[BASIC](#basics) =

[declaration vs definition](#basics_declaration_definition) N4868, 6, 9

[one definition rule](#basics_ODR)

[what expressions are](#basics_what_expressions_are)

[value categories](#basics_value_categories)

["WORDS" in CODE](#words_in_code) =

[reserved key words](#words_reserved_key_words)

[identificators](#words_identificators)

[operators](#words_operators)

[types qualifiers](#words_type_qualifiers)

[specifiers](#words_specifiers)

[modificators](#words_modoficators)

[COMPILE PROCESS](#compile_process) =

program structure

linkage class mamagement

inline

sourse code

preprocessing

compile

assembling

linking

[OBJECTS](#what_objects_are) =

[TYPES INFORMATION](#objects_and_types_info) =

sizeof [...] ( )

SFINAE , <type\_traits>

RTTI , <typeinfo>

auto = template type deduction

decltype

typedef, using

[ERRORS HANDLING =](#errors)

[assert, static\_assert](#errors_asserts)

[exceptions](#errors_exceptions)

[REFS + MOVE SEMANTIC](#refs_and_move_semantic) =

values and references

std::move( )

reference collapsing rule

special template deduction

std::forward( ) + perfect forwarding

memory leaking

move semantic

unique\_ptr

shared\_ptr

weak\_ptr

synopsis

[ITERATORS](#iterators) =

basic info

iterator / adress invalidation

iterators hierarchy

std::insert\_iterator

<iterator\_traits>

[TEMPLATES](#templates) =

[function templates](#templates_function_templates)

[class templates](#templates_class_templates)

[instantiation](#templates_instantiation)

[explicit specialization](#templates_explicit_specialization)

[function template specialization](#templates_function_spacialization)

[deep copying (template for ptr)](#templates_deep_copying)

[SFINAE](#templates_sfinae)

[template recursion](#templates_recursion)

[variadic templates](#templates_variadic_templates)

[fold expression](#templates_fold_expressions)

[specialization + inheritance + variadic](#templates_spec_inherit_variadic)

[tuple example](#templates_tuple_example)

[concepts](#templates_concepts)

[MULTYTHREAD](#multythread) =

basic info

memory fence

thread creation

threads interaction

mutex, lock guard

conditional variables

coroutine

[LOW LEVEL](#low_level) =

[inline assembler](#low_level_inline_asm)

[intrisics](#low_level_intrisics)

ДОПОЛНИТЕЛЬНЫЕ ЦЕЛЬНЫЕ МАТЕРИАЛЫ

ФПМИ, продвинутый С++ [part\_1](https://www.youtube.com/playlist?list=PL4_hYwCyhAvazfCDGyS0wx_hvBmnAAf4h) , [part\_2](https://www.youtube.com/playlist?list=PL4_hYwCyhAvYTzwME4vQoDO8ZINM5trra)

**BASICS**

*more about: …*

СУЩНОСТЬ = абстрактное название "единиц" программы:

переменная

объект

функция

шаблон

концепт …

**DECLARATION vs DEFINITION**

ОБЪЯВЛЕНИЕ (declaration) =

для компилятора

вводит в единицу трансляции идентификатор сущности

поясняет тип и семантические свойства сущностей

ОПРЕДЕЛЕНИЕ (definition) =

для линкера

описывает реализацию сущности

указывает место этой сущности среди единиц трансляции

В соответствии со стандартом, любое объявление есть одновременно и определение, кроме особых случаев [N4868, 6.2.2].

ИНИЦИАЛИЗАЦИЯ (initialization) =

присвоение переменной значения во время выполнения ее конструктора.

**ONE DEFINITION RULE (ODR)**

Правило одного определения. Гласит, что сущность:

- может быть объявлена сколько угодно раз

- может быть определена лишь единожды за всю программу

- исключение - inline

ПРИМЕР – объявление и определение в классе:

1. struct S
2. {
4. // declaration (requires definition outside of class)
5. static T a;
6. static const T b = 1; // initializing is optional
7. S( );
9. // definition - S obj; definition is required to use
10. - default values are allowed
11. T c;
12. const T d;
13. static constexpr T e = 1; // initializing is necessary
14. static inline T f;
15. S( ) { … };
16. };
18. // так как поле b класса это const (и при этом static)
19. T var = S::b; // ок (прямая подстановка значения)
20. T\* ptr = &S::b; // error (так как нет определения)

Обратить внимание, что дефолтное значение в объявлении не делает

это объявление определением (синтаксически похоже, но все же нет). Чтобы определить объявленные поля:

1. T S::a; // zero by default
2. const T S::b; // only one initializing for const
3. S( ) { … }

Определение не обязательно связано с выделением памяти. Например:

1. // here S::e already defined but memory is not allocated yet
2. T var = S::e; // is not allocated as well
3. const T\* ptr = &S::e; // only now it is

Чтобы наглядно увидеть нюансы выделения памяти – см. ассембер

**WHAT EXPRESSIONS ARE**

Expression is a SEQUENCE OF (OPERATORS + OPERANDS)

Expression:

syntax definition = something, that ends with ;

might result in a value and/or cause side effects

every expression has

type

value category

Some examples:

42;

int i;

i = i + 42;

\_f( );

struct S { };

... and so on ...

**VALUE CATEGORIES**

l\_value

gl\_value

expressions x\_value

r\_value

pr\_value

Every expression is belong to one of the three fundamental categories: l\_, x\_ or pr\_value.

***gl*** = evaluation determines the identity of an object or function

***x*** = glvalue that denotes an object whose resources can be reused

***l*** =glvalue that is not an x\_value

***r*** = x\_value + pr\_value

***pr*** = or evaluation that initializes an object

or computes the value of an operand of an operator

or an expression that has type (cv) void

X\_VALUE

General rule is:

unnamed r\_value\_reference = x\_value

named r\_value\_reference = l\_value

r\_value\_reference to function = l\_value

1. struct S
2. {
3. int member;
4. };
6. S&& operator+(S, S);
7. S&& function( );
9. S obj;
10. S&& obj\_2 = static\_cast<S&&>(obj);

x\_values are: function( )

function( ).member

obj + obj

static\_cast<S&&>(obj)

**\*\*\*\*\***

**(с 101 стандарта)**

**https://en.cppreference.com/w/cpp/language/value\_category**

**‘WORDS’ in CODE**

*more about: …*

\*\*\*\*\* **RESERVED KEY WORDS** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Keywords are used by the language

So keywords are not available for re-definition or overloading

**LIST OF KEYWORDS** - [link](https://en.cppreference.com/w/cpp/keyword)

KEYWORDS OF UNRECOGNIZED CATEGORY

\*\*\*\*\* **IDENTIFICATORS** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Identifacator is a name of any object

Rules:

1. only letters + numbers + underscore
2. starts only with letter
3. must be iformative

\*\*\*\*\* **OPERATORS** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Most of C++ operators not actually the words but symbols. But there are some really “word-oretarors” and moreover all of them use keyword “operator” in declaretion.

**SIZEOF( )** watch further in this file

**TYPEID( )** watch further in this file

**T( ) -** C-style cast operator. Overloading syntax (such syntax means that the input expression will be cast to ‘T’ type output expression) **⇨** operator T( ) { return … ; } **⇦** e.g.:

1. struct STRUCT { …
2. int x = 1;
3. int y = 2;
4. **operator int( )** { return x + y; }
6. STRUCT s;
7. int z = **(int)**s; // 3

\*\*\*\*\* **TYPES QUALIFIERS** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Qualifier states additional type info

Make sence only with l-value

Might be used

* only once for one type
* only at the start of declaration

1. const int\* const ptr = &a;

volatile int a, b // but not: volitile int a, const b;

CV-QUALIFIERS

**CONST**  - object cannot be modified at all

**-** only initialization

- both compile- and run-time

- object has internal linkage

**VOLITILE -** object cannot by optimized by compiler – variable would be read every time despite of replacing with it’s value. Important for multithread processes using commonvariable.

1. bool VAR = 0; vs. volatile bool VAR = 0;
2. if (VAR) -> if (0); if (VAR) { … // нет подстановки

ADDITIONAL, NOT IN C++ ISO !

\_\_RESTRICT (or \_\_RESTRICT\_\_) - usage only with ref’s - means that the programmer (!) guarantees that no other pointer in the current scope refers to the same memory location. This helps compiler to optimize code better. Example (specific sintax):

1. int var = 1;
2. int\* \_\_restrict ptr = &var;
4. // will be compiled, but defective logic and undefinied behavior
5. int\* \_\_restrict other = &var;

\*\*\*\*\* **SPECIFIERS**  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**AUTO, DECLTYPE -** auto = type deduction as in templates

- decltype = autonomous type deduction

**REGISTER** any type variable. It is prompt for compiler:

- frequent usage of variable

- to keep this variable in processor register

**-** 1: register int var = 42;

**INLINE** - before = inline expansion (prompt for compiler)

- now = about linking (ODR exception)

**TYEPNAME** 1:use in template definition

4: use with dependent names to make compiler know

the expression is the type but not the member

1. template<typename T> // template syntax
2. struct S { …
3. // meaning is dependent of what is T’s definition
4. typename T::NAME \* ptr;

STORAGE CLASS SPECIFIERS

(only for objects and functions)

**STATIC** - static or thread storage duration and internal linkage

**EXTERN** - static or thread storage duration and external linkage

**THREAD\_LOCAL** - thread storage duration

**MUTABLE** – variable can be changed (for 'const' methods)

CONST… SRECIFIERS

**CONSTINIT** variable must be initialized in compile-time

- in other case = compile error

- only static/thread storage duration variables

- result variable is non-const (!)

**CONSTEXPR** variable or function evaluated in compile-time

- result variable:

is const

has linkage:

internal for globals

week external for class members

must be initialized immidietly

must be literal type

- function might:

be re-used in run-time

non-constexpr variables

include branches and loops

call other constexpr functions

deal with classes (read more)

- function must:

have the only 'return'

return literal type

**CONSTEVAL** the same as constexpr function, but:

- evaluates only in compile-time

- call only constexpr function (not consteval)

\*\*\*\*\* **MODIFICATORS** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**COMPILE PROCESS**

*more about:* …

**PROGRAM STRUCTURE**

TRANSLATION UNIT (единица трансляции) =

файл \*.cpp или \*.c

плюс все его заголовочные файлы \*hpp или \*.h.

после обработки препроцессором

Все программы (кроме самых простейших) состоят из нескольких единиц трансляции.

LINKAGE (связывание) = NO LINK. (нет связывания)

INTERNAL (внутренее)

EXTERNAL (внешнее)

Связывание является характеристикой сущности, которая необходима для линковщика (последняя стадия компиляции программы / подробнее о процессе линковки позже).

Указывает линковщику, является ли эта сущность доступной только для одной конкретной единицы трансляции или другие единицы трансляции также имеют доступ к ней:

no link. = только в области видимости ( = local scope)

internal = только внутри одного юнита

external = доступно всем юнитам программы

По умолчанию, то, что находится в global scope:

internal : const переменные

external : non-const переменные, все функции

**LINKAGE CLASS MANAGEMENT**

EXTERN = принудительно external.

Доступна в любом юните:

- где содержится ее объявление

- объявление должно быть помечено extern

- сущность может быть определена только один раз

STATIC = принудительно internal

Для этой же цели можно использовать анонимное пространство имен:

1. namespace { T var; } // теперь var = internal
2. cout << var << endl; // использовать как обычно

STATIC = change storage duration

Если локальная переменная объявлена как static, это озачает, что она будет единожды создана в момент первого вызова функции и будет уничтожена при завершении программы. Обращаться к этой переменной можно только из той области видимости, в которой она была задана.

1. int function( )
2. {
3. static int i = 0;
4. return i++ ; // 0, 1, 2 …
5. }

Если static используется для объявления поля/метода класса, это означает, что поле/метод принадлежат самому классу, а не отдельным объектам. Доступ:

- осуществляется через оператор области видимости

- возможен в любом юните, где есть объявление

1. // 1.cpp
2. struct S { static int i; } // S::i объявление
3. int S::i = 1; // S::i определение
5. // 2.cpp
6. struct S { static int i; } // S::i объявление
7. cout << S::i << endl; // S::i использование

**INLINE**

inline = исключение из ODR. Позволяет определять одну сущность несколько раз в разных единицах трансляции (но не в одной!).

inline сущности по умолчанию имеют внешнее связывание

- объявление должно быть в любом файле, использующем их

- все объявления должны быть inline

- если несколько определений, они должны быть идентичны

Когда линкер встречачет inline сущность, он оставляет только одно ее определение, а остальные просто игнорирует, что приводит к уменьшению объема кода (если определения будут отличаться, то линкер не выдаст ошибку, но выберет одни из них случайно = UB).

Неявно inline переменная: static constexpr внутри (!) класса

1. constexpr T \_f( ) { … }
2. struct S { …
3. static constexpr T member = \_f( ); // неявно inline

Неявно inline функции:

- полностью определенная внутри определения класса

- любая constexpr функция

- любая =delete функция

inline следует использовать всякий раз, когда очевидно, что определение сущности встретится в нескольких юнитах.

INLINE and TEMPLATES

TEMPLATES LINKAGE

…

TEMPLATES INSTANTIATIONS LINKAGE

Для шаблонов переменных можно указывать inline для надеждности, так как от компилятора зависит, сделать ли инстанциацию inline или нет:

1. // header.h
2. template<typename T> bool b = true; // not safe
3. template<typename T> inline bool = true; // safe

Для шаблонов функций inline бессмыслен:

1. template<typename T>
2. inline T function(T arg) // 'inline' is excess
3. { return arg; }

Для явных инстанциаций – тоже.

1. template
2. inline int function(int); // 'inline' is excess

А вот для полной специализации inline необходим:

1. template< >
2. inline int function(int i)
3. { return i \* 2; }

**ИСХОДНЫЙ КОД**

расширение **\*.cpp**

Исходный код должен быть скомпилирован в:

либо исполняемый файл

либо статическую библиотеку

либо динамическую библиотеку

Рассмотрим процесс компиляции на примере кода:

1. #include <iostream>
2. using namespase std;
3. #define RETURN return 0
4. int main( ) // главная функция
5. {
6. cout << “Hello, world!” << endl;
7. RETURN;
8. }

**ПРЕПРОЦЕССИНГ**

расширение **\*.ii**

Пропроцессинг = обработка кода с помощью МАКРОПРОЦЕССОРА

Макропроцессор = программа, преобразующая входной текст в выходной текст с использованием правил замены последоваельностей символов (правил макроподстановки)

Директивы препроцессора начинаются с символа: # (строки 1 и 3)

Читать о значении директив препроцессора - [link](https://docs.microsoft.com/en-us/cpp/preprocessor/preprocessor-directives?view=msvc-160)

Препроцессор может служить для использования МАКРОСОВ

Последовательность препроцессинга:

- замена комментариев пустыми строками

- текстовое включение файлов - #include<>

- макроподставноки - #define

- обработка директив условной компиляции - #if, #ifdef …

Код после работы препроцессора (так как комментарии также убраются из кода, то пояснения будут писаться теперь с помощью комментариев в стиле Python – не путать с командами препроцессора, которые больше никогда не встретятся)

82374: # ↓↓↓

82375: # от 1 и до сюда вставляется содержание <iostream>

82376: using namespase std;

82377: # строка с define выполнена (см 18170) и удалена

82378:

82379: int main( ) # комментарий отсюда удален

82380: {

82381: cout << “Hello, world!” << endl;

82382: return 0; # произведена замена (см 18165)

82383: }

**КОМПИЛЯЦИЯ**

расширение **\*.s**

На этапе компиляции происходит преобразование полученного после препроцессинга кода в АССЕМБЛЕРНЫЙ код. Ассемблерный код = промежуточное состояние между исходным кодом и машинным кодом.

