Модули в C++20

Введение

- **Зачем нужны заголовки**

  Большинство C++ проектов включают несколько единиц трансляции, которые используют общие объявления и определения. Для этого применяются заголовочные файлы.

- **Минусы использования заголовков**

  Заголовки могут вызывать проблемы с избыточной компиляцией и зависимостями между модулями, а также нет строгих правил относительно использования директивы `#include`. Она просто вставляет содержимое файла в место, где она расположена.

- **Зачем модули**

  Модули это языковая функциональность, позволяющая обмениваться объявлениями и определениями между единицами трансляции. Они являются альтернативой некоторым вариантам использования заголовочных файлов.

Пример использования заголовочных файлов

- *Файл function.hpp:*

```
const char* function();
```

- *Файл function.cpp:*

```
const char* function() {
    return "Hello";
}
```

- *Файл main.cpp:*

```
#include <print>

#include "function.hpp"

int main() {
    std::println("{}", function());
    return 0;
}
```

- *команда компиляции:*

```
g++ -std=c++23 main.cpp function.cpp
```

#include нельзя контролиповать

Можно делать странные вещи и это не вызовет ошибку:

- *Файл return.hpp:*

```
return "Hello";
```

- *Файл function.hpp:*

```
inline const char *function() {
#include return.hpp
}
```

- *Файл main.cpp:*

```
#include <print>

#include "function.hpp"

int main() {
    std::println("{}", function());
    return 0;
}
```

- *команда компиляции:*

```
g++ -std=c++23 main.cpp
```

Нельзя включать файлы два раза

- *Файл foo.hpp:*

```
struct Foo {
    int bar;
    int bazz;
};
```

- *Файл main.cpp:*

```
#include foo.hpp
#include foo.hpp // ошибка переопределения но не повторного включения

int main() {
}
```

Не пользуясь защитой заголовков этот пример вызовет ошибку переопределения(но не повторного включения файла заголовка так как это разрешено)

Файл может исползовать не определенные в нем вещи

- *Файл person.hpp:*

```
class Person {
    std::string name; // std::string нет в фйле ошибка конечно же не возникает
};
```

- *Файл main.cpp:*

```
#include <string>

#include "person.hpp"

int main() {
}
```

Пример использования модулей

- *Файл function.cpp:*

```
export module my_module;

export const char *function() {
    return "Hello";
}
```

- *Файл main.cpp:*

```
import <print>;

import my_module;

int main() {
    std::println("{}", function());
    return 0;
}
```

- *команда компиляции:*

```
g++ -std=c++23 -fmodules-ts -xc++-system-header print
g++ -std=c++23 -fmodules-ts -c function.cpp
g++ -std=c++23 -fmodules-ts -c main.cpp
g++ function.o main.o
```

Сравнение компиляции модулей против заголовчных файлов

- *Файл hello.cpp:*

```
#include <print>
int32_t main(int32_t, const char**) {
    std::println("Hello, world!");
    return EXIT_SUCCESS;
}
```

- *Файл hello_with_modules.cpp:*

```
import <print>;
int32_t main(int32_t, const char**) {
    std::println("Hello, world!");
    return EXIT_SUCCESS;
}
```

Сравниапния длину полученного файла в байтах как результат работы препроцессра

```
clang++ -std=c++23 -E hello.cc | wc -c
1796066
```

против

```
clang++ -std=c++23 -stdlib=libc++ -fmodules -fbuiltin-module-map \
-E hello_modul.cc | wc -c
258
```

Сравнение времени компиляции

Если сравнивать время компиляции то оно тоже будет короче

```
time clang++ -std=c++23 hello.cc
real    0m0,934s
user    0m0,883s
sys     0m0,043s
```

против

```
time clang++ -std=c++23 -stdlib=libc++ -fmodules -fbuiltin-module-map hello_modul.cc
real    0m0,523s
user    0m0,465s
sys     0m0,057s
```

но это с учетом что мы используем прекомпелированные заголовки стандартной библиотеки если будем делать это с компиляцией самих заголовков как модулей:

```
time g++ -std=c++23 hello.cc
real    0m0,991s
user    0m0,951s
sys     0m0,033s
```

