Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Курсова робота

освітній ступінь – бакалавр

на тему: «**Методи та засоби зневадження шаблонного коду**»

Виконав: студент 3-го року навчання,

Спеціальності

121 «Інженерія Програмного Забезпечення»

Студента Охріменка Михайло

Керівник Бублик В. В.

кандидат фіз.-мат. наук, доцент

Київ – 2022

КАЛЕНДАРНИЙ ПЛАН ВИКОНАННЯ КУРСОВОЇ РОБОТИ

Тема: Методи та засоби зневадження шаблонного коду.

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1	Отримання завдання на курсову роботу	15.10.2021	
2	Ознайомлення з існуючою інформацією по темі	20.10.2021	
3	Проєктування прикладів	25.11.2021	
4	Написання теоретичної частини	30.01.2022	
5	Корегування цілої роботи	05.03.2022	

Студент Охріменко М. С.

Керівник Бублик В. В.

“_____” _____

Зміст

Анотація 4

Вступ 5

Розділ 1. Історія розвитку шаблонів 6

Розділ 2. Зневадження коду 9

Розділ 2.1. Static Assertions 11

Розділ 2.2. SFINAE 12

Розділ 2.3. Concepts in C++20 14

Розділ 3. Система модулів 19

Розділ 3.1. Що таке #include 19

Розділ 3.2. Чому його потрібно замінити 19

Розділ 3.3. Що таке модулі 19

Висновки 21

Додаток 22

Список джерел 24

Анотація

У цій роботі розглядається методи та засоби зневадження шаблонного коду. Вони пояснюються на інструментах, які доступні з версії C++11 до C++ 20. Розглядаються причини та наслідки, які привели до нововведень. Показані такі інструменти, як static_assert, SFINAE і enable_if, концепти та система модулів. Розповідається про історію розвитку узагальненого програмування та чого намагаються досягти розробники мови C++.

Вступ

Мова програмування C++ є мультипарадигмовою. Вона підтримує процедурну, об'єктно-орієнтовану, узагальнену та інші парадигми. Якщо до процедурної і об'єктно орієнтованої запитань не виникає, то з узагальненою точилися і точаться суперечки, чи варто її взагалі називати парадигмою. Це ж бо звичайний синтаксичний цукор. Насправді ж узагальнене програмування та його реалізація у C++ — шаблони — значно зменшують кількість коду, що дублюється, а отже і місць, де можна зробити помилку, дозволяють позбутися непотрібних ієрархій, робити обчислення та перевірки типів на етапі компіляції, а отже, лапати помилки раніше**.** Тому і підхід до нього потрібен окремий.

З іншого боку, шаблони мають високий поріг входження, існує навіть термін «магія шаблонів», що означає далеко не їх прозорість та зрозумілість. За неправильного використання призводять до «роздування об'єктного коду», а зрозуміти їх з першого разу не кожному дано. З появою нових версій мови програмування накопичувалися нові конструкції, функції та методи використання шаблонів, але не було систематизації. Зараз, шукаючи на stackoverflow якесь запитання з шаблонів можна наткнутися на величезні пости, де користувачі сперечаються, чи варто використовувати один підхід, а ні цей вже застарілий, але в цьому випадку підходить тільки він. В найкращому випадку можна знайти пост або статтю, де розписано яким методом чи підходом варто користуватися для різних версій C++.

З виходом C++20 розробникам мови вдалося систематизувати роботу з шаблонами та, здебільшого, звести її до конструкцій звичайного безшаблонного програмування.

І ще не менш важливо — використання обрахунків на етапі компіляції помітно пришвидшує виконання, але недоліком цього є відсутність будь-якої можливості інтерактивно зневаджувати код.

Метою мого дослідження є інструменти та прийоми, які можна використати, щоб не допустити помилок в шаблонному коді або ж знайти їх, якщо перше не вдалося. Також ми розглянемо, як можна пришвидшити компіляцію коду і не чекати декілька днів для того, щоб запустити проєкт. Ці рекомендації, інструменти та прийоми будемо дивитися у різних редакціях мови C++, щоб мати змогу працювати як і з новим кодом, так і з legacy.

Розділ 1. Історія розвитку шаблонів

Узагальнене програмування було можливим ще з перших версій C++. Реалізовувалися воно за допомогою макросів і було своєрідною «милицею», яка дісталася у спадок від C.

