module-simple-----");
    HelloWorld();
    TemplateMethod<int>();
    // InnerMethod(); // error: cannot find identifier
                    // (because it's not exported)

    Person person{"Haoyang", 6};
    person.Print();
    Customer customer{ "lrzzzzzzzzzz", 666 };
    customer.Print();
    return 0;
}
```

main.cpp

Modules

- But just like headers, it's allowed to split the declaration and definition.
 - Small interface is easier for maintainers and users to understand; many non-exported methods should be hidden.
 - It may also make compilation time faster when you only change the implementation without changing interface.
- So a module can also have **module implementation unit**.
 - It's quite like source files before C++20.
 - It begins with `module Name;`, and shouldn't have any `export`.
 - It's interface that has the duty to specify what is exported or not; implementation only provides definition.
 - A module can have multiple implementation files, as long as they all begin with `module Name;`.

Modules

Similar to header files, template should still be defined in the interface file to make link success.

```
export module Person; // denote that it's a module interface.

import std;

export class Person
{
public:
    Person(const std::string& init_name, std::uint64_t init_id);
    void Print() const;

private:
    std::string name;
    std::uint64_t id;
};

// You can write definition in module interface; it obeys ODR!
export void HelloWorld();

export template<typename T>
void TemplateMethod()
{
    std::println("This is in template method.");
}
```

Person.mpp

```
module Person; // denote that it's a module implementation.

import std;

Person::Person(const std::string& init_name, std::uint64_t init_id) :
    name{ init_name }, id{ init_id } {}

void Person::Print() const {
    std::println("Person #{}: {}", id, name);
}

// You still need to declare before use, if it's impossible to define before use.
// Here you can just put the definition, we're just showing an example.
void InnerMethod();

void HelloWorld()
{
    std::println("Hello, world!");
    std::println("Calling non-exported method...");
    InnerMethod();
}

void InnerMethod()
{
    std::println("No export, not visible from other modules.");
}
```

Person.cpp

Modules

- But many old libraries are organized as header & source, how can modules cooperate with them?
 - You can directly **import** “Header.h”; they’re called **header unit**.

```
#pragma once
#include <cstdint>

void SomeOldLibFunc(std::uint32_t);
```

Old.h

```
#include <iostream>
#include "Old.h"

void SomeOldLibFunc(std::uint32_t id)
{
    std::cout << "In header unit, id = " << id << '\n';
}
```

Old.cpp

```
module Person;
import std;
import "Old.h";
```

Person.cpp

- Personally, to make the project structure clean, I’d like to group old ones together and compile them to static library, so **.cpp** is always module implementations.

Modules

- Sometimes, a header is partially controlled by macros.

- For example, in [glm](#):

- You can use a macro to disable default ctor.

- But as we said, macros will only affect the current file, so imported header unit won't see that macro.

```
namespace glm
{
    // -- Constructors --

    #if GLM_CONFIG_DEFAULTED_DEFAULT_CTOR == GLM_DISABLE
        template<typename T, qualifier Q>
        GLM_DEFAULTED_DEFAULT_CTOR_QUALIFIER GLM_CONSTEXPR mat<3, 3, T, Q>::mat()
    #if GLM_CONFIG_CTOR_INIT == GLM_CTOR_INITIALIZER_LIST
        : value{col_type(1, 0, 0), col_type(0, 1, 0), col_type(0, 0, 1)}
    #endif
    #endif
}
```

- C++ uses **global module fragment** to solve this historical problem.

- It begins before `export module Name;` or `module Name;`.

```
module; // global module fragment

#define NEED_PARAM
#include "Old.h"

module Person; // denote that it's a
```

Modules

- We can also see it in modules of glm:

```
module;  
  
// #define GLM_GTC_INLINE_NAMESPACE to inline glm::gtc into glm  
// #define GLM_EXT_INLINE_NAMESPACE to inline glm::ext into glm  
// #define GLM_GTX_INLINE_NAMESPACE to inline glm::gtx into glm  
  