Кусочек ассемблерного кода:

1. .file "name.cpp"
2. .local \_ZStL8\_\_ioinit
3. .comm \_ZStL8\_\_ioinit,1,1
4. .section .rodata
5. .LC0:
6. .string "Hello, world!"
7. .text
8. .globl main
9. .type main, @function
10. main:
11. .LFB1021:
12. .cfi\_startproc
13. pushq %rbp
14. .cfi\_def\_cfa\_offset 16
15. .cfi\_offset 6, -16
16. movq %rsp, %rbp

...

... # весь ассемблерный код занял 84 строки

... # ↓↓↓

1. .ident "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609"
2. .section .note.GNU-stack,"",@progbits

Во время компиляции осуществляется:

- лексический анализ

- синтаксический анализ

- семантический анализ

- оптимизация

- генерация кода

**АССЕМБЛИРОВАНИЕ**

расширение **\*.o**

Ассемблер = транслятор, который переводит ассемблерный код в машинный. На выходе получается ОБЪЕКТНЫЙ файл. В одном проекте может быть несколько объектных файлов. Для завершения сборки проекта, необходимо соединить их между собой.

В каждом объектном файле хранится ТАБЛИЦА СИМВОЛОВ = структура данных, в которой хранятся все идентификаторы (и информация, связанная с их объявлением + доступностью снаружи объектного файла (extern))

Все СТАТИЧЕСКИЕ БИБЛИОТЕКИ представляют собой объектные файлы, которые далее будут объедины друг с другом.

**КОМПОНОВКА (линковка)**

расширение **\*.exe** или любой другой исполняемый файл ОС

Компоновщик (линкер), используя таблицы символов, связывает все объектные файлы в один исполняемый файл.

Все ДИНАМИЧЕСКИЕ БИБЛИОТЕКИ представляют из себя исполняемые файлы.

**program.cpp**

**ПРЕПРОЦЕССИНГ**

**program.ii**

**КОМПИЛЯЦИЯ**

**program.s**

**АССЕМБЛИРОВАНИЕ**

**program.o**

**ЛИНКОВКА**

**program.exe**

**headers**

**(прототипы)**

**object files**

**(определения)**

**what OBJECTS are**

*more about:* [cppreference\_object](https://en.cppreference.com/w/cpp/language/object)

**OBJECTS and TYPES**

**INFORMATION**

*more about:* [*type\_support*](https://en.cppreference.com/w/cpp/types)

\*\*\*\*\* **SIZEOF** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Return number of bytes that type (1) or expression (2) takes up in memory. Type of return value is size\_t.

(1)

1. size\_t size = sizeof(T); // ‘T’ is any type
2. // cannot be used with
3. = sizeof(functional( )) // functional objects
4. = sizeof(int[ ]) // incoplete types
5. = sizeof(bits.bits) // bit-field values

(2)

1. T var;
2. size\_t size = sizeof var; // … = sizeof(T)
4. (!) has no side effects
5. int i = 1;
6. sizeof ++i;
7. cout << i << endl; // 1

\*\*\*\*\* **SIZEOF ... (**\*PARAMETER\_PACK or TYPE\_PACK\***)** \*\*\*\*\*\*\*\*\*\*\*\*\*

Returns number of parameter\_pack or type\_pack arguments

1. template<typename ... Types>
2. size\_t function(Types ... args)
3. { return sizeof ... (Types); } // same as sizeof ... (args)

\*\*\*\*\* **<TYPE\_TRAITS> ( SFINAE )** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

SFINAE = (compile time) = substitution failure is not an error

All about type\_traits - [link](https://en.cppreference.com/w/cpp/header/type_traits)

The base class of type\_traits is integral\_constant:

1. template<typename T, T v>
2. struct integral\_constant
3. {
4. static constexpr T value { v };
5. using value\_type = T;
6. using type = integral\_constant<T, v>;
7. constexpr operator value\_type( ) const noexcept
8. { return value; }
10. constexpr value\_type operator( ) ( ) const noexcept
11. { return value; }
12. };

There are three type aliases as well:

1. template<bool B>
2. using bool\_constant = integral\_constant<bool, B>;
4. using true\_type = bool\_constant<true>
5. using false\_type = bool\_constant<false>

All of type\_traits are inherited from one of these aliases, e.g.:

1. template<> struct is\_integral<int> : public true\_type { } ;
2. template<> struct is\_class<vector> : public true\_type { } ;

Usage bool\_constant::value example

1. bool b = is\_integral<int>::value; // true
2. bool b = is\_class<int>::value; // false

Usage bool\_constant::type example

1. template <typename T>
2. void processHelper(const T& t, true\_type)
3. {
4. cout << t << " is an integral type." << endl;
5. }
6. template <typename T>
7. void processHelper(const T& t, false\_type)
8. {
9. cout << t << " is a non-integral type." << endl;
10. }
11. template <typename T>
12. void process(const T& t)
13. {
14. processHelper(t, typename is\_integral<T>::type {});
15. }
17. process(42); // 42 is an integral type
18. process("string"s); // string is a non-integral type

16: is\_integral<T>::type returns true\_type or false\_type

for compiler knows it's a type we use 'typename'

then for instantiate type into anonimous object we use { }

so for e.g. 42 = (42, true\_type { })

\*\*\*\*\* **<TYPEINFO> ( RTTI )** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

RTTI = (run time) = run time type information

Return reference to std::type\_info object, wich contains information about type. Header <typeinfo> must be included.

1. #include <typeinfo>
3. T var;
4. auto& info\_1 = typeid(var); // info about var type ‘T’
5. auto& info\_2 = tipeid(int); // info about type ‘int’
7. cout << info\_1.name( ) << endl; // get type T name
9. // compare types
10. // one type = same hash\_code, but hash codes might repeat!
11. bool b = (info\_1.hash\_code( ) == info\_2.hash\_code( ));

std::type\_info cannot be CopyConstructable nor CopyAssignable

cv-qualifiers are ignored by typeid(T)

\*\*\*\*\* **AUTO** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

compile-time

**Deduce** (выводит) type from given expression.

Syntax: auto var = \*expression\* ;

Механизм вывода идентичен механизму вывода типа в шаблонах:

auto var … **⇦** the same sence as **⇨** template<typename T> …

ПРАВИЛА ВЫВЕДЕНИЯ ТИПОВ

таблица, “cv” обозначает “cv-qualifier”

\*expression\* **⇨** auto **⇨** auto& **⇨** auto\*

(cv) T T err. err.

(cv) T& T (cv) T& err.

(cv) T\* (cv) T\* err. (cv) T\*

\*expression\* **⇨** auto&&

(cv) r-value T (cv) T&&

(cv) T& (cv) T&

AUTO and STD::INITIALIZER\_LIST< >

Отличие от механизма шаблонов: auto совместим с initializer\_list

1. template<typename T>

void funct(T var) { … }

std::vector<T> vec;

auto list = {T, T, T}; // OK. auto = initializer\_list<T>

funct({1, 2, 3}); // compile error

funct(list); // OK

funct(vec, {1, 2, 3}); // OK, but not ({1, 2, 3}, vec). WTF?!

NEW FUNCTION SEMANTIC = AUTO

В настройки компилятора:

-fconcepts or –std=c++20 (GCC 9.2.0)

-fconcepts-ts or -std=c++20 (mingw64)

1. void function (const auto& a, const auto& b)
2. { return a + b; }

AUTO and LAMBDA

...

ОГРАНИЧЕНИЯ В ИСПОЛЬЗОВАНИИ

1. // нельзя использовать для не-статического поля класса
2. class CLASS { …
3. auto var; // compile error
5. // можно – для статического
6. class CLASS { …
7. static const auto var = 42;

\*\*\*\*\* **DECLTYPE** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

compile-time

**Yields** exactly object type (1) or according to the rules (2).

Syntax: decltype(\*obj or expression\*) var;

(1) unparanthesized + identificator or class\_[member\_access](https://en.cppreference.com/w/cpp/language/operator_member_access) expr.

1. T var\_1 = … ;
2. decltype(var\_1) var\_2 = … ;
4. struct { T var\_1; } S;
5. decltype(S.var\_1) var\_2 = … ;

var\_2 type in both examples **⇦** exactly the same as **⇨** T

(2)

1. T var\_1 = … ;
2. prvalue -> T
3. decltype(var\_1 \* 2) var\_2;
4. paranthesized lvalue -> T&
5. decltype((var\_1)) var\_2;
7. xvalue -> T&&
8. decltype(std::move(var\_1)) var\_2;

DECLTYPE(AUTO)

Объединение: сохранение синтаксиса auto

сохранение семантики типа decltype

1. T var\_1 = … ;
2. decltype(auto) var\_2 = var\_1;

NEW FUNCTION SEMANTIC = DECLTYPE

Standard semantic with ‘decltype(auto)’ usage:

1. template<typename T1, typename T2>
2. decltype(auto) funct(T1 a, T2 b)
3. { return … ; }

New semantic: want to explicit ‘decltype(\*expr\*)’ usage:

1. // error, because variables not declared yet
2. template<typename T1, typename T2>

**⤬**

1. decltype(a + b) funct(T1 a, T2 b)
2. { return … ; }
4. // new semantic
5. template<typename T1, typename T2>

**ok**

1. auto funct(T1 a, T2 b) -> decltype(a + b)
2. { return … ; }

ОБЪЕДИНЕНИЕ с AUTO

**ok**

**ok**

В настройки компилятора:

-fconcepts or –std=c++20 (GCC 9.2.0)

-fconcepts-ts or -std=c++20 (mingw64)

1. decltype(auto) function (const auto& a, const auto& b)
2. { return a + b; }

\*\*\*\*\* **TYPEDEF and USING** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TYPEDEF and USING = is to create TYPE ALIAS (псевдоним)

Псевдонимы служат для:

- читабельность кода

- поддержка кода (изменить тип только в одном месте)

- кросплатформенность

- упрощение длинных наименований

ПРИМЕРЫ

typedef: typedef int users\_t;

type alias: using users\_t = int;

Псевдоним, определенный внутри класса, может быть использован через оператор области видимости.

1. struct NAME { …
2. typedef unsigned char counter;
4. NAME::counter i = 0;
5. ++i; // i = 0 … 255

Таким образом, например, можно написать шаблон класса, который принимает в качестве параметра тип какого-либо объекта и добавляет к нему константность.

1. template <typename T>
2. struct ADD\_CONST
3. {
4. typedef T const type;
5. };
7. ADD\_CONST<int>::type i = 0 // const int i = 0
8. ++i; // error

**ERRORS HANDLING**

*more about: …*

**ASSERT and STATIC\_ASSERT**

Способ отследить ошибки в компиляции или работе программы и принудительно прекратить выполняемое действие.

ASSERT = run-time errors

assert( \*NOT\_error\_condition\* && "massage\_string" );

1. #include <cassert>
2. assert(sizeof(obj) != 4 && "Wrong object size")

Если NOT\_error\_condition = true , то ошибки нет

Иначе компилятор выведет сообщение, содержащее:

- полный путь от хранения файла

- строку, в которой возникла ошибка

- "assertion failed: "

- "massage\_string"

Работа программы будет полностью ПРЕКРАЩЕНА, следовательно нужно ВРУЧНУЮ **(!)** следить за следами незавершенной работы программы

1. // отключает все assert'ы (но не static\_assert)
2. #define NDEBUG
3. #include <casssert>

STATIC\_ASSERT = compile time

Выполняет проверку условия. Если условие не соблюдатеся, прекращает компиляцию программы и выводит сообщение об ошибке.

static\_assert( \*NOT\_error\_condition\*, "massage\_string" );

1. #include <cassert>
2. static\_assert(sizeof(void\*) == 4, "You use 64-bit compiler")

Если NOT\_error\_condition = true , то ошибки нет

Иначе компилятор выведет сообщение, содержащее:

- полный путь от хранения файла

- строку, в которой возникла ошибка

- "static assertion failed: "

- "massage\_string"

**EXCEPTIONS**

**REFS and MOVE SEMANTIC**

more about: custom deleter to std::...\_ptr - [link](https://medium.com/pranayaggarwal25/custom-deleters-with-shared-ptr-and-unique-ptr-524bb7bd7262)

**VALUES and REFERENCES**

L-VALUE

слева от оператора присваивания "="

ссылается на память, адрес которой может быть получен через &

невременное

все именованные переменные это l-value

1. int i = 42; // i is l-value
2. int& foo(); // returns l-value
3. i = 43; // ok, i is an l-value
4. int\* p = &i; // ok, i is an l-value
5. foo() = 42; // ok, foo() is an l-value
6. int\* p1 = &foo(); // ok, foo() is an l-value

R-VALUE

справа от оператора присваивания "="

то, что не является l-value

временное

подробнее - [link](https://habr.com/ru/post/322132/)

1. int i = 42; // 42 is r-value
2. int foobar(); // returns r-valu
3. i = 43; // ок, 43 is an rvalue
4. i = foobar(); // oк, foobar() is an rvalue
5. int\* p2 = &foobar(); // error, cant take r-value address

FULL-EXPRESSION RULE

Гласит, что уничтожение временного объекта происходит в самую последнюю очередь при выполнении выражения.

1. void print(int arg) { cout << arg << endl; }
2. int \_f( ) { return 42; }
4. // temporary object inside full-expression
5. print(\_f( )); // ок

(CONST) L-VALUE REFERENCE = (CONST) T&

ссылка на r-value // обязательно const

ссылка на l-value

сама по себе всегда является l-value

В случае с l-value ссылкой на r-value значение const является обязательным, так как только так скомпилируется инициализация. При этом r-value не может быть изменен, так как (const T&) это ссылка на костанту.

Также const обязателен, если l-value само по себе является const.

(CONST) R-VALUE REFERENCE = (CONST) T&&

ссылка на r-value

основное назначение = perfect forwarding и move semantic

В случае с r-value ссылкой const является опциональным, поскольку r-value ссылка изначально и сущестует для иницииализации с помощью r-value значения. При этом, в зависимости от наличия спецификатора const, значение либо может быть изменено, либо нет.

1. T&& ref = 1; // ок
2. const T&& c\_ref = 1; // ок
3. ref++; // ок
4. c\_ref++; // error

Является ли r-value ссылка сама по себе r-value значением? Отличительный критерий:

- если r-value ссылка имеет имя, то это l-value значение

- если r-value ссылка не имеет имени, это r-value значение

То есть правило ничем не отличается от привычного определения

1. T&& \_f( )
2. {
3. T&& ref = 42; // ref = l-value
4. return ref;
5. }
7. \_f( ); // returned = r-value

TEMPORARY MATERIALIZATION

Дополнительный смысл

ссылки на константу на r-value

r-value ссылки

в продлении жизни r-value.

Механика: создается временный объект, который инициализируется значением r-value ⇨ к временному объекту биндится ссылка ⇨ временный объект живет до тех пор, пока живет ссылка на него.

1. // на примере возвращаемого значения из функции
2. int function( ) { return 42; }
3. const int& lref = function( );
4. int&& rref = function( );
5. // на примере прередачи значения в функцию
6. // (подробнее о перегрузках с const T& и T&& позже)
7. void function(const int& arg) { … }
8. void function(int&& arg) { … }
10. int i = 42;
11. function(i);
12. function(42);
13. // не путать с этим
14. // возврат любой ссылки на локальный объект = ошибка!
15. const int& function( ) { …
16. const int& i = 42;
17. return i; // error
19. int&& function( ) { …
20. int&& i = 42;
21. return i; // error

FUNCTION OVERLOAD

Как уже было сказано, основной смысл T&& ссылок – реализация идеальной передачи и семантики перемещения. Об идеальной передаче будет позже. Сейчас на примере обычной перегрузки подготовим предварительное объяснение семантики перемещения.

Суть использования T&& ссылок в том, что можно перегрузить конструктор класса (ctor) и оператор присваивания (operator=) таким образом, чтобы их действия отличалисль пре передаче в качестве параметра l-value и r-value.