против

```
time g++ -std=c++23 -fmodules-ts -xc++-system-header print
real    0m0,811s
user    0m0,772s
sys     0m0,037s
```

и

```
time g++ -std=c++23 -fmodules-ts hello_modul.cc
real    0m0,667s
user    0m0,631s
sys     0m0,029s
```

в данном случае выходит все таки дольше но надобности перекомпелировать заголовок несколько раз нету так что при многократной компиляции время компиляции сокращается в разы

Более подробно о export

- *Файл:*

```
export module my_module;

// фунции
export const char *function();

// типы
export class SomeClass;

// шаблоны
export template<typename T>
T max(T n1, T n2);

export template<typename T>
class anotherClass;
```

также можно обьеденять export в блоки или вовсе не экспортировать те или инные вещи

- *Файл:*

```
export module my_module;

export const char *foo() { return "Foo"; };

// не экспортируется
const char *bar() { return "Bar"; };

// экспортируем много вчего в блоке
export {
    int one() { return 1; }
    int zero() { return 0; }
}

export namespace math {
    int add(int a, int b) {
        return a + b;
    }

    int substract(int a, int b) {
        return a - b;
    }
```

Разделение модулей на разделы

пример1:

- *Файл:*

```
export module A:B; // Объявляет единицу интерфейса модуля для модуля 'A', раздела
```

пример2:

- *Файл A-B.cpp:*

```
export module A:B;
...
```

- *Файл A-C.cpp:*

```
module A:C;
...
```

- *Файл A.cpp:*

```
export module A;

import :C;
export import :B;

...
```

Простой пример использовния модулей

- *Файл person.cpp:*

```
export module person;

import <cstdint>;
import <string>;
import <string_view>;
import <iostream>;

export class Person {
    std::string name;
    uint32_t age;
public:
    Person(std::string_view ="", uint32_t =0);
    const std::string& getName() const;
    uint32_t getAge() const;
};

export std::ostream& operator<<(std::ostream&, const Person&);
```

- *Файл person_impl.cpp:*

```
module person;

Person::Person(std::string_view newName, uint32_t newAge)
    : name{newName}, age{newAge}
{ }

const std::string& Person::getName() const { return name; }

uint32_t Person::getAge() const { return age; }

std::ostream& operator<<(std::ostream& out, const Person& person) {
    return out << person.getName() << " is "
        << person.getAge() << " years old";
}
```

- *Файл main.cpp*

```
import <cstdlib>;
import <cstdint>;
import <iostream>;

import person;

int32_t main(int32_t, const char**) {
    Person mark{ "Mark" };
    Person bob{ "Bob", 20 };
    std::cout << mark << '\n'
        << bob << '\n';
    return EXIT_SUCCESS;
}
```

И тоже самое с использованием заголовочных файлов

- *Файл person.hpp:*

```
#pragma once

#include <string>
#include <string_view>
#include <cstdint>
#include <iostream>

class Person {
    std::string name;
    uint32_t age;
public:
    Person(std::string_view ="", uint32_t =0);
    const std::string& getName() const;
    uint32_t getAge() const;
};

std::ostream& operator<<(std::ostream&, const Person&);
```

- *Файл person.cpp:*

```cpp
#include <string>
#include <string_view>
#include <cstdint>
#include <iostream>



#include "person.hpp"

Person::Person(std::string_view newName, uint32_t newAge)
    : name{newName}, age{newAge}
{ }

const std::string& Person::getName() const { return name; }

uint32_t Person::getAge() const { return age; }

std::ostream& operator<<(std::ostream& out, const Person& person) {
    return out << person.getName() << " is "
        << person.getAge() << " years old";
}
```

- *Файл main.cpp:*

```cpp
#include <iostream>

#include "person.hpp"

int32_t main(int32_t, const char**) {
    Person mark{ "Mark" };
    Person bob{ "Bob", 20 };
    std::cout << mark << '\n'
        << bob << '\n';
    return EXIT_SUCCESS;
}
```