#include <glm/glm.hpp>  
#include <glm/ext.hpp>  
  
export module glm;
```

- Things above are basically supported by all compilers (2024.2).
 - Though gcc doesn't support importing header unit of standard library directly and gcc & (clang with MS-STL) haven't supported `import std;` yet.
 - Clang with libc++ has already support it.
- But things afterwards aren't supported well, so it should only be used in toy projects.

Modules

- If you still find a module too big, you can partition either interface or implementation.
 - **Interface partition unit:** begin with `export module Name:Subname;`
 - **Implementation partition unit:** begin with `module Name:Subname2;`
 - Notice that a module name can have dot (like A.B), but it doesn't have semantic meaning, but just logical meaning.
 - We think it's a "sub part" of A, but principally they're different modules.
- 1. Partition is an inner conception of the module; other modules cannot know whether it has partitions or not.
 - Inside the module, it can use `import :SubName` to import the partition.
 - But in other modules, they cannot use `import Name:SubName`.

Modules

- 2. Unlike module implementation, implementation partition is **not** implementation of interface partition.
 - That is, if there exists `module A:B`; there shouldn't exist `export module A:B`.
 - If you want to split the definitions in the interface partition, do it either in `module A` or another partition `module A:C`.
- 3. Partitions cannot have partitions (depth == 1).

```
module Person:Order;  
import std;  
  
struct Order  
{  
    int count;  
    std::string name;  
    double price;  
  
    Order(int c, const std::string& n, double p)  
        : count{c}, name{n}, price{p} { }  
    void Print() const {  
        std::println("count = {}, name = {}, price = {}", count, name, price);  
    }  
};
```

Person-Order.mpp
An implementation partition

Modules

Notice that implementation partition cannot be exported, only visible inside the module.

```
export module Person:Utils;  
  
import :Order;  
  
export void PrintOrder(){  
    std::print("\t{ }, with order\n\t");  
    Order{1, "apple", 7.3}.Print();  
}
```

Person-Utils.mpp
An interface partition

```
import std;  
import Person;  
  
int main()  
{  
    std::println("-----module-partition-----");  
    Person person{"Haoyang", 6};  
    person.Print();  
    PrintOrder();  
    return 0;  
}
```

```
export module Person; // denote that it's a module interface.  
  
export import :Utils;  
import std;  
  
export class Person  
{  
public:  
    Person(const std::string& init_name, std::uint64_t init_id);  
    void Print() const;  
  
private:  
    std::string name;  
    std::uint64_t id;  
};
```

Primary interface can
choose to export the
interface partition.

Person.mpp
The primary interface

main.cpp

Modules

- Private module fragment:
 - Begins with `module:private;`.
 - When a part (especially for class definition) is put into private module fragment, it's unreachable from other modules.
 - When a module has private module fragment, it can only have a single unit; that is, the primary interface unit.
- Personally, I don't think it's very useful...

Modules

```
import std;
import Person;

int main()
{
    std::println("-----module-private-----");
    auto p = CreatePerson();
    // p->Print(); // error: use undefined type "Person"
    PrintPerson(p);
    DestroyPerson(p);
    return 0;
}
```

main.cpp

```
export module Person; // denote that it's a module interface.

import std;

export class Person;
export Person* CreatePerson();
export void PrintPerson(Person*);
export void DestroyPerson(Person*);

module:private;

class Person
{
public:
    Person(const std::string& init_name, std::uint64_t init_id):
        name{init_name}, id{init_id} {}

    void Print() const {
        std::println("Person #{}: {}", id, name);
    }

private:
    std::string name;
    std::uint64_t id;
};

Person* CreatePerson() { return new Person{"Sheng", 6}; }
void PrintPerson(Person* p) { p->Print(); }
void DestroyPerson(Person* p) { delete p; }
```

Person.mpp

Summary

- Preprocessing and macros
- Translation unit and ODR.
 - Declaration & Definition
 - Header files & source files
 - Special case - template
 - Header guard
 - Inline functions and variables
- Namespace
- Linkage
- Xmake
- Static & shared library
- Inline pitfall!
- Modules
 - Module interface and implementation
 - Header unit, global module fragment
 - Partition, private module fragment

Next lecture...

- We'll cover error handling in C++.
- Err...you'll know what exception is (finally, after lots of "cover in the future").
- You'll also learn some other helpers, like assertion, stacktrace, expected, etc....
- And finally Unit Test! Testing is one of the most significant skills for a professional programmer.