1. int i ;
2. int& ir = i ;
3. const int j = 42 ;
4. const int& jr = j ;
5. int&& k = 42 ;
6. const int&& l = 42 ;
7. **void \_f(T&&) { …** **}**
9. \_f(42); // **ок** (r-value)
10. \_f(i), (ir), (k), (j), (jr), (l); // **error** (any l-value)
11. **void \_f(T& ) { …** **}**
13. \_f(i), (ir), (k); // **ок** (non-const l-value)
14. \_f(42), (j), (jr), (l); // **error** (const l-value, r-value)

1. **void \_f(const T&)** **{ …** **}**
3. // **ок** for everything (any l-value and r-value)
4. \_f(i), (ir), (k), (42), (j), (jr), (l);

1. // overload
2. **void \_f(T&) { 1** **}**
3. **void \_f(const T&) { 2** **}**
4. **void \_f(T&&) { 3** **}**
6. \_f(i), (ir), (k); // 1
7. \_f(j), (jr), (l); // 2
8. \_f(42); // 3

**STD::MOVE( )**

#include <utility>

Функция стандартной библиотеки, которая принимает в качестве параметра l-value объект и трансформирует его в r-value:

1. void \_f(int& ) { 1 }
2. void \_f(int&&) { 2 }
4. int i = 42;
5. \_f(i); // 1
6. \_f(std::move(i)); // 2

std::move( ) turns its argument into an rvalue even if it isn't and it achieves that by hiding the name (см. выше: Является ли r-value reference сама по себе r-value значением?).

РЕАЛИЗАЦИЯ STD::MOVE

1. template<typename S>
2. typename std::remove\_reference<S>::type&&
3. move(S&& arg)
4. {
5. typedef typename std::remove\_reference<S>::type&& RV;
6. return static\_cast<RV>(arg);
7. }

ПРОВЕРКА РАБОТЫ

std::move(l-value) ⇨ S = T&

1. T&& move(T& arg) // T& && = T& arg
2. {
3. return static\_cast<T&&>(arg);
4. }

std::move(r-value) ⇨ S = T&&

1. T&& move(T&& arg) // T&& && = T&& arg
2. {
3. return static\_cast<T&&>(arg);
4. }

Обратить внимание, что нет разницы между (S&& arg) и (S& arg) если заменить первое на второе, - так как тип возвращаемого значения все равно кастуется в стейтменте с return.

**REFERENCE COLLAPSING RULE**

В языке С++ запрещены ссылки на ссылки.

Однако возможны ситуации с

- template’ми

- typedef’ми

когда рядом оказываются несколько символов &

В данном случае T = S&, следовательно, при подстановке получится T& = S&&. Инстнацировать это нельзя, так как

- ссылка на ссылку запрещена

- r-value ссылка тут вообще ни при чем

1. template<typaname T>
2. void \_f (T& arg) { … }
4. const S& ref = S( );
5. \_f(ref);

Аналогично здесь

1. typedef T& ref;
2. ref& var = … ; // ref& = T&& , error

Для таких случаев существует правило схлопывания ссылок

(можно думать как о логическом ИЛИ, где & = 1, && = 0)

& + & = &

& + && = &

&& + & = &

&& + && = &&

**SPECIAL TEMPLATE DEDUCTION**

Есть ситуация, в которой T&& НЕ означает r-value ссылку

Это относится к выведению шаблонного типа

1. template<typename T>
2. void \_f(T&& arg) { … }
4. T = arg =
5. \_f(\*l-value\*); ⇨ T& T& + && ⇨ T& (lv ref)
6. \_f(\*r-value\*); ⇨ T T + && ⇨ T&& (rv ref)
8. int i; \_f(i); // \_f(int& arg)
9. \_f(42); // \_f(int&& arg)

**STD::FORWARD + PERFECT FORWARDING**

#include <utility>

Представим ситуацию, в которой некоторый объект сначала передается в одну функцию, а затем этой функцией должен быть снова передан далее по цепочке.

1. void \_f1( ) { …
2. int i;
3. \_f2(i); // l-value = A
4. \_f2(42); // r-value = тоже A!
6. template<typename T>
7. void \_f2(T&& arg) { … // arg = всегда l-value
8. \_f3(arg); // как исправить строчку?
9. void \_f3(int& ) { A }
10. void \_f3(int&&) { B }

Цепочка выглядит как \_f1 ⇨ \_f2 ⇨ \_f3.

Проблема в функции \_f2, так как какое бы значение ни было передано при ее вызове (l-value или r-value), этим значением будет инициализирован аргумент arg. А сам arg является l-value просто по определению.

Задача идеально передачи в том, чтобы

1. всегда передавать по ссылке, избежав передачи по значению

2. принимать и передавать далее по цепочке идентичный тип:

l-value ⇨ l-value

r-value ⇨ r-value

Для этой цели служит функция std::forward< >( )

С помощью нее строка 8 должна быть переписана так:

8: \_f3(std::forward<T>(arg));

Теперь все будет работать правильно.

РЕАЛИЗАЦИЯ STD::FORWARD

1. template<typename S>
2. S&& forward(typename std::remove\_reference<S>::type& arg)
3. {
4. return static\_cast<S&&>(arg);
5. }

Обратить внимание на то, что std::remove\_reference<S>::type& в реализации std::forward необязательно – теоретически можно написать S& и по правилу схлопывания ссылок получить тот же T&. Однако это добвалено для того, чтобы заставить программиста в угловых скобках явно указывать тип передаваемого аргумента.

ПРОВЕРКА РАБОТЫ

\_f2(l-value) ⇨ T = T& ⇨ std::forward<T&>(arg) ⇨ S = T&

1. T& && forward(T& arg)
2. {
3. return static\_cast<T& &&>(arg);
4. }
5. T& forward(T& arg)
6. {
7. return arg;
8. }

\_f2(r-value) ⇨ T = T ⇨ std::forward<T>(arg) ⇨ S = T

1. T&& forward(T& arg)
2. {
3. return static\_cast<T&&>(arg);
4. }

**MEMORY LEAKING**

ПРОБЛЕМА: утечка памяти, если

забыть освободить память (строка 3)

функция будет завершена досрочно (строка 2)

1. type\* ptr = new type( );

2. … return 0;

3. delete ptr;

ПРЯМОЛИНЕЙНОЕ РЕШЕНИЕ: создание умного указателя. Суть:

- создать класс, полем которого будет указатель

- внутри деструктора класса выполнять возврат памяти указателя

Так как при выходе из области видимости всегда вызывается деструктор класса, это гарантирует возврат динамически выделенной памяти

1. template <class T>
2. class POINTER {…
3. private: T\* m\_ptr;
4. public: POINTER(T\* ptr) : m\_ptr {ptr} { }
5. T& operator\* ( ) { return \*m\_ptr; }
6. T\* operator-> ( ) { return m\_ptr; }
7. POINTER(POINTER& obj) {
8. m\_ptr = obj.m\_ptr;
9. obj.m\_ptr = nullptr;
10. }
11. POINTER& operator= (POINTER& obj) {
12. if(&obj == this) return \*this;
13. delete m\_ptr;
14. m\_ptr = obj.m\_ptr;
15. obj.m\_ptr = nullptr;
16. return \*this;
17. }
18. ~POINTER( ) { delete m\_ptr; }

Комментарий:

3. указатель

5, 6. переопределение операторов = класса как с указатель

7. **семантика перемещения** для конструктора копирования

11. **семантика перемещения** для оператора присваивания

18. деструктор, освобождающий динамически выделенную память

ПРОБЛЕМА ПРЯМОЛИНЕЙНОГО РЕШЕНИЯ:

1. POINTER<int> ptr1 = new int(1);
2. POINTER<int> ptr2 (ptr1);
3. POINTER<int> ptr3; ptr3 = ptr2;
4. std::cout << \*ptr3;

Здесь все хорошо

1 строка – динамическое выделение памяти

2 и 3 – реализация копирования и присваивания

4 строка – работа с объектом класса как с указателем (1)

1. funct(ptr3); // если – void funct(Pointer obj) { … }
2. POINTER<int> ptr4 = new int[10];

Здесь будут проблемы

5 строка – при передаче ptr3 в функцию по значению, будет реализована семантика перемещения от ptr3 к obj. Следовательно сам ptr3 станет нулевым указателем. Кроме того, объект, на который он ссылался, будет удален, т.к. по завершении funct( ) объект obj уничтожается, возвращая динамически выделенную память в систему

6 строка – так как динамически была выделена память под массив, то для корректного еу освобождения, необходимо реализовывать деструктор, в котором вместо delete m\_ptr было бы delete[ ] m\_ptr. А в данном случае возврат памяти будет осуществлен некорректно

**УМНЫЕ УКАЗАТЕЛИ в С++11**

#include <memory>

std::scoped\_ptr;

std::unique\_ptr;

std::weak\_ptr;

std::shared\_ptr.

МОДИФИКАЦИЯ КЛАССА POINTER с использованием ССЫЛОК R-VALUE

1. template <class T>

2. class POINTER {…

3. private: T\* m\_ptr;

4. public: POINTER(T\* ptr) : m\_ptr {ptr} { }

5. T& operator\* ( ) { return \*m\_ptr; }

6. T\* operator-> ( ) { return m\_ptr; }

7.

8. // конструктор копированием для l-value

9. POINTER (const POINTER& obj)

10. : m\_ptr { obj.m\_ptr } { }

11.

12. // оператор присваивания копированием для l-value

13. POINTER& operator= (const POINTER& obj) {

14. if(&obj == \*this) return \*this;

15. delete m\_ptr;

16. m\_ptr = new T;

17. \*m\_ptr = (\*obj).m\_ptr;

18. return \*this;

19. }

20.

21. // конструктор перемещением для r-value

22. POINTER(POINTER&& obj) noexcept {

23. m\_ptr = obj.m\_ptr;

24. obj.m\_ptr = nullptr;

25. }

26.

27. // оператор присваивания перемещением для r-value

28. PONTER& operator= (POINTER&& obj) noexcept {

29. if(&obj == \*this) return \*this;

30. delete m\_ptr;

31. m\_ptr = obj.m\_ptr;

32. obj.m\_ptr = nullptr;

33. return \*this;

34. }

35.

36. // деструктор класса

37. ~POINTER( ) { delete m\_ptr; }

Запретить возможность инициализации/присваивания копированием: POINTER (const POINTER& obj) = delete;

POINTER& operator= (const POINTER& obj) = delete;

КОНСТРУКТОР ПЕРЕМЕЩЕНИЯ, ОПЕРАТОР ПРИСВАИВАНИЯ ПЕРЕМЕЩЕНИЕМ:

не выбрасывать исключений, иметь пометку noexcept

**STD::UNIQUE\_PTR< >**

Используется для управления ДИНАМИЧЕСКИ ВЫДЕЛЕННЫМИ ОБЪЕКТАМИ

Особенности:

должен единолично владеть объектом (unique\_ptr = объект)

использует ТОЛЬКО СЕМАНТИКУ ПЕРЕМЕЩЕНИЯ

неявно преобразуется в bool при проверке

умеет управлять динамическим массивом, но лучше std::vector

1. #include <memory>
2. std::unique\_ptr<type> single\_ptr (new type());
3. std::unique\_ptr<type[]> array\_ptr (new type[]);
4. std::unique\_ptr<type> u\_ptr (std::move(single\_ptr));
5. = std::move(single\_ptr);
6. type\* ptr\_a = new type;
7. std::unique\_ptr<type> ptr\_b (ptr\_a);
8. std::unique\_ptr<type> ptr\_с = std::move(ptr\_a); // error

ОПИСАНИЕ:

2 выделение памяти для обычного объекта

3 выделение памяти для массива (внимание на type[])

4 инициализация другим объектом unique\_ptr

7 инициализация указателем на динамическую память

8 нельзя, operator= использует только объекты unique\_ptr

Синтаксически допустимо, НО ИМЕЕТ НЕОПРЕДЕЛЕННОЕ ПОВЕДЕНИЕ так как delete внутри класса unique\_ptr будет применен к статической / автоматической памяти

type var { … };

type\* ptr\_0 = &var;

std::unique\_ptr<type> ptr\_1 (ptr\_0);

Синтаксически допустимо, НО ИМЕЕТ НЕОПРЕДЕЛЕННОЕ ПОВЕДЕНИЕ так как

при уничтожении ptr\_1 и ptr\_2 их delete применяется к одной и той же памяти

type\* ptr\_0 = new type;

std::unique\_ptr<type> ptr\_1 (ptr\_0);

std::unique\_ptr<type> ptr\_2 (ptr\_0);

Синтаксически допустимо, НО ИМЕЕТ НЕОПРЕДЕЛЕННОЕ ПОВЕДЕНИЕ так как delete внутри класса unique\_ptr будет применен к удаленной памяти

type\* ptr\_0 = new type;

std::unique\_ptr<type> ptr\_1 (ptr\_0);

delete ptr\_0;

Принудительное освобождение памяти умного указателя: ptr\_1.reset( )

**STD::MAKE\_UNIQUE()**

СОЗДАНИЕ УМНОГО УКАЗАТЕЛЯ unique\_ptr

Чтобы избежать трудностей с инициализацией unique\_ptr указателем или другим объектом unique\_ptr + упростить код используется шаблон функции make\_unique. То есть функция make\_unique( ) создает объект unique\_ptr.

1. #include <memory>
2. auto ptr = std::make\_unique<type>(arguments);
3. или std::unique\_ptr<type> ptr (new type(arguments));
5. // для массива type[42]
6. auto ptr = std::make\_unique<type[ ]>(42)

ВОЗВРАТ ИЗ ФУНКЦИИ

всегда ПО ЗНАЧЕНИЮ

никогда по указателю

в исключительных случаях по ссылке

1. std::unique\_ptr<type> funct\_out( ) { …
2. return std::make\_unique<type>(arguments);
3. int main( ) { …
4. auto ptr = funct\_return( );

ПЕРЕДАЧА В ФУНКЦИЮ

по значению, если нужно передать владение

по ссылке, если нужно только предоставить доступ

используя метод .get() для прямого доступа к объекту указателя

1. void funct\_in(std::unique\_ptr ptr) {…
2. int main( ) {…
3. auto ptr = std::make\_unique<type>(parameters);
4. funct\_in(std::move(ptr));
5. void funct\_in(const std::unique\_ptr &ptr) {…
6. int main( ) { …
7. auto ptr = std::make\_unique<type>(parameters);
8. funct\_in(ptr);
9. void funct\_in(type\* ptr) {…
10. int main( ) { …
11. auto ptr = std::make\_unique<type>(parameters);
12. funct\_in(ptr.get());

**STD::SHARED\_PTR< >**

Счетчик реализован как АТОМАРНАЯ переменная = замедление!

Используется для управления ДИНАМИЧЕСКИ ВЫДЕЛЕННЫМИ ОБЪЕКТАМИ

Особенности:

совместное владение объектом с другими shared\_ptr

иcпользует семантику копирования. Для корректной работы, 2 и далее указатели shared\_ptr должны быть инициализированы с помощью уже существующего shared\_ptr

1. #include <memory>
2. std::shared\_ptr<type> ptr\_1 (new type());
3. std::shared\_ptr<type> ptr\_2 (ptr\_1);
4. type\* ptr\_0 = new type();
5. std::shared\_ptr<type> ptr\_1 (ptr\_0);
6. std::shared\_ptr<type> ptr\_2 (ptr\_1);

**STD::MAKE\_SHARED()**

СОЗДАНИЕ УМНОГО УКАЗАТЕЛЯ shared\_ptr

Чтобы избежать трудностей с инициализацией shared\_ptr указателем или другим объектом shared\_ptr + упростить код используется шаблон функции make\_shared. То есть функция make\_unique( ) создает объект shared\_ptr.