Ось так, наприклад, можна визначити функцію піднесення двох чисел до квадрату.

```
#define SQUARE(a) (a * a)
```

І використаємо її . Виведеться число 9.

```
int main()
{
	cout << SQUARE(3) << '\n';
	//cout << (3 * 3) << '\n'
}
```

Тепер спробуємо запустити такий код.

```
int a{3};
cout << SQUARE(++a) << '\n';
//cout << (++a * ++a) << '\n';
```

*В результаті мало б вийти **16**, а виходить **25**.*

Макрос — це звичайна текстова підстановка. Компілятор не вважає макрос окремою функцію, а тому наші інтуїтивні очікування від макросу-функції будуть відрізнятися від результату. Помилки, що виникають під час роботи з макросами важко знайти. І тому така можливість узагальненого програмування була невдалою.

Наступним етапом були «необмежені шаблони», тобто неможливо було задати обмеження на аргументи.

Ці шаблони:

- Максимально загальні і гнучкі
- Накладні витрати мінімальні
- Мають занадто загальні інтерфейси
- Організація коду можлива тільки в заголовних файлах.

Останні 2 пункти звісно ж погані. Але, як зазначається в [1], перші були настільки добрими, що узагальнений код, побудований на таких шаблонах, став основою для високопродуктивного коду. З цього ж етапу стало можливим обрахування на етапі компіляції, а узагальнене програмування стало основою для STL(standard template library).

Ось так тепер виглядатиме функція піднесення до квадрату.

```
template<typename T>
```

```
T square(T a) {return a * a;}
```

І її використання. Результатом буде те ж число 9.

```
int a{3};
cout << square(a) << '\n';
```

Тепер стало можливим робити ті ж операції, що й зі звичними нешаблонними функціями.

```
int a{3};
cout << square(++a) << '\n';
```

Результатом буде число 16.

Але через загальність інтерфейсів, стала неможлива перевірка типів. Що станеться, якщо ми спробуємо піднести до квадрату тип, для якого не визначено операції множення.

```
struct A{};
```

Піднесення типу, для якого невизначена операція множення.

```
int a{3};
cout << square(A()) << '\n';
```

Ось таке повідомлення про помилку ми отримаємо:
binary '*': 'T' does not define this operator or a conversion to a type acceptable to the predefined operator

Звісно, тут одразу зрозуміло, що не так. Але цей код може бути бібліотечним і тоді доведеться шукати поміж сотень стрічок, де і що могло піти не так. Натомість у нешаблонному програмуванні ми обмежуємо типи аргументів.

Крайній етап — C++20. Це поки одна з найбільш вдалих стосовно узагальненого програмування версій мови. Розробники структурували та додали декілька функцій, які наближують шаблонне програмування до звичайного. Додали концепти(concepts), які задають обмеження для типового параметру. Додали модулі, які мають замінити вже давно застарілі директиви препроцесора(#include). Вони дозволяють уникнути циклічних залежностей та прискорити компіляцію коду в декілька разів.

Ось так тепер виглядатиме функція піднесення до степеню.

```
template<typename T>
concept multiplicable = requires(T a){
	a * a;
};

template<multiplicable T>
T square(T a) {return a * a;}
```

*Введемо концепт, який визначатиме, чи має тип визначений оператор **.*

А ось таке повідомлення про помилку ми отримаємо, якщо спробуємо застосувати цю функцію до типу, який не має оператора *:
'square': the associated constraints are not satisfied

Тепер просто дивлячись на інтерфейс ми можемо зрозуміти, який тип може приймати функція.

Зараз вже більшість ide та компіляторів підтримують новий стандарт, хоча і є проблеми з бібліотеками та доповненнями від сторонніх розробників.

Розділ 2. Зневадження коду.

Розглядаючи найчастіші помилки в шаблонах ми не братимемо до уваги макроси. Ще з початку існування мови вони одразу створювали багато проблем і зараз їхнє використання вважається поганим тоном.

Шаблони створюють 2 типи викликів, які потрібно подолати. З боку того, хто їх проєктує — як можна впевнитися, що шаблон працюватиме для будь-яких типових параметрів, що задовольняють заявлені умови? З боку користувача — як знайти вимоги до параметрів, які було порушено, якщо шаблон працює не так, як заявлено.

Розглянемо такий фрагмент коду з [2].

```
template<typename T>
void clear (T& p)
{
	*p = 0;
}

template<typename T>
void core(T& p)
{
	clear(p);
}

template<typename T>
void shell (T const& env)
{
	typename T::Index i;
	middle<T>(i);
}

template<typename T>
void middle(typename T::Index p)
{
	core(p);
}
```

Тут показано типові компоненти-шари в програмній інженерії: високорівнева функція *shell()* користується компонентою *middle()*, а та в свою чергу використовує базові можливості *core()*. Під час конкретизації *shell()*, інші шари також каскадно конкретизуються. Помилка стається в останньому шарі: core() конкретизується типом *int* і намагається розіменувати його, що призводить до помилки. Її можна виявити тільки на етапі конкретизації.