Различия во времени компиляции

Впемя компеляции решения с модулями с прекомпеляцией заголловочных файлов

```
time (for header in iostream string string_view cstdint cstdlib; \
do g++ -std=c++23 -fmodules-ts -xc++-system-header $header; done && \
g++ -std=c++23 -fmodules-ts -c person.cpp person_impl.cpp main.cpp && \
g++ main.o person.o person_impl.o)
real    0m1,964s
user    0m1,862s
sys     0m0,098s
```

Впемя компеляции решения с модулями с прекомпелированными заголловочными файлами

```
time (g++ -std=c++23 -fmodules-ts -c person.cpp person_impl.cpp main.cpp && \
g++ main.o person.o person_impl.o)
real    0m0,531s
user    0m0,460s
sys     0m0,069s
```

а теперь время компиляции решения с зоголовочными файлами

```
time (g++ -std=c++23 -c person.cpp main.cpp && g++ main.o person.o)
real    0m1,206s
user    0m1,149s
sys     0m0,055s
```

Если есть прекомпилированные файлы или компеляция будет проводится много раз то эффективнее использовать модули так как с использованием заголовочных файлов препроцессором включается огромное количество кода

Можно использовать заголовочные файлы вместе с моддулями

тоесть import и #include одновременно: иногда это необходимо так как иногда импорт нзаголовчных файлов невозможен (особенно, когда заголовочный файл использует макросы предварительной обработки в качестве конфигурации). (Но обычно все заголовочные файлы можно прекомпилировать или использовать уже прекомпелированные)

как пример модно изменить person.cpp и main.cpp из решения с модулями

- *Файл person.cpp:*

```
module;
#include <cstdint>;
#include <string>;
#include <string_view>;
#include <iostream>;
export module person;
```

```
export class Person {
    std::string name;
    uint32_t age;
public:
    Person(std::string_view ="", uint32_t =0);
    const std::string& getName() const;
    uint32_t getAge() const;
};

export std::ostream& operator<<(std::ostream&, const Person&);
```

- *Файл main.cpp:*

```
#include <cstdint>;
#include <string>;
#include <string_view>;
#include <iostream>;

import person;

int32_t main(int32_t, const char**) {
    Person mark{ "Mark" };
    Person bob{ "Bob", 20 };
    std::cout << mark << '\n'
        << bob << '\n';
    return EXIT_SUCCESS;
}
```

при таком использовании можно не прекомпелировать файлы заголовков как модули но это решение будет тоже долго компилироватся

Итоги

- Модули решают проблемы, связанные с избыточной компиляцией и зависимостями.
- Модули позволяют значительно сократить время компиляции при многократных сборках. (особенно при использовании состем сборки но порядок их компеляции важен что усложняет их использование)
- При первом использовании модулей возможно увеличение времени из-за необходимости предварительной компиляции заголовков.
- Модули поддерживают явный экспорт (export), что позволяет точно контролировать видимость символов.
- Возможность комбинировать модули с заголовочными файлами сохраняет совместимость с существующим кодом.
- Модули легко делятся на части, что упрощает управление сложными проектами.
- Модули эффективны в крупных проектах с большим количеством единиц трансляции.
- Вовсе не обязательно использовать модули так как некотрые IDE до сих пор не понимают этот синтаксис и подсвечивают как ошибке

Далнейшее изучение по теме (и используемые источники информации)

https://youtu.be/WRCwciJ5MTE?si=7abukqak4n90knBG

https://youtu.be/_x9K9_q2ZXE?si=1b1MeF0O2UUJvP2d

https://youtu.be/-p9lvvV8F-w?si=J7LSGOP2NbTrNQnA

https://en.cppreference.com/w/cpp/language/modules

https://habr.com/ru/companies/otus/articles/575954/

https://cpp-kt.github.io/cpp-notes/32_modules.html

https://learn.microsoft.com/ru-ru/cpp/cpp/modules-cpp?view=msvc-170

https://releases.llvm.org/18.1.4/tools/clang/docs/StandardCPlusPlusModules.html

Спасибо за внимание