1. std::shared\_ptr<type> ptr (new type(arguments));
3. // эквивалентно
4. auto ptr\_1 = std::make\_shared<type>(arguments);
5. auto ptr\_2 = ptr\_1;

MAKE\_SHARED() и МАССИВЫ

можно: std::shared\_ptr<type[]> ptr (new type[]);

нельзя: auto ptr = std::make\_shared<type[]>();

SHARED\_PTR и UNIQUE\_PTR

можно: std::unique\_ptr<type> ptr\_u (new type());

std::shared\_ptr<type> ptr\_s (std::move(ptr\_u));

теперь ptr\_u.get() == nullptr;

нельзя: безопасно трансформировать обратно

**STD::WEAK\_PTR<>**

СОЗДАНИЕ УМНОГО УКАЗАТЕЛЯ weak\_ptr

Используется для управления ДОСТУПОМ к объекту,

которым владеет shared\_ptr. Особенности:

- не владеет объектом (не учитывается в счетчике shared\_ptr)

- инициализация только через shared\_ptr

- при уничтожении weak\_ptr объект не уничтожается

1. #include <memory>
2. std::shared\_ptr<type> ptr\_s\_1 (new type());
3. std::weak\_ptr<type> ptr\_w (ptr\_s);
4. std::shared\_ptr<type> ptr\_s\_1 (new type());
5. std::weak\_ptr<type> ptr\_w;
6. ptr\_w = ptr\_s;

ИСПОЛЬЗОВАНИЕ – только через преобразование в объект shared\_ptr

либо std::shared\_ptr<type> ptr\_s\_2 (ptr\_w);

либо std::shared\_ptr<type> ptr\_s\_2 = ptr\_w.lock();

**СИНОПСИС**

ДИНАМИЧЕСКОЕ ВЫДЕЛЕНИЕ ПАМЯТИ

ПРОБЛЕМА: нужно следить за выделенной памятью

1. type\* ptr = new type();
2. delete ptr;

САМОПИСНЫЙ УМНЫЙ УКАЗАТЕЛЬ = ФОКУС ВНИМАНИЯ ВНУТРИ КЛАССА

ПРЕИМУЩЕСТВА:

- всегда вызывается деструктор – память точно освободится

- реализовано глубокое копирование

1. template <class T>
2. class POINTER { …
3. private: T\* m\_ptr;
4. public: POINTER (T\* ptr = nullptr) : m\_ptr { ptr } { }
5. \*... перегрузка operator\* .....................\*
6. \*... перегрузка operator-> ....................\*
7. \*... глубокое копирование конструктором .......\*
8. \*... глубокое копирование через operator= .....\*
9. ~ POINTER ( ) { delete m\_ptr; }
11. POINTER<type> ptr (new type( ));

ТЕПЕРЬ ВМЕСТО УКАЗАТЕЛЯ ИСПОЛЬЗУЕТСЯ ОБЪЕКТ КЛАССА POINTER

ПРОБЛЕМА: производительность при анонимных объектах POINTER

1. POINTER<type> funct (POINTER<type> f\_pointer) { … }
2. POINTER<type> ptr = funct(POINTER<type>(\*arguments\*));

СЕМАНТИКА ПЕРЕМЕЩЕНИЯ = ПЕРЕДАЧА ВЛАДЕНИЯ вместо КОПИРОВАНИЯ

ИСПОЛЬЗОВАНИЕ ССЫЛОК R-VALUE

1. class POINTER { дополнительно:
2. \*... конструктор, семантика перемещения .......\*
3. \*... operator= , семантика перемещения .......\*

ЕСТЬ ВОЗМОЖНОСТЬ:

отключить конструктор и operator= копированием = delete

использовать POINTER l-value как r-value, если нужно std::move( )

ИТОГ: стандартные классы с реализованными решениями:

unique\_ptr, shared\_ptr, week\_ptr

**ITERATORS**

*more about:*

**BASIC INFO**

Итераторы обеспечивают доступ к элементам контейнера. Компилятор испоьзует принцип утиной типизации, чтобы определить итератор. Простейший состав итератора следующий:

1. // собственно контейнер
2. template<typename T>
3. class CONTAINER
4. {
5. public:
6. class ITERATOR
7. {
8. T regular\_element; // очередной элемент
9. public:
10. \* перегрузка операторов
11. \* для доступа к элементам, например:
12. T& operator++ ( ) { return … }
14. };
16. ITERATOR begin ( ) { … } // первый элемент
17. ITERATOR end ( ) { … } // после последнего элемент
18. };

Если operator== позволяет сравнить текущий итератор с вовзращаемым из метода end( ) так, что последний вовсе не обязателен, допустимо:

bool operator== (std::default\_sentinel\_t) { … }

std::default\_sentinel\_t end( ) { return { }; }

CONST\_ITERATOR реализован с использованием std::conditional таким образом, что можно получить также и обычный оператор.

template<bool IsConst>

class ITERATOR

{

std::conditional<IsConst, const T, T> regular element;

std::conditional<IsConst, const T&, T&> operator++( );

\*и так далее\*

}

using iter = CONTAINER<T>::ITERATOR<true>

using citer = CONTAINER<T>::ITERATOR<false>

**ITERATOR / ADRESS INVALIDATION**

В каждом из стандартных контейнеров реализованы итераторы. Некоторые стандартные контейнеры устроены таким образом, что при манипуляции с ними могут возникнуть ситуации, когда итератор становится невалидным:

iterator element I

\*do\_something\_with\_container\*

iterator element X

Инвалидация ссылки / указателя имеет похожий смысл. Если получить адрес какого-то элемента контейнера, а затем произвести с контейнером какие-либо действия, адрес может стать невалидным:

adress element I

\*do\_something\_with\_container\*

adress random value in memory

ТАБЛИЦА ИНВАЛИДАЦИЙ ДЛЯ КОНТАЙНЕРОВ

iterator adress

std::vector **⤬** **⤬**

std::deque **⤬** **ok**

std::list **ok ok**

std::map **ok ok**

std::unordered\_map **⤬ ok**

std::set

**⤬** = может произойти инвалидация

**ok** = всегда остается валидным

**HIERARCHY of ITERATORS CATEGORIES**

INPUT\_ITERATOR ровно один проход

операция инкримента

операция сравнения на равенство

разыменование только для чтения (rvalue)

swapable

OUTPUT\_ITERATOR ровно один проход

операция инкримента

разыменование только для записи (value)

swapable

FORWARD\_ITERATOR предыдущие + дополнительно

неограниченное количество проходов

использование: std::forward\_list

std::unordered\_map std::unordered\_set

BIDIRACTIONAL\_ITERATOR предыдущий + дополнительно

операция дикремента

использование: std::list

std::map

std::set

RANDOM\_ACCESS\_ITERATOR предыдущий + дополнительно

операции (it += n) и (it -= n)

операции сравнения любые

операция разности итераторов (it2 – it1)

использование: std::deque

CONTIGUOUS\_ITERATOR предыдущий + дополнительно

гарантия, что объекты под итератором последовательно в памяти

по сути эквивалентен указателю

использование: std::vector

std::array

**INSERT\_ITERATOR**

По своей сути output\_iterator = итератор, который предоставляет к элементам доступ "только для записи". Любой итератор стандартного контейнера позволяет записывать в контейнер, однако существует проблема контроля выхода за границы контейнера.

Например:

1. std::vector<int> v\_1 {0, 1, 2, 3, 4}; // v\_1.size() = 5
2. std::vector<int> v\_2 (3); // v\_2.size() = 0
4. std::copy(v\_1.begin( ), v\_1.end( ), \*итератор v\_2\*);

Если подать третьим аргументом обыкновенный итератор (например, v\_2.begin( )), возникнет ошибка, так как размер v\_1 > v\_2.

Первый вариант решения: убеждаться в том, что размер контейнера- назначения позволяет вместить данные из контейнера-источника.

Второй вариант решения заключается в том, чтобы заменить операцию присваивания нового значения конкретногому элементу контейнера назначения v\_2 на операцию добавления нового элемента в этот контейнер:

вместо условного v\_2[4] = 3; // ошибка

сделать, например v\_2.insert(v\_2.end( ), 3); // ок

Для этого и существует std::insert\_iterator (пример его внутренней реализации находится на следующей странице) плюс вспомогательная функция, которая сама выводит тип контейнера.

std::insert\_iterator<\*container\_type\*>(\*container\*, \*iterator\*)

std::inserter(\*container\*, \*iterator\*)

- тип контейнера в качестве шаблонного параметра

- первый аргумент - сам контейнер

- второй контейнер - итератор места для вставки

Частными случаями std::insert\_operator являются:

std::back\_insert\_iterator<\*container\_type\*>(\*container\*)

std::back\_inserter(\*container\*)

std::front\_insert\_iterator<\*container\_type\*>(\*container\*)

std::front\_inserter(\*container\*)

Алгоритм последовательного добавления элементов с помощью итератора состоит из трех шагов:

- сдвинуть итератор

- разыменовать итератор для получения доступа к элементу

- присвоить элементу новое значение

И именно такие операции выполняет приведенный в изначальном примере std::copy. Более того, именно по перегрузке соответствующих операторов компилятор может понять, что какая-то сущность является итератором. Однако, как уже выяснили, при вставке нельзя ни двигать итератор по уже существующим элементам, ни писать в них.

Поэтому реализация insert\_iterator содержит соответствующие перегрузки, но она отличается от перегрузок в обычных итераторах (в приведенной ниже реализации нет прописан конструктор):

1. template<typename CONTAINER>
2. struct insert\_operator
3. {
4. // псевдонимы для удобства и необходимые поля
5. using ii = insert\_iterator<CONTAINER>;
6. using ct = typename CONTAINER::value\_type;
7. using ci = typename CONTAINER::iterator;
8. CONTAINER& container;
9. ci iter;
10. // ничего никуда не смещается
11. // ничего не разыменовывается
12. ii& operator++ ( ) { return this\*; }
13. ii& operator\* ( ) { return this\*; }
15. // вместо присвоения – вставка нового элемента
16. ct& operator= (ct& value, ci& iter)
17. {
18. container.insert(value, iter);
19. ++iter;
20. return \*this;
21. }
22. };
24. // изменяем изначальный пример
25. std::copy(v\_1.begin( ), v\_1.end( ),
26. std::insert\_operator<vector<int>>(v\_2, v\_2.begin( )) );

**<ITERATOR\_TRAITS>**

Заголовочный файл <itarator\_traits> содержит одноименную структуру

template<typename Iterator>

struct iterator\_traits { };

которая служит для поучения информации о итераторе. Возможная информация будет представлена ниже. Чтобы получить доступ к ней, нужно использовать:

std::iterator\_traits<Iterator>::\*info\_name\*

::ITERATOR\_CATEGORY( )

Для различения категорий итераторов, когда это необходимо, существует набо соответствующих тэгов = пустых структур.

struct std::input\_iterator\_tag { };

struct std::forward\_iterator\_tag { };

struct std::bidiractional\_iterator\_tag { };

struct std::random\_access\_iterator\_tag { };

struct std::contiguous\_iterator\_tag { };

Категория итератора выводится компилятором автоматически исходя из того, определены для итератора соответствующие операции.

**TEMPLATES**

*more about:* [шаблонная фабрика объектов](https://habr.com/ru/post/229993/)

[шаблоны переменных](https://ru.stackoverflow.com/questions/737944/Шаблоны-переменных-в-c14)

[спецификаторы и квалификаторы в шабонах](https://habr.com/ru/company/jugru/blog/506104/)

[template<auto var> + deduction guide](https://habr.com/ru/company/pvs-studio/blog/340014/)

metaprogramming [link](https://habr.com/ru/post/218229/), [link](https://habr.com/ru/post/218341/), [link](https://habr.com/ru/post/337590/)

**FUNCTION TEMPLATES**

1. template <typename T1, typename T2 ← >
2. const T1& \_funct (const T2 &par, const T1 &par ← )
3. { … }
4. int \_funct<int, double ← > (3.7, 1 ← ) { return … }
5. int \_funct(1, 3.7) { return … }

DESCRIPTION

1: ключевые слова: template <typename> (или <class> - разницы нет)

2: определение функции идет сразу же после объявления шаблона

типичное написание:

* + параметр шаблона буквой T либо Type
  + имя шаблонной функции начинать с нижнего подчеркивания

2, 5:

параметров шаблона может быть несколько и

они могут быть указаны в любой последовательности, но:

* в параметрах функции ← соответствие передаваемым аргументам
* в параметрах шаблона ← соответствие самой последовательности

5: передача типов явным образом

6: нет передачи типов = автоматическое выведение типа

NON\_TYPE PARAMETERS

1. template <typename T, int N>
2. void \_funct(T& var) { …
3. for(int i = 0; i < N; ++i) ++var;

T = typename / class parameter

N = non-type parameter

В таком случае обязательна явная передача:

1. int z = 0;
2. \_funct<int, 42>(z);

**CLASS TEMPLATES**

1. template <typename T, int N>
2. class Name { …
3. private: T m\_var;
4. int n = N;
5. public:
6. T \_method\_A ( … ) { … }
7. void \_method\_B ( … );
8. template <typename T>
9. T Name<T>::\_method\_B ( … ) { B }
11. Name<char, 42> obj;

DESCRIPRION

3, 8-9:

Метод, определяемый вне класса, требует своего шаблона (\_method\_B ←)

12: Обязательна явная передача параметров (даже если нет non-type)

**TEMPLATES INSTANTIATION**

Шаблоны являются инструментами compile-time.

Однако сам по себе шаблон не компилируется напрямую.

Определение шаблона представляет из себя инструкцию для компилятора, по какому образцу создать непосредственно рабочую функцию или класс.

ЭКЗЕМПЛЯР = конкретная рабочая единица на основе шаблона

ИНСТАНЦИРОВАНИЕ = процесс создания экземпляра по шаблонному образцу

Инстанцированные с разными параметрами экземпляры класса представляют из себя РАЗНЫЕ КЛАССЫ

EXPLICIT INSTANTIATION (явное)

1. template
2. void \_funct<char>(char&);
4. template<typename T>
5. class Name { …
6. void \_method(T& var) { … }
8. template
9. class Name<int>;
11. template
12. void Name<double>::\_method<T>( );

7-8, 14-15, 17-18 явное инстанцирование

Явное инстанцирование метода класса (17-18) имеет смысл (и возможно) только в том случае, если нет явного инстанцирования класса для того же типа данных (в строках 14-15 тип данных другой)

IMPLICIT INSTANTIATION (неявное)

1. template<typename T>
2. void \_funct(T& var) { … };
4. int main( ) { …
5. int i = 0;
6. \_funct<int>(i)

6: неявное инстанцирование. Компилятор:

видит тип параметра ⇨ создает экземпляр

IF CONSTEXPR

Для того, чтобы определить некоторое условие, от выполнения которого зависит состав инстанциации шаблона, можно использовать if constexpr ( )

1. template<int N>
2. void function( )
3. {
4. if constexpr (N > 0) function<N - 1>( );
5. cout << N << endl;
6. }
8. function<3>( ); // 0 1 2 3

При рекурсивной инстанциации все экзепляры, для которых N > 0 будут содержать в своем теле помимо вывода числа обращение к очередному шаблону. И, следовательно продолжение рекурсии.

И только последний шаблон при N = 0 будет содержать только вывод числа, так как 0 > 0 = false и поэтому function<-1>( ); не будет включено в инстанциацию.

Таким образом мы обозначили дно рекурсии.

ШАБЛОНЫ и СТАТИЧЕСКИЕ БИБЛИОТЕКИ

При попытке разделить шаблонный класс (функцию) на header и .cpp файл возникнет проблема, которая заключается в следующем: в момент компиляции только объявление класса (прототип) функции будет помещен препроцессором из заголовочного файла непосредственно в main.cpp.

Когда настанет этап компиляции main.cpp и компилятор будет создавать экземпляры шаблонов, он сможет инстанцировать только экземпляры объявлений (прототипов) так как определения классов (функций) содержаться в .cpp и станут доступны лишь на этапе линковки.

В то же время, когда компилятор будет компилировать .cpp файл библиотеки, он никак не сможет неявно инстанцировать определения классов (функций), так как необходимые для этого стейтменты находятся в main.cpp.