Тут показано

```
class Client
{
public:
	using Index = int;
};

int main()
{
```

```
	Client mainClient;
	shell(mainClient);
}
```

У відповідь отримуємо таку помилку.

```
1>C:\git\Course_work\Drafts\Drafts\InstantiantionChain.h(5,2): error C2100: illegal
indirection
1>C:\git\Course_work\Drafts\Drafts\InstantiantionChain.h(11): message : see reference
to function template instantiation 'void clear<T>(T &)' being compiled
1>        with
1>        [
1>            T=Client::Index
1>        ]
1>C:\git\Course_work\Drafts\Drafts\InstantiantionChain.h(24): message : see reference
to function template instantiation 'void core<Client::Index>(T &)' being compiled
1>        with
1>        [
1>            T=Client::Index
1>        ]
1>C:\git\Course_work\Drafts\Drafts\InstantiantionChain.h(18): message : see reference
to function template instantiation 'void middle<T>(Client::Index)' being compiled
1>        with
1>        [
1>            T=Client
1>        ]
1>C:\git\Course_work\Drafts\Drafts\main.cpp(25): message : see reference to function
template instantiation 'void shell<Client>(const T &)' being compiled
1>        with
1>        [
1>            T=Client
1>        ]
```

Важко одразу зрозуміти про що тут йдеться. Звісно хороша діагностика шаблонів має покривати всі кроки, які призводять до проблеми, але такий об’єм інформації здається занадто громіздким.

Рішенням цього може бути *поверхнева конкретизація(shallow instantiation)*. Вона досягається додаванням коду, який буде тільки провокувати помилку, якщо шаблон конкретизується аргументами, які не задовольняють вимог наступних рівнів.

У попередньому прикладі до *shell()* можна додати код, який намагається розіменувати значення типу *T::index*.

```
template<typename T>
void ignore(T const&){}

template<typename T>
void shell(T const& env)
{
	class ShallowChecks
	{
		void deref(typename T::Index ptr)
		{
			ignore(*ptr);
		}
	};
	typename T::Index i;
	middle<T>(i);
```

```
}
```

І тепер повідомлення про помилку виглядає так:

```
1>C:\git\Course_work\Drafts\Drafts\InstantiationChainShallow.h(25,12): error C2100:
illegal indirection
1>C:\git\Course_work\Drafts\Drafts\main.cpp(22): message : see reference to function
template instantiation 'void
InstantiationChainShallow::shell<InstantiationChainShallow::Client>(const T &)' being
compiled
1>        with
1>        [
1>            T=InstantiationChainShallow::Client
1>        ]
```

Тепер помилка спрацьовує в локальному класі ShallowChecks. Цей клас ніколи не використовується, тому він не впливає на продуктивність функції *shell()*. На жаль, багато компіляторів будуть попереджувати про те, що ShallowChecks ніколи не використовувався. Можна використати трюк з додаванням шаблону ignore(), щоб уникнути таких попереджень. Проте такі трюки ускладнюють код і роблять його громіздким.

Розділ 2.1. Static Assertions

Макрос *assert()* часто використовується в C та C++ щоб перевірити певну умову. Якщо умова не виконується, програма «падає» і можна одразу побачити, де виникла помилка.

C++11 додає ключове слово static_assert, яке робить те саме, тільки під час компіляції. Якщо умова, яка має бути `constexpr`, не виконується, компілятор видасть повідомлення про помилку. Повідомлення буде містити стрічку (аргумент static_assert), яка повідомлятиме програмісту, що пішло не так. Наприклад, цей static_assert перевіряє, чи платформа, на якій компілюється програма, 32 бітна.

```
static_assert(sizeof(void*) * CHAR_BIT == 32, "Target platform is not 32-bit");
```

Static_assert використовується для того, щоб виводити корисні повідомлення про помилку, якщо аргументи не задовольняють обмеження шаблону.

Зробимо type_trait, який визначає, чи можна розіменувати тип, використовуючи техніку SFINAE(substitution failure is not an error).