Выходит, что определния классов (шаблонов) так никогда и не будут инстанцированы. Варианты исправленя:

– явное инстанцирование в .cpp (если типов немного и они известны)

– определение классов (функций) непосредственно в заголовочном файле

Если при написании шаблона появилась необходимость модифицировать определение для случая, когда передается какая-то конкретная комбинация типов, можно использовать:

- ENTIRE SPECIALIZATION функции и классы

- PARTIAL SPECIALIZATION только классы

- TEMPLATE OVERLOADING только функции

**EXPLICIT (явная) TEMPLATES SPICIALIZATION**

Каждая специализация шаблона класса является НОВЫМ ШАБЛОНОМ

Одинаковое имя шаблонов = указание компилятору на специализацию

Специализация – как и инстанцирование – может быть неявная (выводится из типов при вызове шаблонной функции \ объявлении объекта шаблонного класса) и явная (специально задается программистом).

Преимущество явной специализации в том, что можно изменить определение функции / класса (даже добавить новый метод)

ОБЩИЙ СИНТАКИС СПЕИАЛИЗАЦИИ:

1. template<typename T, typename S>
2. class NAME { };
4. template< **\*something\*** >
5. class NAME< **\*something\*** > { };

Правило 1: есть в **\*something\*** - должен быть в **\*something\***

Правило 2: нет в **\*something\*** - не должен быть в **\*something\***

Правило 3: есть в **\*something\*** - можеть не быть в **\*something\***

ENTIRE (полная) SPECIALIZATION

1. template<class T, int size> // класс
2. class Name { …
3. public: T m\_array[size];
4. template<typename T, int size> // функция
5. void print(Name<T, size> &name)
6. {
7. for(int i = 0; i < size; i++)
8. std::cout << name.m\_array[i] << “ “;
9. }

Если необходимо полностью специализировать шаблон,

то все его параметры должны быть явно определены:

- как typename

- так и non-type !

Ограничение на специализацию класса: нельзя добавить новый метод \ поле

Ограничение на специализацию функцию: доступна толко полная специализация

1. // класс
2. template<> <- синтаксис полной специализации
3. class Name<int, 42> { …
4. public: int m\_array[42];
6. // функция
7. template<>
8. void print(Name<char, 10> &name)
9. {
10. for(int i = 0; i < 10; i++)
11. std::cout << name.m\_array[i];
12. }

ПОЛНАЯ СПЕЦИАЛИЗАЦИЯ (только) МЕТОДА КЛАССА (без всего класса)

1. template<typename T, int size>
2. class CLASS { …
3. public:
4. T method( )
5. {
6. std::cout << size << std::endl;
7. }
9. template<>
10. char CLASS<char, 1>::method( )
11. {
12. std::cout << “another\_method” <<std::endl;
13. }

PARTIAL (частичная) SPECIALIZATION

Отдельная специализация = отдельный класс

1. template<typename T1, typename T2>
2. class CLASS { …
4. template<typename T1>
5. class CLASS<T1, char> { …

НЮАНС С ЧАСТИЧНОЙ И ПОЛНОЙ СПЕЦИАЛИЗАЦИЕЙ

1. template<typename T> void f1(T t); // (1)
2. template<typename T> void f1(T\* t); // (2)
3. template< > void f1(int\* t); // (3)
5. template<typename T> void f2(T t); // (4)
6. template< > void f2(int\* t); // (5)
7. template<typename T> void f2(T\* t); // (6)
9. int i;
10. f1(&i); // 3
11. f2(&i); // 6

Порядок выбора между шаблонами:

1. выбирается наиболее подходящий из базовых шаблонов

2. проверяется, есть ли у выбранного еще и подходящая

специализация – если есть, выбирает она

Нюанс в том, что специализация относится к тому базовому шаблону, непосредственно после которого она идет. То есть:

1. для функции f1 ситуация: (1), (2) –> (3)

2. для функции f2 ситуация: (4) -> (5), (6)

Соответственно, при вызове:

1. f1

из базовых (1) и (2) выбрана (2)

у (2) есть специализация (3)

2. f2

из базовых (4) и (6) выбрана (6)

у (6) нет никакой специализации

**FUNCTION TEMPLATE SPECIALIZATION**

Как уже было сказано, для функции доступна только полная специализация шаблона. Так как метода класса также является функцией, то для него также доступна только полная специализация. Частичная специализация – запрещена.

ЧАСТИЧНАЯ СПЕЦИАЛИЗАЦИЯ ШАБЛОНА ФУНКЦИИ ЗАПРЕЩЕНА

1. template <typename A, typename B>
2. void function (const A &a, const B &b) { … }
3. template <typename A, char> **= нельзя**
4. void function (const A &a, const char &b) { … }

ЧАСТИЧНАЯ СПЕЦИАЛИЗАЦИЯ МЕТОДА КЛАССА ЗАПРЕЩЕНА

1. template<typename T, int size>
2. class Name { …
3. T method( ) { }
5. template<int size>
6. char Name<char, size>::method( ) **= нельзя**

Однако есть возможность сымитировать:

частичную специализацию шаблона функции = TEMPLATE OVERLOAD

частичную специализацию метода класса =

OUTER TEMPLATE “METHOD” (friend function)

or INHERITANCE

TEMPLATE OVERLOAD

1. template <typename A, typename B>
2. void function (const A &a, const B &b)
3. { BASE\_TEMPL }
4. template <typename A>
5. void function (const A &a, const char &b)
6. { OVER\_TEMPL }
7. void function (const int &a, const char &b)
8. { NONE\_TEMPL }
10. function(3.3, 0 ) - int, double == BASE\_TEMPL
11. function(3.3, ‘a’) - double, char == OVER\_TEMPL
12. function( 0, ‘a’) - int, char == NONE\_TEMPL

С помощью перегрузки шаблонов, можно добиться имитации частичной перегрузки шаблона функции. Порядок работы компилятора при генерации кода:

СНАЧАЛА = поиск НЕ\_шаблонной функции (строки 9 + 13)

ЗАТЕМ = шабонная функция, с НЕ\_шаблонными параметрами (6 + 14)

ПОСЛЕДНЕЕ = поиск полностью ШАБЛОННОЙ функции (1 + 15)

TEMPLATE OVERLOAD = только для функций

PARTIAL SPECIALIZATION = только для классов

OUTER TEMPLATE ‘METHOD’ = friend function

По своему существу метод ничем не отличается от обыкновенной функции. Единственная их разница в том, что метод принимает в качестве параметра скрытый указатель \*this на объект того класса, с которым вызывается.

Соответственно, если есть необходимость сделать какой-то частично специализированный метод, можно вместо него сделать шаблонную дружественную функцию и затем явно передавать ей нужный объект частично специализированного шаблона.

1. template<typename T1, typename T2>
2. class CLASS { …
3. private: int var = 42;
4. public: void method( )
5. {
6. std::cout << “inner, ” << var << std::endl;
7. }
9. template<typename T3>
10. friend void method(Name<T3, int>& obj);

13. template<typename T3>
14. void method(Name<T3, int>& obj);
15. {
16. std::cout << “outer, ” << obj.var << std::endl;
17. }

20. CLASS<char, char> a;
21. a.method( ); // inner: 42
22. method(a); // compile error
23. CLASS<char, int> b;
24. b.method( ) // inner: 42
25. method(b); // outer: 42

21, 24: вызывается внутренний метод

25: вызывается friend функция, поэтому доступно private поле

22: friend функция не может быть вызвана (т.к. строка 20)

9: T3 т.к.

- дублирование внутри класса запрещено

- т.к. все методы являются шаблонными <T1, T2>

- а нам необходимо как-то сделать специализацию

- T3 никак не скажется на работе friend функции

INHERITANCE

1. template<class T1, class T2>
2. class CLASS { …
3. T1 x = … ;
4. virtual void method(T2 y) { cout << x + y << endl; }

Необходимо сделать так, чтобы method(T2) вел себя иначе, если только T1 останется в качетсве параметра шаблона, а вместо T2 будет конкретный тип int.

1. template<class T1>
2. void CLASS<T1, int>::method( ) { … } – НЕ ПОЛУЧИТСЯ

так как это является частичной специализацией функции, что запрещено. Возможное решение – частичная специализация класса (но это плохо, так как придется полностью дублировать даже неизменяемый код)

Правильное решение = наследование + частичная специализация

Сначала нужно полностью унаследоваться от исходного шаблона

template<class T1, class T2>

class CLASS\_modif: public CLASS<T1, T2>

{ \*пустой класс\* };

А затем частично специализировать пустой класс: наследование с тем же именем, но только с одним шаблонным параметром

1. template<class T1> // T1 остается шаблонным параметром
2. class CLASS\_modif<T1, int> : public CLASS<T1, int> { …
3. void method(int y) override
4. { cout << (\*this).x – y << endl; }

Пример работы:

1. CLASS<double, double> obj(40.2);
2. obj.method(1.8); // 40.2 + 1.8 = 42.0
4. CLASS\_modif<double> obj\_m(43.0);
5. obj\_m.method(1); // 43.0 – 1 = 42.0

**DEEP COPYING (TEMPLATE FOR PTR)**

Общий шаблон класса

1. template <typename T>
2. class Name { …
3. private: T m\_var;
4. public:
5. Name (T var) : m\_var = var;
6. ~Name ( ) { }

Шаблон класса для указателей (реализация глубокого копирования)

1. template <typename T>
2. class Name<T\*> { …
3. private: T\* m\_var;
4. public:
5. Name (T\* var) { m\_var = new T {\*var}; }
6. ~Name ( ) { delete m\_var; }

Полная специализация шаблонов конструктора и деструктора конкретно для char\* (строка C-style, а не обыкновенная переменная)

1. template< >
2. Name<char\*>::Name(char\* var)
3. {
4. // определить размер строки
5. int size {0};
6. while (var[size] != ‘\0’) ++size;
8. // для нуль-терминатора
9. ++size;
10. // глубокое копирование
11. m\_var = new char[size];
12. for (int i = 0; i < size; i++) m\_var[i] = var[i];
13. }
15. template< >
16. Name<char\*>::~Name( )
17. { delete[] m\_var; }

**SFINAE**

SFINAE = substitution failure is not an error

Изначально sfinae это о шаблонах функций. Применяется при составлении списка кандидатов (overload set) функций при перегрузке. Основная идея:

если во время подстановки (substitution) параметров шаблона в параметры функции возникает сбой (невозможно подставить), не будет выдана ошибка компиляции – вместо этого шаблон функции просто исключается из списка кандидатов.

1. template<typename T>
2. struct IS\_METHOD
3. {
4. void detect( … ); // ellipsis
6. template<typename S>
7. static decltype(std::declval<S>( ).method(42)) detect(const S&);
8. static constexpr bool value =
9. std::is\_same<int, decltype(detect(std::declval<S>( )))>::value;
10. };

Выше мы проверяем, есть ли в каком-либо типе метод, имеющий имя и сигнатуру method(int) и вовращающий знчение типа int. Когда будет вызвана проверка: IS\_METHOD<T>::value , компилятор:

- сделает попытку вычислить значение value

- для этого ему необходимо определить тип decltype( )

- для этого необходимо инстанцировать detect(S( ))

- будет проверен шаблон <typename S> функции detect( )

- он инстанцируется только если сработает подстановка

- если подставка не сработает, будет выбран void detector( … )

Результат проверки окажется успешным, только если:

- подстановка в <typename S> сработает = есть имя и сигнатура

- decltype( ) в шаблоне вернет тип int = есть возвращаемый тип

Важное замечание! Так как шаблонный тип (typename T) может содержаться так же и в типе возврата функции, то на этапе подстановки (substitution) ТИП ВОЗАРАТА ЯВЛЯЕТСЯ частью СИГНАТУРЫ ФУНКЦИИ.

Однако выбор между инстанцированием того или иного шаблона класса также стал восприниматься как частный случай sfinae.

1. template<typename T>
2. struct IS\_CONST
3. { bool value = false; };
5. template<typename T>
6. struct IS\_CONST<const T>
7. { bool value = true; };
8. bool b1 = IS\_CONST<int>::value; // false
9. bool b2 = IS\_CONST<const int>::value; // true

Подробнее про SFINAE - [link](https://en.cppreference.com/w/cpp/language/sfinae)

**TEMPLATE RECURSION**

Для создания рекурсии шаблона необходимо:

- определить базовый шаблон

- задать правило рекурсии

- определить дно рекурсии (последний шаблон без рекурсии)

Наглядный – бесполезный на практике - пример

ВЫЧИСЛЕНИЕ СУММЫ

1. // базовый шаблон
2. template<int N, int M>
3. struct ADD
4. {
5. // через статическое поле …
6. static const int result = ADD<N – 1, M + 1>::result;
8. // … или через анонимный объект
9. int result = ADD<N – 1, M + 1>( ).result;
10. };
12. // дно рекурсии = специализация
13. template<int N>
14. struct ADD<N, 0>
15. {
16. int result = N;
17. };
18. int result = ADD<5, 8>::result; // 13
19. int result = ADD<5, 8>( ).result;

Полезный пример

СОЗДАНИЕ N-МЕРНОЙ СЕТКИ (n-dimentional grid)

1. template <typename T, size\_t N>
2. class NDGrid
3. {
5. private: std::vector<NDGrid<T, N - 1>> m\_elements;
6. public: static const size\_t DefaultSize {10};
8. explicit NDGrid(size\_t size = DefaultSize) { resize(size); }
9. virtual ~NDGrid( ) = default;
11. NDGrid<T, N-1>& operator[] (size\_t el) { return m\_elements[el]; }
12. void resize(size\_t size)
13. {
14. m\_elements.resize(size);
15. for(auto& el : m\_elements)
16. el.resize(size);
17. }
19. };
20. template<typename T>
21. class NDGrid<T, 1>
22. {
24. public: static const size\_t DefaultSize {10};
25. private:
26. std::vector<T> m\_elements;
27. explicit NDGrid(size\_t size = DefaultSize) { resize(size); }
28. virtual ~NDGrid( ) = default;
30. T& operator[] (size\_t el) { return m\_elements[el]; }
31. void resize(size\_t size) { m\_elements.resize(size); }
33. };
35. NDGrid<int, 3> my3DGrid {4};
36. my3DGrid[1][2][3] = 1;
37. my3DGrid[2][1][0] = 1;
38. cout << my3DGrid[0][0][0] << endl; // 0

**VARIADIC TEMPLATES**

В этом разделе эллипсис ( **...** ) - синтаксис, который обозначает, что работа, так или иначе, ведется с переменным числом аргументов. При этом эллипсис относится к примыкающему к нему слева выражению:

**typename ...** = совокупность шаблонных параметров

**types ...** = совокупность типов

**args ...** = совокупность параметров

Все это будет рассмотренно далее.

PARAMETER\_PACK (пакет типов)

Вариативный шаблон начинается, как и обычный, с template<typename>

1. template <typename **...** types>
2. \*класс или функция\*

**... =** parameter\_pack **-** цепочка 0+ произвольных типов

Parameter\_pack позволяет работать с любым количеством передаваемых параметров шаблона, включая их отсутствие. Задавать нижнюю границу количества параметров шаблона можно, явно прописывая привычные одиночные парамтеры:

1. // нижняя граница теперь = 2 параметра
2. template <typename A, typename B, typename … types>

FUNCTION\_PARAMETER\_PACK (пакет параметров)

1. template <typename ... types>
2. int function(**types ...** **args**)

**types ...** **args –** вся конструкция целиком -function\_parameter\_pack **–** цепочка 0+ произвольных аргументов, стоящих в соответствии с цепочкой произвольных типов **types**.

Благодаря тому, что во время инстанцирования экземпляров шаблонных функций происходит автоопределение типов …

template<typename T>

T funct(T var) { }

funct(42); // инстанцируется -> int funct(int var) { }

… для вариативного шаблона доступен вызов функции с любым количеством аргументов любого типа.