Розглянемо приклад з [2].

```
template<typename T>
class HasDerefernce
{
private:
	template<typename U> struct Identity;
	template<typename U> static std::true_type
		test(Identity<decltype(*std::declval<U>())>*);
	template<typename U> static std::false_type
		test(...);
```

```
public:
	static constexpr bool value = decltype(test<T>(nullptr))::value;
};
```

І використаємо в функції shell().

```
template<typename T>
void shell(T const& env)
{
	static_assert(HasDerefernce<T>::value, "T is not dereferenceable");

	typename T::Index i;
	middle<T>(i);
}
```

Зараз компілятор видає набагато зрозуміліше повідомлення про те, що тип T не можна розіменувати. Static assertions значно покращують досвід роботи з різними бібліотеками, бо роблять повідомлення про помилки коротшими і прямолінійними.

Розділ 2.2. SFINAE

Коли виникає потреба довизначити(overload) шаблонні функції, у компілятора можуть виникати проблеми.

Розглянемо приклад з [3].

```
void func(unsigned i)
{
	std::cout << "unsigned " << i << '\n';
}

template<typename T>
void func(const T& t)
{
	std::cout << "template " << t << '\n';
}
```

Коли ми викличемо `func(3)`, виведеться «template 3». Причина цьому те, що за умовчуванням цілий число є знаковим типом. Коли компілятор визначає, яку функцію використати, він бачить, що перша функція вимагає приведення типу, а друга підходить ідеально. Тому він її і вибирає.

Коли компілятор вибирає серед шаблонних функцій, він мусить їх конкретизувати. Це не завжди закінчується успіхом.

Розглянемо ще один приклад з [3].

```
int negate(int i) { return -i; }

template<typename T>
typename T::value_type negate(const T& t) { return -T(t); }
```

Коли викликається `negate(32)`, компілятор вибере першу реалізацію. Проте, коли він вибиратиме, він мусить врахувати шаблонну версію функції. Він конкретизує параметр `T` типом `int` і отримає такий вираз:

```
int::value_type negate(const T& t) { return -T(t); }
```

Такий код некоректний, тому що `int` не має члена класу `value_type`. Але компілятор не видає помилку і компіляція закінчується успіхом. У цьому йому допомагає принцип SFINAE(Substitution Failure Is Not An Error).

Стандарт визначає його так: при конкретизації довизначених шаблонних функцій помилкові конкретизації не призводять до помилки компіляції, а просто відкидаються зі списку кандидатів на найбільш вдалу функцію. З цього ми можемо виділити те, що SFINAE працює тільки для шаблонних функцій, які були довизначені; він відкидає шаблони, які під час конкретизації призводять до синтаксичної помилки, відкинути можна тільки шаблон, цей принцип перевіряє тільки заголовки функцій, а помилки в тілі вважатимуться помилками часу компіляції.

SFINAE став настільки корисним, що на його базі робили різні прийоми, один з найбільш помітних став `std::enable_if`.

Він визначається так:

```
template <bool, typename T = void>
struct enable_if {};

template<typename T>
struct enable_if<true, T> { using type = T };
```

Розглянемо приклад з [3].

```
template<typename T,
	typename std::enable_if<std::is_integral<T>::value, T> ::type* = nullptr>
void do_stuff(T& t)
{
	//An implementation for integral types (int, char, unsigned, etc.)
	std::cout << "do_stuff integral\n";
}

template<typename T,
	typename std::enable_if<std::is_class<T>::value, T>::type* = nullptr>
void do_stuff(T& t)
{
	//An implementation for class types
}
```

Коли ми робимо виклик do_stuff(<int var>), компілятор вибирає першу функцію, тому що виконується умова `std::is_integral<int>`. Друга функція відкидається, бо результат `std::is_class<int>` — `false`.

До C++11 `enable_if` був частиною бібліотеки boost. Часто визначення `enable_if` було занадто багатослівним, тому з C++14 стала доступна скорочена версія `enable_if_t`.

```
template<bool B, typename T = void>
using enable_if_t = typename enable_if<B, T>::type;
```

Розглянемо приклад `do_stuff_t` з `enable_if_t`.

```
template<typename T,
typename std::enable_if_t<std::is_integral<T>::value>* = nullptr>
void do_stuff_t(T& t)
{
	//An implementation for integral types (int, char, unsigned, etc.)
	std::cout << "do_stuff integral\n";
}

template<typename T,
	typename std::enable_if<std::is_class<T>::value>* = nullptr>
void do_stuff_t(T& t)
{
	//An implementation for class types
}
```

Так ми змогли трохи спростити визначення шаблонної функції.