1. {
2. return sizeof...(types); // количество типов
3. return sizeof...(args); // количество аргументов
4. }
6. int var = function(1, ‘A’, 3.14, myclassobj); // var = 4

Конструкцию (types ... args) можно разделить на:

<types ...> = TYPE\_PACK\_EXPANSION

совокупность типов, разворачивающихся в список через запятую

возможна передача в шаблон <typename ... types>

(args ...) = PARAMETER\_PACK\_EXPANSION

совокупность параметров, разворачивающихся в список через запятую

возможна передача в функцию (types ... args)

Пример:

1. // создаем шаблонный класс
2. template <**typename ... types**>
3. struct STRUCT
4. {
5. STRUCT(**types ... args**) { cout << “ctor” << endl; }
6. };
8. // создаем шаблонную функцию
9. template <typename ... types>
10. void function(**types ... args**)
11. { **⇩ ⇩**
12. STRUCT<**types ...**> obj(**args ...**);
13. }
15. // вызываем функцию с любым набором аргументов
16. function(**1, ‘A’, 3.14, myclassobj**); // ctor

Сам по себе пример бессмысленный, так как не присходит никакой реальной обработки переданных в шаблонную функцию (а затем и в шаблонный класс) аргументов. Однако пример нужен для демонстрации употребления синтаксиса и правил совместимости.

Кроме того, можно проследить, как во время компиляции шаблоны класса и функции будут неявно инстанцированы в экземпляры:

void function(**int** **a1**, **char** **a2**, **double** **a3**, **MYCLASS** **a4**)

{

STRUCT<**int, char, double, MYCLASS**> obj(**a1, a2, a3, a4**);

}

RECURSION and SIDE FUNCTION CALLING

Реальный пример – рекурсивное раскручивание function\_parameter\_pack. Для реализации необходимо:

- variadic template, который будет рекурсивно вызывать себя

- template: для инстанцирования при достижении дна рекурсии

1. using namespace std;
3. template<typename T>
4. auto \_double(T var) { return var \* 2; }
6. template<typename T> // дно рекурсии
7. ostream& \_print(ostream& out, const T& next)
8. {
9. out << next;
10. return out;
11. }
13. template<typename T, typename … Types> // рекурсивный шаблон
14. ostream& \_print(ostream& out, const T& next, const Types& ... other)
15. {
16. out << next << “ ” ;
17. return \_print(out, other ...);
18. }
20. template<typename ... Types>
21. void \_doubleprint(ostream& out, const Types& ... args)
22. {
23. \_print(out, \_double(args) ...);
24. }
26. \_print(cout, 1, ‘A’, 3.14, myclassobj);
27. \_doubleprint(cout, 1, 2, 3, myclassobj);

6-11 и 13-18:

Обратить внимание, что объявления идут именно в таком порядке, так как в момент, когда рекурсия вызовов из строки 17 дойдет до своего дна, компилятор должен иметь информацию об этом дне.

17, 14, 7:

(out, **other ...**) ⇨ (ostream& out, **T next, Types ... other**) до тех пор, пока **other ...** содержит более одного элемента. Например, (out, **1**, **2, 3**) ⇨ (ostream& out, **1**, **2, 3**)

⇩

**«откусывание головы».** Как только в parameter\_pack\_expansion останется только один аргумент, будет инстанцирован шаблон, соответствующий дну рекурсии.

3-4, 20-24 и 27:

Еще одна возможность – вызывать функцию для каждого члена parameter\_expansion\_pack поочередно. Например, следующая запись:

(cout, \_double(args) ... ) ⇨ (out, \_double(1), \_double(2), \*итд\*)

**FOLD EXPRESSIONS**

ВЫЗОВ ФУНКЦИИ + ОПЕРАТОР "ЗАПЯТАЯ"

1. template<typename … Types>
2. void \_function(Types … args)
3. {
4. \_f1(args ...); // \_f1(arg1, arg2, arg3);
5. **(**\_f2(args)**,** ...**)**; // \_f2(arg1), \_f2(arg2), \_f3(arg3), …
6. }

ОПЕРАТОРЫ

1. args <op> ... // arg1 + (arg2 + (arg3 + arg4))
2. ... <op> args // ((arg1 + arg2) + arg3) + arg4
3. args <op> ... <op> i // arg1 + (arg2 + (arg3 + i))
4. i <op> ... <op> args // ((i + arg1) + arg2) + arg3

Availible operators list – [link](https://en.cppreference.com/w/cpp/language/fold)

VARIABLE NUMBER OF BASE CLASSES

1. struct S1
2. {
3. int s1;
4. S1(int i) : s1{i} { }
5. };
7. struct S2
8. {
9. int s2;
10. S2(int i) : s2{i} { }
11. };
13. template<typename ... Classes>
14. struct MY : public Classes ...
15. {
16. MY(const Classes& ... objs) : Classes {objs} ... { }
17. virtual ~MY( ) = default;
18. };
20. MY<S1, S2> my{ S1{1}, S2{2} };
21. cout << my.s1 << " " << my.s2 << endl;

**SPECIALIZATION and INHERITANCE and VARIADIC**

Наследование допустимо не только от шаблона с другим идентификатором. Также возможно наследование от шаблона с аналогчным идентификатором при условии:

- производный шаблон = полностью или частично специализирован

- базовый шаблон = соответствовать сигнатуре шаблона

ONE PARAMETER

1. template<typename T>
2. struct NAME { };
4. // partial – one
5. template<typename T>
6. struct NAME<T&> : or NAME<T > // not specialized
7. or NAME<T\* > // partial
8. or NAME<int> // full
10. // full
11. template< >
12. struct NAME<double> : NAME<int> // full only

SOME PARAMETERS

1. template<typename T, typename S>
2. struct NAME { };
4. //partial – two parameters in base template!
5. template<typename T>
6. struct NAME<T, int> : NAME<T, T> // and so one …

ОБРАТИТЬ ВНИМАНИЕ

При наследовании необходимо указывать в базовом шаблонном классе ровно такое же количество параметров, сколько этот шаблон содержит сам по себе (и затем уточнять их, если необходимо)

1. template<typename T, typename S>
2. struct NAME { };

**⤬**

2. template<typename T>
3. struct NAME<T&, int> : NAME<T> … // error

При специализации количество параметров специализируемого шаблона должно быть равным или меньшим количества параметров основного шаблона

**⤬**

1. template<typename T>
2. struct NAME { };
4. template<typename T, typename S> : … // error

VARIADIC

Однако при использовании вариативных шаблонов, в силу их всеядности относительно количества передаваемых аргументов (которых может быть от 0 до N), скомпилируются такие конструкции:

1. template<typename T, typename … S>

**ok**

1. struct NAME { };
3. template<typename T>
4. struct NAME<T&, int> : NAME<S … > { }; // 2 vs 1, ok

**ok**

1. template<typename … T>
2. struct NAME { };
4. template<typename S, typename … T> : … // 1 vs 2, ok

На этой особенности основана возможность реализации кортежа: контейнера, хранящего любое количество переменных любого типа.

**TUPLE – пример реализации через метапрограммирование**

1. template<typename ... Args>
2. struct \_tuple;
4. template<typename Head, typename ... Tail>
5. \_tuple<Head, Tail ...> : \_tuple<Tail ...>
6. {
7. typedef \_tuple<Tail ...> base\_type;
8. base\_type& base = static\_cast<base\_type&>(\*this);
10. typedef Head head\_type;
11. Head head\_;
13. \_tuple(Head h\_, Tail ... args)
14. : \_tuple(args ...), head\_ {h\_}
15. { }
16. };
18. template< >
19. struct \_tuple< >
20. { };
22. template<size\_t Index, typename Head, typename ... Tail>
23. struct \_getter
24. {
25. typedef typename
26. \_getter<Index – 1, Head, Tail ...>::return\_type return\_type;
28. static return\_type get(\_tuple<Head, Tail ...> \_t)
29. { return \_getter<Index – 1, Tail ...>::\_get(\_t); }
30. };
32. template<typename Head, typename ... Tail>
33. struct \_getter<0, Head, Tail ...>
34. {
35. typedef typename
36. \_tuple<Head, Tail ...>::head\_type return\_type;
37. static return\_type get(tuple<Head, Tail ...> \_t)
38. { return \_t.head\_; }
39. };
41. template<size\_t Index, typename Head, typename ... Tail>
42. typename \_getter<Index, Head, Tail ...>::return\_type
43. get(\_tuple<Head, Tail ...> \_t)
44. { return \_getter<Index, Head, Tail ...>::get(\_t): }

Общая логика. Разместить подаваемые на вход значения кортежа в памяти с помощью наследования следующим образом

- имена родительского и всех производных классов одинаковые

- в производном класса есть указатель на базовый класс = base

- одинаковое имя переменной для хранения данных = head\_

[ базовый ][ наследники ]

[head\_ = valueN][ … ][head\_ = value1][head\_ = value0]

**⇧** **⇧**

\_tuple::base:: N ::base::head\_ \_tuple::head\_

1-2: Шаблон, который нужен для организации наследования. Позволяет написать то, чтонаписано в 5: строке. Без него строка имела бы вид: \_tuple<Head, Tail ...> : \_tuple<Head, Tail ...> // error

4-5: Частичная специализация + наследование. Позволяет «откусывать» голову от передаваемого набора параметров и передавать далее «хвост». При этом, в конечном счете, инстанцирование будет начато именно с самого базового класса (то есть последнего входящего элемента)

7-8: Одинаковое имя переменной ‘base’ при различии \_tuple<Tail ...>

10: Понадобится при написании функции геттера – чтобы сделать эту функцию также шаблонной и способной возвращать любой тип данных (35-36)

22-30: Оборачиваем функцию-геттер ‘get( )’ в структуру, чтобы обойти запрет на частичную специализацию функции (нужно получать Index – 1)

25-26, 35-36: В строках 32-39 находится дно рекурсии для структуры ‘\_getter’. В этой структуре есть функция ‘get( )’, которая и подвергается частичной специализации. Именна эта функция – со дна рекурсии – получает значение из поля ‘head\_’ и затем возвращает его в раскручивающуюся рекурсию. Необходимо обеспечить совместимость вовзращаемого значения из ‘get( )’ со дна рекурсии (37) с возвращаемым значением из ‘get( )’ внутри самой рекурсии (28). Поэтому и вводится одинаковый псевдоним возвращаемого типа ‘return\_type’

41-44: Доступ до значений в \_tuple осущеcтвляется следующим образом:

\_tuple \_t = {1, ‘A’, 3.14};

auto a = \_getter<1, int, char, double>::get(\_t);

Чтобы избежать громоздкости, написана шаблонная функция get< >( ). Она сама вызывает скоснструированный из передаваемого индекса и передаваемого исходного кортежа класс ‘\_getter’

auto a = get<1>(\_t);

**CONCEPTS**

ШАБЛОН БЕЗ СЛОВА TEMPLATE<>

Подробнее: Gregoire Mark – Professional C++, 5th edition, 12 ch

В настройки компилятора:

-fconcepts or –std=c++20 (GCC 9.2.0)

-fconcepts-ts or -std=c++20 (mingw64)

1. decltype(auto) temp (const auto& a, const auto& b)
2. { return a + b; }

CONCEPT = инструмент проверки типа, передаваемого в качестве параметра шаблона на соответствие каким-либо критериям.

GENERIC SYNTAX

1. #include <concepts>
3. template<\*parameters\*>
4. concept NAME =
5. \*constraints\_expression\* ;
7. // using in templates
8. template<NAME T>
9. class ... ;
11. // using as bool
12. bool b = NAME<T>;

FLAGS for compiler: -fconceptc or –std=c++20

constraint = ограничение

requirement = требование

ОГРАНИЧЕНИЯ при определении концепта:

1) только compile-time внутри

2) внутри template<> нельзя использовать другие концепты

CONSTRAINTS EXPRESSION

Вариант 1

A. boolean value = if true or false

Вариант 2

Выражения:

B. simple requirements = if it compiles

C. type requirements = if the type is valid

D. compound requirements = if exceptions / which return type

E. nasted requirements = (вложенные)

Синтаксис:

1. template<typename T>
2. concept NAME =
3. requires(\*args\*)
4. {
5. \*requirement\* ;
6. } ;

3: args are optional

5: semicolon after every requirement

EXAMPLES

1. A. boolean
2. template<typename T>
3. concept NAME =
4. sizeof(T) == 4 && !std::is\_same<T, int>::value
5. ;
7. B. simple
8. template<typename T>
9. concept NAME =
10. requires(T t) { t++; ++t; }
11. ;
13. C. type
14. template<typename T> using REF = T&;
15. template<typename T>
16. cocept NAME =
17. requires
18. {
19. typename T::value\_type;
20. typename REF<T>;
21. } ;
23. D. compound, syntax:
24. { expression } noexcept -> type-constraint ;
25. both ‘noexept’ and ‘type-constraint’ are optional
26. template<typename T>
27. concept NAME =
28. requires(T t)
29. {
30. { t.method\_1( ) } noexept ;
31. { t.method\_2( ) } -> std::same\_as<int> ;
32. } ;
34. E. nasted
35. template<typename T>
36. concept NAME =
37. requires(T t)
38. {
39. requires sizeof(t) == 4 ;
40. t++; ++t;
41. } ;

TYPE CONSTRAINTS

Type constraints = концепт с нулем и более аргументов

Особенность = принимает на один аргумент меньше, чем в определении

1. // concept
2. template<typename T, typename S>
3. concept SAME\_TYPE =
4. std::is\_same<T, S>::value
5. ;
7. // using in other concept
8. template<typename T, typename S>
9. concept NAME =
10. requires(T t, S s)
11. {
12. {t + s} -> SAME\_TYPE<int> ;
13. } ;
14. // using in function/class template
15. template<typename T, SAME\_TYPE<T> S>
16. void function(T t, S s)
17. { cout << “same” << endl; }

В обоих случаяя (12 и 16 строки) выражения идентичны и будут приведенны к единому формату по правилу:

{ A } -> CONCEPT<B> to CONCEPT<A, B>

CONCEPT<B> A to CONCEPT<A, B>

В СТАНДАРТНОЙ БИБЛИОТЕКЕ вместе с концептами поставляется набор готовых type\_constraints четырех типов:

- core language same\_as, convertible\_to …

- comparison equality\_comparable …

- object movable, copyble …

- callable invocable, predicate …

Кроме того, концепты добавлены в <iterator>

Подробнее: Gregoire Mark – Professional C++, 5th edition, p 457

TYPE-CONSTRAINED AUTO

Концепт можно ипользовать совместно со словом auto чтобы ограничить возможные варианты вывода типа. Выводимые auto типы, не прошедшие проверку, не скомпилируются и выдадут ошибку.