Розділ 2.3. Concepts in C++20

Версія мови C++20 принесла *концепти*. Ця функція стала логічною еволюцією звичайних, дуже загальних шаблонів. Розробники мови вже давно хотіли знайти спосіб обмеження параметрів шаблону. Перша версія системи концептів була готовою ще на етапі C++11, але через різні причини її не стали вносити в редакцію. Те ж сталося з C++14 та C++17. Проте концепти реалізовували сторонні розробники, як-от в бібліотеці boost.

Концепти — це іменований набір обмежень для типових параметрів. Їх можна використовувати для того, щоб керувати перевизначенням функції та частковою спеціалізацією шаблону. Для цього в C++20 з'явилися нові ключові слова ***requires*** та ***concept*** і початковий набір концептів в STL. Також концепти можна комбінувати.

Наприклад, в нас є потреба побудувати функцію, яка перевірятиме, чи 2 числа достатньо близькі між собою. Цілі числа будемо перевіряти на рівність, а числа з плаваючою крапкою — що модуль їх різниці менший якось α. Цю задачу можна вирішити трюком SFINAE і написати 2 функції з використанням *std::enable_if*. Коли відбуватиметься конкретизація шаблону, компілятор відкидатиме реалізації з помилками.

Розглянемо приклад з [4].

```
#include <type_traits>

using namespace std;

//Function that should check whether two numbers are close to each other
namespace CloseDistantSFINAE
{
	constexpr double ALPHA{1e-5};

	template<typename T>
	T Abs(T x)
	{
		return x >= 0 ? x : -x;
	}
```

```
	//Check for float
	template<typename T>
	std::enable_if_t<std::is_floating_point_v<T>, bool>
	ElementsAreClose(T a, T b)
	{
		return Abs(a - b) < static_cast<T>(ALPHA);
	}

	//Check for other objects
	template<typename T>
	std::enable_if_t<!std::is_floating_point_v<T>, bool>
	ElementsAreClose(T a, T b)
	{
		return a == b;
	}
};
```

В цьому випадку достатньо *std::enable_if*. Але ми втрачаємо *тип повернення*. Він мав бути `bool`, а насправді — `std::enable_if_t`.

Тепер потрібно написати функцію *Print*, яка виводитиме на екран все, що можливо передати. Для контейнерів мають виводитися всі елементи, для звичайних типів — виведе їх. Для такої функції трюк з SFINAE реалізувати буде важко, а код буде величезний.

```
template<typename T>
void Print(std::ostream&, const std::vector<T>& v)
{
	for (const auto& elem : v)
	{
		out << elem << std::endl;
	}
}

	//Here shoud be defined functions for map, set, list, etc.

template<typename T>
void Print(std::ostream& out, const T& v)
{
	out << v;
}
```

Ці проблеми можна вирішити новими конструкціями. Якщо концепти — це ім’я для обмежень, то самі обмеження — це шаблонно булеві вирази. Ми вже обмежували параметр умовою «бути числом з плаваючою крапкою».

Розглянемо покращення першого прикладу з [4].

```
	//Check for float
	template<typename T>
	requires std::is_floating_point_v<T>
	bool ElementsAreClose(T a, T b)
	{
		return Abs(a - b) < static_cast<T>(ALPHA);
	}

	//Check for other objects
	template<typename T>
	bool ElementsAreClose(T a, T b)
	{
		return a == b;
```

```
}
```

Ми явно бачимо тип повернення(`bool`), немає зайвих конструкцій з *enable_if*, а код став чистішим.

Тепер розберемося з тим, як працюють концепти. Почнемо з найпростішого обмеження — `true`, воно задовольняє будь-який тип.

```
template<typename T> concept Simplest = true;
```

Для створення концептів можна використати будь-які булеві операції або й інші концепти.

```
template<typename T>
concept Number = std::is_floating_point_v<T> || std::is_integral_v<T>;
```

Можна використовувати вирази й функції, проте вони мають бути `constexpr`.

```
template<typename T>
concept NumberWordSize = Number<T> && sizeof(T) == sizeof(void*);
```

Всередині блоку `requires` можна записати вираз. Якщо він не зможе скомпілюватися, концепт одразу стане `false`. В блоці також можна використати інше `requires`. Тоді вираз буде перевірятися не на коректність, а на те, чи рівний він `true` або `false`. Якщо він виявиться `false`, весь вираз стане `false`.

```
template<typename T>
concept IntegerMultiplicable = requires (T a, T b)
{
	a * b; //expression, must compile
	requires std::same_as<T, int>; //predicate, must be true
};
```

Всередині `requires` можна використовувати вкладений блок `requires`.

Розглянемо приклад з [5].

```
template<typename T>
concept C1 = requires(T v)
{
	requires (typename T::value_type x) { ++x; };
};
```

Проте такий код не спрацює. В цьому випадку `requires` є звичайним виразом і тому просто перевіряється, чи може він скомпілюватися. В деяких компіляторах це може працювати, проте Visual C++ вважає це помилкою. Для того, щоб такий блок спрацював потрібно додати ще одне `requires`.

```
template<typename T>
concept C1 = requires(T v)
{
	requires requires (T x) { ++x; };
};
```

Можна використовувати й складніші вирази.

Розглянемо приклад з [4].

```
template<typename T, typename U = T>
concept Swappable = requires(T&& t, U&& u)
{
	swap(std::forward<T>(t), std::forward<U>(u));
	swap(std::forward<U>(u), std::forward<T>(t));
};
```

Концепт буде `true` тільки тоді, коли вирази успішно скомпілюються.

Розглянемо ще один приклад з [4].

```
template<typename T>
concept C = requires(T x)
{
	{*x} -> std::convertible_to<typename T::InnerType>;
	{x * 1} -> std::convertible_to<T>;
};
```

Перше обмеження гарантує, що:

- `*x` — коректний, тобто x можна розіменувати;
- Тип `T::InnerType` коректний і існує;
- `*x` и можна конвертувати в `T::InnerType`

Друге — гарантує, що вираз `x * 1` коректний, а його результат конвертується в `T`.

З концептами можна явно вказати інтерфейс, який має бути в нашому класі, а отже всі вимоги будуть в коді, а не в документації.

Розглянемо приклад з [6]. *Автор визначає обмеження того, що шаблонний тип має підтримувати операцію хешування.*

```
template <typename T>
concept Hashable = requires(T t)
{
	{std::hash<T>{}(t) } -> std::convertible_to<std::size_t>;
};
```

Тепер розберемося, де можна використовувати концепти. Перш за все, `requires` можна використовувати для обмеження типового параметру.

```
//require declaration
template<typename T>
requires Number<T>
void func(T arg);

//require tail-declaration(just for functions)
template<typename T>
void func(T arg) requires Number<T>;
```

Другий варіант можна застосувати тільки для функцій. Також ім’я концепту можна використовувати замість `typename`.

```
//concept name instead of typename
template<Number T>
void func(T arg);
```

Або ж можна визначити неявний шаблон, який став доступний в сигнатурах будь-яких функцій в C++20.

```
void func(Number auto v);
```

Всі попередні приклади еквівалентні.

В блоці `requires` можна використати декілька предикатів об'єднаних логічними операторами.

Розглянемо приклад з [7].

```
//concept name instead of typename
template<typename T>
	requires is_standard_layout_v<T> && is_trivial_v<T>
void func(T v);
```

Якщо ж ми спробуємо інвертувати одну з умов, компілятор видасть повідомлення про помилку.

```
template<typename T>
	requires !is_standard_layout_v<T> && is_trivial_v<T>
void func(T v);
```

Це виникає через неоднозначність, яка виникає під час розбору певних виразів. *Такий приклад наводять в* [7].

```
template<typename T>
	requires (bool)&T::operator short unsigned int f();
```

Тут незрозуміло до чого віднести `unsigned` — до оператора чи типу повернення функції. Тому розробники вирішили, що без круглих дужок, аргументами `requires` можуть бути тільки літерали `true`, `false` або булеві імена полів виду value, T::value, T::trait::value, імена концептів і блоки `requires`.

Для шаблонного параметру можна одразу визначити блок `requires` і не створювати концепт. Хоча інколи такий прийом може стати в нагоді, зловживати ним не варто. Для уникнення дублювання коду обмеження краще виділяти в окремі концепти. В такому випадку код буде легше сприймати, адже за назвою легше визначити типи, які потрібно передати.

```
//Block requires without separate concept. Use wisely.
template<typename T>
requires requires(T v)
{
	{v * v};
}
```

Згідно з [8], вже визначені концепти містяться в заголовних файлах `<concepts>`, `<iterator>`, `<ranges>`. Також в файлі `<type_traits>` є класи-шаблони, які можна використати під час створення нових концептів.

Розділ 3. Система модулів

Ще одною довгоочікуваною функцією C++20 стала система модулів. Вона має на меті замінити директиву препроцесора `#include`, яка дісталася у спадок від мови C, і є частиною стратегії розробників з повного видалення препроцесора з мови.