1. template<typename T>
2. concept INCR =
3. requires(T t) { t++; ++t; }
4. ;
6. INCR auto var {1} ; // ok
7. INCR auto var {“str”} ; // ok
8. INCR auto var {“str”s}; // error

SOLO REQUIRES

Ключевое слово requires может быть использовано самостоятельно:

A. в шаблонах функций/классов

1. // concept
2. template<typename T, typename S>
3. concept SAME\_TYPE =
4. std::is\_same<T, S>::value
5. ;
7. // change this …
8. template<typename T, SAME\_TYPE<T> S>
9. void function(T t, S s) …
11. // … for this
12. template<typename T, typename S>
13. requires SAME<T, S>
14. void function(T t, S s) …

B. в теле функций

1. auto var = obj.method( );
2. if( requires { sizeof(var) == 4; } )
3. { … }

CONCEPTS and TEMPLATE OVERLOAD

Concepts cannot be overloaded

But functions/class templates might be:

- overloaded using concepts

- or subsumpted

(not at the same time with one concept)

1. template<typename T>
2. concept NAME =
3. std::is\_pointer<T>::value
4. ;
6. template<typename T>
7. void function(T)
8. { std::cout << “is\_pointer? no” << std::endl; }
10. // overloading or …
11. template<NAME T>
12. void function(T)
13. { std::cout << “is\_pointer? yes” << std””endl; }
15. // … or subsumption
16. template<typename T>
17. requires NAME<T>
18. void function(T)
19. { std::cout << “is\_pointer? yes” << std””endl; }
21. int i;
22. int\* p;
23. function(i); // is\_pointer? no
24. function(p); // is\_pointer? yes

TEMPLATE with CONCEPT + AUTO

1. template<typename T>
2. concept \_concept =
3. ...
4. ;
6. void \_function(const \_consept auto& arg) { … }
8. // equal
9. template<\_concept T>
10. void \_function(const T& arg) { … }

**MULTYTHREAD**

*more about: mail.ru* [*video*](https://www.youtube.com/playlist?list=PLrCZzMib1e9oag2Qgg76foi0x88gFs0rh) *course*

[*async and parallelism*](https://habr.com/ru/company/jugru/blog/446562/)

*threads and bloks* [*link*](https://habr.com/ru/post/182610/) *+* [*link*](https://habr.com/ru/post/182626/)

[*memory model and CLR*](https://itnan.ru/post.php?c=1&p=541362)

*atomics* [*link*](https://www.youtube.com/watch?app=desktop&v=_-3syPxgwqs&list=PLQC2_0cDcSKAIWkuo8yft2DlelRUljFy4&index=3)*,* [*link*](https://habr.com/ru/post/517918/)

[*thread safety objects*](https://habr.com/ru/post/328348/)

[*lock free data*](https://habr.com/ru/users/khizmax/posts/)

*coroutine:* [*video*](https://www.youtube.com/watch?v=OE45F3iKtv4)

[*video*](https://www.youtube.com/watch?v=xkshva7DKfQ&t=1112s)

*video* [*part\_1*](https://www.youtube.com/watch?v=R_gZQJC-uv0) *+* [*part\_2*](https://www.youtube.com/watch?v=-cnibVwLCrI)

[*link*](https://habr.com/ru/company/yandex_praktikum/blog/559642/)

[*link*](https://habr.com/ru/company/piter/blog/491996/)

**BASIC INFO**

КОНТЕКСТ ПРОЦЕССА:

системный контекст

регистровый контекст

пользовательский контекст

Дополнительно - <https://textarchive.ru/c-2601643-pall.html>

ПРОЦЕСС • каждый процесс является автономным

• каждый процесс действует в своем контексте

• невозможно взаимодействие напрямую

• создание/переключение/уничтожение = затратны

ПОТОК • часть процесса, делят его адресное пространство

• взаимодействие напрямую

• под поток выделяется его собственный стек

• создание / переключение / уничтожение – дешевые

ВАРИАНТЫ ОБМЕНА ДАННЫМИ (только один из двух!)

активный источник появились данные

сигнал потребителю

прием, если возможно

активный потребитель нужны данные

запрос у источник

отдача всего, что готово

МНОГОПОТОЧНОСТЬ

ГЛАВНОЕ ПРАВИЛО №1 = уменьшать количество точек синхронизации, т.е. ситуаций, в которых необходимо упорядочивать потоки, приводя к последовательному их выполнению

Не смотря на то, что создание потока дешевле создания процесса, выделение стека все же является дорогостоящей операцией. Поэтому для многопоточного приложения есть смысл создавать ПУЛ ПОТОКОВ – набор пустых потоков, которые создаются заранее, а затем могут быть использованы в нужный момент без необходимости задействовать ресурсы на их ситуативное создание

СПОСОБЫ РЕАЛИЗАЦИИ

посредством планировщика =

принудительное переключение контекстов

плюс: надежно, от программы ничего не зависит

минус: переключение контекста дорогое, не всегда есть смысл

кооперативная многозадачность =

за передачу управления отвечает код

плюс: дешевое переключение или даже просто переход по адресу

минус: повышенные требования к надежности кода

**MEMORY FENCE**

ПРОБЛЕМЫ многопоточности – ОПТИМИЗАЦИЯ КОМПИЛЯТОРОМ

так как в контексте функции значение FLAG не меняется, компилятор выбрасывает проверку while(!FLAG) и заменяет на:

void thread\_1 ( )

{

FLAG = false;

while (1) …

1. bool FLAG = false;
2. void thread\_1 ( )
3. {
4. FLAG = false;
5. while (!FLAG)
6. \*do\_something\*
7. }
8. void thread\_2 ( )
9. {
10. \*do\_something\*
11. FLAG = true;
12. }

СПОСОБ РЕШЕНИЯ – ключевое слово volatile

компилятор не будет оптимизировать эту переменную:

1. volatile bool FLAG = false;

ПРОБЛЕМЫ многопоточности – СТРУКТУРА РАБОТЫ ПРОЦЕССОРА

Так как любое обращение к оперативной памяти – дорогая операция, каждое ядро процессора содержит кэш нескольких уровней (буфер, хранящий данные первой необходимости и имеющий максимальную скорость доступа к этим данным)

CORE 0 CORE 1

Registers: EAX, EBX, … ESP Registers: EAX, EBX, … ESP

CACHE, level 1 CACHE, level 1

16 Kb, instructions 16 Kb, instructions

16 Kb, data 16 Kb, data

CACHE, level 2 CACHE, level 2

256 Kb 256 Kb

----------- CACHE, level 3 ------------ common cache ----------

--------------- 8 Mb ------------------------------------------

Однако передача данных от одного ядра к другому так же является дорогой операцией (а именно она необходима, чтобы данные, измененные одним ядром, стали доступны другому ядру). Эта операция называется ИНВАЛИДАЦИЯ КЭША и она может оптимизироваться процессором таким образом, чтобы максимально отсрочить изменение общих для нескольких потоков данных

Поэтому возможны ситуации:

void thread\_1( ) { …

var = funct( );

FLAG = true;

var = funct();

или

void thread\_2( ) { …

while (!FLAG) ;

std::cout …

while (!FLAG) ;

bool FLAG = false;

int var = 0;

int funct( ) { …

return 42;

void thread\_1( ) { …

var = funct( );

FLAG = true;

void thread\_2( ) { …

while (!FLAG) ;

std::cout << “got!”;

СПОСОБ РЕШЕНИЯ – использование барьера памяти

Барьер памяти = инструкция, которая указывает компилятору / процессору (?) строгую последовательность выполнения кода

Операции с памятью делятся на:

load = чтение в регистр из памяти (получение данных)

store = запись из регистра в память (передача данных)

Барьер имеет вид X \_ Y

это означает, что:

все операции все операции

типа X перед барьером типа Y после барьера

будут выполнены будут выполнены

раньше операций после операций

типа Y после барьера типа X перед барьером

КЛЮЧЕВЫЕ КОМБИНАЦИИ БАРЬЕРА

**ACQUIRE** = **«захват»**

семантика LoadLoad \_ LoadStore

значение сначала (перед): все операции чтения

потом (после): любые операции

особенность: не упорядочивает Store операции перед барьером

они могут проникнуть в пространство после барьера

**RELEASE = «освобождение»**

семантика LoadStore \_ StoreStore

значение сначала (перед): все операции

потом (после): операции записи

особенность: не упорядочивает Load операции после барьера

они могут проникнуть в пространство перед барьером

СПОСОБЫ ВЫСТАВЛЕНИЯ БАРЬЕРА

ВАРИАНТ 1 – вручную

# include <atomic>

void atomic\_thread\_fence (\*std::memory\_order order\*); // барьер

enum memory\_order // возможные парaметры

{

memory\_order\_relaxed,

memory\_order\_consume,

memory\_order\_acquire, <- acquire

memory\_order\_release, <- release

memory\_order\_acq\_rel,

memory\_order\_seq\_cst, <- sequence consistency

}

**SEQUENCE CONSISTENCY** (default)

Гарантирует, что код будет исполнен точно в том порядке, в котором он написан программистом. Дорогостоящая операция (> 1 микросекунды). Целесообразнее использовать конкретные барьеры, если есть понимание их работы

ГЛАВНАЯ РЕКОМЕНДАЦИЯ – не разделять изменяемые данные между потоками

ИСПРАВЛЕНИЕ ПРОГРАММЫ

1. bool FLAG = false;
2. int var = 0;
3. int funct( ) { …
4. return 42;
5. void thread\_1( ) { …
6. var = funct( ); // сначала все операции перед барьером
7. std::atomic\_thread\_fence(std::memory\_order\_relese); барьер
8. FLAG = true; // теперь можно выполнить store-операцию
9. void thread\_2( ) { …
10. while (!FLAG) ; // сначала load-операции перед барьером
11. std::atomic\_thread\_fence(std::memory\_order\_aquire); барьер
12. std::cout << “got!”; // теперь можно выполнять код после

ВАРИАНТ 2 – атомарные переменные

#**include** <atomic>

std::atomic<T> var = … ;

Класс ATOMIC имеет

перегруженные операторы для привычной работы с переменными +

методы:

T load(std::memory\_order = memory\_order\_seq\_cst) …

void store(T value, std::memory\_order = memory\_order\_seq\_cst) …

которые автоматически выставляют барьер при выполнении операции с переменной. Барьер по-прежнему дефолтный (целесообразнее пользоваться вручную и конкретными барьерами)

АТОМАРНАЯ ПЕРЕМЕННАЯ ГАРАНТИРУЕТ, ЧТО:

если она будет читаться,

то текущий поток считает актуальное значение

если в нее будет запись,

другие потоки тоже увидят актуальное значение

ИСПРАВЛЕНИЕ ПРОГРАММЫ

1. std::atomic<bool> FLAG = false;
2. int var = 0;
3. int funct( ) { …
4. return 42;
5. void thread\_1( ) { …
6. var = funct( );
7. FLAG.store(true); \\ записываем в переменную
8. void thread\_2( ) { …
9. while (!FLAG.load()) ; \\ читаем из переменной
10. std::cout << “got!”;

**THREAD CREATION**

1. include <thread>
2. **std::thread th( … );**
3. // функция не возвращает значений
4. // если передавать по ссылке, то только: (const type& var)
5. void f\_no\_ret(\*parameters\*) { … }
6. // функция возвращает значение по ссылке
7. void f\_ref\_ret(type& var) { var = … }
8. // функция возвращает значение через return
9. type f\_return(type& var) { return var; }
10. int main( ) { …
11. // функция без возврата
12. std::thread th(f\_no\_ret, \*arguments\*);
13. // возврат значения по ссылке
14. type var = … ;
15. std::thread th(f\_ref\_ret, std::ref(var));
16. // возврат значения с использованием lambda-выражения
17. type var = … ;
18. auto lambda = [&var]( ){ var = f\_return(\*arguments\*); }
19. std::thread th(lambda)
20. // каждому потоку должно соответствовать
21. **th.detach( ); или th.join( );**

.JOIN( ) родительский поток будет дожидаться окончания

выполнения производного потока в той точке,

где использован этот метод

.DETACH( ) производный поток будет выполняться на фоне,

даже если происходит выход из области видимости

внутри родительского потока

Если не вызван один из этих методов, деструктор класса thread вызовет std::terminate( )

1. class CLASS { …
2. public:
3. // static метод
4. static void f\_stat( ) { … }
5. // метод не возвращает значений
6. void f\_no\_ret(\*parameters\*) { … }
7. // метод возвращает значение через return
8. type f\_return(type& var) { return var; }
9. int main( ) { …
10. CLASS obj;
12. // static метод = как обычная функция (с оператором ::)
13. std::thread th(CLASS::f\_stat, \*arguments\*);
14. // метод без возврата
15. std::thread th(**&**CLASS::f\_no\_ret, obj, \*arguments\*);
16. // возврат значения с использованием lambda-выражения
17. type var = … ;
18. auto lambda = [&var]( ){var = obj.f\_return(\*arguments\*);}
19. std::thread th(lambda)
20. // каждому потоку должно соответствовать
21. th.detach( ); или th.join( );

Важно! Если необходимо передавать аргументы в метод класса, вызываемый в потоке, то они должны передаваться:

1. class PARAM { … } p;
2. class CLASS { …
3. void method(PARAM … ) { … } } c;

- либо по значению thread(&CLASS::method, c, p);

- либо как указатели thread(&CLASS::method, c, &p);

- либо std::ref( ) thread(&CLASS::method, c, std::ref(p));

USER’s CLASS

Чтобы передать в поток функцию, параметром которой является пользовательский тип данных, необходимо использовать указатель

1. class CLASS { …
2. void function(CLASS\* obj) { …
3. CLASS obj;
4. std::thread th(function, &obj);
5. th.join( );

TEMPLATE

1. template <class T>
2. class CLASS { …
3. template <typedef T>
4. void function(T\* obj) { …

Чтобы вызывать подобную функцию в потоке, необходимо явно указать специализацию шаблона при вызове. При этом, если принимаемый параметр тоже является шаблонным типом, то и его спеиализация также должна быть указана явным образом

1. CLASS<int> obj;
2. std::tread th(function<CLASS<int>>, 42);
3. th.join( );

ФУНКЦИОНАЛ ПОТОКОВ через использование std::this\_thread

// задержка времени

include <chrono>

std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

// уникальный ID потока

const std::thread::id = std::this\_thread::get\_id;

// указать планировщику снять поток с выполнения до след. раза

// (другими словами, не-до-использовать свой квант времени -

// когда очередь дойдет снова, все продолжится штатно)

std::this\_thread::yield( );

ФУНКЦИОНАЛ ПОТОКОВ через использование std::async

СОЗДАНИЕ ПОТОКА – ВРУЧНУЮ или на усмотрение КОМПИЛЯТОРА

#include <async>

std::async(std::launch::async, \*functor\*, \*arguments\*)

std::async(std::launch::deffered, \*functor\*, \*arguments\*)

std::launch::async = гарантированно новый поток

std::launch::deffered = на усмотрение компилятора

Преимущество – не думать о .join() и .detach()

**THREADS INTERACTION**

ФУНКЦИОНАЛ ПОТОКОВ через использование std::future и std::promise

ОБМЕН ИНФОРМАЦИЕЙ МЕЖДУ ПОТОКАМИ

ВАРИАНТ\_1 = только STD::FUTURE<>

(std::promise<> находится под капотом sdt::async)

#include <future>

**1** std::future<T> fut = std::async(std::launch::async, [](){return T;})

**2** fut.wait( );

**3** T var = fut.get( );

**4** auto status = fut.wait\_for(std::chrono:milliseconds(100));

1. создается объект, принимающий возвращаемое из потока значение

можно использовать ключевое слово auto вместо std::future<T>

2. дождаться окончания работы потока – как .join( )

возвращенное значение можено получить потом

3. одновременно:

дождаться окончания работы потока - как .join( )

и получить возвращенное значение в переменную

ВАЖНО: после первого извлечения метод .get( ) становится невалидным

Чтобы проверить, вызвать - fut.valid( ) - вернет 0 или 1

4. получить статус потока:

status == std::future\_status::deffered <- еще не стартовал

status == std::future\_status::timeout <- результата пока нет

status == std::future\_status::ready <- все готово

ВАРИАНТ\_2 = STD::FUTURE<> + STD::PROMISE<>

Пример функции, чья задача – запускать параллельный поток

1. #include <future>
2. #include <thread>
3. std::future<T> LAUNCHER ( \*parameters\* )
4. {
5. **1.** std::promise<T> prom;
6. **2.** std::future<T> fut = **3.** prom.get\_future( );
8. **4.** auto lambda = [ ] (**5.** std::promise<T>&& prom\_l)
9. {
10. **6.** prom\_l.set\_value( **VAL** );
11. };
13. **7.** std::thread th (lambda, prom);
14. th.detach( );
15. **8.** return fut;
16. }
17. int main( )
18. {
19. std::future<T> fut = LAUNCHER( \*arguments\* );
20. **9. T var** = fut.get( );
21. std::cout << var << std::endl;
22. }
23. создать объект класса std::promise<\*тип возврата\*>
24. создать объект класса std::future <\*тип возврата\*>
25. метод .get\_future( ) получает значение «из будущего»
26. создать функтор, который будет запущен в отдельном потоке
27. параметр = std::promise<> из пункта 1 (именно перемещение)
28. метод .set\_value( ) см. пункт 3 – то самое значение из будущего:теперь в объекте класса std::future<> содержится ожидаемое значение
29. запуск отдельного потока – в качестве аргумента – объект std::promise<>
30. теперь можно вернуть ожидаемое значение из LAUNCHER( ) в main( )
31. результат: **T var = VAL**

ОБРАБОТКА ИСКЛЮЧЕНИЯ ВНУТРИ ПОТОКА

КЛЮЧЕВАЯ ИДЕЯ – использование STD::FUTURE

1. void EXCEPTION(std::promise<T>&& prom\_e)
2. {
3. \*exception\_start\*
4. try
5. {
6. \*exception\_try\*
7. throw std::runtime\_error(“\_exception\_error\_”);
8. throw -42;
9. }
10. catch(...)
11. {
12. \*exception\_catch\*
13. prom\_e.set\_exception(std::current\_exception( ));
14. }
15. \*exception\_end\*
16. }
17. int main( )
18. {
19. \*main\_start\*
20. std::promise<T> prom\_m;
21. std::future<T> fut\_m = prom.get\_future( );
22. std::thread th(EXCEPTION, std::move(prom\_m));
23. th.join;
24. try
25. {
26. \*main\_try\*
27. fut\_m.get( );
28. }
29. catch(std::exception& exc)
30. {
31. \*main\_catch\_std\*
32. std::cout << exc.what( ) << std::endl;
33. }
34. catch(int) <- для варианта строки 7
35. {
36. \*main\_catch\_int\*
37. }
38. \*main\_end\*
39. }

Результат выполнения: main\_start

exception\_start

exception\_try

exception\_catch

exception\_end

main\_try

main\_catch\_std

\_exception\_error\_

main\_end\*

Описание:

20: создание объекта std::promise<> для возврата значения из потока – в данном примере (хотя и указан ожидаемый тип <T> ), оно используется только в контексте работы с исключением, поэтому во всем примере отсутствует работа с возвратом к-либо полезного значения

21: создание объекта std::future<> для приема значения из потока

23: создание потока – в качестве параметра передается объект std::promise<>. Этот объект должен использовать семантику

перемещения, поэтому он оборачивается в std::move( ) - только таким образом он сможет вернуть значение из потока

1: функция, выполняющаяся в отдельном потоке. Тип возвращаемого значения void, так как для возврата значения используется принимаемый объект std::promise<>&&

7: внутри блока try выбрасывается исключение

оно может быть любого типа

9: внутри функции параллельного потока не нужно использовать конкретные catch – подойдет catch\_all, так как …

12: … так как std::current\_exception( ) принимает исключение любого типа (используя под капотом std::exception\_ptr).

А само исключение с помощью std::promise::set\_exception( ) и посредством связи future-promise (см. описание стр. 21) помещается в объект std::future<>

26: исключение, выброшенное в потоке было в потоке обработано таким образом, чтобы только передать его в main( ) – поэтому для полноценной его обработки необходим try

29: при вызове метода std::future::get( ) выбрасывается то самое исключение из потока, находящееся в объекте std::future<>. Обратить внимание, что тип объекта std::future<T>, хотя и предполагает работу с конкретным типом данных для полезной работы (см. описание стр. 20), все равно способно хранить и выбрасывать исключение

31: полноценная обработка внутри main( ) того исключения, которое было выброшено в потоке в строке 7 – именно здесь и необходимы конкретные catch-блоки в зависимости от того типа данных, которого на самом деле является исключение (например, для альтернативного варианта строки 7 необходим и альтернативный catch-блок). Обратить внимание, что исключение типа std::exception& принимается по ссылке – в противном случае будет выведено не сообщение из строки 7, а просто «std::exception»

36: после обработки исключения продолжается выполнение кода основного потока (в данном случае, просто конец программы)

СОЗДАНИЕ ЛАУНЧЕРА для запуска ФУНКЦИЙ в отдельном ПОТОКЕ

и ВОЗВРАТ значения / исключения из этого потока в основной:

файл: *«thread\_exception.cpp»*

STD::PACKAGED\_TASK< ( )>

Для вызова функтора в параллельном потоке существует обертка

аналогичная по своей сути обертке std::function< ( )> :

позволяет вернуть значение из потока

позволяет вернуть сгенерированное в потоке исключение

реализованная засчет future и promise

можно использовать массив из оберток

1. int EXCEPTION(int parameter)
2. {
3. if (parameter) return 42;
4. else throw -1;
5. }
6. std::future<int> LAUNCHER( )
7. {
8. std::packaged\_task<int(int)> task(EXCEPTION);
9. auto fut = task.get\_future( );
10. std::thread th(task, 1); <- или
11. std::thread th(task, 0); <- или
12. th.detach( );
13. return fut;
14. }
15. int main( )
16. {
17. int result = 0;
18. auto fut = LAUNCHER( );
20. try
21. {
22. result = fut.get( );
23. }
24. catch(int)
25. {
26. std::cout << “int\_error” << std::endl;
27. }
29. std::cout << “the result is: ” << result << std::endl;
30. }

**MUTEX**

1. std::mutex mtx;
2. void function\_1( ) { …
3. mtx.lock( );
4. \*защищаемый\_код\*
5. mtx.unlock( );
6. void function\_2( ) { …
7. mtx.lock( );
8. \*защищаемый\_код\*
9. mtx.unlock( );
10. int main( ) { …
11. std::thread th\_1(function\_1);
12. std::thread th\_2(function\_2);

Теперь, какой бы из потоков ни добрался до любого mtx.lock( ), по ВСЕМУ коду, все участки между всеми mtx.lock( ) и mtx.unlock( ) будут недоступны для любого из потоков.

Другими словами, mutex защищает не сами объекты в памяти, а инструкции кода, которые к этим данным обращаются. Поэтому задача помнить о том, каким участки памяти / объекты где используются остается за программистом

Использование времени при работе с mutex’ом

1. std::timed\_mutex mut;
2. mut.lock( );
3. if (mut.try\_lock( )) // попытка .lock( ) – вернет 0 или 1
4. mut.unlock( ); // если успешно, то не забыть .unlock( )
5. // если не успешно, можно продолжать выполнение кода здесь
6. // проверяет mutex «в течение … »
7. auto period = std::chrono::milliseconds(100);
8. if (mut.try\_lock\_for(period)) mut.unlock;
9. // проверяет mutex «до момента … »
10. // steady\_clock – часы с защитой от перевода системного времени
11. auto now = std::chrono::steady\_clock::now( );
12. mut.try\_lock\_until(now + std::chrono::seconds(1));

RECURSIVE\_MUTEX

Для рекурсивных функций вместо std::mutex mtx

необходимо использовать std::recursive\_mutex r\_mtx

Особенность в том, что он позволяет одному потоку обращаться к защищенному коду неограниченно количество раз – однако при этом для разблокировки r\_mtx.unlock( ) необходимо использовать ровно столько же раз, сколько раз было использовано r\_mtx.lock( )

LOCK\_GUARD

1. std::mutex mtx;
2. void function( )
3. {
4. \*код\*
5. std::lock\_guard<std::mutex> guard(mtx); = mtx.lock()
6. \*защищаемый\_код\*
7. } = mtx.unlock( ) здесь

Опасность mutex’а: после mtx.lock( ) при невыполнении защищаемого кода до конца (например, выброшенное исключение или return из ветвления) не будет исполнен mtx.unlock( )

lock\_guard позволяет не заботиться об этом, так как объект этого класса уничтожается при выходе из области видимости (как и любой объект любого класса) – а в его деструкторе вызывается тот самый mtx.unlock( )

Для изменения области видимости можно просто использовать **{ … }**.

1. void function( )
2. {
3. \*код\*
4. **{**
5. std::lock\_guard<std::mutex> guard(mtx);
6. \*защищаемый\_код\*
7. **}**
8. \*код\*
9. }

Альтернатива: UNIQUE\_LOCK

приоритет: lock\_guard

частные случаи: unique\_lock

1. std::mutex mtx; // создать mutex
2. // если просто (mtx), то mtx.lock( ) вызовется немедленно
3. std::unique\_lock<std::mutex> unique(mtx, std::defer\_lock)
4. {
5. unique.lock( );
6. \*защищаемый\_код\*
7. unique.unlock( );
8. } // если без unique.unlock( ), то автоматически здесь

Методы:

.lock( )

.unlock( )

.try\_lock( )

.try\_lock\_for( )

.try\_lock\_until( )

.swap( )

.release( )

Для атомарного захвата двух mutex’ов:

1. // создать два mutex’a
2. std::mutex mtx1;
3. std::mutex mtx2;
4. // создать два unique\_lock’а
5. std::unique\_lock<std::mutex> unique1(mtx1, std::defer\_lock);
6. std::unique\_lock<std::mutex> unique2(mtx2, std::defer\_lock);
7. // использовать:
8. std::lock(unique1, unique2);

**CONDITIONAL VARIABLES**

КОГДА НЕОБХОДИМО 1ый ПОТОК УВЕДОМИТЬ О ЧЕМ-ТО ИЗ 2ого ПОТОКА

1. #include <condition\_variable>
2. std::mutex mtx;
3. std::conditional\_variable cond;
4. int data = 0;
5. // поток, который будет ждать
6. void f\_wait( )
7. {
8. std::unique\_lock<std::mutex> unique(mtx);
9. while (!data) // если данные еще не готовы
10. cond.wait(unique); // поток засыпает
11. }
12. // поток, который выполняет операцию
13. void f\_do( )
14. {
15. {
16. std::lock\_guard<std::mutex> unique(mtx);
17. data = 1; // подготовиваются данные
18. }
19. cond.notify\_one( ); // разбудить случ. спящий поток
20. cond.notify\_all( ); // … или все спящие потоки
21. }

**COROUTINE**

ПРОБЛЕМЫ

1. Из функции сложно вернуть несколько значений – их приходится либо упаковывать в контейнер, либо пользоваться итераторами …

2. В высоконагруженных приложениях (например, на сервере) время ожидания ответа превосходит реальную полезную работу.

СПОСОБ РЕШЕНИЯ

Реализовать такой способ работы функций, когда каждая конкретная функция не привязана к потоку. Это можно сделать, используя в функции (грубо говоря) такую "точку" в которой функция прекращает (так как ей необходимо ждать) свое выполнение, а поток тем временем берется за выполнение другой (ожидающей) функции. Затем выполнение функции продолжается с того самой "точки".

Возможны даже ситуации, когда функция начинает выполнение в одном потоке, а закнчивает его в другом ([link](https://en.cppreference.com/w/cpp/language/coroutines#Example)).

COROUTINE SYNTAX

Корутина = функция, которая содержит одно из ключевых слов:

co\_await - "точка" прерывания-продолжения

co\_yield - возврат значения + co­\_await

co\_return - завершение работы функции

Тип возвращаемого значения = proxy-класс. Все взаимодействие осуществляется через объект этого класса, который был возвращен при запуске функции. С точки зрения коллера функция обычная.

прокси класс далее = PRX произвольное имя

объект далее = crt произвольное имя

PRX может иметь произвольное определение, но обязан содержать

вложенную структуру struct PRX::promise\_type { }

поле std::coroutine\_handle<promise\_type> member;

struct PRX::promise\_type { } = описание характеристик корутины

может иметь следующие методы =

PRX get\_return\_object ( ) ; обязательно

static void unhandled\_exception ( ) ; обязательно

? initial\_suspend ( ) ; обязательно

? final\_suspend ( ) ; обязательно

? yield\_value (? value) ; if co\_yield is used

? return\_value (? value) ; if co\_return is used

? await\_transform (? value) ; if co\_await is used

? await\_resume ( ) ; if co\_await is used

std::coroutine\_handle<promise\_type> получает на вход в качестве шаблонного параметра структуру с описанием характеристик корутины – таким образом принимает управление корутиной на себя. Взаимодействие с корутиной осуществляется через 'member'

описывает использует для управления

взаимодействует

Программист promise\_type std::coroutine\_handle<promise\_type>

1. template<std::movable T>
2. class PRX
3. {
4. std::coroutine\_handle<promise\_type> m\_handler;
6. public:
7. struct promise\_type
8. {
9. std::optional<T> current\_value;
11. PRX<T> get\_return\_object( )
12. {
13. return PRX{std::coroutine\_handle<promise\_type>::from\_promise(\*this);
14. }
16. [[noreturn]] static void unhandled\_exeption( ) { throw; }
18. static std::suspend\_always initial\_suspend( ) noexcept { return { }; }
19. static std::suspend\_always final\_suspend( ) noexcept { return { }; }
21. std::suspend\_never return\_value(T value) noexcept
22. {
23. current\_value = std::move(value);
24. return { };
25. }
26. };
28. explicit PRX(const std::coroutine\_handle<promise\_type> handler)
29. : m\_handler {handler} { }
31. PRX(PRX&& other) noexcept
32. : m\_handler { std::exchange(other.m\_handler, { }) } { }
34. PRX(const PRX&) = delete;
35. PRX& operator= (const PRX&) = delete;
37. ~PRX( ) { if(m\_handler) m\_handler.destroy( ); }
39. T get\_value( )
40. {
41. if(!m\_handler.promise( ).current\_value) m\_handler.resume( );
42. return std::move(\*m\_handler.promise( ).current\_value);
43. }
44. };
46. template<std::integral T>
47. PRX<T> lazy\_sum(T x, T y)
48. { co\_return x + y; }
50. int main( )
51. {
52. PRX<int> crt = lazy\_sum(42, 58);
53. std::cout << crt.get\_value( ) << std::endl;
54. }

53 вызов функции

48 функция = корутина, так как

48 возвращает прокси-класс

49 есть одно из ключевых слов (co\_return)

7 по характеристикам из promise\_type

29 при вызове функции создается корутина

4 и связывается с handler'ом

18 инициализации корутины она всегда приостанавливается

19 при окончании корутины она всегда приостанавливается

21 при вовзрате значения из функции работает этот метод

23 помещает возвращаемое значение

9 в current\_value,

21 а сама корутина не приостанавливается

54 запрос возвращенного из функции значения

39 вызывается этот метод

41 обращение к корутине через метод handler'а

42 возврат запрашиваемого из main( ) значения

**LOW LEVEL**

*more about: …*

**INLINE ASSEMBLER**

Формат встроенного ассемблерного кода

1. \_\_asm\_\_
2. (
4. // назначение синтаксиса
5. ".intel\_syntax noprefix \n"
6. // ассемблерный код
7. " <command> \n"
8. " <command> \n"
9. " <command> \n"
10. // дополнительная информация
11. : "=<constraint>" (<var>) // output
12. : "<constraint>" (<var>) // input
13. : "<clobberd\_register>" // clobered
14. );

Команды кода должны быть:

- в кавычках

- заканчиваться ; либо \n

Дополнительная информация:

- необязательно указывать

- обязательно прописать все три двоеточия

- <constraint> = register constraint (ограничение),

Можно написать: a ⇨ rax, eax, ax, al

b ⇨ rbx, ebx, bx, bl

c ⇨ rcx, ecx, cx, cl

d ⇨ rdx, edx, dx, dl

S ⇨ rsi, esi, si

D ⇨ rdi, edi, di

r ⇨ любой регистр

- (<var>) = переменная, ассоциированная с регистром

- <clobberd … > = разрушаемый регистр

и не указанный как input/output

Пример

1. int i = 1, j = 2, sum;
2. \_\_asm\_\_
3. (
5. ".intel\_syntax noprefix \n"
6. " mov rdx, rbx \n"
7. " add rdx, rcx \n"
8. " mov rax, rdx \n"
9. : "=a" (sum)
10. : "b" (i), "c" (j)
11. : "rdx"
12. );

При использовании 32-битных регистров доступны номера операндов

" mov edx, %1 \n"

" add edx, %2 \n"

" mov %0, edx \n"

: "=a" (sum) // %0

: "b" (i), "c" (j) // %1, %2

: "edx"

**INTRISICS**

доступные интрисики - [link](https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=SSE) или «Intel, intrisics»

Разбор на примере сравнения строк

\_mm\_cmpistri , стр. 1583

\_mm\_loadu\_si128 , стр. 1648

1. #include <intrin.h>
2. const char\* find = "ys";
3. const char\* in = "any symbols";
5. \_\_m128i a = \_mm\_loadu\_si128((\_\_m128i\*) find);
6. \_\_m128i b = \_mm\_loadu\_si128((\_\_m128i\*) in );
8. const int mode = 0;
9. int idx = \_mm\_cmpistri(a, b, mode); // 2

в строках 4 и 5 правильно : reinterpret\_cast<const \_\_m128i\*>( )