Розділ 3.1. Що таке #include

`#include` — це директива препроцесора, яка дозволяє розділити код на логічні частини. Зокрема, на інтерфейс(зазвичай розташовується в заголовних файлах з розширенням «.h») і реалізацію(зазвичай розташовується в файлах з розширенням «.cpp»). Автор [9] виділяє такі переваги:

1. Створює бібліотеки коду, які можна використовувати багаторазово.
2. Розділяє інтерфейс і реалізацію
3. Ділить код на модулі, що за правильного використання дозволяє прискорити час компіляції.
4. Організує код в логічні компоненти.

Розділ 3.2. Чому його потрібно замінити

За своєю суттю `#include` є звичайним копіюванням одного файлу в інший. Саме тому часто виникають проблеми, які додають роботи програмісту.

Наприклад, нам потрібно слідкувати, щоб не включати заголовні файли двічі в один і той самий файл. Для захисту від цього, ми обгортаємо наші заголовки директивою `#ifndef`, або ж `#pragma once`.

Ще одною проблемою заголовних файлів є те, що вони можуть значно сповільнити компіляцію складного шаблонного коду, такого як контейнери STL. Контейнери використовують часто, тому компілятору доводиться розбирати заголовний файл кожного разу, коли його включають в інший.

Розділ 3.3. Що таке модулі

Модулі стали спробою згрупувати код в скомпільовані двійкові файли або “modules”, які містять імена типів, функцій і т. д. та використовуються у коді, що їх імпортує. Одним з найскладніших процесів компіляції є побудова абстрактного синтаксичного дерева. AST — це серіалізоване представлення без втрат всіх імен функцій, класів, шаблонів і т. д. Компілятор ж використовує цю інформацію для побудови кінцевого двійкового модуля.

З заголовними файлами відбувається те саме, проте генерація дерева дуже повільний процес і він відбувається кожного разу, коли заголовний файл кудись включається. Модулі ж роблять цей крок тільки один раз.

Все, що знаходиться всередині модуля і не розкривається за допомогою export, стає недоступним під час імпортування. Проте, вони можуть працювати в комбінації з заголовними файлами, щоб підтримувати зворотню сумісність. Також модулі дозволяють зменшити кількість файлів вдвічі. Тепер нам не обов'язково потрібно записувати реалізацію в окремий файл, ми просто можемо не експортувати непотрібні функції, класи чи типи.

Розлянемо приклад модуля з [9].

```
//NumberCruncher.ixx -> .ixx is requires for visual studio, for others it can be .cppm

export module NubmerCruncher;

//We can import other modules
import logger;

//Any macros could be defined here
#define PI 3.14

namespace numbers
{
	//This thing can't be imported as we don't use export
	float applyCrunchFactor(float number)
	{
		return number * PI;
	}

	//But here we use `export` and so can import it to another module
	export float crunch(float number)
	{
		//Calls an internal function.
		auto crunched = applyCrunchFactor(number);

		logger::info("Crunched {} with result of {}", number, crunched);
		return crunched;
	}
}

export void MyFunc();
```

Тут ми визначаємо іменований модуль за допомогою `export module`, а також функцію, яку ми хочемо показати користувачу, за допомогою `export`.

І ось так ми його використовуємо.

```
import NubmerCruncher;

int main()
{
	auto value = numbers::crunch(42);
}
```

Модулі стали одним з основних можливостей для нової редакції мови. Стандартна бібліотека вже переписана і готова до роботи з модулям, проте старі версії бібліотек ще довго будуть використовуватися і повноцінний перехід на модульну систему вимагає часу.

Висновки

У цій роботі описані прийоми та інструменти, якими найчастіше користуються розробники, щоб не допустити або все ж знайти помилку в шаблонному коді. Було досліджено історію розвитку шаблонів в мові C++, проблеми, з якими стикаються розробники. Та в якому напрямку розвивається мова.

Проаналізовані можливості нової редакції мови C++, зокрема концепти та систему модулів. Описані способи прискорення компіляції, систему SFINAE та інструмент на її базі — enable_if.

У майбутніх дослідженнях може розглядатися розробка доповнення на кшталт IntelliSense, яке зможе в реальному часі сканувати шаблонний код і давати підказки як його покращити.

Додаток

Фрагмент коду з тестуванням.

```
#define SQUARE(a) (a * a)
#include <typeinfo>
import <iostream>;
import <type_traits>;
import <vector>;
import <type_traits>;
import ConceptFunctions;
import MyConcepts;
import Print;
import SFINAE;
import CloseDistant;
import InstantiationChain;

using namespace std;

//Won't compile
//template<typename T>
//void foo(T t)
//{
//    undeclared();
//    undeclared(t);
//    //static_assert(sizeof(int) > 10, "int too small");
//    static_assert(sizeof(T) > 3, "T too small");
//}

template<typename T>
void system_check()
{
	//Won't compile on not 64 bit system
	static_assert(sizeof(void*) * CHAR_BIT == 64, "Target platform is not 32-bit");
}

struct TrivialType
{
	struct value_type{};
};

int main()
{
	cout << std::boolalpha;

	std::vector<int> v{ 1, 2, 3, 4 };

	cout << "Macros square: " << SQUARE(9) << '\n';

	//Just the same
	func1(5);
	func1('d');
	func2(5);
	func3(5);
	funcAnonymousType(5);
	func(3);
	funcMultiplication(3);
	//Won't compile, constraints are not satisfied
	//func1(std::vector<int>());
	//funcMultiplication(nullptr);
	cout << '\n';

	//Disjunction with inner types
```

```
	foo(TrivialType(), TrivialType());
	cout << '\n';

	//Check whether types' size is the same as bit in the machine
	cout << "int is NumberWordSize: " << NumberWordSize<int> << '\n';
	cout << "long long is NumberWordSize: " << NumberWordSize<long long> << '\n';

	//Check whether integer multiplication can be made
	cout << "int can be multiplied: " << IntegerMultiplicable<int> << '\n';
	cout << "long can't be multiplied: " << IntegerMultiplicable<long> << '\n';

	//Swappable types
	int a{1};
	int b{2};
	cout << "swap(" << a << ',' << b << "): ";
	MySwap(a, b);
	cout << "(" << a << ',' << b << ")\n";
	cout << '\n';

	//Hash inteface
	cout << "Hashable: " << Hashable<std::string> << '\n';
	cout << "Not hashable: " << Hashable<TrivialType> << '\n';
	cout << '\n';

	//Simple print
	cout << "Simple print\nVector:\n";
	SimplePrint::Print(cout, v);
	cout << "Not container:\n";
	SimplePrint::Print(cout, 54);
	cout << "\n\n";

	//Concept print
	cout << "Concept print\nVector:\n";
	ConceptPrint::Print(cout, v);
	cout << "Not container:\n";
	ConceptPrint::Print(cout, 54);
	cout << "\n\n";
	//They're the same

	//Integral SFINAE
	cout << "Do stuff for integral\n";
	SFINAE::do_stuff(a);
	SFINAE::do_stuff(v);
	cout << '\n';

	//Close distant SFINAE
	cout << "3, 3 is close(SFINAE): " << CloseDistantSFINAE::ElementsAreClose(3, 3)
<< '\n';
	cout << "3.2, 3.1 is close(SFINAE): " <<
CloseDistantSFINAE::ElementsAreClose(3.2, 3.1) << '\n';
	cout << "3, 3 is close(Concept): " << CloseDistantConcept::ElementsAreClose(3,
3) << '\n';
	cout << "3.2, 3.1 is close(Concept): " <<
CloseDistantConcept::ElementsAreClose(3.2, 3.1) << "\n\n";

	//Instantiation chain
	InstantiationChain::Client mainClientSimple;
	InstantiationChainShallow::Client mainClientShallow;
	InstantiationChainAssert::Client mainClientAssert;
	//Uncomment one shell functions to see error which will be generated
	//shell(mainClientSimple);
	//shell(mainClientShallow);
	//shell(mainClientAssert);
}
```

Список джерел

[1] B. Stroustrup, «Bjarne Stroustrup: C++20 Generic Programming,» https://www.youtube.com/watch?v=03BtQljH2oA.

[2] D. Vandevoorde, N. M. Josuttis та D. Gregor, C++ Templates: The Complete Guide 2nd Edition, Addison-Wesley Professional, 2017.

[3] E. Bendersky, «SFINAE and enable_if,» https://eli.thegreenplace.net/2014/sfinae-and-enable_if/.

[4] «Стандарт C++20: огляд нових можливостей C++. Частина 3 "Концепти",» https://habr.com/ru/company/yandex_praktikum/blog/556816/.

[5] A. Krzemieński, «Requires-expression,» https://akrzemi1.wordpress.com/2020/01/29/requires-expression/.

[6] T. Doumler, «How C++20 Changes the Way We Write Code,» https://www.youtube.com/watch?v=wKc1cigKlH0.

[7] A. Krzemieński, «Requires-clause,» https://akrzemi1.wordpress.com/2020/03/26/requires-clause/.

[8] Cppreference, «Concepts library,» https://en.cppreference.com/w/cpp/concepts.

[9] A. Nayar, «C++20 in 2020: Modules,» https://pspdfkit.com/blog/2020/cpp20-in-2020-modules/.